FLEMMING NIELSON
HANNE RIIS NIELSON
CHRIS HANKIN

Principles of Program Analysis

Springer

Principles of Program Analysis

This One
YR6O-C9Q-UD3R

Flemming Nielson Hanne Riis Nielson
Chris Hankin

Principles of Program Analysis

With 56 Figures and 51 Tables

Springer

Flemming Nielson
Hanne Riis Nielson

The Technical University of Denmark
Informatics and Mathematical Modelling
Richard Petersens Plads, Bldg. 321
2800 Kongens Lyngby, Denmark
E-mail: {nielson, riis}@imm.dtu.dk
WWW: http://www.imm.dtu.dk/~riis/PPA/ppa.html

Chris Hankin

Department of Computing
The Imperial College of Science, Technology, and Medicine
180 Queen's Gate, London SW7 2BZ, UK
E-mail: clh@doc.ic.ac.uk

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Nielson, Flemming:
Principles of program analysis: with 51 tables/F. Nielson; H.R. Nielson; C. Hankin. – Berlin; Heidelberg; New York; Barcelona; Hong Kong; London; Milan; Paris; Singapore; Tokyo: Springer, 1999
ISBN 3-540-65410-0

ACM Computing Classification (1998): F.3.2, D.3, F.3, D.2

ISBN 3-540-65410-0 Springer Berlin Heidelberg New York
1st edition 1999. Corrected 2nd printing 2005

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable for prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media
springeronline.com

© Springer-Verlag Berlin Heidelberg 1999, 2005
Printed in Germany

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by the authors
Cover design: Erich Kirchner, Heidelberg
Production: LE-TeX Jelonek, Schmidt & Vöckler GbR, Leipzig
Printed on acid-free paper 45/3142/YL - 5 4 3 2 1 0

Preface

Aims of the book. Our motivation for writing this book is twofold. One is that for quite some time we have lacked a textbook for our own courses on program analysis; instead we have been forced to base the courses on conference papers supplemented with the occasional journal paper or chapter from a text book out of print. The other is the growing feeling that the various subcommunities in the field often study similar problems without being sufficiently aware of insights or developments generally known in other subcommunities. The idea then emerged that perhaps a text book could be written that both would be useful for advanced courses on program analysis and that would help increase the awareness in the field about the many similarities between the various approaches.

There is an important analogy to complexity theory here. Consider researchers or students looking through the literature for a clever algorithm or heuristics to solve a problem on graphs. They might come across papers on clever algorithms or heuristics for solving problems on boolean formulae and might dismiss them on the grounds that graphs and boolean formulae surely are quite different things. Yet complexity theory tells us that this is quite a mistaken point of view. The notions of log-space reduction between problems and of NP-complete problems lead to realising that problems may appear unrelated at first sight and nonetheless be so closely related that a good algorithm or heuristics for one will give rise to a good algorithm or heuristics for the other. We believe it to be a sad fact that students of programming languages in general, and program analysis in particular, are much too often allowed to get away with making similarly naive decisions and arguments. Program analysis is still far from being able to precisely relate ingredients of different approaches to one another but we hope that this book will help to bring this about. In fact, we hope to shock quite a number of readers by convincing them that there are important similarities between the approaches of their own work and other approaches deemed to be of no relevance.

This book concentrates on what we believe to be the four main approaches to program analysis: Data Flow Analysis; Constraint Based Analysis; Abstract

Interpretation; and Type and Effect Systems. For each approach we aim at identifying and describing the important general principles, rather than presenting a cook-book of techniques and language constructs, with a view to how the principles scale up for more complex programming languages and analyses.

As a consequence we have deliberately decided that this book should not treat a number of interesting approaches, some of which are dear to the heart of the authors, in order to be able to cover the four main approaches to some depth; the approaches not covered include denotationally based program analysis, projection analysis, and logical formulations based on Stone dualities. For reasons of space, we have also had to omit material that would have had a natural place in the book; this includes a deeper treatment of set constraints, fast techniques for implementing type based analyses, single static assignment form, a broader treatment of pointer analyses, and the interplay between analysis and transformation and how to efficiently recompute analysis information invalidated by the transformation.

How to read the book. The book is relatively self-contained although the reader will benefit from previous exposure to discrete mathematics and compilers. The main chapters are generally rigorous in the early parts where the basic techniques are covered, but less so in the later parts where more advanced techniques are covered.

Chapter 1 is intended to be read quickly. The purpose of the chapter is to give an overview of the main approaches to program analysis, to stress the approximate nature of program analysis, and to make it clear that seemingly different approaches may yet have profound similarities. We recommend reading through all of Chapter 1 even though the reader might want to specialise in only parts of the book.

Chapter 2 introduces Data Flow Analysis. Sections 2.1 to 2.4 cover the basic techniques for intraprocedural analysis, including the Monotone Frameworks as generalisations of the Bit Vector Frameworks, and a worklist algorithm for computing the information efficiently. Section 2.2 covers more theoretical properties (semantic correctness) and can be omitted on a first reading. The presentation makes use of notions from lattice theory and for this we refer to Appendix A.1, A.2 and parts of A.3. — Section 2.5 is a more advanced section that gives an overview of interprocedural analysis including a treatment of call string based methods and methods based on assumption sets; since Section 2.5 is used as a stepping stone to Chapter 3 we recommend to read at least up to Subsection 2.5.2. Section 2.6 is an advanced section that illustrates how the relatively simple techniques introduced so far can be combined to develop a very complex shape analysis, but the material is not essential for the remainder of the book.

Chapter 3 covers Constraint Based Analysis. The treatment makes a clear distinction between determining the safety of an analysis result and how to compute the best safe result; it also stresses the need to analyse open systems. Sections 3.1, 3.3 and 3.4 cover the basic techniques; these include coinduction which is likely to be new to most readers and we refer to the treatment in Appendix B (building upon Tarski's theorem as covered in Appendix A.4). Section 3.2 covers more theoretical properties (semantic correctness and the existence of best solutions) and can be omitted on a first reading. — Sections 3.5 and 3.6 extend the development so as to link up with the treatment of Data Flow Analysis. Section 3.5 shows how to incorporate Monotone Frameworks (Section 2.3) and Section 3.6 shows how to add context in the manner of call strings and assumption sets (Section 2.5).

Chapter 4 covers Abstract Interpretation in a programming language independent fashion in order to stress that it can be integrated both with Data Flow Analysis and Constraint Based Analysis. Section 4.1 introduces some of the key considerations and is used to motivate some of the technical definitions in later sections. Section 4.2 deals with the use of widening and narrowing for approximating fixed points and Sections 4.3 deals with Galois connections; this order of presentation has been chosen to stress the fundamental nature played by widenings but the sections are largely independent of one another. — Sections 4.4 and 4.5 study how to build Galois connections in a systematic manner and how to use them for inducing approximate analyses; the material is not essential for the remainder of the book.

Chapter 5 covers Type and Effect Systems which is an approach to program analysis that is often viewed as having a quite different flavour from the approaches covered so far. Section 5.1 presents the basic approach (by linking back to the Constraint Based Analyses studied in Chapter 3) and suffices for getting an impression of the approach. Section 5.2 studies more theoretical properties (semantic correctness) and Section 5.3 studies algorithmic issues (soundness and completeness of a variation of algorithm $\mathcal{W}$) and these sections can be omitted on a first reading. — Sections 5.4 and 5.5 gradually introduce more and more advanced Type and Effect Systems.

Chapter 6 presents algorithms for Data Flow Analysis and Constraint Based Analysis. The treatment concentrates on general techniques for solving systems of inequations. We emphasise the fact that, to a large extent, the same set of techniques can be used for a number of different approaches to program analysis. Section 6.1 presents a general worklist algorithm, where the operations on the worklist constitute an abstract data type, and its correctness and complexity is established. Section 6.2 organises the worklist so that iteration takes place in reverse postorder and the Round Robin Algorithm is obtained as a special case. Section 6.3 then further identifies strong components and iterates through each strong component in reverse postorder before considering the next.

Appendices A and C review the concepts from partially ordered sets, graphs and regular expressions that are used throughout the book. Appendix B is more tutorial in nature since coinduction is likely to be a new concept for most readers.

To help the reader when navigating through the book we provide a table of *contents*, a *list of tables*, and a *list of figures* at the front of the book and an *index* of the main concepts and an *index of notation* at the end of the book. The index of notation is divided into three parts: first the mathematical symbols (with an indication of where they take their parameters), then the notation beginning with a greek letter, and finally the notation beginning with a letter in the latin alphabet (regardless of the font used). The book concludes with a *bibliography*.

Our notation is mostly standard and is explained when introduced. However, it may be helpful to point out that we will be using "iff" as an abbreviation for "if and only if", that we will be using $\cdots[\cdots \mapsto \cdots]$ to mean both syntactic substitution as well as update of an environment or store, and that we will be writing $\cdots \rightarrow_{\mathsf{fin}} \cdots$ for the set of finitary functions: these are the partial functions with a finite domain. We also use λ-notation for functions when it improves the clarity of presentation: $\lambda x. \cdots x \cdots$ stands for the unary function f defined by $f(x) = \cdots x \cdots$.

Most proofs and some technical lemmas and facts are in small print in order to aid the reader in navigating through the book.

How to teach from the book. The book contains more material than can be covered in a one semester course. The pace naturally depends on the background of the students; we have taught the course at different paces to students in their fourth year as well as to Ph.D.-students with a variety of backgrounds. Below we summarise our experiences on how many lectures are needed for covering the various parts of the book; it supplements the guide-lines presented above for how to read the book.

Two or three lectures should suffice for covering all of Chapter 1 and possibly some of the simpler concepts from Appendix A.1 and A.2.

Sections 2.1, 2.3 and 2.4 are likely to be central to any course dealing with data flow analysis. Three to four lectures should suffice for covering Sections 2.1 to 2.4 but five lectures may be needed if the students lack the necessary background in operational semantics or lattice theory (Appendix A.1, A.2 and parts of A.3). — Sections 2.5 and 2.6 are more advanced. One or two lectures suffice for covering Section 2.5 but it is hard to pay justice to Section 2.6 in less than two lectures.

Four or five lectures should suffice for a complete treatment of both Chapter 3 and Appendix B; however, the concept of coinduction takes some time to get used to and should be explained more than once.

Four or five lectures should suffice for a complete treatment of Chapter 4 as well as Appendix A.4.

About four lectures should suffice for a complete treatment of Chapter 5.

Two lectures should suffice for Chapter 6 but three lectures may be needed if large parts of Appendix C need to be reviewed.

We have always covered the appendices as an integrated part of the other chapters; indeed, some of the foundations for partial orders are introduced gently in Chapter 1 and most of our students have had some prior exposure to partial orders, graphs and regular expressions.

The book contains numerous exercises and several mini projects which are small projects dealing with practical or theoretical aspects of the development. Many of the important links and analogies between the various chapters are studied in the exercises and mini projects. Some of the harder exercises are starred.

Acknowledgements. We should like to thank Reinhard Wilhelm for his long and lasting interest in this project and for his many encouraging and constructive remarks. We have also profited greatly from discussions with Alan Mycroft, Mooly Sagiv and Helmut Seidl about their perspective on selected parts of the manuscript. Many other colleagues have influenced the writing of the book and we should like to thank them all; in particular Alex Aiken, Torben Amtoft, Patrick Cousot, Laurie Hendren, Suresh Jagannathan, Florian Martin, Barbara Ryder, Bernhard Steffen. We are grateful to Schloss Dagstuhl for having hosted two key meetings: a one-week meeting among the authors in March of 1997 (during which we discovered the soothing nature of Vangelis' *1492*) and our advanced course in November of 1998. Warm thanks go to the many students attending the advanced course in Dagstuhl, and to the many students in Aarhus, London, Saarbrücken and Tel Aviv that have tested the book as it evolved ever so slowly. Finally, we should like to thank Alfred Hofmann at Springer for a very satisfactory contract and René Rydhof Hansen for his help in tuning the LaTeX commands.

Aarhus and London
August, 1999

Flemming Nielson
Hanne Riis Nielson
Chris Hankin

Preface to the second printing. In this second printing we have corrected all errors and shortcomings pointed out to us. We should like to thank Torben Amtoft, John Tang Boyland, Jurriaan Hage and Mirko Luedde as well as our students for their observations.

Official web page. Further information about the book is available at the web page `http://www.imm.dtu.dk/~riis/PPA/ppa.html`. Here we will

provide information about availability of the book, a list of misprints (initially empty), pointers to web based tools that can be used in conjunction with the book, and transparencies and other supplementary material. Instructors are encouraged to send us teaching material for inclusion on the web page.

Lyngby and London
October, 2004

Flemming Nielson
Hanne Riis Nielson
Chris Hankin

Contents

1 Introduction 1

1.1 The Nature of Program Analysis 1

1.2 Setting the Scene 3

1.3 Data Flow Analysis 5

1.3.1 The Equational Approach 5

1.3.2 The Constraint Based Approach 8

1.4 Constraint Based Analysis 10

1.5 Abstract Interpretation 13

1.6 Type and Effect Systems 17

1.6.1 Annotated Type Systems 18

1.6.2 Effect Systems 22

1.7 Algorithms 25

1.8 Transformations 27

Concluding Remarks 29

Mini Projects 29

Exercises 31

2 Data Flow Analysis 35

2.1 Intraprocedural Analysis 35

2.1.1 Available Expressions Analysis 39

2.1.2 Reaching Definitions Analysis 43

2.1.3 Very Busy Expressions Analysis 46

2.1.4 Live Variables Analysis 49

2.1.5 Derived Data Flow Information 52

2.2 Theoretical Properties 54
2.2.1 Structural Operational Semantics 54
2.2.2 Correctness of Live Variables Analysis 60
2.3 Monotone Frameworks 65
2.3.1 Basic Definitions 67
2.3.2 The Examples Revisited 70
2.3.3 A Non-distributive Example 72
2.4 Equation Solving 74
2.4.1 The MFP Solution 74
2.4.2 The MOP Solution 78
2.5 Interprocedural Analysis 82
2.5.1 Structural Operational Semantics 85
2.5.2 Intraprocedural versus Interprocedural Analysis 88
2.5.3 Making Context Explicit 90
2.5.4 Call Strings as Context 95
2.5.5 Assumption Sets as Context 99
2.5.6 Flow-Sensitivity versus Flow-Insensitivity 101
2.6 Shape Analysis 104
2.6.1 Structural Operational Semantics 105
2.6.2 Shape Graphs 109
2.6.3 The Analysis 115
Concluding Remarks 128
Mini Projects 132
Exercises 135

3 Constraint Based Analysis 141
3.1 Abstract 0-CFA Analysis 141
3.1.1 The Analysis 143
3.1.2 Well-definedness of the Analysis 150
3.2 Theoretical Properties 153
3.2.1 Structural Operational Semantics 153
3.2.2 Semantic Correctness 158
3.2.3 Existence of Solutions 162

3.2.4 Coinduction versus Induction 165
3.3 Syntax Directed 0-CFA Analysis 168
3.3.1 Syntax Directed Specification 169
3.3.2 Preservation of Solutions 171
3.4 Constraint Based 0-CFA Analysis 173
3.4.1 Preservation of Solutions 175
3.4.2 Solving the Constraints 176
3.5 Adding Data Flow Analysis 182
3.5.1 Abstract Values as Powersets 182
3.5.2 Abstract Values as Complete Lattices 185
3.6 Adding Context Information 189
3.6.1 Uniform k-CFA Analysis 191
3.6.2 The Cartesian Product Algorithm 196
Concluding Remarks 198
Mini Projects 202
Exercises 205

4 Abstract Interpretation 211
4.1 A Mundane Approach to Correctness 211
4.1.1 Correctness Relations 214
4.1.2 Representation Functions 216
4.1.3 A Modest Generalisation 219
4.2 Approximation of Fixed Points 221
4.2.1 Widening Operators 224
4.2.2 Narrowing Operators 230
4.3 Galois Connections 233
4.3.1 Properties of Galois Connections 239
4.3.2 Galois Insertions 242
4.4 Systematic Design of Galois Connections 246
4.4.1 Component-wise Combinations 249
4.4.2 Other Combinations 253
4.5 Induced Operations 258
4.5.1 Inducing along the Abstraction Function 258

4.5.2 Application to Data Flow Analysis 262
4.5.3 Inducing along the Concretisation Function 267
Concluding Remarks 270
Mini Projects 274
Exercises 276

5 Type and Effect Systems 283
5.1 Control Flow Analysis 283
5.1.1 The Underlying Type System 284
5.1.2 The Analysis 287
5.2 Theoretical Properties 291
5.2.1 Natural Semantics 292
5.2.2 Semantic Correctness 294
5.2.3 Existence of Solutions 297
5.3 Inference Algorithms 300
5.3.1 An Algorithm for the Underlying Type System 300
5.3.2 An Algorithm for Control Flow Analysis 306
5.3.3 Syntactic Soundness and Completeness 312
5.3.4 Existence of Solutions 317
5.4 Effects 319
5.4.1 Side Effect Analysis 319
5.4.2 Exception Analysis 325
5.4.3 Region Inference 330
5.5 Behaviours 339
5.5.1 Communication Analysis 339
Concluding Remarks 349
Mini Projects 353
Exercises 359

6 Algorithms 365
6.1 Worklist Algorithms 365
6.1.1 The Structure of Worklist Algorithms 368
6.1.2 Iterating in LIFO and FIFO 372
6.2 Iterating in Reverse Postorder 374

6.2.1 The Round Robin Algorithm 378
6.3 Iterating Through Strong Components 381
Concluding Remarks 384
Mini Projects 387
Exercises 389

A Partially Ordered Sets 393
A.1 Basic Definitions 393
A.2 Construction of Complete Lattices 397
A.3 Chains 398
A.4 Fixed Points 402
Concluding Remarks 404

B Induction and Coinduction 405
B.1 Proof by Induction 405
B.2 Introducing Coinduction 407
B.3 Proof by Coinduction 411
Concluding Remarks 415

C Graphs and Regular Expressions 417
C.1 Graphs and Forests 417
C.2 Reverse Postorder 421
C.3 Regular Expressions 426
Concluding Remarks 427

Index of Notation 429

Index 433

Bibliography 439

List of Tables

1.1 Reaching Definitions information for the factorial program. . 4
1.2 Reaching Definitions: annotated base types. 19
1.3 Reaching Definitions: annotated type constructors. 21
1.4 Call-tracking Analysis: Effect System. 24
1.5 Chaotic Iteration for Reaching Definitions. 26
1.6 Constant Folding transformation. 27

2.1 Available Expressions Analysis. 40
2.2 Reaching Definitions Analysis. 44
2.3 Very Busy Expressions Analysis. 47
2.4 Live Variables Analysis. 50
2.5 Semantics of expressions in WHILE. 55
2.6 The Structural Operational Semantics of WHILE. 56
2.7 Constant Propagation Analysis. 73
2.8 Algorithm for solving data flow equations. 75
2.9 Iteration steps of the worklist algorithm. 76
2.10 The instrumented semantics of WHILE. 133

3.1 Abstract Control Flow Analysis (Subsections 3.1.1 and 3.1.2). 146
3.2 The Structural Operational Semantics of FUN (part 1). . . . 155
3.3 The Structural Operational Semantics of FUN (part 2). . . . 156
3.4 Abstract Control Flow Analysis for intermediate expressions. 158
3.5 Syntax directed Control Flow Analysis. 170
3.6 Constraint based Control Flow Analysis. 174
3.7 Algorithm for solving constraints. 178

3.8 Abstract values as powersets. 184
3.9 Abstract values as complete lattices. 188
3.10 Uniform k-CFA analysis. 192

5.1 The Underlying Type System. 285
5.2 Control Flow Analysis. 288
5.3 Equality of Annotations. 290
5.4 Natural Semantics for FUN. 292
5.5 Algorithm $\mathcal{W}_{\mathsf{UL}}$ for the underlying type system. 303
5.6 Unification of underlying types. 305
5.7 Unification of simple types. 307
5.8 Algorithm $\mathcal{W}_{\mathsf{CFA}}$ for Control Flow Analysis. 310
5.9 Side Effect Analysis. 322
5.10 Exception Analysis. 328
5.11 Natural Semantics for extended expressions. 334
5.12 Region Inference Analysis and Translation (part 1). 337
5.13 Region Inference Analysis and Translation (part 2). 338
5.14 The sequential semantics. 341
5.15 The concurrent semantics. 343
5.16 Communication Analysis (part 1). 346
5.17 Communication Analysis (part 2). 347
5.18 Ordering on behaviours. 348

6.1 The Abstract Worklist Algorithm. 369
6.2 Iterating in last-in first-out order (LIFO). 373
6.3 Iterating in reverse postorder. 376
6.4 The Round Robin Algorithm. 379
6.5 Pseudocode for constraint numbering. 382
6.6 Iterating through strong components. 384

C.1 The DFSF Algorithm. 421

List of Figures

1.1 The nature of approximation: erring on the safe side. 2
1.2 Flow graph for the factorial program. 6
1.3 The adjunction (α, γ). 15

2.1 Flow graph for the **power** program. 38
2.2 A schematic flow graph. 41
2.3 A schematic flow graph (in reverse). 48
2.4 Preservation of analysis result. 61
2.5 The correctness result for *live*. 62
2.6 Instances for the four classical analyses. 70
2.7 Flow graph for the Fibonacci program. 85
2.8 Analysis of procedure call: the forward case. 93
2.9 Analysis of procedure call: ignoring calling context. 94
2.10 Analysis of procedure call: merging of context. 95
2.11 Procedure call graph for example program. 103
2.12 Reversal of a list of five elements. 106
2.13 Shape graphs corresponding to Figure 2.12. 112
2.14 Sharing information. 113
2.15 The single shape graph in the extremal value ι for the list reversal program. 115
2.16 The effect of $[x{:=}\texttt{nil}]^{\ell}$. 116
2.17 The effect of $[x{:=}y]^{\ell}$ when $x \neq y$. 118
2.18 The effect of $[x{:=}y.sel]^{\ell}$ in Case 2 when $x \neq y$. 120
2.19 The effect of $[x{:=}y.sel]^{\ell}$ in a special case (part 1). 123
2.20 The effect of $[x{:=}y.sel]^{\ell}$ in a special case (part 2). 124

2.21 The effect of $[x.sel{:}{=}\texttt{nil}]^{\ell}$ when $\#into(n_U, \mathsf{H}') \leq 1$. 126
2.22 The effect of $[x.sel{:}{=}y]^{\ell}$ when $\#into(n_Y, \mathsf{H}') < 1$. 127
2.23 *du*-and *ud*-chains. 132

3.1 Pictorial illustration of the clauses [*let*] and [*var*]. 148
3.2 Pictorial illustration of the clauses [*app*], [*fn*] and [*var*]. 149
3.3 Preservation of analysis result. 159
3.4 Initialisation of data structures for example program. 179
3.5 Iteration steps of example program. 180
3.6 Control Flow and Data Flow Analysis for example program. . 185

4.1 Correctness relation R *generated by* representation function β. 218
4.2 The complete lattice **Interval** = (**Interval**, $\sqsubseteq$). 221
4.3 Fixed points of f. 224
4.4 The widening operator ∇ applied to f. 226
4.5 The narrowing operator Δ applied to f. 231
4.6 The Galois connection (L, α, γ, M). 235
4.7 The complete lattice $\mathcal{P}(\mathbf{Sign}) = (\mathcal{P}(\mathbf{Sign}), \subseteq)$. 238
4.8 The Galois insertion (L, α, γ, M). 243
4.9 The reduction operator $\varsigma : M \to M$. 245
4.10 The complete lattice $(\mathbf{Sign}', \sqsubseteq)$. 276

5.1 The memory model for the stack-based implementation of FUN. 331
5.2 The pipeline produced by `pipe [f₁, f₂] inp out`. 340

6.1 Example: LIFO iteration. 374
6.2 (a) Graphical representation. (b) Depth-first spanning tree. . 376
6.3 Example: Reverse postorder iteration. 377
6.4 Example: Round Robin iteration. 379
6.5 (a) Strong components. (b) Reduced graph. 382
6.6 Example: Strong component iteration. 384

A.1 Two complete lattices. 394
A.2 Two partially ordered sets. 400
A.3 Fixed points of f. 403

C.1 A flow graph. 422
C.2 A DFSF for the graph in Figure C.1. 423
C.3 An irreducible graph. 425

Chapter 1

Introduction

In this book we shall introduce four of the main approaches to program analysis: Data Flow Analysis, Constraint Based Analysis, Abstract Interpretation, and Type and Effect Systems. Each of Chapters 2 to 5 deals with one of these approaches at some length and generally treats the more advanced material in later sections. Throughout the book we aim at stressing the many similarities between what may at a first glance appear to be very unrelated approaches. To help to get this idea across, and to serve as a gentle introduction, this chapter treats all of the approaches at the level of examples. The technical details are worked out but it may be difficult to apply the techniques to related examples until some of the material of later chapters has been studied.

1.1 The Nature of Program Analysis

Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer. A main application is to allow compilers to generate code avoiding *redundant* computations, e.g. by reusing available results or by moving loop invariant computations out of loops, or avoiding *superfluous* computations, e.g. of results known to be not needed or of results known already at compile-time. Among the more recent applications is the validation of software (possibly purchased from sub-contractors) to reduce the likelihood of malicious or unintended behaviour. Common for these applications is the need to combine information from different parts of the program.

A main aim of this book is to give an overview of a number of approaches to program analysis, all of which have a quite extensive literature, and to show

Figure 1.1: The nature of approximation: erring on the safe side.

that there is a large amount of *commonality* among the approaches. This should help in cultivating the ability to choose the right approach for the right task and in exploiting insights developed in one approach to enhance the power of other approaches.

One common theme behind all approaches to program analysis is that in order to remain computable one can only provide *approximate answers*. As an example consider a simple language of statements and the program

```
read(x); (if x>0 then y:=1 else (y:=2;S)); z:=y
```

where S is some statement that does not contain an assignment to y. Intuitively, the values of y that can reach `z:=y` will be 1 or 2.

Now suppose an analysis claims that the only value for y that can reach `z:=y` is in fact 1. While this seems intuitively wrong, it is in fact correct in the case where S is known never to terminate for $x \leq 0$ and $y = 2$. But since it is *undecidable* whether or not S terminates, we normally do not expect our analysis to attempt to detect this situation. So in general, we expect the program analysis to produce a possibly *larger set* of possibilities than what will ever happen during execution of the program. This means that we shall also accept a program analysis claiming that the values of y reaching `z:=y` are among 1, 2 or 27, although we will clearly prefer the analysis that gives the more precise answer that the values are among 1 or 2. This notion of safe approximation is illustrated in Figure 1.1. Clearly the challenge is not to

produce the safe "$\{d_1, \cdots, d_N\}$" too often as the analysis will then be utterly useless. Note, that although the analysis does not give precise information it may still give useful information: knowing that the value of `y` is one of 1, 2 and 27 just before the assignment `z:=y` still tells us that `z` will be positive, and that `z` will fit within 1 byte of storage etc. To avoid confusion it may help to be precise in the use of terminology: it is better to say "the values of `y` possible at `z:=y` are among 1 and 2" than the slightly shorter and more frequently used "the values of `y` possible at `z:=y` are 1 and 2".

Another common theme, to be stressed throughout this book, is that all program analyses should be *semantics based*: this means that the information obtained from the analysis can be proved to be safe (or correct) with respect to a semantics of the programming language. It is a sad fact that new program analyses often contain subtle bugs, and a formal justification of the program analysis will help finding these bugs sooner rather than later. However, we should stress that we do *not* suggest that program analyses be *semantics directed*: this would mean that the structure of the program analysis should reflect the structure of the semantics and this will be the case only for a few approaches which are not covered in this book.

1.2 Setting the Scene

Syntax of the WHILE language. We shall consider a simple imperative language called WHILE. A program in WHILE is just a statement which may be, and normally will be, a sequence of statements. In the interest of simplicity, we will associate data flow information with single assignment statements, the tests that appear in conditionals and loops, and `skip` statements. We will require a method to identify these. The most convenient way of doing this is to work with a labelled program – as indicated in the syntax below. We will often refer to the labelled items (assignments, tests and `skip` statements) as *elementary blocks*. In this chapter we will assume that distinct elementary blocks are initially assigned distinct labels; we could drop this requirement, in which case some of the examples would need to be slightly reformulated and the resultant analyses would be less accurate.

We use the following syntactic categories:

a	$\in$	**AExp**	arithmetic expressions
b	$\in$	**BExp**	boolean expressions
S	$\in$	**Stmt**	statements

We assume some countable set of variables is given; numerals and labels will not be further defined and neither will the operators:

x, y	$\in$	**Var**	variables
n	$\in$	**Num**	numerals
ℓ	$\in$	**Lab**	labels

ℓ	$\mathsf{RD}_{entry}(\ell)$	$\mathsf{RD}_{exit}(\ell)$
1	(x, ?), (y, ?), (z, ?)	(x, ?), (y, 1), (z, ?)
2	(x, ?), (y, 1), (z, ?)	(x, ?), (y, 1), (z, 2)
3	(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)	(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)
4	(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)	(x, ?), (y, 1), (y, 5), (z, 4)
5	(x, ?), (y, 1), (y, 5), (z, 4)	(x, ?), (y, 5), (z, 4)
6	(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)	(x, ?), (y, 6), (z, 2), (z, 4)

Table 1.1: Reaching Definitions information for the factorial program.

$$\begin{array}{lcll} op_a & \in & \mathbf{Op}_a & \text{arithmetic operators} \\ op_b & \in & \mathbf{Op}_b & \text{boolean operators} \\ op_r & \in & \mathbf{Op}_r & \text{relational operators} \end{array}$$

The syntax of the language is given by the following *abstract syntax*:

$$\begin{array}{lcl} a & ::= & x \mid n \mid a_1 \; op_a \; a_2 \\ b & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{not}\; b \mid b_1 \; op_b \; b_2 \mid a_1 \; op_r \; a_2 \\ S & ::= & [x := a]^\ell \mid [\texttt{skip}]^\ell \mid S_1; S_2 \mid \\ & & \texttt{if}\; [b]^\ell \;\texttt{then}\; S_1 \;\texttt{else}\; S_2 \mid \texttt{while}\; [b]^\ell \;\texttt{do}\; S \end{array}$$

One way to think of the abstract syntax is as specifying the parse trees of the language; it will then be the purpose of the *concrete syntax* to provide sufficient information to enable unique parse trees to be constructed. In this book we shall *not* be concerned with concrete syntax: whenever we talk about some syntactic entity we will always be talking about the abstract syntax so there will be no ambiguity with respect to the form of the entity. We shall use a textual representation of the abstract syntax and to disambiguate it we shall use parentheses. For statements one often writes **begin** $\cdots$ **end** or $\{\cdots\}$ for this but we shall feel free to use $(\cdots)$. Similarly, we use brackets $(\cdots)$ to resolve ambiguities in other syntactic categories. To cut down on the number of brackets needed we shall use the familiar relative precedences of arithmetic, boolean and relational operators.

Example 1.1 An example of a program written in this language is the following which computes the factorial of the number stored in **x** and leaves the result in **z**:

$$[\texttt{y:=x}]^1;\; [\texttt{z:=1}]^2;\; \texttt{while}\; [\texttt{y>1}]^3\; \texttt{do}\; ([\texttt{z:=z*y}]^4;\; [\texttt{y:=y-1}]^5);\; [\texttt{y:=0}]^6 \qquad \blacksquare$$

Reaching Definitions Analysis. The use of distinct labels allows us to identify the primitive constructs of a program without explicitly constructing a flow graph (or flow chart). It also allows us to introduce a program analysis to be used throughout the chapter: *Reaching Definitions Analysis*, or as it should be called more properly, reaching assignments analysis:

> An assignment (called a definition in the classical literature) of the form $[x := a]^\ell$ *may reach* a certain program point (typically the entry or exit of an elementary block) if there is an execution of the program where x was last assigned a value at ℓ when the program point is reached.

Consider the factorial program of Example 1.1. Here $[\mathtt{y:=x}]^1$ reaches the entry to $[\mathtt{z:=1}]^2$; to allow a more succinct presentation we shall say that (y,1) reaches the entry to 2. Also we shall say that (x,?) reaches the entry to 2; here "?" is a special label not appearing in the program and it is used to record the possibility of an uninitialised variable reaching a certain program point.

Full information about reaching definitions for the factorial program is then given by the pair $\mathsf{RD} = (\mathsf{RD}_{entry}, \mathsf{RD}_{exit})$ of functions in Table 1.1. Careful inspection of this table reveals that the entry and exit information agree for elementary blocks of the form $[b]^\ell$ whereas for elementary blocks of the form $[x := a]^\ell$ they may differ on pairs (x, ℓ'). We shall come back to this when formulating the analysis in subsequent sections.

Returning to the discussion of safe approximation note that if we modify Table 1.1 to include the pair (z,2) in $\mathsf{RD}_{entry}(5)$ and $\mathsf{RD}_{exit}(5)$ we still have safe information about reaching definitions but the information is more approximate. However, if we remove (z,2) from $\mathsf{RD}_{entry}(6)$ and $\mathsf{RD}_{exit}(6)$ then the information will no longer be safe – there exists a run of the factorial program where the set {(x,?),(y,6),(z,4)} does not correctly describe the reaching definitions at the exit of label 6.

1.3 Data Flow Analysis

In *Data Flow Analysis* it is customary to think of a program as a graph: the nodes are the elementary blocks and the edges describe how control might pass from one elementary block to another. Figure 1.2 shows the flow graph for the factorial program of Example 1.1. We shall first illustrate the more common *equational approach* to Data Flow Analysis and then a *constraint based approach* that will serve as a stepping stone to Section 1.4.

1.3.1 The Equational Approach

The equation system. An analysis like Reaching Definitions can be specified by extracting a number of equations from a program. There are two classes of equations. One class of equations relate exit information of a node to entry information for the same node. For the factorial program

$$[\mathtt{y:=x}]^1;\ [\mathtt{z:=1}]^2;\ \mathtt{while}\ [\mathtt{y>1}]^3\ \mathtt{do}\ ([\mathtt{z:=z*y}]^4;\ [\mathtt{y:=y-1}]^5);\ [\mathtt{y:=0}]^6$$

Figure 1.2: Flow graph for the factorial program.

we obtain the following six equations:

$$
\begin{array}{lcl}
\mathsf{RD}_{exit}(1) & = & (\mathsf{RD}_{entry}(1)\backslash\{(\mathbf{y},\ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathbf{y},1)\} \\
\mathsf{RD}_{exit}(2) & = & (\mathsf{RD}_{entry}(2)\backslash\{(\mathbf{z},\ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathbf{z},2)\} \\
\mathsf{RD}_{exit}(3) & = & \mathsf{RD}_{entry}(3) \\
\mathsf{RD}_{exit}(4) & = & (\mathsf{RD}_{entry}(4)\backslash\{(\mathbf{z},\ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathbf{z},4)\} \\
\mathsf{RD}_{exit}(5) & = & (\mathsf{RD}_{entry}(5)\backslash\{(\mathbf{y},\ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathbf{y},5)\} \\
\mathsf{RD}_{exit}(6) & = & (\mathsf{RD}_{entry}(6)\backslash\{(\mathbf{y},\ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathbf{y},6)\}
\end{array}
$$

These are instances of the following schema: for an assignment $[x := a]^{\ell'}$ we exclude all pairs (x, ℓ) from $\mathsf{RD}_{entry}(\ell')$ and add (x, ℓ') in order to obtain $\mathsf{RD}_{exit}(\ell')$ – this reflects that x is redefined at ℓ. For all other elementary blocks $[\cdots]^{\ell'}$ we let $\mathsf{RD}_{exit}(\ell')$ equal $\mathsf{RD}_{entry}(\ell')$ – reflecting that no variables are changed.

The other class of equations relate entry information of a node to exit information of nodes from which there is an edge to the node of interest; that is, entry information is obtained from all the exit information where control could have come from. For the example program we obtain the following equations:

$$\mathsf{RD}_{entry}(2) \quad = \quad \mathsf{RD}_{exit}(1)$$

$$\begin{aligned}
\mathsf{RD}_{entry}(3) &= \mathsf{RD}_{exit}(2) \cup \mathsf{RD}_{exit}(5) \\
\mathsf{RD}_{entry}(4) &= \mathsf{RD}_{exit}(3) \\
\mathsf{RD}_{entry}(5) &= \mathsf{RD}_{exit}(4) \\
\mathsf{RD}_{entry}(6) &= \mathsf{RD}_{exit}(3)
\end{aligned}$$

In general, we write $\mathsf{RD}_{entry}(\ell) = \mathsf{RD}_{exit}(\ell_1) \cup \cdots \cup \mathsf{RD}_{exit}(\ell_n)$ if $\ell_1, \cdots, \ell_n$ are all the labels from which control might pass to ℓ. We shall consider more precise ways of explaining this in Chapter 2. Finally, let us consider the equation

$$\mathsf{RD}_{entry}(1) = \{(x, ?) \mid x \text{ is a variable in the program}\}$$

that makes it clear that the label "?" is to be used for uninitialised variables; so in our case

$$\mathsf{RD}_{entry}(1) = \{(\mathbf{x}, ?), (\mathbf{y}, ?), (\mathbf{z}, ?)\}$$

The least solution. The above system of equations defines the twelve sets

$$\mathsf{RD}_{entry}(1), \cdots, \mathsf{RD}_{exit}(6)$$

in terms of each other. Writing $\overrightarrow{\mathsf{RD}}$ for this twelve-tuple of sets we can regard the equation system as defining a function F and demanding that:

$$\overrightarrow{\mathsf{RD}} = F(\overrightarrow{\mathsf{RD}})$$

To be more specific we can write

$$F(\overrightarrow{\mathsf{RD}}) = (F_{entry}(1)(\overrightarrow{\mathsf{RD}}), F_{exit}(1)(\overrightarrow{\mathsf{RD}}), \cdots, F_{entry}(6)(\overrightarrow{\mathsf{RD}}), F_{exit}(6)(\overrightarrow{\mathsf{RD}}))$$

where e.g.:

$$F_{entry}(3)(\cdots, \mathsf{RD}_{exit}(2), \cdots, \mathsf{RD}_{exit}(5), \cdots) = \mathsf{RD}_{exit}(2) \cup \mathsf{RD}_{exit}(5)$$

It should be clear that F operates over twelve-tuples of sets of pairs of variables and labels; this can be written as

$$F : (\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12} \to (\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12}$$

where it might be natural to take $\mathbf{Var}_\star = \mathbf{Var}$ and $\mathbf{Lab}_\star = \mathbf{Lab}$. However, it will simplify the presentation in this chapter to let $\mathbf{Var}_\star$ be a *finite* subset of $\mathbf{Var}$ that contains the variables occurring in the program $S_\star$ of interest and similarly for $\mathbf{Lab}_\star$. So for the example program we might have $\mathbf{Var}_\star = \{\mathbf{x}, \mathbf{y}, \mathbf{z}\}$ and $\mathbf{Lab}_\star = \{1, \cdots, 6, ?\}$.

It is immediate that $(\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12}$ can be partially ordered by setting

$$\overrightarrow{\mathsf{RD}} \sqsubseteq \overrightarrow{\mathsf{RD}}' \quad \text{iff} \quad \forall i : \mathsf{RD}_i \subseteq \mathsf{RD}'_i$$

where $\overrightarrow{\mathsf{RD}} = (\mathsf{RD}_1, \cdots, \mathsf{RD}_{12})$ and similarly $\overrightarrow{\mathsf{RD}}' = (\mathsf{RD}'_1, \cdots, \mathsf{RD}'_{12})$. This turns $(\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12}$ into a complete lattice (see Appendix A) with least element

$$\vec{\emptyset} = (\emptyset, \cdots, \emptyset)$$

and binary least upper bounds given by:

$$\overrightarrow{\mathsf{RD}} \sqcup \overrightarrow{\mathsf{RD}}' = (\mathsf{RD}_1 \cup \mathsf{RD}'_1, \cdots, \mathsf{RD}_{12} \cup \mathsf{RD}'_{12})$$

It is easy to show that F is in fact a monotone function (see Appendix A) meaning that:

$$\overrightarrow{\mathsf{RD}} \sqsubseteq \overrightarrow{\mathsf{RD}}' \text{ implies } F(\overrightarrow{\mathsf{RD}}) \sqsubseteq F(\overrightarrow{\mathsf{RD}}')$$

This involves calculations like

$$\mathsf{RD}_{exit}(2) \subseteq \mathsf{RD}'_{exit}(2) \text{ and } \mathsf{RD}_{exit}(5) \subseteq \mathsf{RD}'_{exit}(5)$$

imply

$$\mathsf{RD}_{exit}(2) \cup \mathsf{RD}_{exit}(5) \subseteq \mathsf{RD}'_{exit}(2) \cup \mathsf{RD}'_{exit}(5)$$

and the details are left to the reader.

Consider the sequence $(F^n(\vec{\emptyset}))_n$ and note that $\vec{\emptyset} \sqsubseteq F(\vec{\emptyset})$. Since F is monotone, a straightforward mathematical induction (see Appendix B) gives that $F^n(\vec{\emptyset}) \sqsubseteq F^{n+1}(\vec{\emptyset})$ for all n. All the elements of the sequence will be in $(\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12}$ and since this is a finite set it cannot be the case that all elements of the sequence are distinct so there must be some n such that:

$$F^{n+1}(\vec{\emptyset}) = F^n(\vec{\emptyset})$$

But since $F^{n+1}(\vec{\emptyset}) = F(F^n(\vec{\emptyset}))$ this just says that $F^n(\vec{\emptyset})$ is a *fixed point* of F and hence that $F^n(\vec{\emptyset})$ is a solution to the above equation system.

In fact we have obtained the *least solution* to the equation system. To see this suppose that $\overrightarrow{\mathsf{RD}}$ is some other solution, i.e. $\overrightarrow{\mathsf{RD}} = F(\overrightarrow{\mathsf{RD}})$. Then a straightforward mathematical induction shows that $F^n(\vec{\emptyset}) \sqsubseteq \overrightarrow{\mathsf{RD}}$. Hence the solution $F^n(\vec{\emptyset})$ contains the fewest pairs of reaching definitions that is consistent with the program, and intuitively, this is also the solution we want: while we can add additional pairs of reaching definitions without making the analysis semantically unsound, this will make the analysis less usable as discussed in Section 1.1. In Exercise 1.7 we shall see that the least solution is in fact the one displayed in Table 1.1.

1.3.2 The Constraint Based Approach

The constraint system. An alternative to the equational approach above is to use a *constraint based approach.* The idea is here to extract a number of inclusions (or inequations or constraints) out of a program. We

shall present the constraint system for Reaching Definitions in such a way that the relationship to the equational approach becomes apparent; however, it is not a general phenomenon that the constraints are naturally divided into two classes as was the case for the equations.

For the factorial program

$$[\mathtt{y:=x}]^1;\ [\mathtt{z:=1}]^2;\ \mathtt{while}\ [\mathtt{y>1}]^3\ \mathtt{do}\ ([\mathtt{z:=z*y}]^4;\ [\mathtt{y:=y-1}]^5);\ [\mathtt{y:=0}]^6$$

we obtain the following constraints for expressing the effect of elementary blocks:

$$\begin{array}{lcl}
\mathsf{RD}_{exit}(1) & \supseteq & \mathsf{RD}_{entry}(1)\backslash\{(\mathtt{y},\ell) \mid \ell \in \mathbf{Lab}\} \\
\mathsf{RD}_{exit}(1) & \supseteq & \{(\mathtt{y},1)\} \\
\mathsf{RD}_{exit}(2) & \supseteq & \mathsf{RD}_{entry}(2)\backslash\{(\mathtt{z},\ell) \mid \ell \in \mathbf{Lab}\} \\
\mathsf{RD}_{exit}(2) & \supseteq & \{(\mathtt{z},2)\} \\
\mathsf{RD}_{exit}(3) & \supseteq & \mathsf{RD}_{entry}(3) \\
\mathsf{RD}_{exit}(4) & \supseteq & \mathsf{RD}_{entry}(4)\backslash\{(\mathtt{z},\ell) \mid \ell \in \mathbf{Lab}\} \\
\mathsf{RD}_{exit}(4) & \supseteq & \{(\mathtt{z},4)\} \\
\mathsf{RD}_{exit}(5) & \supseteq & \mathsf{RD}_{entry}(5)\backslash\{(\mathtt{y},\ell) \mid \ell \in \mathbf{Lab}\} \\
\mathsf{RD}_{exit}(5) & \supseteq & \{(\mathtt{y},5)\} \\
\mathsf{RD}_{exit}(6) & \supseteq & \mathsf{RD}_{entry}(6)\backslash\{(\mathtt{y},\ell) \mid \ell \in \mathbf{Lab}\} \\
\mathsf{RD}_{exit}(6) & \supseteq & \{(\mathtt{y},6)\}
\end{array}$$

By considering this system a certain methodology emerges: for an assignment $[x := a]^{\ell'}$ we have one constraint that excludes all pairs (x, ℓ) from $\mathsf{RD}_{entry}(\ell')$ in reaching $\mathsf{RD}_{exit}(\ell')$ and we have one constraint for incorporating (x, ℓ'); for all other elementary blocks $[\cdots]^{\ell'}$ we just have one constraint that allows everything in $\mathsf{RD}_{entry}(\ell')$ to reach $\mathsf{RD}_{exit}(\ell')$.

Next consider the constraints for more directly expressing how control may flow through the program. For the example program we obtain the constraints:

$$\begin{array}{lcl}
\mathsf{RD}_{entry}(2) & \supseteq & \mathsf{RD}_{exit}(1) \\
\mathsf{RD}_{entry}(3) & \supseteq & \mathsf{RD}_{exit}(2) \\
\mathsf{RD}_{entry}(3) & \supseteq & \mathsf{RD}_{exit}(5) \\
\mathsf{RD}_{entry}(5) & \supseteq & \mathsf{RD}_{exit}(4) \\
\mathsf{RD}_{entry}(6) & \supseteq & \mathsf{RD}_{exit}(3)
\end{array}$$

In general, we have a constraint $\mathsf{RD}_{entry}(\ell) \supseteq \mathsf{RD}_{exit}(\ell')$ if it is possible for control to pass from ℓ' to ℓ. Finally, the constraint

$$\mathsf{RD}_{entry}(1) \supseteq \{(\mathtt{x},?), (\mathtt{y},?), (\mathtt{z},?)\}$$

records that we cannot be sure about the definition point of uninitialised variables.

The least solution revisited. It is not hard to see that a solution to the equation system presented previously will also be a solution to the above constraint system. To make this connection more transparent we can rearrange the constraints by *collecting* all constraints with the same left hand side. This means that for example

$$\begin{array}{lcl} \mathsf{RD}_{exit}(1) & \supseteq & \mathsf{RD}_{entry}(1) \backslash \{(\mathsf{y}, \ell) \mid \ell \in \mathbf{Lab}\} \\ \mathsf{RD}_{exit}(1) & \supseteq & \{(\mathsf{y}, 1)\} \end{array}$$

will be replaced by

$$\mathsf{RD}_{exit}(1) \supseteq (\mathsf{RD}_{entry}(1) \backslash \{(\mathsf{y}, \ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathsf{y}, 1)\}$$

and clearly this has no consequence for whether or not $\overrightarrow{\mathsf{RD}}$ is a solution. In other words we obtain a version of the previous equation system except that all equalities have been replaced by inclusions. Formally, whereas the equational approach demands that $\overrightarrow{\mathsf{RD}} = F(\overrightarrow{\mathsf{RD}})$, the constraint based approach demands that $\overrightarrow{\mathsf{RD}} \sqsupseteq F(\overrightarrow{\mathsf{RD}})$ for the *same* function F. It is therefore immediate that a solution to the equation system is also a solution to the constraint system whereas the converse is not necessarily the case.

Luckily we can show that both the equation system and the constraint system have the same *least solution.* Recall that the least solution to $\overrightarrow{\mathsf{RD}} = F(\overrightarrow{\mathsf{RD}})$ is constructed as $F^n(\vec{\emptyset})$ for a value of n such that $F^n(\vec{\emptyset}) = F^{n+1}(\vec{\emptyset})$. If $\overrightarrow{\mathsf{RD}}$ is a solution to the constraint system, that is $\overrightarrow{\mathsf{RD}} \sqsupseteq F(\overrightarrow{\mathsf{RD}})$, then $\vec{\emptyset} \sqsubseteq \overrightarrow{\mathsf{RD}}$ is immediate and the monotonicity of F and mathematical induction then gives $F^n(\vec{\emptyset}) \sqsubseteq \overrightarrow{\mathsf{RD}}$. Since $F^n(\vec{\emptyset})$ is a solution to the constraint system this shows that it is also the least solution to the constraint system.

In summary, we have thus seen a very strong connection between the equational approach and the constraint based approach. This connection is not always as apparent as it is here: one of the characteristics of the constraint based approach is that often constraints with the same left hand side are generated at many different places in the program and therefore it may require serious work to collect them.

1.4 Constraint Based Analysis

The purpose of *Control Flow Analysis* is to determine information about what "elementary blocks" may lead to what other "elementary blocks". This information is immediately available for the WHILE language unlike what is the case for more advanced imperative, functional and object-oriented languages. Often Control Flow Analysis is expressed as a Constraint Based Analysis as will be illustrated in this section.

Consider the following functional program:

```
let   f = fn x => x 1;
      g = fn y => y+2;
      h = fn z => z+3
in    (f g) + (f h)
```

It defines a higher-order function `f` with formal parameter `x` and body `x 1`; then it defines two functions `g` and `h` that are given as actual parameters to `f` in the body of the `let`-construct. Semantically, `x` will be bound to each of these two functions in turn so both `g` and `h` will be applied to `1` and the result of the computation will be the value 7.

An application of `f` will transfer control to the body of `f`, i.e. to `x 1`, and this application of `x` will transfer control to the body of `x`. The problem is that we cannot immediately point to the body of `x`: we need to know what parameters `f` will be called with. This is exactly the information that the Control Flow Analysis gives us:

For each function application, which functions may be applied.

As is typical of functional languages, the labelling scheme used would seem to have a very different character than the one employed for imperative languages because the "elementary blocks" may be nested. We shall therefore label *all* subexpressions as in the following simple program that will be used to illustrate the analysis.

Example 1.2 Consider the program:

$$[[\texttt{fn x => } [\texttt{x}]^1]^2\ [\texttt{fn y => } [\texttt{y}]^3]^4]^5$$

It calls the identity function `fn x => x` on the argument `fn y => y` and clearly evaluates to `fn y => y` itself (omitting all $[\cdots]^\ell$). ∎

We shall now be interested in associating information with the labels themselves, rather than with the entries and exits of the labels – thereby we exploit the fact that there are no side-effects in our simple functional language. The Control Flow Analysis will be specified by a pair $(\widehat{\mathsf{C}}, \widehat{\rho})$ of functions where $\widehat{\mathsf{C}}(\ell)$ is supposed to contain the values that the subexpression (or "elementary block") labelled ℓ may evaluate to and $\widehat{\rho}(x)$ contain the values that the variable x can be bound to.

The constraint system. One way to specify the Control Flow Analysis then is by means of a collection of constraints and we shall illustrate this for the program of Example 1.2. There are three classes of constraints. One class of constraints relate the values of function abstractions to their labels:

$$\{\texttt{fn x => } [\texttt{x}]^1\} \subseteq \widehat{\mathsf{C}}(2)$$
$$\{\texttt{fn y => } [\texttt{y}]^3\} \subseteq \widehat{\mathsf{C}}(4)$$

These constraints state that a function abstraction evaluates to a closure containing the abstraction itself. So the general pattern is that an occurrence of $[\texttt{fn}\ x\ \texttt{=>}\ e]^\ell$ in the program gives rise to a constraint $\{\texttt{fn}\ x\ \texttt{=>}\ e\} \subseteq \widehat{\mathsf{C}}(\ell)$.

The second class of constraints relate the values of variables to their labels:

$$\widehat{\rho}(\mathsf{x}) \subseteq \widehat{\mathsf{C}}(1)$$
$$\widehat{\rho}(\mathsf{y}) \subseteq \widehat{\mathsf{C}}(3)$$

The constraints state that a variable always evaluates to its value. So for each occurrence of $[x]^\ell$ in the program we will have a constraint $\widehat{\rho}(x) \subseteq \widehat{\mathsf{C}}(\ell)$.

The third class of constraints concerns function application: for each application point $[e_1\ e_2]^\ell$, and for each possible function $[\texttt{fn}\ x\ \texttt{=>}\ e]^{\ell'}$ that could be called at this point, we will have: (i) a constraint expressing that the formal parameter of the function is bound to the actual parameter at the application point, and (ii) a constraint expressing that the result obtained by evaluating the body of the function is a possible result of the application.

Our example program has just one application $[[\cdots]^2\ [\cdots]^4]^5$, but there are two candidates for the function, i.e. $\widehat{\mathsf{C}}(2)$ is a subset of the set $\{\texttt{fn x => } [\mathsf{x}]^1, \texttt{fn y => } [\mathsf{y}]^3\}$. If the function $\texttt{fn x => } [\mathsf{x}]^1$ is applied then the two constraints are $\widehat{\mathsf{C}}(4) \subseteq \widehat{\rho}(\mathsf{x})$ and $\widehat{\mathsf{C}}(1) \subseteq \widehat{\mathsf{C}}(5)$. We express this as *conditional constraints*:

$$\{\texttt{fn x => } [\mathsf{x}]^1\} \subseteq \widehat{\mathsf{C}}(2) \Rightarrow \widehat{\mathsf{C}}(4) \subseteq \widehat{\rho}(\mathsf{x})$$
$$\{\texttt{fn x => } [\mathsf{x}]^1\} \subseteq \widehat{\mathsf{C}}(2) \Rightarrow \widehat{\mathsf{C}}(1) \subseteq \widehat{\mathsf{C}}(5)$$

Alternatively, the function being applied could be $\texttt{fn y => } [\mathsf{y}]^3$ and the corresponding conditional constraints are:

$$\{\texttt{fn y => } [\mathsf{y}]^3\} \subseteq \widehat{\mathsf{C}}(2) \Rightarrow \widehat{\mathsf{C}}(4) \subseteq \widehat{\rho}(\mathsf{y})$$
$$\{\texttt{fn y => } [\mathsf{y}]^3\} \subseteq \widehat{\mathsf{C}}(2) \Rightarrow \widehat{\mathsf{C}}(3) \subseteq \widehat{\mathsf{C}}(5)$$

The least solution. As in Section 1.3 we shall be interested in the least solution to this set of constraints: the smaller the sets of values given by $\widehat{\mathsf{C}}$ and $\widehat{\rho}$, the more precise the analysis is in predicting which functions are applied. In Exercise 1.2 we show that the following choice of $\widehat{\mathsf{C}}$ and $\widehat{\rho}$ gives a solution to the above constraints:

$$\begin{array}{rcl}
\widehat{\mathsf{C}}(1) & = & \{\texttt{fn y => } [\mathsf{y}]^3\} \\
\widehat{\mathsf{C}}(2) & = & \{\texttt{fn x => } [\mathsf{x}]^1\} \\
\widehat{\mathsf{C}}(3) & = & \emptyset \\
\widehat{\mathsf{C}}(4) & = & \{\texttt{fn y => } [\mathsf{y}]^3\} \\
\widehat{\mathsf{C}}(5) & = & \{\texttt{fn y => } [\mathsf{y}]^3\} \\
\widehat{\rho}(\mathsf{x}) & = & \{\texttt{fn y => } [\mathsf{y}]^3\} \\
\widehat{\rho}(\mathsf{y}) & = & \emptyset
\end{array}$$

Among other things this tells us that the function abstraction `fn y => y` is never applied (since $\widehat{\rho}(\mathtt{y}) = \emptyset$) and that the program may only evaluate to the function abstraction `fn y => y` (since $\widehat{\mathsf{C}}(5) = \{\mathtt{fn\ y\ =>\ [y]}^3\}$).

Note the similarities between the constraint based approaches to Data Flow Analysis and Constraint Based Analysis: in both cases the syntactic structure of the program gives rise to a set of constraints whose least solution is desired. The main difference is that the constraints for the Constraint Based Analysis have a more complex structure than those for the Data Flow Analysis.

1.5 Abstract Interpretation

The theory of *Abstract Interpretation* is a general methodology for calculating analyses rather than just specifying them and then relying on a posteriori validation. To some extent the application of Abstract Interpretation is independent of the specification style used for presenting the program analysis and so applies not only to the Data Flow Analysis formulation to be used here.

Collecting semantics. As a preliminary step we shall formulate a so-called *collecting semantics* that records the set of *traces tr* that can reach a given program point:

$$tr \in \mathbf{Trace} = (\mathbf{Var} \times \mathbf{Lab})^*$$

Intuitively, a trace will record where the variables have obtained their values in the course of the computation. So for the factorial program

$$[\mathtt{y:=x}]^1;\ [\mathtt{z:=1}]^2;\ \mathtt{while}\ [\mathtt{y>1}]^3\ \mathtt{do}\ ([\mathtt{z:=z*y}]^4;\ [\mathtt{y:=y-1}]^5);\ [\mathtt{y:=0}]^6$$

we will for example have the trace

$$((\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?),(\mathtt{y},1),(\mathtt{z},2),(\mathtt{z},4),(\mathtt{y},5),(\mathtt{z},4),(\mathtt{y},5),(\mathtt{y},6))$$

corresponding to a run of the program where the body of the `while`-loop is executed twice.

The traces contain sufficient information that we can extract a set of *semantically reaching definitions*:

$$\mathsf{SRD}(tr)(x) = \ell \quad \text{iff} \quad \text{the rightmost pair } (x, \ell') \text{ in } tr \text{ has } \ell = \ell'$$

We shall write $\mathsf{DOM}(tr)$ for the set of variables for which $\mathsf{SRD}(tr)$ is defined, i.e. $x \in \mathsf{DOM}(tr)$ iff some pair (x, ℓ) occurs in tr.

In order for the Reaching Definitions Analysis to be correct (or safe) we shall require that it captures the semantic reaching definitions, that is, if tr is a

possible trace just before entering the elementary block labelled ℓ then we shall demand that

$$\forall x \in \mathsf{DOM}(tr) : (x, \mathsf{SRD}(tr)(x)) \in \mathsf{RD}_{entry}(\ell)$$

in order to trust the information in $\mathsf{RD}_{entry}(\ell)$ about the set of definitions that may reach the entry to ℓ. In later chapters, we will conduct proofs of results like this.

The collecting semantics will specify a *superset* of the possible traces at the various program points. We shall specify the collecting semantics CS in the style of the Reaching Definitions Analysis in Section 1.3; more precisely, we shall specify a twelve-tuple of elements from $(\mathcal{P}(\mathbf{Trace}))^{12}$ by means of a set of equations. First we have

$$\begin{array}{rcl}
\mathsf{CS}_{exit}(1) & = & \{tr : (\mathbf{y}, 1) \mid tr \in \mathsf{CS}_{entry}(1)\} \\
\mathsf{CS}_{exit}(2) & = & \{tr : (\mathbf{z}, 2) \mid tr \in \mathsf{CS}_{entry}(2)\} \\
\mathsf{CS}_{exit}(3) & = & \mathsf{CS}_{entry}(3) \\
\mathsf{CS}_{exit}(4) & = & \{tr : (\mathbf{z}, 4) \mid tr \in \mathsf{CS}_{entry}(4)\} \\
\mathsf{CS}_{exit}(5) & = & \{tr : (\mathbf{y}, 5) \mid tr \in \mathsf{CS}_{entry}(5)\} \\
\mathsf{CS}_{exit}(6) & = & \{tr : (\mathbf{y}, 6) \mid tr \in \mathsf{CS}_{entry}(6)\}
\end{array}$$

showing how the assignment statements give rise to extensions of the traces. Here we write $tr : (x, \ell)$ for appending an element (x, ℓ) to a list tr, that is $((x_1, \ell_1), \cdots, (x_n, \ell_n)) : (x, \ell)$ equals $((x_1, \ell_1), \cdots, (x_n, \ell_n), (x, \ell))$. Furthermore, we have

$$\begin{array}{rcl}
\mathsf{CS}_{entry}(2) & = & \mathsf{CS}_{exit}(1) \\
\mathsf{CS}_{entry}(3) & = & \mathsf{CS}_{exit}(2) \cup \mathsf{CS}_{exit}(5) \\
\mathsf{CS}_{entry}(4) & = & \mathsf{CS}_{exit}(3) \\
\mathsf{CS}_{entry}(5) & = & \mathsf{CS}_{exit}(4) \\
\mathsf{CS}_{entry}(6) & = & \mathsf{CS}_{exit}(3)
\end{array}$$

corresponding to the flow of control in the program; more detailed information about the values of the variables would allow us to define the sets $\mathsf{CS}_{entry}(4)$ and $\mathsf{CS}_{entry}(6)$ more precisely but the above definitions are sufficient for illustrating the approach. Finally, we take

$$\mathsf{CS}_{entry}(1) \;=\; \{((\mathbf{x}, ?), (\mathbf{y}, ?), (\mathbf{z}, ?))\}$$

corresponding to the fact that all variables are uninitialised in the beginning.

In the manner of the previous sections we can rewrite the above system of equations in the form

$$\overrightarrow{\mathsf{CS}} \;=\; G(\overrightarrow{\mathsf{CS}})$$

Figure 1.3: The adjunction (α, γ).

where $\overrightarrow{\mathsf{CS}}$ is a twelve-tuple of elements from $(\mathcal{P}(\mathbf{Trace}))^{12}$ and where G is a monotone function of functionality:

$$G : (\mathcal{P}(\mathbf{Trace}))^{12} \rightarrow (\mathcal{P}(\mathbf{Trace}))^{12}$$

As is explained in Appendix A there is a body of general theory that ensures that the equation system in fact has a least solution; we shall write it as $\mathit{lfp}(G)$. However, since $(\mathcal{P}(\mathbf{Trace}))^{12}$ is not finite we cannot simply use the methods of the previous sections in order to construct $\mathit{lfp}(G)$.

Galois connections. As we have seen the collecting semantics operates on sets of traces whereas the Reaching Definitions Analysis operates on sets of pairs of variables and labels. To relate these "worlds" we define an abstraction function α and a concretisation function γ as illustrated in:

$$\mathcal{P}(\mathbf{Trace}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}(\mathbf{Var} \times \mathbf{Lab})$$

The idea is that the *abstraction function* α extracts the reachability information present in a set of traces; it is natural to define

$$\alpha(X) = \{(x, \mathsf{SRD}(tr)(x)) \mid x \in \mathsf{DOM}(tr) \ \wedge tr \ \in X\}$$

where we exploit the notion of semantically reaching definitions.

The *concretisation function* γ then produces all traces tr that are consistent with the given reachability information:

$$\gamma(Y) = \{tr \mid \forall x \in \mathsf{DOM}(tr) : (x, \mathsf{SRD}(tr)(x)) \in Y\}$$

Often it is demanded that α and γ satisfy the condition

$$\alpha(X) \subseteq Y \Leftrightarrow X \subseteq \gamma(Y)$$

and we shall say that (α, γ) is an *adjunction*, or a *Galois connection*, whenever this condition is satisfied; this is illustrated in Figure 1.3. We shall leave it to the reader to verify that (α, γ) as defined above does in fact fulfil this condition.

Induced analysis. We shall now show how the collecting semantics can be used to *calculate* (as opposed to "guess") an analysis like the one in Section 1.3; we shall say that the analysis is an *induced analysis*. For this we define

$$\begin{array}{rcl} \vec{\alpha}(X_1, \cdots, X_{12}) & = & (\alpha(X_1), \cdots, \alpha(X_{12})) \\ \vec{\gamma}(Y_1, \cdots, Y_{12}) & = & (\gamma(Y_1), \cdots, \gamma(Y_{12})) \end{array}$$

where α and γ are as above and we consider the function $\vec{\alpha} \circ G \circ \vec{\gamma}$ of functionality:

$$(\vec{\alpha} \circ G \circ \vec{\gamma}) : (\mathcal{P}(\mathbf{Var} \times \mathbf{Lab}))^{12} \to (\mathcal{P}(\mathbf{Var} \times \mathbf{Lab}))^{12}$$

This function defines a Reaching Definitions analysis in an indirect way. Since G is specified by a set of equations (over $\mathcal{P}(\mathbf{Trace})$) we can use $\vec{\alpha} \circ G \circ \vec{\gamma}$ to calculate a new set of equations (over $\mathcal{P}(\mathbf{Var} \times \mathbf{Lab})$). We shall illustrate this for one of the equations:

$$\mathsf{CS}_{exit}(4) = \{tr : (\mathbf{z}, 4) \mid tr \in \mathsf{CS}_{entry}(4)\}$$

The corresponding clause in the definition of G is:

$$G_{exit}(4)(\cdots, \mathsf{CS}_{entry}(4), \cdots) = \{tr : (\mathbf{z}, 4) \mid tr \in \mathsf{CS}_{entry}(4)\}$$

We can now calculate the corresponding clause in the definition of $\vec{\alpha} \circ G \circ \vec{\gamma}$:

$$\begin{array}{l} \alpha(G_{exit}(4)(\vec{\gamma}(\cdots, \mathsf{RD}_{entry}(4), \cdots))) \\ \quad = \alpha(\{tr : (\mathbf{z}, 4) \mid tr \in \gamma(\mathsf{RD}_{entry}(4))\}) \\ \quad = \{(x, \mathsf{SRD}(tr : (\mathbf{z}, 4))(x)) \\ \qquad \mid x \in \mathsf{DOM}(tr : (\mathbf{z}, 4)), \\ \qquad\quad \forall y \in \mathsf{DOM}(tr) : (y, \mathsf{SRD}(tr)(y)) \in \mathsf{RD}_{entry}(4)\} \\ \quad = \{(x, \mathsf{SRD}(tr : (\mathbf{z}, 4))(x)) \\ \qquad \mid x \neq \mathbf{z},\ x \in \mathsf{DOM}(tr : (\mathbf{z}, 4)), \\ \qquad\quad \forall y \in \mathsf{DOM}(tr) : (y, \mathsf{SRD}(tr)(y)) \in \mathsf{RD}_{entry}(4)\} \\ \qquad \cup \{(x, \mathsf{SRD}(tr : (\mathbf{z}, 4))(x)) \\ \qquad \mid x = \mathbf{z},\ x \in \mathsf{DOM}(tr : (\mathbf{z}, 4)), \\ \qquad\quad \forall y \in \mathsf{DOM}(tr) : (y, \mathsf{SRD}(tr)(y)) \in \mathsf{RD}_{entry}(4)\} \\ \quad = \{(x, \mathsf{SRD}(tr)(x)) \\ \qquad \mid x \neq \mathbf{z},\ x \in \mathsf{DOM}(tr), \\ \qquad\quad \forall y \in \mathsf{DOM}(tr) : (y, \mathsf{SRD}(tr)(y)) \in \mathsf{RD}_{entry}(4)\} \\ \qquad \cup \{(\mathbf{z}, 4) \\ \qquad \mid \forall y \in \mathsf{DOM}(tr) : (y, \mathsf{SRD}(tr)(y)) \in \mathsf{RD}_{entry}(4)\} \\ \quad = (\mathsf{RD}_{entry}(4) \setminus \{(\mathbf{z}, \ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathbf{z}, 4)\} \end{array}$$

The resulting equation

$$\mathsf{RD}_{exit}(4) = (\mathsf{RD}_{entry}(4) \setminus \{(\mathbf{z}, \ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathbf{z}, 4)\}$$

is as in Section 1.3. Similar calculations can be performed for the other equations.

The least solution. As explained in Appendix A the equation system

$$\overrightarrow{\mathsf{RD}} \quad = \quad (\vec{\alpha} \circ G \circ \vec{\gamma})(\overrightarrow{\mathsf{RD}})$$

has a least solution; we shall write it as $\mathit{lfp}(\vec{\alpha} \circ G \circ \vec{\gamma})$. It is interesting to note that if one replaces the infinite sets **Var** and **Lab** with finite sets $\mathbf{Var}_\star$ and $\mathbf{Lab}_\star$ as before, then the least fixed point of $\vec{\alpha} \circ G \circ \vec{\gamma}$ can be obtained as $(\vec{\alpha} \circ G \circ \vec{\gamma})^n(\vec{\emptyset})$ just as was the case for F previously.

In Exercise 1.4 we shall show that $\vec{\alpha} \circ G \circ \vec{\gamma} \sqsubseteq F$ and that $\vec{\alpha}(G^n(\vec{\emptyset})) \sqsubseteq (\vec{\alpha} \circ G \circ \vec{\gamma})^n(\vec{\emptyset}) \sqsubseteq F^n(\vec{\emptyset})$ holds for all n. In fact it will be the case that

$$\vec{\alpha}(\mathit{lfp}(G)) \sqsubseteq \mathit{lfp}(\vec{\alpha} \circ G \circ \vec{\gamma}) \sqsubseteq \mathit{lfp}(F)$$

and this just says that the least solution to the equation system defined by $\vec{\alpha} \circ G \circ \vec{\gamma}$ is correct with respect to the collecting semantics, and similarly that the least solution to the equation system of Section 1.3 is also correct with respect to the collecting semantics. Thus it follows that we will only need to show that the collecting semantics is correct – the correctness of the induced analysis will follow for free.

For some analyses one is able to prove the stronger result $\vec{\alpha} \circ G \circ \vec{\gamma} = F$. Then the analysis is *optimal* (given the choice of approximate properties it operates on) and clearly $\mathit{lfp}(\vec{\alpha} \circ G \circ \vec{\gamma}) = \mathit{lfp}(F)$. In Exercise 1.4 we shall study whether or not this is the case here.

1.6 Type and Effect Systems

A simple type system. The ideal setting for explaining *Type and Effect Systems* is to consider a typed functional or imperative language. However, even our simple toy language can be considered to be typed: a statement S maps a state to a state (in case it terminates) and may therefore be considered to have type $\Sigma \to \Sigma$ where Σ denotes the type of states; we write this as the judgement:

$$S : \Sigma \to \Sigma$$

One way to formalise this is by the following utterly trivial system of axioms and inference rules:

$$[x := a]^\ell \ : \ \Sigma \to \Sigma$$

$$[\texttt{skip}]^\ell \ : \ \Sigma \to \Sigma$$

$$\frac{S_1 : \Sigma \to \Sigma \quad S_2 : \Sigma \to \Sigma}{S_1; S_2 \ : \ \Sigma \to \Sigma}$$

$$\frac{S_1 : \Sigma \to \Sigma \quad S_2 : \Sigma \to \Sigma}{\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \ : \ \Sigma \to \Sigma}$$

$$\frac{S : \Sigma \to \Sigma}{\texttt{while } [b]^\ell \texttt{ do } S \ : \ \Sigma \to \Sigma}$$

Often a Type and Effect System can be viewed as the amalgamation of two ingredients: an *Effect System* and an *Annotated Type System*. In an Effect System we typically have judgements of the form $S : \Sigma \xrightarrow{\varphi} \Sigma$ where the *effect* φ tells something about what happens when S is executed: for example this may be which errors might occur, which exceptions might be raised, or which files might be modified. In an Annotated Type System we typically have judgements of the form $S : \Sigma_1 \to \Sigma_2$ where the Σ_i describe certain *properties of states*: for example this may be that a variable is positive or that a certain invariant is maintained. We shall first illustrate the latter approach for the WHILE language and then illustrate the Effect Systems using the functional language.

1.6.1 Annotated Type Systems

Annotated base types. To obtain our first specification of Reaching Definitions we shall focus on a formulation where the *base types* are annotated. Here we will have judgements of the form

$$S : \mathsf{RD}_1 \to \mathsf{RD}_2$$

where $\mathsf{RD}_1, \mathsf{RD}_2 \in \mathcal{P}(\mathbf{Var} \times \mathbf{Lab})$ are sets of reaching definitions. Based on the trivial axioms and rules displayed above we then obtain the more interesting ones in Table 1.2.

To explain these rules let us first explain the meaning of $S : \mathsf{RD}_1 \to \mathsf{RD}_2$ in terms of the developments performed in Section 1.3. For this we first observe that any statement S will have one elementary block at its entry, denoted *init*(S), and one or more elementary blocks at its exit, denoted *final*(S); for a statement like `if` $[\texttt{x<y}]^1$ `then` $[\texttt{x:=y}]^2$ `else` $[\texttt{y:=x}]^3$ we thus get *init*$(\cdots) = 1$ and *final*$(\cdots) = \{2, 3\}$.

Our first (and not quite successful) attempt at explaining the meaning of

$$[ass] \quad [x := a]^{\ell'} : \mathsf{RD} \to ((\mathsf{RD} \backslash \{(x,\ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(x,\ell')\})$$

$$[skip] \quad [\mathtt{skip}]^{\ell'} : \mathsf{RD} \to \mathsf{RD}$$

$$[seq] \quad \frac{S_1 : \mathsf{RD}_1 \to \mathsf{RD}_2 \qquad S_2 : \mathsf{RD}_2 \to \mathsf{RD}_3}{S_1; S_2 : \mathsf{RD}_1 \to \mathsf{RD}_3}$$

$$[if] \quad \frac{S_1 : \mathsf{RD}_1 \to \mathsf{RD}_2 \qquad S_2 : \mathsf{RD}_1 \to \mathsf{RD}_2}{\mathtt{if}\ [b]^{\ell}\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2 : \mathsf{RD}_1 \to \mathsf{RD}_2}$$

$$[wh] \quad \frac{S : \mathsf{RD} \to \mathsf{RD}}{\mathtt{while}\ [b]^{\ell}\ \mathtt{do}\ S\ :\ \mathsf{RD} \to \mathsf{RD}}$$

$$[sub] \quad \frac{S : \mathsf{RD}_2 \to \mathsf{RD}_3}{S : \mathsf{RD}_1 \to \mathsf{RD}_4} \text{ if } \mathsf{RD}_1 \subseteq \mathsf{RD}_2 \text{ and } \mathsf{RD}_3 \subseteq \mathsf{RD}_4$$

Table 1.2: Reaching Definitions: annotated base types.

$S : \mathsf{RD}_1 \to \mathsf{RD}_2$ then is to say that:

$$\mathsf{RD}_1 = \mathsf{RD}_{entry}(init(S))$$
$$\bigcup\{\mathsf{RD}_{exit}(\ell) \mid \ell \in final(S)\} = \mathsf{RD}_2$$

This suffices for explaining the axioms for assignment and **skip**: here the formulae after the arrows correspond exactly to the similar equations in the equational formulation of the analysis in Section 1.3. Also the rule for sequencing now seems rather natural. However, the rule for conditional is more dubious: considering the statement `if` $[\mathtt{x<y}]^1$ `then` $[\mathtt{x:=y}]^2$ `else` $[\mathtt{y:=x}]^3$ once more, it seems impossible to achieve that the **then**-branch gives rise to the same set of reaching definitions as the **else**-branch does.

Our second (and successful) attempt at explaining the intended meaning of $S : \mathsf{RD}_1 \to \mathsf{RD}_2$ then is to say that:

$$\mathsf{RD}_1 \subseteq \mathsf{RD}_{entry}(init(S))$$
$$\forall \ell \in final(S) : \mathsf{RD}_{exit}(\ell) \subseteq \mathsf{RD}_2$$

This formulation is somewhat closer to the development in the constraint based formulation of the analysis in Section 1.3 and it explains why the last rule, called a *subsumption rule*, is unproblematic. Actually, the subsumption rule will solve our problem with the conditional because even when the **then**-branch gives a different set of reaching definitions than the **else**-branch we can enlarge both results to a common set of reaching definitions. Finally, consider the rule for the iterative construct. Here we simply express that RD is a consistent guess concerning what may reach the entry and exits of S – this expresses a fixed point property.

Example 1.3 To analyse the factorial program

$$[\mathtt{y:=x}]^1;\ [\mathtt{z:=1}]^2;\ \mathtt{while}\ [\mathtt{y>1}]^3\ \mathtt{do}\ ([\mathtt{z:=z*y}]^4;\ [\mathtt{y:=y-1}]^5);\ [\mathtt{y:=0}]^6$$

of Example 1.1 we will proceed as follows. We shall write $\mathsf{RD_f}$ for the set $\{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{z},2),(\mathtt{z},4)\}$ and consider the body of the `while`-loop. The axiom [*ass*] gives

$$[\mathtt{z:=z*y}]^4\colon \mathsf{RD_f} \to \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{z},4)\}$$
$$[\mathtt{y:=y-1}]^5 : \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{z},4)\} \to \{(\mathtt{x},?),(\mathtt{y},5),(\mathtt{z},4)\}$$

so the rule [*seq*] gives:

$$([\mathtt{z:=z*y}]^4;\ [\mathtt{y:=y-1}]^5)\colon \mathsf{RD_f} \to \{(\mathtt{x},?),(\mathtt{y},5),(\mathtt{z},4)\}$$

Now $\{(\mathtt{x},?),(\mathtt{y},5),(\mathtt{z},4)\} \subseteq \mathsf{RD_f}$ so the subsumption rule gives:

$$([\mathtt{z:=z*y}]^4;\ [\mathtt{y:=y-1}]^5)\colon \mathsf{RD_f} \to \mathsf{RD_f}$$

We can now apply the rule [*wh*] and get:

$$\mathtt{while}\ [\mathtt{y>1}]^3\ \mathtt{do}\ ([\mathtt{z:=z*y}]^4;\ [\mathtt{y:=y-1}]^5)\colon \mathsf{RD_f} \to \mathsf{RD_f}$$

Using the axiom [*ass*] we get:

$$[\mathtt{y:=x}]^1\colon \{(\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?)\} \to \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},?)\}$$
$$[\mathtt{z:=1}]^2 : \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},?)\} \to \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},2)\}$$
$$[\mathtt{y:=0}]^6\colon \mathsf{RD_f} \to \{(\mathtt{x},?),(\mathtt{y},6),(\mathtt{z},2),(\mathtt{z},4)\}$$

Since $\{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},2)\} \subseteq \mathsf{RD_f}$ we can apply the rules [*seq*] and [*sub*] to get

$$([\mathtt{y:=x}]^1;\ [\mathtt{z:=1}]^2;\ \mathtt{while}\ [\mathtt{y>1}]^3\ \mathtt{do}\ ([\mathtt{z:=z*y}]^4;\ [\mathtt{y:=y-1}]^5);\ [\mathtt{y:=0}]^6)\colon$$
$$\{(\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?)\} \to \{(\mathtt{x},?),(\mathtt{y},6),(\mathtt{z},2),(\mathtt{z},4)\}$$

corresponding to the result in Table 1.1. ∎

The system in Table 1.2 suffices for manually analysing a given program. To obtain an implementation it will be natural to extract a set of constraints similar to those considered in Section 1.3, and then solve them in the same way as before. This will be the idea behind the approach taken in Chapter 5.

Annotated type constructors. Another approach to Reaching Definitions has a little bit of the flavour of Effect Systems in that it is the type constructors (arrow in our case) that are annotated. Here we will have judgements of the form

$$S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma$$

$$[ass] \quad [x := a]^\ell : \Sigma \xrightarrow[\{(x,\ell)\}]{\{x\}} \Sigma$$

$$[skip] \quad [\texttt{skip}]^\ell : \Sigma \xrightarrow[\emptyset]{\emptyset} \Sigma$$

$$[seq] \quad \frac{S_1 : \Sigma \xrightarrow[\mathsf{RD}_1]{X_1} \Sigma \qquad S_2 : \Sigma \xrightarrow[\mathsf{RD}_2]{X_2} \Sigma}{S_1; S_2 : \Sigma \xrightarrow[(\mathsf{RD}_1 \setminus X_2) \cup \mathsf{RD}_2]{X_1 \cup X_2} \Sigma}$$

$$[if] \quad \frac{S_1 : \Sigma \xrightarrow[\mathsf{RD}_1]{X_1} \Sigma \qquad S_2 : \Sigma \xrightarrow[\mathsf{RD}_2]{X_2} \Sigma}{\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 : \Sigma \xrightarrow[\mathsf{RD}_1 \cup \mathsf{RD}_2]{X_1 \cap X_2} \Sigma}$$

$$[wh] \quad \frac{S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma}{\texttt{while } [b]^\ell \texttt{ do } S : \Sigma \xrightarrow[\mathsf{RD}]{\emptyset} \Sigma}$$

$$[sub] \quad \frac{S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma}{S : \Sigma \xrightarrow[\mathsf{RD}']{X'} \Sigma} \quad \text{if } X' \subseteq X \text{ and } \mathsf{RD} \subseteq \mathsf{RD}'$$

Table 1.3: Reaching Definitions: annotated type constructors.

where X denotes the set of variables that *definitely* will be assigned in S and RD denotes the set of reaching definitions that S *might* produce. The axioms and rules are shown in Table 1.3 and are explained below.

The axiom for assignment simply expresses that the variable x definitely will be assigned and that the reaching definition (x, ℓ) is produced. In the rule for sequencing the notation $\mathsf{RD} \setminus X$ means $\{(x, \ell) \in \mathsf{RD} \mid x \notin X\}$. The rule expresses that we take the union of the reaching definitions after having removed entries from S_1 that are definitely redefined in S_2. Also we take the union of the two sets of assigned variables. In the rule for conditional we take the union of information about reaching definitions whereas we take the intersection (rather than the union) of the assigned variables because we are not completely sure what path was taken through the conditional. A similar comment holds for the rule for the `while`-loop; here we can think of $\emptyset$ as the intersection between $\emptyset$ (when the body is not executed) and X.

We have included a *subsumption rule* because this is normally the case for such systems as we shall see in Chapter 5. However, in the system above there is little need for it, and if one excludes it then implementation becomes very straightforward: simply perform a syntax directed traversal of the program where the sets X and RD are computed for each subprogram.

Example 1.4 Let us once again consider the analysis of the factorial program

$$[\texttt{y:=x}]^1;\ [\texttt{z:=1}]^2;\ \texttt{while } [\texttt{y>1}]^3 \texttt{ do } ([\texttt{z:=z*y}]^4;\ [\texttt{y:=y-1}]^5);\ [\texttt{y:=0}]^6$$

For the body of the `while`-loop we get

$$[\texttt{z:=z*y}]^4\colon \Sigma \xrightarrow[\{(\texttt{z},4)\}]{\{\texttt{z}\}} \Sigma$$

$$[\texttt{y:=y-1}]^5\colon \Sigma \xrightarrow[\{(\texttt{y},5)\}]{\{\texttt{y}\}} \Sigma$$

so the rule [*seq*] gives:

$$([\texttt{z:=z*y}]^4;\ [\texttt{y:=y-1}]^5)\colon \Sigma \xrightarrow[\{(\texttt{y},5),(\texttt{z},4)\}]{\{\texttt{y},\texttt{z}\}} \Sigma$$

We can now apply the rule [*wh*] and get:

$$\texttt{while}\ [\texttt{y>1}]^3\ \texttt{do}\ ([\texttt{z:=z*y}]^4;\ [\texttt{y:=y-1}]^5)\colon \Sigma \xrightarrow[\{(\texttt{y},5),(\texttt{z},4)\}]{\emptyset} \Sigma$$

In a similar way we get

$$([\texttt{y:=x}]^1;\ [\texttt{z:=1}]^2)\colon \Sigma \xrightarrow[\{(\texttt{y},1),(\texttt{z},2)\}]{\{\texttt{y},\texttt{z}\}} \Sigma$$

$$[\texttt{y:=0}]^6\colon \Sigma \xrightarrow[\{(\texttt{y},6)\}]{\{\texttt{y}\}} \Sigma$$

so using the rule [*seq*] we obtain

$$([\texttt{y:=x}]^1;\ [\texttt{z:=1}]^2;\ \texttt{while}\ [\texttt{y>1}]^3\ \texttt{do}\ ([\texttt{z:=z*y}]^4;\ [\texttt{y:=y-1}]^5);\ [\texttt{y:=0}]^6)\colon$$
$$\Sigma \xrightarrow[\{(\texttt{y},6),(\texttt{z},2),(\texttt{z},4)\}]{\{\texttt{y},\texttt{z}\}} \Sigma$$

showing that the program definitely will assign to `y` and `z` and that the final value of `y` will be assigned at 6 and the final value of `z` at 2 or 4. ∎

Compared with the previous specifications of Reaching Definitions analysis the flavour of Table 1.3 is rather different: the analysis of a statement expresses how information present at the entry will be *modified* by the statement – we may therefore view the specification as a higher-order formulation of Reaching Definitions analysis.

1.6.2 Effect Systems

A simple type system. To give the flavour of Effect Systems let us once more turn to the functional language. As above, the idea is to annotate a traditional type system with analysis information, so let us start by presenting a simple type system for a language with variables x, function abstraction $\texttt{fn}_\pi\ x\ \texttt{=>}\ e$ (where π is the name of the abstraction), and function application $e_1\ e_2$. The judgements have the form

$$\Gamma \vdash e : \tau$$

where Γ is a *type environment* that gives types to all free variables of e and τ is the *type* of e. For simplicity we shall assume that types are either base

types such as `int` and `bool` or they are function types written $\tau_1 \to \tau_2$. The type system is given by the following axioms and rules:

$$\Gamma \vdash x : \tau_x \quad \text{if } \Gamma(x) = \tau_x$$

$$\frac{\Gamma[x \mapsto \tau_x] \vdash e : \tau}{\Gamma \vdash \mathtt{fn}_\pi\ x \mathtt{\ =>\ } e : \tau_x \to \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau, \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$$

So the axiom for variables just expresses that the type of x is obtained from the assumptions of the type environment. The rule for function abstraction requires that we "guess" a type τ_x for the formal parameter x and we determine the type of the body of the abstraction under that additional assumption. The rule for function application requires that we determine the type of the operator as well as the argument and it implicitly expresses that the operator must have a function type by requiring the type of e_1 to have the form $\tau_2 \to \tau$. Furthermore the two occurrences of τ_2 in the rule implicitly express that the type of the actual parameter must equal the type expected by the formal parameter of the function.

Example 1.5 Consider the following version of the program of Example 1.2

$$(\mathtt{fn}_{\mathsf{X}}\ \mathtt{x\ =>\ x})\ (\mathtt{fn}_{\mathsf{Y}}\ \mathtt{y\ =>\ y})$$

where we now have given `fn x => x` the name X and `fn y => y` the name Y. To see that this program has type `int` $\to$ `int` we first observe that $[\mathtt{y} \mapsto \mathtt{int}] \vdash \mathtt{y} : \mathtt{int}$ so:

$$[\,] \vdash \mathtt{fn}_{\mathsf{Y}}\ \mathtt{y\ =>\ y} : \mathtt{int} \to \mathtt{int}$$

Similarly, we have $[\mathtt{x} \mapsto \mathtt{int} \to \mathtt{int}] \vdash \mathtt{x} : \mathtt{int} \to \mathtt{int}$ so:

$$[\,] \vdash \mathtt{fn}_{\mathsf{X}}\ \mathtt{x\ =>\ x} : (\mathtt{int} \to \mathtt{int}) \to (\mathtt{int} \to \mathtt{int})$$

The rule for application then gives:

$$[\,] \vdash (\mathtt{fn}_{\mathsf{X}}\ \mathtt{x\ =>\ x})\ (\mathtt{fn}_{\mathsf{Y}}\ \mathtt{y\ =>\ y}) : \mathtt{int} \to \mathtt{int}$$

■

Effects. The analysis we shall consider is a *Call-Tracking Analysis*:

> For each subexpression, which function abstractions may be applied during its evaluation.

$$[var] \quad \widehat{\Gamma} \vdash x : \widehat{\tau}_x \;\&\; \emptyset \;\text{ if } \widehat{\Gamma}(x) = \widehat{\tau}_x$$

$$[fn] \quad \frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_x] \vdash e : \widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash \mathtt{fn}_\pi\ x \texttt{ => } e : \widehat{\tau}_x \xrightarrow{\varphi \cup \{\pi\}} \widehat{\tau} \;\&\; \emptyset}$$

$$[app] \quad \frac{\widehat{\Gamma} \vdash e_1 : \widehat{\tau}_2 \xrightarrow{\varphi} \widehat{\tau} \;\&\; \varphi_1 \qquad \widehat{\Gamma} \vdash e_2 : \widehat{\tau}_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash e_1\ e_2 : \widehat{\tau} \;\&\; \varphi_1 \cup \varphi_2 \cup \varphi}$$

Table 1.4: Call-tracking Analysis: Effect System.

The set of function names constitutes the *effect* of the subexpression. To determine this information we shall annotate the function types with their *latent effect* so for example we shall write $\mathtt{int} \xrightarrow{\{\mathsf{X}\}} \mathtt{int}$ for the type of a function mapping integers to integers and with effect $\{\mathsf{X}\}$ meaning that when executing the function it may apply the function named X. More generally, the annotated types $\widehat{\tau}$ will either be base types or they will have the form

$$\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$$

where φ is the effect, i.e. the names of the function abstractions that we might apply when applying a function of this type.

We specify the analysis using judgements of the form

$$\widehat{\Gamma} \vdash e : \widehat{\tau} \;\&\; \varphi$$

where $\widehat{\Gamma}$ is the type environment that now gives the annotated type of all free variables, $\widehat{\tau}$ is the annotated type of e, and φ is the effect of evaluating e. The analysis is specified by the axioms and rules in Table 1.4 which will be explained below.

In the axiom [*var*] for variables we produce an empty effect because we assume that the parameter mechanism is call-by-value and therefore no evaluation takes place when mentioning a variable. Similarly, in the rule [*fn*] for function abstractions we produce an empty effect: no evaluation takes place because we only construct a closure. The body of the abstraction is analysed in order to determine its annotated type and effect. This information is needed to annotate the function arrow: all the names of functions in the effect of the body and the name of the abstraction itself may be involved when this particular abstraction is applied.

Next, consider the rule [*app*] for function application $e_1\ e_2$. Here we obtain annotated types and effects from the operator e_1 as well as the operand e_2. The effect of the application will contain the effect φ_1 of the operator (because we have to evaluate it before the application can take place), the effect φ_2 of

the operand (because we employ a call-by-value semantics so this expression has to be evaluated too) and finally we need the effect φ of the function being applied. But this is exactly the information given by the annotation of the arrow in the type $\widehat{\tau}_2 \xrightarrow{\varphi} \widehat{\tau}$ of the operand. Hence we produce the union of these three sets as the overall effect of the application.

Example 1.6 Returning to the program of Example 1.5 we have:

$$[\,] \vdash \texttt{fn}_{\mathsf{Y}}\ \texttt{y => y} : \texttt{int} \xrightarrow{\{\mathsf{Y}\}} \texttt{int}\ \&\ \emptyset$$

$$[\,] \vdash \texttt{fn}_{\mathsf{X}}\ \texttt{x => x} : (\texttt{int} \xrightarrow{\{\mathsf{Y}\}} \texttt{int}) \xrightarrow{\{\mathsf{X}\}} (\texttt{int} \xrightarrow{\{\mathsf{Y}\}} \texttt{int})\ \&\ \emptyset$$

$$[\,] \vdash (\texttt{fn}_{\mathsf{X}}\ \texttt{x => x})\ (\texttt{fn}_{\mathsf{Y}}\ \texttt{y => y}) : \texttt{int} \xrightarrow{\{\mathsf{Y}\}} \texttt{int}\ \&\ \{\mathsf{X}\}$$

This shows that our example program may (in fact it will) apply the function `fn x => x` but that it will not apply the function `fn y => y`. ∎

For a more general language we will also need to introduce some form of subsumption rule in the manner of Tables 1.2 and 1.3; there are different approaches to this and we shall return to this later. Effect Systems are often implemented as extensions of type inference algorithms and, depending on the form of the effects, it may be possible to calculate them on the fly; alternatively, sets of constraints can be generated and solved subsequently. We refer to Chapter 5 for more details.

1.7 Algorithms

Let us now reconsider the problem of computing the least solution to the program analysis problems considered in Data Flow Analysis and Constraint Based Analysis.

Recall from Section 1.3 that we consider twelve-tuples $\overrightarrow{\mathsf{RD}} \in (\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12}$ of pairs of variables and labels where each label indicates an elementary block in which the corresponding variable was last assigned. The equation or constraint system gives rise to demanding the least solution to an equation $\overrightarrow{\mathsf{RD}} = F(\overrightarrow{\mathsf{RD}})$ or inclusion $\overrightarrow{\mathsf{RD}} \sqsupseteq F(\overrightarrow{\mathsf{RD}})$ where F is a monotone function over $(\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12}$. Due to the finiteness of $(\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12}$ the desired solution is in both cases obtainable as $F^n(\vec{\emptyset})$ for any n such that $F^{n+1}(\vec{\emptyset}) = F^n(\vec{\emptyset})$ and we know that such an n does in fact exist.

Chaotic Iteration. Naively implementing the above procedure soon turns out to require an overwhelming amount of work. In later chapters we shall see much more efficient algorithms and in this section we shall illustrate the principle of *Chaotic Iteration* that lies at the heart of many of them. For

INPUT:	Example equations for Reaching Definitions	
OUTPUT:	The least solution: $\overrightarrow{\mathsf{RD}} = (\mathsf{RD}_1, \cdots, \mathsf{RD}_{12})$	
METHOD:	Step 1:	Initialisation $\mathsf{RD}_1{:=}\emptyset; \cdots; \mathsf{RD}_{12}{:=}\emptyset$
	Step 2:	Iteration while $\mathsf{RD}_j \neq F_j(\mathsf{RD}_1, \cdots, \mathsf{RD}_{12})$ for some j do $\mathsf{RD}_j{:=}F_j(\mathsf{RD}_1, \cdots, \mathsf{RD}_{12})$

Table 1.5: Chaotic Iteration for Reaching Definitions.

this let us write

$$\begin{aligned} \overrightarrow{\mathsf{RD}} &= (\mathsf{RD}_1, \cdots, \mathsf{RD}_{12}) \\ F(\overrightarrow{\mathsf{RD}}) &= (F_1(\overrightarrow{\mathsf{RD}}), \cdots, F_{12}(\overrightarrow{\mathsf{RD}})) \end{aligned}$$

and consider the non-deterministic algorithm in Table 1.5. It is immediate that there exists j such that $\mathsf{RD}_j \neq F_j(\mathsf{RD}_1, \cdots, \mathsf{RD}_{12})$ if and only if $\overrightarrow{\mathsf{RD}} \neq F(\overrightarrow{\mathsf{RD}})$. Hence if the algorithm terminates it will produce a fixed point of F; that is, a solution to the desired equations or constraints.

Properties of the algorithm. To further analyse the algorithm we shall exploit that

$$\vec{\emptyset} \sqsubseteq \overrightarrow{\mathsf{RD}} \sqsubseteq F(\overrightarrow{\mathsf{RD}}) \sqsubseteq F^n(\vec{\emptyset})$$

holds at all points in the algorithm (where n is determined by $F^{n+1}(\vec{\emptyset}) = F^n(\vec{\emptyset})$): it clearly holds initially and, as will be shown in Exercise 1.6, it is maintained during iteration. This means that if the algorithm terminates we will have obtained not only a fixed point of F but in fact the least fixed point (i.e. $F^n(\vec{\emptyset})$).

To see that the algorithm terminates note that if j satisfies

$$\mathsf{RD}_j \neq F_j(\mathsf{RD}_1, \cdots, \mathsf{RD}_{12})$$

then in fact $\mathsf{RD}_j \subset F_j(\mathsf{RD}_1, \cdots, \mathsf{RD}_{12})$ and hence the size of $\overrightarrow{\mathsf{RD}}$ increases by at least one as we perform each iteration. This ensures termination since we assumed that $(\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12}$ is finite.

The above algorithm is suitable for *manually* solving data flow equations and constraint systems. To obtain an algorithm that is suitable for implementation we need to give more details about the choice of j so as to avoid an extensive search for the value; we shall return to this in Chapters 2 and 6.

$$[ass_1] \quad \mathsf{RD} \vdash [x := a]^\ell \ \rhd \ [x := a[y \mapsto n]]^\ell$$
$$\text{if} \begin{cases} y \in FV(a) \ \wedge \ (y, ?) \notin \mathsf{RD}_{entry}(\ell) \ \wedge \\ \forall (z, \ell') \in \mathsf{RD}_{entry}(\ell) : (z = y \Rightarrow [\cdots]^{\ell'} \text{ is } [y := n]^{\ell'}) \end{cases}$$

$$[ass_2] \quad \mathsf{RD} \vdash [x := a]^\ell \ \rhd \ [x := n]^\ell$$
$$\text{if } FV(a) = \emptyset \ \wedge \ a \notin \mathbf{Num} \ \wedge \ a \text{ evaluates to } n$$

$$[seq_1] \quad \frac{\mathsf{RD} \vdash S_1 \ \rhd \ S_1'}{\mathsf{RD} \vdash S_1; S_2 \ \rhd \ S_1'; S_2}$$

$$[seq_2] \quad \frac{\mathsf{RD} \vdash S_2 \ \rhd \ S_2'}{\mathsf{RD} \vdash S_1; S_2 \ \rhd \ S_1; S_2'}$$

$$[if_1] \quad \frac{\mathsf{RD} \vdash S_1 \ \rhd \ S_1'}{\mathsf{RD} \vdash \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \ \rhd \ \texttt{if } [b]^\ell \texttt{ then } S_1' \texttt{ else } S_2}$$

$$[if_2] \quad \frac{\mathsf{RD} \vdash S_2 \ \rhd \ S_2'}{\mathsf{RD} \vdash \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \ \rhd \ \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2'}$$

$$[wh] \quad \frac{\mathsf{RD} \vdash S \ \rhd \ S'}{\mathsf{RD} \vdash \texttt{while } [b]^\ell \texttt{ do } S \ \rhd \ \texttt{while } [b]^\ell \texttt{ do } S'}$$

Table 1.6: Constant Folding transformation.

1.8 Transformations

A major application of program analysis is to transform the program (at the source level or at some intermediate level inside a compiler) so as to obtain better performance. To illustrate the ideas we shall show how Reaching Definitions can be used to perform a transformation known as *Constant Folding*. There are two ingredients in this. One is to replace the use of a variable in some expression by a constant if it is known that the value of the variable will always be that constant. The other is to simplify an expression by partially evaluating it: subexpressions that contain no variables can be evaluated.

Source to source transformation. Consider a program $S_\star$ and let RD be a solution (preferable the least) to the Reaching Definitions Analysis for $S_\star$. For a sub-statement S of $S_\star$ we shall now describe how to transform it into a "better" statement S'. We do so by means of judgements of the form

$$\mathsf{RD} \vdash S \ \rhd \ S'$$

expressing *one step* of the transformation process. We may define the transformation using the axioms and rules in Table 1.6; they are explained below.

The first axiom $[ass_1]$ expresses the first ingredient in Constant Folding as explained above – the use of a variable can be replaced with a constant if it is known that the variable always will be that constant; here we write $a[y \mapsto n]$ for the expression that is as a except that all occurrences of y have been replaced by n; also we write $FV(a)$ for the set of variables occurring in a.

The second axiom $[ass_2]$ expresses the second ingredient of the transformation – expressions can be partially evaluated; it uses the fact that if an expression contains no variables then it will always evaluate to the same value.

The last five rules in Table 1.6 simply say that if we can transform a substatement then we can transform the statement itself. Note that the rules (e.g. $[seq_1]$ and $[seq_2]$) do not prescribe a specific transformation order and hence many different transformation sequences may exist. Also note that the relation $\mathsf{RD} \vdash \cdot \ \rhd \ \cdot$ is neither reflexive nor transitive because there are no rules that forces it to be so. Hence we shall often want to perform an entire sequence of transformations.

Example 1.7 To illustrate the use of the transformation consider the program:

$$[\mathtt{x:=10}]^1; [\mathtt{y:=x+10}]^2; [\mathtt{z:=y+10}]^3$$

The least solution to the Reaching Definitions Analysis for this program is:

$$\begin{array}{rcl}
\mathsf{RD}_{entry}(1) & = & \{(\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?)\} \\
\mathsf{RD}_{exit}(1) & = & \{(\mathtt{x},1),(\mathtt{y},?),(\mathtt{z},?)\} \\
\mathsf{RD}_{entry}(2) & = & \{(\mathtt{x},1),(\mathtt{y},?),(\mathtt{z},?)\} \\
\mathsf{RD}_{exit}(2) & = & \{(\mathtt{x},1),(\mathtt{y},2),(\mathtt{z},?)\} \\
\mathsf{RD}_{entry}(3) & = & \{(\mathtt{x},1),(\mathtt{y},2),(\mathtt{z},?)\} \\
\mathsf{RD}_{exit}(3) & = & \{(\mathtt{x},1),(\mathtt{y},2),(\mathtt{z},3)\}
\end{array}$$

Let us now see how to transform the program. From the axiom $[ass_1]$ we have

$$\mathsf{RD} \vdash [\mathtt{y:=x+10}]^2 \ \rhd \ [\mathtt{y:=10+10}]^2$$

and therefore the rules for sequencing gives:

$$\mathsf{RD} \vdash [\mathtt{x:=10}]^1; [\mathtt{y:=x+10}]^2; [\mathtt{z:=y+10}]^3 \rhd [\mathtt{x:=10}]^1; [\mathtt{y:=10+10}]^2; [\mathtt{z:=y+10}]^3$$

We can now continue and obtain the following transformation sequence:

$$\begin{array}{lll}
\mathsf{RD} & \vdash & [\mathtt{x:=10}]^1; [\mathtt{y:=x+10}]^2; [\mathtt{z:=y+10}]^3 \\
 & \rhd & [\mathtt{x:=10}]^1; [\mathtt{y:=10+10}]^2; [\mathtt{z:=y+10}]^3 \\
 & \rhd & [\mathtt{x:=10}]^1; [\mathtt{y:=20}]^2; [\mathtt{z:=y+10}]^3 \\
 & \rhd & [\mathtt{x:=10}]^1; [\mathtt{y:=20}]^2; [\mathtt{z:=20+10}]^3 \\
 & \rhd & [\mathtt{x:=10}]^1; [\mathtt{y:=20}]^2; [\mathtt{z:=30}]^3
\end{array}$$

after which no more steps are possible. ■

Successive transformations. The above example shows that we shall want to perform many successive transformations:

$$\mathsf{RD} \vdash S_1 \rhd S_2 \rhd \cdots \rhd S_{n+1}$$

This could be costly because once S_1 has been transformed into S_2 we might have to *recompute* Reaching Definitions Analysis for S_2 before the transformation can be used to transform it into S_3 etc. It turns out that it is sometimes possible to use the analysis for S_1 to obtain a reasonable analysis for S_2 without performing the analysis from scratch. In the case of Reaching Definitions and Constant Folding this is very easy: if RD is a solution to Reaching Definitions for S_i and $\mathsf{RD} \vdash S_i \rhd S_{i+1}$ then RD is also a solution to Reaching Definitions for S_{i+1} – intuitively, the reason is that the transformation only changed things that were not observed by the Reaching Definitions Analysis.

Concluding Remarks

In this chapter we have briefly illustrated a few approaches (but by no means all) to program analysis. Clearly there are many differences between the four approaches. However, the main aim of the chapter has been to suggest that there are also more *similarities* than one would perhaps have expected at first sight: in particular, the interplay between the use of equations versus constraints. It is also interesting to note that some of the techniques touched upon in this chapter have close connections to other approaches to reasoning about programs; especially, some versions of Annotated Type Systems are closely related to Hoare's logic for partial correctness assertions.

As mentioned earlier, the approaches to program analysis covered in this book are *semantics based* rather than *semantics directed.* The semantics directed approaches include the denotational based approaches [27, 86, 115, 117] and logic based approaches [19, 20, 81, 82].

Mini Projects

Mini Project 1.1 Correctness of Reaching Definitions

In this mini project we shall increase our faith in the Type and Effect System of Table 1.3 by proving that it is correct. This requires knowledge of regular expressions and homomorphisms to the extent covered in Appendix C.

First we shall show how to associate a regular expression with each statement. We define a function $\mathcal{S}$ such that $\mathcal{S}(S)$ is a regular expression for each statement $S \in \mathbf{Stmt}$. It is defined by structural induction (see Appendix B) as follows:

$$\begin{array}{rcl}
\mathcal{S}([x{:=}a]^\ell) & = & !_x^\ell \\
\mathcal{S}([\mathtt{skip}]^\ell) & = & \Lambda \\
\mathcal{S}(S_1; S_2) & = & \mathcal{S}(S_1) \cdot \mathcal{S}(S_2) \\
\mathcal{S}(\mathtt{if}\ [b]^\ell\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2) & = & \mathcal{S}(S_1) + \mathcal{S}(S_2) \\
\mathcal{S}(\mathtt{while}\ b\ \mathtt{do}\ S) & = & (\mathcal{S}(S))^*
\end{array}$$

The alphabet is $\{!_x^\ell \mid x \in \mathbf{Var}_\star, \ell \in \mathbf{Lab}_\star\}$ where $\mathbf{Var}_\star$ and $\mathbf{Lab}_\star$ are finite and non-empty sets that contain all the variables and labels, respectively, of the statement $S_\star$ of interest. As an example, for $S_\star$ being

$$\mathtt{if}\ [\mathtt{x>0}]^1\ \mathtt{then}\ [\mathtt{x:=x+1}]^2\ \mathtt{else}\ ([\mathtt{x:=x+2}]^3; [\mathtt{x:=x+3}]^4)$$

we have $\mathcal{S}(S_\star) = !_{\mathtt{x}}^2 + (!_{\mathtt{x}}^3 \cdot !_{\mathtt{x}}^4)$.

Correctness of X. To show the correctness of the X component in $S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma$ we shall for each $y \in \mathbf{Var}_\star$ define a homomorphism

$$h_y : \{!_x^\ell \mid x \in \mathbf{Var}_\star, \ell \in \mathbf{Lab}_\star\} \to \{!\}^*$$

as follows:

$$h_y(!_x^\ell) = \begin{cases} ! & \text{if } y = x \\ \Lambda & \text{if } y \neq x \end{cases}$$

As an example $h_{\mathtt{x}}(\mathcal{S}(S_\star)) = ! + (! \cdot !)$ and $h_{\mathtt{y}}(\mathcal{S}(S_\star)) = \Lambda$ using that $\Lambda \cdot \Lambda = \Lambda$ and $\Lambda + \Lambda = \Lambda$. Next we write

$$h_y(\mathcal{S}(S)) \subseteq ! \cdot !^*$$

to mean that the language $\mathcal{L}[\![h_y(\mathcal{S}(S))]\!]$ defined by the regular expression $h_y(\mathcal{S}(S))$ is a subset of the language $\mathcal{L}[\![! \cdot !^*]\!]$ defined by $! \cdot !^*$; this is equivalent to

$$\neg\exists w \in \mathcal{L}[\![h_y(\mathcal{S}(S))]\!] : h_y(w) = \Lambda$$

and intuitively says that y is always assigned in S. Prove that

$$\text{if } S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma \text{ and } y \in X \text{ then } h_y(\mathcal{S}(S)) \subseteq ! \cdot !^*$$

by induction on the shape of the inference tree establishing $S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma$ (see Appendix B for an introduction to the proof principle).

Correctness of RD. To show the correctness of the RD component in $S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma$ we shall for each $y \in \mathbf{Var}_\star$ and $\ell' \in \mathbf{Lab}_\star$ define a homomorphism

$$h_y^{\ell'} : \{!_x^\ell \mid x \in \mathbf{Var}_\star, \ell \in \mathbf{Lab}_\star\} \to \{!, ?\}^*$$

as follows:

$$h_y^{\ell'}(!_x^\ell) = \begin{cases} ! & \text{if } y = x \ \wedge \ \ell = \ell' \\ ? & \text{if } y = x \ \wedge \ \ell \neq \ell' \\ \Lambda & \text{if } y \neq x \end{cases}$$

As an example $h_{\mathsf{x}}^2(\mathcal{S}(S_\star)) = ! + (? \cdot ?)$ and $h_{\mathsf{y}}^5(\mathcal{S}(S_\star)) = \Lambda$. Next

$$h_y^{\ell'}(\mathcal{S}(S)) \subseteq ((!+?)^* \cdot ?) + \Lambda$$

is equivalent to

$$\neg\exists w \in \mathcal{L}[\![\mathcal{S}(S)]\!] : h_y^{\ell'}(w) \text{ ends in } !$$

and intuitively means than the last assignment to y could not have been performed at the statement labelled ℓ'. Prove that

$$\text{if } S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma \text{ and } (y, \ell') \notin \mathsf{RD} \text{ then } h_y^{\ell'}(\mathcal{S}(S)) \subseteq ((!+?)^* \cdot ?) + \Lambda$$

by induction on the shape of the inference tree establishing $S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma$. ■

Exercises

Exercise 1.1 A variant of Reaching Definitions replaces $\mathsf{RD} \in \mathcal{P}(\mathbf{Var} \times \mathbf{Lab})$ by $\mathsf{RL} \in \mathcal{P}(\mathbf{Lab})$; the idea is that given the program, a label should suffice for finding the variables that may be assigned in some elementary block bearing that label. Use this as the basis for modifying the equation system given in Section 1.3 for $\overrightarrow{\mathsf{RD}}$ to an equation system for $\overrightarrow{\mathsf{RL}}$. (Hint: It may be appropriate to think of $\mathsf{RD} = \{(x_1, ?), \cdots, (x_n, ?)\}$ as meaning $\mathsf{RD} = \{(x_1, ?_{x_1}), \cdots, (x_n, ?_{x_n})\}$ and then use $\mathsf{RL} = \{?_{x_1}, \cdots, ?_{x_n}\}$.) ■

Exercise 1.2 Show that the solution displayed for the Control Flow Analysis in Section 1.4 is a solution. Also show that it is in fact the least solution. (Hint: Consider the demands on $\widehat{\mathsf{C}}(2)$, $\widehat{\mathsf{C}}(4)$, $\widehat{\rho}(\mathbf{x})$, $\widehat{\mathsf{C}}(1)$ and $\widehat{\mathsf{C}}(5)$.) ■

Exercise 1.3 Let (α, γ) be an adjunction, or a Galois connection, as explained in Section 1.5; this just means that $\alpha(X) \subseteq Y \Leftrightarrow X \subseteq \gamma(Y)$ holds for all X and Y. Show that α uniquely determines γ in the sense that $\gamma = \gamma'$ whenever (α, γ') is an adjunction. Also show that γ uniquely determines α for (α, γ) being an adjunction. ■

Exercise 1.4 For F as in Section 1.3 and α, $\vec{\alpha}$, γ, $\vec{\gamma}$ and G as in Section 1.5 show that $\vec{\alpha} \circ G \circ \vec{\gamma} \sqsubseteq F$; this involves showing that

$$\alpha(G_j(\gamma(\mathsf{RD}_1), \cdots, \gamma(\mathsf{RD}_{12}))) \subseteq F_j(\mathsf{RD}_1, \cdots, \mathsf{RD}_{12})$$

for all j and $(\mathsf{RD}_1, \cdots, \mathsf{RD}_{12})$. Determine whether or not $F = \vec{\alpha} \circ G \circ \vec{\gamma}$. Prove by numerical induction on n that $(\vec{\alpha} \circ G \circ \vec{\gamma})^n(\vec{\emptyset}) \sqsubseteq F^n(\vec{\emptyset})$. Also prove that $\vec{\alpha}(G^n(\vec{\emptyset})) \sqsubseteq (\vec{\alpha} \circ G \circ \vec{\gamma})^n(\vec{\emptyset})$ using that $\vec{\alpha}(\vec{\emptyset}) = \vec{\emptyset}$ and $G \sqsubseteq G \circ \vec{\gamma} \circ \vec{\alpha}$. ■

Exercise 1.5 Consider the Annotated Type System for Reaching Definitions defined in Table 1.2 in Section 1.6 and suppose that we want to stick to the first (and unsuccessful) explanation of what $S : \mathsf{RD}_1 \rightarrow \mathsf{RD}_2$ means in terms of Data Flow Analysis. Can you change Table 1.2 (by modifying or removing axioms and rules) such that this becomes possible? ■

Exercise 1.6 Consider the Chaotic Iteration algorithm of Section 1.7 and suppose that

$$\vec{\emptyset} \sqsubseteq \overrightarrow{\mathsf{RD}} \sqsubseteq F(\overrightarrow{\mathsf{RD}}) \sqsubseteq F^n(\vec{\emptyset}) = F^{n+1}(\vec{\emptyset})$$

holds immediately before the assignment to RD_j; show that is also holds afterwards. (Hint: Write $\overrightarrow{\mathsf{RD}}'$ for $(\mathsf{RD}_1, \cdots, F_j(\overrightarrow{\mathsf{RD}}), \cdots, \mathsf{RD}_{12})$ and use the monotonicity of F and $\overrightarrow{\mathsf{RD}} \sqsubseteq F(\overrightarrow{\mathsf{RD}})$ to establish that $\overrightarrow{\mathsf{RD}} \sqsubseteq \overrightarrow{\mathsf{RD}}' \sqsubseteq F(\overrightarrow{\mathsf{RD}}) \sqsubseteq F(\overrightarrow{\mathsf{RD}}')$.) ■

Exercise 1.7 Use the Chaotic Iteration scheme of Section 1.7 to show that the information displayed in Table 1.1 is in fact the least fixed point of the function F defined in Section 1.3. ■

Exercise 1.8 Consider the following program

$$[\texttt{z:=1}]^1; \texttt{while } [\texttt{x>0}]^2 \texttt{ do } ([\texttt{z:=z*y}]^3; [\texttt{x:=x-1}]^4)$$

computing the x-th power of the number stored in y. Formulate a system of data flow equations in the manner of Section 1.3. Next use the Chaotic Iteration strategy of Section 1.7 to compute the least solution and present it in a table (like Table 1.1). ■

Exercise 1.9 Perform Constant Folding upon the program

$$[\texttt{x:=10}]^1; [\texttt{y:=x+10}]^2; [\texttt{z:=y+x}]^3$$

so as to obtain

$$[\texttt{x:=10}]^1; [\texttt{y:=20}]^2; [\texttt{z:=30}]^3$$

How many ways of obtaining the result are there? ■

Exercise 1.10 The specification of Constant Folding in Section 1.8 only considers arithmetic expressions. Extend it to deal also with boolean expressions. Consider adding axioms like

$$\mathsf{RD} \vdash ([\texttt{skip}]^{\ell}; S) \ \rhd \ S$$

$$\mathsf{RD} \vdash (\texttt{if}\ [\texttt{true}]^{\ell}\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \ \rhd \ S_1$$

and discuss what complications arise. ■

Exercise 1.11 Consider adding the axiom

$$\mathsf{RD} \vdash [x := a]^{\ell} \ \rhd \ [x := a[y \mapsto a']]^{\ell}$$
$$\text{if} \begin{cases} y \in FV(a) \ \wedge \ (y, ?) \notin \mathsf{RD}_{entry}(\ell) \ \wedge \\ \forall (z, \ell') \in \mathsf{RD}_{entry}(\ell) : (y = z \Rightarrow [\cdots]^{\ell'} \text{ is } [y := a']^{\ell'}) \end{cases}$$

to the specification of Constant Folding given in Section 1.8 and discuss whether or not this is a good idea. ■

Chapter 2

Data Flow Analysis

In this chapter we introduce techniques for Data Flow Analysis. Data Flow Analysis is the traditional form of program analysis which is described in many textbooks on compiler writing. We will present analyses for the simple imperative language WHILE that was introduced in Chapter 1. This includes a number of classical Data Flow Analyses: Available Expressions, Reaching Definitions, Very Busy Expressions and Live Variables. We introduce an operational semantics for WHILE and demonstrate the correctness of the Live Variables Analysis. We then present the notion of Monotone Frameworks and show how the examples may be recast as such frameworks. We continue by presenting a worklist algorithm for solving flow equations and we study its termination and correctness properties. The chapter concludes with a presentation of some advanced topics, including Interprocedural Data Flow Analysis and Shape Analysis.

Throughout the chapter we will clarify the distinctions between intraprocedural and interprocedural analyses, between forward and backward analyses, between may and must analyses (or union and intersection analyses), between flow sensitive and flow insensitive analyses, and between context sensitive and context insensitive analyses.

2.1 Intraprocedural Analysis

In this section we present a number of example Data Flow Analyses for the WHILE language. The analyses are each defined by pairs of functions that map labels to the appropriate sets; one function in each pair specifies information that is true on *entry* to the block, the second specifies information that is true at the *exit*.

Initial and final labels. When presenting examples of Data Flow Analyses we will use a number of operations on programs and labels. The first of these is

$$init : \mathbf{Stmt} \to \mathbf{Lab}$$

which returns the *initial label* of a statement:

$$\begin{aligned}
init([x := a]^\ell) &= \ell \\
init([\texttt{skip}]^\ell) &= \ell \\
init(S_1; S_2) &= init(S_1) \\
init(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2) &= \ell \\
init(\texttt{while } [b]^\ell \texttt{ do } S) &= \ell
\end{aligned}$$

We will also need a function

$$final : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab})$$

which returns the set of *final labels* in a statement; whereas a sequence of statements has a single entry, it may have multiple exits (as for example in the conditional):

$$\begin{aligned}
final([x := a]^\ell) &= \{\ell\} \\
final([\texttt{skip}]^\ell) &= \{\ell\} \\
final(S_1; S_2) &= final(S_2) \\
final(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2) &= final(S_1) \cup final(S_2) \\
final(\texttt{while } [b]^\ell \texttt{ do } S) &= \{\ell\}
\end{aligned}$$

Note that the `while`-loop terminates immediately after the test has evaluated to false.

Blocks. To access the statements or tests associated with a label in a program we use the function

$$blocks : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Blocks})$$

where **Blocks** is the set of statements, or *elementary blocks*, of the form $[x{:=}a]^\ell$ or $[\texttt{skip}]^\ell$ as well as the set of tests of the form $[b]^\ell$. It is defined as follows:

$$\begin{aligned}
blocks([\texttt{x} := a]^\ell) &= \{[\texttt{x} := a]^\ell\} \\
blocks([\texttt{skip}]^\ell) &= \{[\texttt{skip}]^\ell\} \\
blocks(S_1; S_2) &= blocks(S_1) \cup blocks(S_2) \\
blocks(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2) &= \{[b]^\ell\} \cup blocks(S_1) \cup blocks(S_2) \\
blocks(\texttt{while } [b]^\ell \texttt{ do } S) &= \{[b]^\ell\} \cup blocks(S)
\end{aligned}$$

Then the set of *labels* occurring in a program is given by

$$\mathit{labels} : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab})$$

where

$$\mathit{labels}(S) = \{\ell \mid [B]^\ell \in \mathit{blocks}(S)\}$$

Clearly $\mathit{init}(S) \in \mathit{labels}(S)$ and $\mathit{final}(S) \subseteq \mathit{labels}(S)$.

Flows and reverse flows. We will need to operate on edges, or *flows*, between labels in a statement. We define a function

$$\mathit{flow} : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab} \times \mathbf{Lab})$$

which maps statements to sets of flows:

$$\begin{array}{rcl}
\mathit{flow}([x := a]^\ell) & = & \emptyset \\
\mathit{flow}([\mathtt{skip}]^\ell) & = & \emptyset \\
\mathit{flow}(S_1; S_2) & = & \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \\
 & & \cup\{(\ell, \mathit{init}(S_2)) \mid \ell \in \mathit{final}(S_1)\} \\
\mathit{flow}(\mathtt{if}\ [b]^\ell\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2) & = & \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \\
 & & \cup\{(\ell, \mathit{init}(S_1)), (\ell, \mathit{init}(S_2))\} \\
\mathit{flow}(\mathtt{while}\ [b]^\ell\ \mathtt{do}\ S) & = & \mathit{flow}(S) \cup \{(\ell, \mathit{init}(S))\} \\
 & & \cup\{(\ell', \ell) \mid \ell' \in \mathit{final}(S)\}
\end{array}$$

Thus $\mathit{labels}(S)$ and $\mathit{flow}(S)$ will be a representation of the *flow graph* of S.

Example 2.1 Consider the following program, power, computing the x-th power of the number stored in y:

$$[\mathtt{z:=1}]^1; \mathtt{while}\ [\mathtt{x>0}]^2\ \mathtt{do}\ ([\mathtt{z:=z*y}]^3; [\mathtt{x:=x-1}]^4)$$

We have $\mathit{init}(\mathsf{power}) = 1$, $\mathit{final}(\mathsf{power}) = \{2\}$ and $\mathit{labels}(\mathsf{power}) = \{1, 2, 3, 4\}$. The function *flow* produces the following set

$$\{(1,2), (2,3), (3,4), (4,2)\}$$

which corresponds to the flow graph in Figure 2.1. ■

The function *flow* is used in the formulation of *forward analyses*. Clearly $\mathit{init}(S)$ is the (unique) entry node for the flow graph with nodes $\mathit{labels}(S)$ and edges $\mathit{flow}(S)$. Also

$$\mathit{labels}(S) = \{\mathit{init}(S)\} \cup \{\ell \mid (\ell, \ell') \in \mathit{flow}(S)\} \cup \{\ell' \mid (\ell, \ell') \in \mathit{flow}(S)\}$$

and for composite statements (meaning those not simply of the form $[B]^\ell$) the equation remains true when removing the $\{\mathit{init}(S)\}$ component.

Figure 2.1: Flow graph for the power program.

In order to formulate *backward analyses* we require a function that computes *reverse flows*:

$$\mathit{flow}^R : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab} \times \mathbf{Lab})$$

It is defined by:

$$\mathit{flow}^R(S) = \{(\ell, \ell') \mid (\ell', \ell) \in \mathit{flow}(S)\}$$

Example 2.2 For the power program of Example 2.1, flow^R produces

$$\{(2,1), (2,4), (3,2), (4,3)\}$$

which corresponds to a modified version of the flow graph in Figure 2.1 where the direction of the arcs has been reversed. ■

In case $\mathit{final}(S)$ contains just one element that will be the unique entry node for the flow graph with nodes $\mathit{labels}(S)$ and edges $\mathit{flow}^R(S)$. Also

$$\mathit{labels}(S) = \mathit{final}(S) \cup \{\ell \mid (\ell, \ell') \in \mathit{flow}^R(S)\} \cup \{\ell' \mid (\ell, \ell') \in \mathit{flow}^R(S)\}$$

and for composite statements the equation remains true when removing the $\mathit{final}(S)$ component.

The program of interest. We will use the notation $S_\star$ to represent the program that we are analysing (the "top-level" statement), $\mathbf{Lab}_\star$ to represent the labels ($\mathit{labels}(S_\star)$) appearing in $S_\star$, $\mathbf{Var}_\star$ to represent the variables ($FV(S_\star)$) appearing in $S_\star$, $\mathbf{Blocks}_\star$ to represent the elementary blocks

($blocks(S_\star)$) occurring in $S_\star$, and $\mathbf{AExp}_\star$ to represent the set of *non-trivial* arithmetic subexpressions in $S_\star$; an expression is trivial if it is a single variable or constant. We will also write $\mathbf{AExp}(a)$ and $\mathbf{AExp}(b)$ to refer to the set of non-trivial arithmetic subexpressions of a given arithmetic, respectively boolean, expression.

To simplify the presentation of the analyses, and to follow the traditions of the literature, we shall frequently assume that the program $S_\star$ has *isolated entries*; this means that:

$$\forall \ell \in \mathbf{Lab} : (\ell, init(S_\star)) \notin flow(S_\star)$$

This is the case whenever $S_\star$ does not start with a `while`-loop. Similarly, we shall frequently assume that the program $S_\star$ has *isolated exits*; this means that:

$$\forall \ell_1 \in final(S_\star)\ \forall \ell_2 \in \mathbf{Lab} : (\ell_1, \ell_2) \notin flow(S_\star)$$

A statement, S, is *label consistent* if and only if:

$$[B_1]^\ell, [B_2]^\ell \in blocks(S) \text{ implies } B_1 = B_2$$

Clearly, if all blocks in S are uniquely labelled (meaning that each label occurs only once), then S is label consistent. When S is label consistent the clause "where $[B]^\ell \in blocks(S)$" is unambiguous in defining a partial function from labels to elementary blocks; we shall then say that ℓ *labels* the block B. We shall exploit this when defining the example analyses below.

Example 2.3 The power program of Example 2.1 has isolated entries but not isolated exits. It is clearly label consistent as well as uniquely labelled. ■

2.1.1 Available Expressions Analysis

The *Available Expressions Analysis* will determine:

> For each program point, which expressions *must* have already been computed, and not later modified, on all paths to the program point.

This information can be used to avoid the re-computation of an expression. For clarity, we will concentrate on arithmetic expressions.

Example 2.4 Consider the following program:

$$[\texttt{x:=a+b}]^1; [\texttt{y:=a*b}]^2; \texttt{while } [\texttt{y>a+b}]^3 \texttt{ do } ([\texttt{a:=a+1}]^4; [\texttt{x:=a+b}]^5)$$

It should be clear that the expression `a+b` is available every time execution reaches the test (label 3) in the loop; as a consequence, the expression need not be recomputed. ■

kill and *gen* functions

$$
\begin{array}{rcl}
kill_{\mathsf{AE}}([x := a]^\ell) & = & \{a' \in \mathbf{AExp}_\star \mid x \in FV(a')\} \\
kill_{\mathsf{AE}}([\mathtt{skip}]^\ell) & = & \emptyset \\
kill_{\mathsf{AE}}([b]^\ell) & = & \emptyset \\
gen_{\mathsf{AE}}([x := a]^\ell) & = & \{a' \in \mathbf{AExp}(a) \mid x \notin FV(a')\} \\
gen_{\mathsf{AE}}([\mathtt{skip}]^\ell) & = & \emptyset \\
gen_{\mathsf{AE}}([b]^\ell) & = & \mathbf{AExp}(b)
\end{array}
$$

data flow equations: $\mathsf{AE}^=$

$$
\begin{array}{rcll}
\mathsf{AE}_{entry}(\ell) & = & \begin{cases} \emptyset & \text{if } \ell = init(S_\star) \\ \bigcap\{\mathsf{AE}_{exit}(\ell') \mid (\ell', \ell) \in flow(S_\star)\} & \text{otherwise} \end{cases} & \\
\mathsf{AE}_{exit}(\ell) & = & (\mathsf{AE}_{entry}(\ell) \backslash kill_{\mathsf{AE}}(B^\ell)) \cup gen_{\mathsf{AE}}(B^\ell) & \\
 & & \text{where } B^\ell \in blocks(S_\star) &
\end{array}
$$

Table 2.1: Available Expressions Analysis.

The analysis is defined in Table 2.1 and explained below. An expression is *killed* in a block if any of the variables used in the expression are modified in the block; we use the function

$$kill_{\mathsf{AE}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{AExp}_\star)$$

to produce the set of non-trivial expressions killed in the block. Test and `skip` blocks do not kill any expressions and assignments kill any expression that uses the variable that appears in the left hand side of the assignment. Note that in the clause for $[x{:=}a]^\ell$ we have used the notation $a' \in \mathbf{AExp}_\star$ to denote the fact that a' is a non-trivial arithmetic expression appearing in the program.

A *generated* expression is an expression that is evaluated in the block and where none of the variables used in the expression are later modified in the block. The set of non-trivial generated expressions is produced by the function:

$$gen_{\mathsf{AE}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{AExp}_\star)$$

The analysis itself is now defined by the functions AE_{entry} and AE_{exit} that each map labels to sets of expressions:

$$\mathsf{AE}_{entry}, \mathsf{AE}_{exit} : \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{AExp}_\star)$$

For a label consistent program $S_\star$ (with isolated entries) the functions can be defined as in Table 2.1.

Figure 2.2: A schematic flow graph.

The analysis is a *forward analysis* and, as we shall see, we are interested in the *largest* sets satisfying the equation for AE_{entry} – an expression will be considered available if no path kills it. No expression is available at the start of the program. Subsequently, the expressions that are available at the entry to a block are any expressions that are available at *all* of the exits from blocks that flow to the block; if there are no such blocks the formula evaluates to $\mathbf{AExp}_\star$. Given a set of expressions that are available at the entry, the expressions available at the exit of the block are computed by removing killed expressions and adding any new expression generated by the block.

To see why we require the largest solution, consider Figure 2.2 which shows the flow graph for a program in a schematic way. Such a flow graph might correspond to the following program:

$$[\mathtt{z:=x+y}]^{\ell};\mathtt{while}\ [\mathtt{true}]^{\ell'}\ \mathtt{do}\ [\mathtt{skip}]^{\ell''}$$

The set of expressions generated by the first assignment is $\{\mathtt{x+y}\}$; the other blocks do not generate expressions and no block kills any expressions. The equations for AE_{entry} and AE_{exit} are as follows:

$$\begin{aligned}
\mathsf{AE}_{entry}(\ell) &= \emptyset \\
\mathsf{AE}_{entry}(\ell') &= \mathsf{AE}_{exit}(\ell) \cap \mathsf{AE}_{exit}(\ell'') \\
\mathsf{AE}_{entry}(\ell'') &= \mathsf{AE}_{exit}(\ell') \\
\mathsf{AE}_{exit}(\ell) &= \mathsf{AE}_{entry}(\ell) \cup \{\mathtt{x+y}\} \\
\mathsf{AE}_{exit}(\ell') &= \mathsf{AE}_{entry}(\ell') \\
\mathsf{AE}_{exit}(\ell'') &= \mathsf{AE}_{entry}(\ell'')
\end{aligned}$$

After some simplification, we find that:

$$\mathsf{AE}_{entry}(\ell') = \{\texttt{x+y}\} \cap \mathsf{AE}_{entry}(\ell')$$

There are two solutions to this equation: {x+y} and $\emptyset$. Consideration of the example and the definition of available expressions shows that the most informative solution is {x+y} – the expression is available every time we enter ℓ'. Thus we require the *largest* solution to the equations.

Example 2.5 For the program

$$[\texttt{x:=a+b}]^1; [\texttt{y:=a*b}]^2; \texttt{while } [\texttt{y>a+b}]^3 \texttt{ do } ([\texttt{a:=a+1}]^4; [\texttt{x:=a+b}]^5)$$

of Example 2.4, $kill_{\mathsf{AE}}$ and gen_{AE} are defined as follows:

ℓ	$kill_{\mathsf{AE}}(\ell)$	$gen_{\mathsf{AE}}(\ell)$
1	$\emptyset$	{a+b}
2	$\emptyset$	{a*b}
3	$\emptyset$	{a+b}
4	{a+b, a*b, a+1}	$\emptyset$
5	$\emptyset$	{a+b}

We get the following equations:

$$\begin{aligned}
\mathsf{AE}_{entry}(1) &= \emptyset \\
\mathsf{AE}_{entry}(2) &= \mathsf{AE}_{exit}(1) \\
\mathsf{AE}_{entry}(3) &= \mathsf{AE}_{exit}(2) \cap \mathsf{AE}_{exit}(5) \\
\mathsf{AE}_{entry}(4) &= \mathsf{AE}_{exit}(3) \\
\mathsf{AE}_{entry}(5) &= \mathsf{AE}_{exit}(4)
\end{aligned}$$

$$\begin{aligned}
\mathsf{AE}_{exit}(1) &= \mathsf{AE}_{entry}(1) \cup \{\texttt{a+b}\} \\
\mathsf{AE}_{exit}(2) &= \mathsf{AE}_{entry}(2) \cup \{\texttt{a*b}\} \\
\mathsf{AE}_{exit}(3) &= \mathsf{AE}_{entry}(3) \cup \{\texttt{a+b}\} \\
\mathsf{AE}_{exit}(4) &= \mathsf{AE}_{entry}(4) \backslash \{\texttt{a+b}, \texttt{a*b}, \texttt{a+1}\} \\
\mathsf{AE}_{exit}(5) &= \mathsf{AE}_{entry}(5) \cup \{\texttt{a+b}\}
\end{aligned}$$

Using an analogue of the Chaotic Iteration discussed in Chapter 1 (starting with $\mathbf{AExp}_\star$ rather than $\emptyset$) we can compute the following solution:

ℓ	$\mathsf{AE}_{entry}(\ell)$	$\mathsf{AE}_{exit}(\ell)$
1	$\emptyset$	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	$\emptyset$
5	$\emptyset$	{a+b}

Note that, even though a is redefined in the loop, the expression a+b is re-evaluated in the loop and so it is always available on entry to the loop. On the other hand, a*b is available on the first entry to the loop but is killed before the next iteration. ■

2.1.2 Reaching Definitions Analysis

As mentioned in Chapter 1, the *Reaching Definitions Analysis* should more properly be called reaching assignments but we will use the traditional name. This analysis is analogous to the previous one except that we are interested in:

> For each program point, which assignments *may* have been made and not overwritten, when program execution reaches this point along some path.

A main application of Reaching Definitions Analysis is in the construction of direct links between blocks that produce values and blocks that use them; we shall return to this in Subsection 2.1.5.

Example 2.6 Consider the following program:

$$[\texttt{x:=5}]^1; [\texttt{y:=1}]^2; \texttt{while } [\texttt{x>1}]^3 \texttt{ do } ([\texttt{y:=x*y}]^4; [\texttt{x:=x-1}]^5)$$

All of the assignments reach the entry of 4 (the assignments labelled 1 and 2 reach there on the first iteration); only the assignments labelled 1, 4 and 5 reach the entry of 5. ■

The analysis is specified in Table 2.2. The function

$$\mathit{kill}_{\mathsf{RD}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star^?)$$

produces the set of pairs of variables and labels of assignments that are destroyed by the block. An assignment is destroyed if the block assigns a new value to the variable, i.e. the left hand side of the assignment. To deal with uninitialised variables we shall, as in Chapter 1, use the special label "?" and we set $\mathbf{Lab}_\star^? = \mathbf{Lab}_\star \cup \{?\}$.

The function

$$\mathit{gen}_{\mathsf{RD}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star^?)$$

produces the set of pairs of variables and labels of assignments generated by the block; only assignments generate definitions.

The analysis itself is now defined by the pair of functions RD_{entry} and RD_{exit} mapping labels to sets of pairs of variables and labels (of assignment blocks):

$$\mathsf{RD}_{entry}, \mathsf{RD}_{exit} : \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star^?)$$

kill and *gen* functions

$$
\begin{aligned}
kill_{\mathsf{RD}}([x := a]^{\ell}) &= \{(x, ?)\} \\
&\quad \cup \{(x, \ell') \mid B^{\ell'} \text{ is an assignment to } x \text{ in } S_\star\} \\
kill_{\mathsf{RD}}([\mathtt{skip}]^{\ell}) &= \emptyset \\
kill_{\mathsf{RD}}([b]^{\ell}) &= \emptyset \\
gen_{\mathsf{RD}}([x := a]^{\ell}) &= \{(x, \ell)\} \\
gen_{\mathsf{RD}}([\mathtt{skip}]^{\ell}) &= \emptyset \\
gen_{\mathsf{RD}}([b]^{\ell}) &= \emptyset
\end{aligned}
$$

data flow equations: $\mathsf{RD}^{=}$

$$
\begin{aligned}
\mathsf{RD}_{entry}(\ell) &= \begin{cases} \{(x, ?) \mid x \in FV(S_\star)\} & \text{if } \ell = init(S_\star) \\ \bigcup\{\mathsf{RD}_{exit}(\ell') \mid (\ell', \ell) \in flow(S_\star)\} & \text{otherwise} \end{cases} \\
\mathsf{RD}_{exit}(\ell) &= (\mathsf{RD}_{entry}(\ell) \backslash kill_{\mathsf{RD}}(B^{\ell})) \cup gen_{\mathsf{RD}}(B^{\ell}) \\
&\quad \text{where } B^{\ell} \in blocks(S_\star)
\end{aligned}
$$

Table 2.2: Reaching Definitions Analysis.

For a label consistent program $S_\star$ (with isolated entries) the functions are defined as in Table 2.2.

Similar to the previous example, this is a *forward analysis* but, as we shall see, we are interested in the *smallest* sets satisfying the equation for RD_{entry}. An assignment reaches the entry of a block if it reaches the exit of *any* of the blocks which precede it; if there are none the formula evaluates to $\emptyset$. The computation of the set of assignments reaching the exit of a block is analogous to the Available Expressions Analysis.

We motivate the requirement for the smallest solution by consideration of the program $[\mathtt{z:=x+y}]^{\ell}; \mathtt{while}\ [\mathtt{true}]^{\ell'}\ \mathtt{do}\ [\mathtt{skip}]^{\ell''}$ corresponding to Figure 2.2 again. The equations for RD_{entry} and RD_{exit} are as follows:

$$
\begin{aligned}
\mathsf{RD}_{entry}(\ell) &= \{(\mathtt{x}, ?), (\mathtt{y}, ?), (\mathtt{z}, ?)\} \\
\mathsf{RD}_{entry}(\ell') &= \mathsf{RD}_{exit}(\ell) \cup \mathsf{RD}_{exit}(\ell'') \\
\mathsf{RD}_{entry}(\ell'') &= \mathsf{RD}_{exit}(\ell') \\
\mathsf{RD}_{exit}(\ell) &= (\mathsf{RD}_{entry}(\ell) \setminus \{(\mathtt{z}, ?)\}) \cup \{(\mathtt{z}, \ell)\} \\
\mathsf{RD}_{exit}(\ell') &= \mathsf{RD}_{entry}(\ell') \\
\mathsf{RD}_{exit}(\ell'') &= \mathsf{RD}_{entry}(\ell'')
\end{aligned}
$$

Once again, we concentrate on the entry of the block labelled ℓ', $\mathsf{RD}_{entry}(\ell')$; after some simplification we get

$$\mathsf{RD}_{entry}(\ell') = \{(\mathtt{x}, ?), (\mathtt{y}, ?), (\mathtt{z}, \ell)\} \cup \mathsf{RD}_{entry}(\ell')$$

but this equation has many solutions: we can take $\mathsf{RD}_{entry}(\ell')$ to be any superset of $\{(\mathtt{x}, ?), (\mathtt{y}, ?), (\mathtt{z}, \ell)\}$. However, since ℓ' does not generate any new definitions, the most precise solution is $\{(\mathtt{x}, ?), (\mathtt{y}, ?), (\mathtt{z}, \ell)\}$ – we require the *smallest* solution to the equations.

Sometimes, when the Reaching Definitions Analysis is presented in the literature, one has $\mathsf{RD}_{entry}(init(S_\star)) = \emptyset$ rather than $\mathsf{RD}_{entry}(init(S_\star)) = \{(x, ?) \mid x \in FV(S_\star)\}$. This is correct only for programs that always assign variables before their first use; incorrect optimisations may result if this is not the case. The advantage of our formulation, as will emerge from Mini Project 2.2, is that it is always semantically sound.

Example 2.7 The following table summarises the assignments killed and generated by each of the blocks in the program

$$[\mathtt{x:=5}]^1; [\mathtt{y:=1}]^2; \mathtt{while}\ [\mathtt{x>1}]^3\ \mathtt{do}\ ([\mathtt{y:=x*y}]^4; [\mathtt{x:=x-1}]^5)$$

of Example 2.6:

ℓ	$kill_{\mathsf{RD}}(\ell)$	$gen_{\mathsf{RD}}(\ell)$
1	$\{(\mathtt{x}, ?), (\mathtt{x}, 1), (\mathtt{x}, 5)\}$	$\{(\mathtt{x}, 1)\}$
2	$\{(\mathtt{y}, ?), (\mathtt{y}, 2), (\mathtt{y}, 4)\}$	$\{(\mathtt{y}, 2)\}$
3	$\emptyset$	$\emptyset$
4	$\{(\mathtt{y}, ?), (\mathtt{y}, 2), (\mathtt{y}, 4)\}$	$\{(\mathtt{y}, 4)\}$
5	$\{(\mathtt{x}, ?), (\mathtt{x}, 1), (\mathtt{x}, 5)\}$	$\{(\mathtt{x}, 5)\}$

The analysis gives rise to the following equations:

$$\begin{aligned}
\mathsf{RD}_{entry}(1) &= \{(\mathtt{x}, ?), (\mathtt{y}, ?)\} \\
\mathsf{RD}_{entry}(2) &= \mathsf{RD}_{exit}(1) \\
\mathsf{RD}_{entry}(3) &= \mathsf{RD}_{exit}(2) \cup \mathsf{RD}_{exit}(5) \\
\mathsf{RD}_{entry}(4) &= \mathsf{RD}_{exit}(3) \\
\mathsf{RD}_{entry}(5) &= \mathsf{RD}_{exit}(4)
\end{aligned}$$

$$\begin{aligned}
\mathsf{RD}_{exit}(1) &= (\mathsf{RD}_{entry}(1) \backslash \{(\mathtt{x}, ?), (\mathtt{x}, 1), (\mathtt{x}, 5)\}) \cup \{(\mathtt{x}, 1)\} \\
\mathsf{RD}_{exit}(2) &= (\mathsf{RD}_{entry}(2) \backslash \{(\mathtt{y}, ?), (\mathtt{y}, 2), (\mathtt{y}, 4)\}) \cup \{(\mathtt{y}, 2)\} \\
\mathsf{RD}_{exit}(3) &= \mathsf{RD}_{entry}(3) \\
\mathsf{RD}_{exit}(4) &= (\mathsf{RD}_{entry}(4) \backslash \{(\mathtt{y}, ?), (\mathtt{y}, 2), (\mathtt{y}, 4)\}) \cup \{(\mathtt{y}, 4)\} \\
\mathsf{RD}_{exit}(5) &= (\mathsf{RD}_{entry}(5) \backslash \{(\mathtt{x}, ?), (\mathtt{x}, 1), (\mathtt{x}, 5)\}) \cup \{(\mathtt{x}, 5)\}
\end{aligned}$$

Using Chaotic Iteration we may compute the solution:

ℓ	$\mathsf{RD}_{entry}(\ell)$	$\mathsf{RD}_{exit}(\ell)$
1	$\{(x,?),(y,?)\}$	$\{(y,?),(x,1)\}$
2	$\{(y,?),(x,1)\}$	$\{(x,1),(y,2)\}$
3	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$
4	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(y,4),(x,5)\}$
5	$\{(x,1),(y,4),(x,5)\}$	$\{(y,4),(x,5)\}$

■

2.1.3 Very Busy Expressions Analysis

An expression is *very busy* at the exit from a label if, no matter what path is taken from the label, the expression must always be used before any of the variables occurring in it are redefined. The aim of the *Very Busy Expressions Analysis* is to determine:

> For each program point, which expressions *must* be very busy at the exit from the point.

A possible optimisation based on this information is to evaluate the expression at the block and store its value for later use; this optimisation is sometimes called *hoisting* the expression.

Example 2.8 Consider the program:

$$\texttt{if } [\texttt{a>b}]^1 \texttt{ then } ([\texttt{x:=b-a}]^2; [\texttt{y:=a-b}]^3) \texttt{ else } ([\texttt{y:=b-a}]^4; [\texttt{x:=a-b}]^5)$$

The expressions `a-b` and `b-a` are both very busy at the start of the conditional; they can be hoisted to the start of the conditional resulting in a space saving in the size of the code generated for this program. ■

The analysis is specified in Table 2.3. We have already defined the notion of an expression being killed when we presented the Available Expressions Analysis; we use an equivalent function here:

$$kill_{\mathsf{VB}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{AExp}_\star)$$

By analogy with the previous analyses, we also need to define how a block generates additional very busy expressions. For this we use:

$$gen_{\mathsf{VB}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{AExp}_\star)$$

All of the expressions that appear in a block are very busy at the entry to the block (unlike what was the case for Available Expressions).

kill and *gen* functions

$$
\begin{array}{rcl}
\mathit{kill}_{\mathsf{VB}}([x := a]^{\ell}) & = & \{a' \in \mathbf{AExp}_{\star} \mid x \in FV(a')\} \\
\mathit{kill}_{\mathsf{VB}}([\mathtt{skip}]^{\ell}) & = & \emptyset \\
\mathit{kill}_{\mathsf{VB}}([b]^{\ell}) & = & \emptyset \\
\mathit{gen}_{\mathsf{VB}}([\mathtt{x} := a]^{\ell}) & = & \mathbf{AExp}(a) \\
\mathit{gen}_{\mathsf{VB}}([\mathtt{skip}]^{\ell}) & = & \emptyset \\
\mathit{gen}_{\mathsf{VB}}([b]^{\ell}) & = & \mathbf{AExp}(b)
\end{array}
$$

data flow equations: $\mathsf{VB}^{=}$

$$
\begin{array}{rcl}
\mathsf{VB}_{\mathit{exit}}(\ell) & = & \begin{cases} \emptyset & \text{if } \ell \in \mathit{final}(S_{\star}) \\ \bigcap\{\mathsf{VB}_{\mathit{entry}}(\ell') \mid (\ell', \ell) \in \mathit{flow}^{R}(S_{\star})\} & \text{otherwise} \end{cases} \\
\mathsf{VB}_{\mathit{entry}}(\ell) & = & (\mathsf{VB}_{\mathit{exit}}(\ell) \backslash \mathit{kill}_{\mathsf{VB}}(B^{\ell})) \cup \mathit{gen}_{\mathsf{VB}}(B^{\ell}) \\
 & & \text{where } B^{\ell} \in \mathit{blocks}(S_{\star})
\end{array}
$$

Table 2.3: Very Busy Expressions Analysis.

The analysis itself is defined by the pair of functions $\mathsf{VB}_{\mathit{entry}}$ and $\mathsf{VB}_{\mathit{exit}}$ mapping labels to sets of expressions:

$$\mathsf{VB}_{\mathit{entry}}, \mathsf{VB}_{\mathit{exit}} : \mathbf{Lab}_{\star} \rightarrow \mathcal{P}(\mathbf{AExp}_{\star})$$

For a label consistent program $S_{\star}$ (with isolated exits) they are defined as in Table 2.3.

The analysis is a *backward analysis* and, as we shall see, we are interested in the *largest* sets satisfying the equation for $\mathsf{VB}_{\mathit{exit}}$. The functions propagate information *against* the flow of the program: an expression is very busy at the exit from a block if it is very busy at the entry to *every* block that follows; if there are none the formula evaluates to $\mathbf{AExp}_{\star}$. However, no expressions are very busy at the exit from any final block.

To motivate the fact that we require the largest set, we consider the situation where we have a flow graph as shown in Figure 2.3; this flow graph might correspond to the program:

$$(\mathtt{while}\ [\mathtt{x>1}]^{\ell}\ \mathtt{do}\ [\mathtt{skip}]^{\ell'}); [\mathtt{x:=x+1}]^{\ell''}$$

The equations for this program are

$$
\begin{array}{rcl}
\mathsf{VB}_{\mathit{entry}}(\ell) & = & \mathsf{VB}_{\mathit{exit}}(\ell) \\
\mathsf{VB}_{\mathit{entry}}(\ell') & = & \mathsf{VB}_{\mathit{exit}}(\ell') \\
\mathsf{VB}_{\mathit{entry}}(\ell'') & = & \{\mathtt{x+1}\}
\end{array}
$$

Figure 2.3: A schematic flow graph (in reverse).

$$\begin{aligned}
\mathsf{VB}_{exit}(\ell) &= \mathsf{VB}_{entry}(\ell') \cap \mathsf{VB}_{entry}(\ell'') \\
\mathsf{VB}_{exit}(\ell') &= \mathsf{VB}_{entry}(\ell) \\
\mathsf{VB}_{exit}(\ell'') &= \emptyset
\end{aligned}$$

and, for the exit conditions of ℓ, we calculate:

$$\mathsf{VB}_{exit}(\ell) = \mathsf{VB}_{exit}(\ell) \cap \{\texttt{x+1}\}$$

Any subset of $\{\texttt{x+1}\}$ is a solution but $\{\texttt{x+1}\}$ is the most informative. Hence we want the *largest* solution to the equations.

Example 2.9 To analyse the program

$$\texttt{if } [\texttt{a>b}]^1 \texttt{ then } ([\texttt{x:=b-a}]^2; [\texttt{y:=a-b}]^3) \texttt{ else } ([\texttt{y:=b-a}]^4; [\texttt{x:=a-b}]^5)$$

of Example 2.8, we calculate the following killed and generated sets:

ℓ	$kill_{\mathsf{VB}}(\ell)$	$gen_{\mathsf{VB}}(\ell)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{\texttt{b-a}\}$
3	$\emptyset$	$\{\texttt{a-b}\}$
4	$\emptyset$	$\{\texttt{b-a}\}$
5	$\emptyset$	$\{\texttt{a-b}\}$

We get the following equations:

$$\begin{aligned}
\mathsf{VB}_{entry}(1) &= \mathsf{VB}_{exit}(1) \\
\mathsf{VB}_{entry}(2) &= \mathsf{VB}_{exit}(2) \cup \{\texttt{b-a}\} \\
\mathsf{VB}_{entry}(3) &= \{\texttt{a-b}\} \\
\mathsf{VB}_{entry}(4) &= \mathsf{VB}_{exit}(4) \cup \{\texttt{b-a}\} \\
\mathsf{VB}_{entry}(5) &= \{\texttt{a-b}\}
\end{aligned}$$

$$
\begin{aligned}
\mathsf{VB}_{exit}(1) &= \mathsf{VB}_{entry}(2) \cap \mathsf{VB}_{entry}(4) \\
\mathsf{VB}_{exit}(2) &= \mathsf{VB}_{entry}(3) \\
\mathsf{VB}_{exit}(3) &= \emptyset \\
\mathsf{VB}_{exit}(4) &= \mathsf{VB}_{entry}(5) \\
\mathsf{VB}_{exit}(5) &= \emptyset
\end{aligned}
$$

We can then use an analogue of Chaotic Iteration (starting with $\mathbf{AExp}_\star$ rather than $\emptyset$) to compute:

ℓ	$\mathsf{VB}_{entry}(\ell)$	$\mathsf{VB}_{exit}(\ell)$
1	{a-b, b-a}	{a-b, b-a}
2	{a-b, b-a}	{a-b}
3	{a-b}	$\emptyset$
4	{a-b, b-a}	{a-b}
5	{a-b}	$\emptyset$

■

2.1.4 Live Variables Analysis

A variable is *live* at the exit from a label if there exists a path from the label to a use of the variable that does not re-define the variable. The *Live Variables Analysis* will determine:

> For each program point, which variables *may* be live at the exit from the point.

We shall take the view that no variables are live at the end of the program; for some applications it might be better to assume that all variables are live at the end of the program

Live Variables analysis may be used as the basis for *Dead Code Elimination.* If the variable is not live at the exit from a label then, if the elementary block is an assignment to the variable, the elementary block can be eliminated.

Example 2.10 Consider the following expression:

$$[\mathtt{x:=2}]^1; [\mathtt{y:=4}]^2; [\mathtt{x:=1}]^3; (\mathtt{if}\ [\mathtt{y>x}]^4\ \mathtt{then}\ [\mathtt{z:=y}]^5\ \mathtt{else}\ [\mathtt{z:=y*y}]^6); [\mathtt{x:=z}]^7$$

The variable `x` is not live at the exit from label 1; the first assignment of the program is redundant. Both `x` and `y` are live at the exit from label 3. ■

The analysis is defined in Table 2.4. The variable that appears on the left hand side of an assignment is killed by the assignment; tests and `skip` statements do not kill variables. This is expressed by the function:

$$kill_{\mathsf{LV}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{Var}_\star)$$

kill and *gen* functions

$$
\begin{array}{rcl}
\mathit{kill}_{\mathsf{LV}}([x := a]^{\ell}) & = & \{x\} \\
\mathit{kill}_{\mathsf{LV}}([\mathtt{skip}]^{\ell}) & = & \emptyset \\
\mathit{kill}_{\mathsf{LV}}([b]^{\ell}) & = & \emptyset \\
\mathit{gen}_{\mathsf{LV}}([x := a]^{\ell}) & = & FV(a) \\
\mathit{gen}_{\mathsf{LV}}([\mathtt{skip}]^{\ell}) & = & \emptyset \\
\mathit{gen}_{\mathsf{LV}}([b]^{\ell}) & = & FV(b)
\end{array}
$$

data flow equations: $\mathsf{LV}^{=}$

$$
\begin{array}{rcll}
\mathsf{LV}_{\mathit{exit}}(\ell) & = & \begin{cases} \emptyset & \text{if } \ell \in \mathit{final}(S_\star) \\ \bigcup\{\mathsf{LV}_{\mathit{entry}}(\ell') \mid (\ell', \ell) \in \mathit{flow}^R(S_\star)\} & \text{otherwise} \end{cases} \\
\mathsf{LV}_{\mathit{entry}}(\ell) & = & (\mathsf{LV}_{\mathit{exit}}(\ell) \backslash \mathit{kill}_{\mathsf{LV}}(B^{\ell})) \cup \mathit{gen}_{\mathsf{LV}}(B^{\ell}) & \\
 & & \text{where } B^{\ell} \in \mathit{blocks}(S_\star) &
\end{array}
$$

Table 2.4: Live Variables Analysis.

The function

$$\mathit{gen}_{\mathsf{LV}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{Var}_\star)$$

produces the set of variables that appear in the block.

The analysis itself is defined by the pair of functions $\mathsf{LV}_{\mathit{entry}}$ and $\mathsf{LV}_{\mathit{exit}}$ mapping labels to sets of variables:

$$\mathsf{LV}_{\mathit{exit}}, \mathsf{LV}_{\mathit{entry}} : \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{Var}_\star)$$

For a label consistent program $S_\star$ (with isolated exits) they can be defined as in Table 2.4. The equation for $\mathsf{LV}_{\mathit{exit}}(\ell)$ includes a variable in the set of live variables (at the exit from ℓ) if it is live at the entry to any of the blocks that follow ℓ; if there are none then the formula evaluates to $\emptyset$.

The analysis is a *backward analysis* and, as we shall see, we are interested in the *smallest* sets satisfying the equation for $\mathsf{LV}_{\mathit{exit}}$. To see why we require the smallest set, consider once again the program

$$(\mathtt{while}\ [\mathtt{x>1}]^{\ell}\ \mathtt{do}\ [\mathtt{skip}]^{\ell'}); [\mathtt{x:=x+1}]^{\ell''}$$

corresponding to the flow graph in Figure 2.3. The equations for the program are:

$$
\begin{array}{rcl}
\mathsf{LV}_{\mathit{entry}}(\ell) & = & \mathsf{LV}_{\mathit{exit}}(\ell) \cup \{\mathtt{x}\} \\
\mathsf{LV}_{\mathit{entry}}(\ell') & = & \mathsf{LV}_{\mathit{exit}}(\ell')
\end{array}
$$

$$
\begin{aligned}
\mathsf{LV}_{entry}(\ell'') &= \{\mathtt{x}\} \\
\mathsf{LV}_{exit}(\ell) &= \mathsf{LV}_{entry}(\ell') \cup \mathsf{LV}_{entry}(\ell'') \\
\mathsf{LV}_{exit}(\ell') &= \mathsf{LV}_{entry}(\ell) \\
\mathsf{LV}_{exit}(\ell'') &= \emptyset
\end{aligned}
$$

Suppose that we are interested in $\mathsf{LV}_{exit}(\ell)$; after some calculation we get:

$$\mathsf{LV}_{exit}(\ell) = \mathsf{LV}_{exit}(\ell) \cup \{\mathtt{x}\}$$

Any superset of $\{\mathtt{x}\}$ is a solution. Optimisations based on this analysis are based on "dead" variables – the smaller the set of live variables, the more optimisations are possible. Hence we shall be interested in the *smallest* solution $\{\mathtt{x}\}$ to the equations. Correctness of the analysis will be established in Section 2.2.

Example 2.11 Returning to the program

$$[\mathtt{x:=2}]^1; [\mathtt{y:=4}]^2; [\mathtt{x:=1}]^3; (\mathtt{if}\ [\mathtt{y>x}]^4\ \mathtt{then}\ [\mathtt{z:=y}]^5\ \mathtt{else}\ [\mathtt{z:=y*y}]^6); [\mathtt{x:=z}]^7$$

of Example 2.10, we can compute $kill_{\mathsf{LV}}$ and gen_{LV} as:

ℓ	$kill_{\mathsf{LV}}(\ell)$	$gen_{\mathsf{LV}}(\ell)$
1	$\{\mathtt{x}\}$	$\emptyset$
2	$\{\mathtt{y}\}$	$\emptyset$
3	$\{\mathtt{x}\}$	$\emptyset$
4	$\emptyset$	$\{\mathtt{x},\mathtt{y}\}$
5	$\{\mathtt{z}\}$	$\{\mathtt{y}\}$
6	$\{\mathtt{z}\}$	$\{\mathtt{y}\}$
7	$\{\mathtt{x}\}$	$\{\mathtt{z}\}$

We get the following equations:

$$
\begin{aligned}
\mathsf{LV}_{entry}(1) &= \mathsf{LV}_{exit}(1)\backslash\{\mathtt{x}\} \\
\mathsf{LV}_{entry}(2) &= \mathsf{LV}_{exit}(2)\backslash\{\mathtt{y}\} \\
\mathsf{LV}_{entry}(3) &= \mathsf{LV}_{exit}(3)\backslash\{\mathtt{x}\} \\
\mathsf{LV}_{entry}(4) &= \mathsf{LV}_{exit}(4) \cup \{\mathtt{x},\mathtt{y}\} \\
\mathsf{LV}_{entry}(5) &= (\mathsf{LV}_{exit}(5)\backslash\{\mathtt{z}\}) \cup \{\mathtt{y}\} \\
\mathsf{LV}_{entry}(6) &= (\mathsf{LV}_{exit}(6)\backslash\{\mathtt{z}\}) \cup \{\mathtt{y}\} \\
\mathsf{LV}_{entry}(7) &= \{\mathtt{z}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{LV}_{exit}(1) &= \mathsf{LV}_{entry}(2) \\
\mathsf{LV}_{exit}(2) &= \mathsf{LV}_{entry}(3) \\
\mathsf{LV}_{exit}(3) &= \mathsf{LV}_{entry}(4) \\
\mathsf{LV}_{exit}(4) &= \mathsf{LV}_{entry}(5) \cup \mathsf{LV}_{entry}(6)
\end{aligned}
$$

$$\begin{aligned}
\mathsf{LV}_{exit}(5) &= \mathsf{LV}_{entry}(7) \\
\mathsf{LV}_{exit}(6) &= \mathsf{LV}_{entry}(7) \\
\mathsf{LV}_{exit}(7) &= \emptyset
\end{aligned}$$

We can then use Chaotic Iteration to compute the solution:

ℓ	$\mathsf{LV}_{entry}(\ell)$	$\mathsf{LV}_{exit}(\ell)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{\mathtt{y}\}$
3	$\{\mathtt{y}\}$	$\{\mathtt{x},\mathtt{y}\}$
4	$\{\mathtt{x},\mathtt{y}\}$	$\{\mathtt{y}\}$
5	$\{\mathtt{y}\}$	$\{\mathtt{z}\}$
6	$\{\mathtt{y}\}$	$\{\mathtt{z}\}$
7	$\{\mathtt{z}\}$	$\emptyset$

Note that we have assumed that all variables are dead at the end of the program. Some authors assume that the variables of interest are output at the end of the program; in that case $\mathsf{LV}_{exit}(7)$ should be $\{\mathtt{x},\mathtt{y},\mathtt{z}\}$ which means that $\mathsf{LV}_{entry}(7)$, $\mathsf{LV}_{exit}(5)$ and $\mathsf{LV}_{exit}(6)$ should all be $\{\mathtt{y},\mathtt{z}\}$. ■

2.1.5 Derived Data Flow Information

It is often convenient to directly link labels of statements that produce values to the labels of statements that use them. Links that, for each use of a variable, associate all assignments that reach that use are called *Use-Definition chains* or *ud*-chains. Links that, for each assignment, associate all uses are called *Definition-Use chains* or *du*-chains.

In order to make these definitions more precise, we will use the notion of a *definition clear path* with respect to some variable. The idea is that $\ell_1, \cdots, \ell_n$ is a definition clear path for x if none of the blocks labelled $\ell_1, \cdots, \ell_n$ assigns a value to x and if ℓ_n uses x. Formally, for a label consistent program $S_\star$ we define the predicate *clear*:

$$\begin{aligned}
\mathit{clear}(x, \ell, \ell') \quad = \quad & \exists \ell_1, \cdots, \ell_n : \\
& (\ell_1 = \ell) \wedge (\ell_n = \ell') \wedge (n > 0) \wedge \\
& (\forall i \in \{1, \cdots, n-1\} : (\ell_i, \ell_{i+1}) \in \mathit{flow}(S_\star)) \wedge \\
& (\forall i \in \{1, \cdots, n-1\} : \neg \mathit{def}(x, \ell_i)) \wedge \mathit{use}(x, \ell_n)
\end{aligned}$$

Here the predicate *use* checks whether the variable is used in a block

$$\mathit{use}(x, \ell) = (\exists B : [B]^\ell \in \mathit{blocks}(S_\star) \wedge x \in \mathit{gen}_{\mathsf{LV}}([B]^\ell))$$

and the predicate *def* checks whether the variable is assigned in a block:

$$\mathit{def}(x, \ell) = (\exists B : [B]^\ell \in \mathit{blocks}(S_\star) \wedge x \in \mathit{kill}_{\mathsf{LV}}([B]^\ell))$$

Armed with these definitions, we can define the functions

$$ud, du : \mathbf{Var}_\star \times \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{Lab}_\star)$$

as follows:

$$\begin{array}{rcl} ud(x,\ell') & = & \{\ell \mid def(x,\ell) \wedge \exists \ell'' : (\ell,\ell'') \in flow(S_\star) \wedge clear(x,\ell'',\ell')\} \\ & \cup & \{? \mid clear(x, init(S_\star), \ell')\} \end{array}$$

$$du(x,\ell) = \begin{cases} \{\ell' \mid def(x,\ell) \wedge \exists \ell'' : (\ell,\ell'') \in flow(S_\star) \wedge clear(x,\ell'',\ell')\} \\ \qquad\qquad \text{if } \ell \neq ? \\ \{\ell' \mid clear(x, init(S_\star), \ell')\} \\ \qquad\qquad \text{if } \ell = ? \end{cases}$$

So $ud(x,\ell')$ will return the labels where an occurrence of x at ℓ' might have obtained its value; this may be at a label ℓ in $S_\star$ or x may be uninitialised as indicated by the occurrence of "?". And $du(x,\ell)$ will return the labels where the value assigned to x at ℓ might be used; again we distinguish between the case where x gets its value within the program and the case where it is uninitialised. It turns out that:

$$du(x,\ell) = \{\ell' \mid \ell \in ud(x,\ell')\}$$

Before showing how *ud*- and *du*-chains can be used, we illustrate the functions by a simple example.

Example 2.12 Consider the program:

$$[\texttt{x:=0}]^1; [\texttt{x:=3}]^2; (\texttt{if } [\texttt{z=x}]^3 \texttt{ then } [\texttt{z:=0}]^4 \texttt{ else } [\texttt{z:=x}]^5); [\texttt{y:=x}]^6; [\texttt{x:=y+z}]^7$$

Then we get:

$ud(x,\ell)$	x	y	z
1	$\emptyset$	$\emptyset$	$\emptyset$
2	$\emptyset$	$\emptyset$	$\emptyset$
3	$\{2\}$	$\emptyset$	$\{?\}$
4	$\emptyset$	$\emptyset$	$\emptyset$
5	$\{2\}$	$\emptyset$	$\emptyset$
6	$\{2\}$	$\emptyset$	$\emptyset$
7	$\emptyset$	$\{6\}$	$\{4,5\}$

$du(x,\ell)$	x	y	z
1	$\emptyset$	$\emptyset$	$\emptyset$
2	$\{3,5,6\}$	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$	$\{7\}$
5	$\emptyset$	$\emptyset$	$\{7\}$
6	$\emptyset$	$\{7\}$	$\emptyset$
7	$\emptyset$	$\emptyset$	$\emptyset$
?	$\emptyset$	$\emptyset$	$\{3\}$

The table for *ud* shows that the occurrence of **x** in block 3 will get its value in block 2 and the table for *du* shows that the value assigned to **x** in block 2 may be used in block 3, 5 and 6. ∎

One application of *ud*- and *du*-chains is for *Dead Code Elimination*; for the program of Example 2.12 we may remove the block labelled 1 for example

because there will be no use of the value assigned to x before it is reassigned in the next block. Another application is in *Code Motion*; in the example program the block labelled 6 can be moved to just in front of the conditional because it only uses variables assigned in earlier blocks and the conditional does not use the variable assigned in block 6.

The definitions of *ud*- and *du*-chains do not give any hints as to how to compute the chains – the definitions are not constructive. It is possible to give constructive definitions which re-use some of the functions that we have defined in the earlier examples. In order to define *ud*-chains we can use RD_{entry}, which records the assignments reaching a block and define

$$\mathsf{UD} : \mathbf{Var}_\star \times \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{Lab}_\star)$$

by:

$$\mathsf{UD}(x,\ell) = \begin{cases} \{\ell' \mid (x,\ell') \in \mathsf{RD}_{entry}(\ell)\} & \text{if } x \in gen_{\mathsf{LV}}(B^\ell) \\ \emptyset & \text{otherwise} \end{cases}$$

Similarly, we can define a function $\mathsf{DU} : \mathbf{Var}_\star \times \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{Lab}_\star)$ for *du*-chains based on the functions we have seen previously. We shall leave this to Mini Project 2.1 where we also consider the formal relationship between the UD and DU functions and the functions *ud* and *du*.

2.2 Theoretical Properties

In this section we will show that the Live Variables Analysis of Subsection 2.1.4 is indeed correct; the correctness of the Reaching Definitions Analysis is the topic of Mini Project 2.2. We shall begin by presenting a formal semantics for WHILE.

The material of this section may be skimmed through on a first reading; however, it is frequently when conducting the correctness proof that the final and subtle errors in the analysis are found and corrected! In other words, proving the semantic correctness of the analysis should not be considered a dispensable development that is merely of interest for theoreticians.

2.2.1 Structural Operational Semantics

We choose to use a (so-called small step) *Structural Operational Semantics* because it allows us to reason about intermediate stages in a program execution and it also allows us to deal with non-terminating programs.

Configurations and transitions. First define a *state* as a mapping from variables to integers:

$$\sigma \in \mathbf{State} \;=\; \mathbf{Var} \to \mathbf{Z}$$

$$\mathcal{A} : \mathbf{AExp} \to (\mathbf{State} \to \mathbf{Z})$$

$$\begin{aligned}
\mathcal{A}[\![x]\!]\sigma &= \sigma(x) \\
\mathcal{A}[\![n]\!]\sigma &= \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![a_1 \ op_a \ a_2]\!]\sigma &= \mathcal{A}[\![a_1]\!]\sigma \ \mathbf{op}_a \ \mathcal{A}[\![a_2]\!]\sigma
\end{aligned}$$

$$\mathcal{B} : \mathbf{BExp} \to (\mathbf{State} \to \mathbf{T})$$

$$\begin{aligned}
\mathcal{B}[\![\texttt{not}\ b]\!]\sigma &= \neg\mathcal{B}[\![b]\!]\sigma \\
\mathcal{B}[\![b_1 \ op_b \ b_2]\!]\sigma &= \mathcal{B}[\![b_1]\!]\sigma \ \mathbf{op}_b \ \mathcal{B}[\![b_2]\!]\sigma \\
\mathcal{B}[\![a_1 \ op_r \ a_2]\!]\sigma &= \mathcal{A}[\![a_1]\!]\sigma \ \mathbf{op}_r \ \mathcal{A}[\![a_2]\!]\sigma
\end{aligned}$$

Table 2.5: Semantics of expressions in WHILE.

A *configuration* of the semantics is either a pair consisting of a statement and a state or it is a state; a terminal configuration is a configuration that simply is a state. The *transitions* of the semantics are of the form

$$\langle S, \sigma\rangle \to \sigma' \quad \text{and} \quad \langle S, \sigma\rangle \to \langle S', \sigma'\rangle$$

and express how the configuration is changed by *one step* of computation. So in the configuration $\langle S, \sigma\rangle$ one of two things may happen:

- the execution terminates after one step and we record that by giving the resulting state σ', or
- the execution does not terminate after one step and we record that by a new configuration $\langle S', \sigma'\rangle$ where S' is the rest of the program and σ' is the updated state.

To deal with arithmetic and boolean expressions we require the semantic functions

$$\begin{aligned}
\mathcal{A} &: \ \mathbf{AExp} \to (\mathbf{State} \to \mathbf{Z}) \\
\mathcal{B} &: \ \mathbf{BExp} \to (\mathbf{State} \to \mathbf{T})
\end{aligned}$$

whose definition are given in Table 2.5. Here we assume that $\mathbf{op}_a$, $\mathbf{op}_b$ and $\mathbf{op}_r$ are the semantic counterparts of the corresponding syntax. We have also assumed the existence of $\mathcal{N} : \mathbf{Num} \to \mathbf{Z}$ which defines the semantics of numerals. For simplicity we have assumed that no errors can occur; this means that division by 0 will have to produce an integer for example. One can modify the definition so as to allow errors but this will complicate the correctness proof to be performed below. Note that the value of an expression is only affected by the variables appearing in it, that is:

$$\text{if } \forall x \in FV(a) : \sigma_1(x) = \sigma_2(x) \text{ then } \mathcal{A}[\![a]\!]\sigma_1 = \mathcal{A}[\![a]\!]\sigma_2$$

$$\text{if } \forall x \in FV(b) : \sigma_1(x) = \sigma_2(x) \text{ then } \mathcal{B}[\![b]\!]\sigma_1 = \mathcal{B}[\![b]\!]\sigma_2$$

$[ass]$	$\langle [x := a]^\ell, \sigma\rangle \rightarrow \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$	
$[skip]$	$\langle [\texttt{skip}]^\ell, \sigma\rangle \rightarrow \sigma$	
$[seq_1]$	$\dfrac{\langle S_1, \sigma\rangle \rightarrow \langle S_1', \sigma'\rangle}{\langle S_1; S_2, \sigma\rangle \rightarrow \langle S_1'; S_2, \sigma'\rangle}$	
$[seq_2]$	$\dfrac{\langle S_1, \sigma\rangle \rightarrow \sigma'}{\langle S_1; S_2, \sigma\rangle \rightarrow \langle S_2, \sigma'\rangle}$	
$[if_1]$	$\langle \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2, \sigma\rangle \rightarrow \langle S_1, \sigma\rangle$	if $\mathcal{B}[\![b]\!]\sigma = \mathit{true}$
$[if_2]$	$\langle \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2, \sigma\rangle \rightarrow \langle S_2, \sigma\rangle$	if $\mathcal{B}[\![b]\!]\sigma = \mathit{false}$
$[wh_1]$	$\langle \texttt{while } [b]^\ell \texttt{ do } S, \sigma\rangle \rightarrow \langle (S; \texttt{while } [b]^\ell \texttt{ do } S), \sigma\rangle$	if $\mathcal{B}[\![b]\!]\sigma = \mathit{true}$
$[wh_2]$	$\langle \texttt{while } [b]^\ell \texttt{ do } S, \sigma\rangle \rightarrow \sigma$	if $\mathcal{B}[\![b]\!]\sigma = \mathit{false}$

Table 2.6: The Structural Operational Semantics of WHILE.

These results can easily be proved by structural induction on expressions (or by mathematical induction on their size); see Appendix B for a brief introduction to these proof principles.

The detailed definition of the semantics of the statements may be found in Table 2.6; it is explained below.

The clause $[ass]$ specifies that the assignment $x := a$ is executed in one step; here we write $\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$ for the state that is as σ except that x is mapped to $\mathcal{A}[\![a]\!]\sigma$, i.e. the value that a will evaluate to in the state σ. Formally:

$$(\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma])y = \begin{cases} \mathcal{A}[\![a]\!]\sigma & \text{if } x = y \\ \sigma(y) & \text{otherwise} \end{cases}$$

The semantics of sequencing is given by the two rules $[seq_1]$ and $[seq_2]$ and the idea is as follows. The first step of executing $S_1; S_2$ is the first step of executing S_1. It may be that only one step is needed for S_1 to terminate and then the rule $[seq_2]$ applies and says that the new configuration is $\langle S_2, \sigma'\rangle$ reflecting that we are ready to start executing S_2 in the next step. Alternatively, S_1 may not terminate in just one step but gives rise to some other configuration $\langle S_1', \sigma'\rangle$; then the rule $[seq_1]$ applies and it expresses that the rest of S_1 and all of S_2 still have to be executed: the next configuration is $\langle S_1'; S_2, \sigma'\rangle$.

The semantics of the conditional is given by the two axioms $[if_1]$ and $[if_2]$ expressing that the first step of computation will select the appropriate branch

based on the current value of the boolean expression.

Finally, the semantics of the `while`-construct is given by the two axioms $[wh_1]$ and $[wh_2]$; the first axiom expresses that if the boolean expressions evaluates to true then the first step is to unroll the loop and the second axiom expresses that the execution terminates if the boolean expression evaluates to false.

Derivation sequences. A *derivation sequence* for a statement S_1 and a state σ_1 can take one of two forms:

- It is a *finite* sequence of configurations $\langle S_1, \sigma_1\rangle, \cdots, \langle S_n, \sigma_n\rangle, \sigma_{n+1}$ satisfying $\langle S_i, \sigma_i\rangle \rightarrow \langle S_{i+1}, \sigma_{i+1}\rangle$ for $i = 1, \cdots, n-1$ and $\langle S_n, \sigma_n\rangle \rightarrow \sigma_{n+1}$; this corresponds to a terminating computation.
- It is an *infinite* sequence of configurations $\langle S_1, \sigma_1\rangle, \cdots, \langle S_i, \sigma_i\rangle, \cdots$ satisfying $\langle S_i, \sigma_i\rangle \rightarrow \langle S_{i+1}, \sigma_{i+1}\rangle$ for all $i \geq 1$; this corresponds to a looping computation.

Example 2.13 We illustrate the semantics by showing an execution of the factorial program of Example 1.1. In the following we assume that the state $\sigma_{n_x n_y n_z}$ maps `x` to n_x, `y` to n_y and `z` to n_z. We then get the following finite derivation sequence:

$$\langle [\texttt{y:=x}]^1; [\texttt{z:=1}]^2; \texttt{while } [\texttt{y>1}]^3 \texttt{ do } ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5); [\texttt{y:=0}]^6, \sigma_{300}\rangle$$

$$\rightarrow \langle [\texttt{z:=1}]^2; \texttt{while } [\texttt{y>1}]^3 \texttt{ do } ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5); [\texttt{y:=0}]^6, \sigma_{330}\rangle$$

$$\rightarrow \langle \texttt{while } [\texttt{y>1}]^3 \texttt{ do } ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5); [\texttt{y:=0}]^6, \sigma_{331}\rangle$$

$$\rightarrow \langle [\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5; \texttt{while } [\texttt{y>1}]^3 \texttt{ do } ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5); [\texttt{y:=0}]^6, \sigma_{331}\rangle$$

$$\rightarrow \langle [\texttt{y:=y-1}]^5; \texttt{while } [\texttt{y>1}]^3 \texttt{ do } ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5); [\texttt{y:=0}]^6, \sigma_{333}\rangle$$

$$\rightarrow \langle \texttt{while } [\texttt{y>1}]^3 \texttt{ do } ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5); [\texttt{y:=0}]^6, \sigma_{323}\rangle$$

$$\rightarrow \langle [\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5; \texttt{while } [\texttt{y>1}]^3 \texttt{ do } ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5); [\texttt{y:=0}]^6, \sigma_{323}\rangle$$

$$\rightarrow \langle [\texttt{y:=y-1}]^5; \texttt{while } [\texttt{y>1}]^3 \texttt{ do } ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5); [\texttt{y:=0}]^6, \sigma_{326}\rangle$$

$$\rightarrow \langle \texttt{while } [\texttt{y>1}]^3 \texttt{ do } ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5); [\texttt{y:=0}]^6, \sigma_{316}\rangle$$

$$\rightarrow \langle [\texttt{y:=0}]^6, \sigma_{316}\rangle$$

$$\rightarrow \sigma_{306}$$

Note that labels have no impact on the semantics: they are merely carried along and never inspected. ∎

Properties of the semantics. We shall first establish a number of properties of the operations on programs and labels that we have used in the formulation of the analyses. In the course of the computation the set of flows, the set of final labels and the set of elementary blocks of the statements of the configurations will be modified; Lemma 2.14 shows that the sets will decrease:

Lemma 2.14

(i) If $\langle S, \sigma\rangle \rightarrow \sigma'$ then $\mathit{final}(S) = \{\mathit{init}(S)\}$.

(ii) If $\langle S, \sigma\rangle \rightarrow \langle S', \sigma'\rangle$ then $\mathit{final}(S) \supseteq \mathit{final}(S')$.

(iii) If $\langle S, \sigma\rangle \rightarrow \langle S', \sigma'\rangle$ then $\mathit{flow}(S) \supseteq \mathit{flow}(S')$.

(iv) If $\langle S, \sigma\rangle \rightarrow \langle S', \sigma'\rangle$ then $\mathit{blocks}(S) \supseteq \mathit{blocks}(S')$ and if S is label consistent then so is S'. ■

Proof The proof of (i) is by induction on the shape of the inference tree used to establish $\langle S, \sigma\rangle \rightarrow \sigma'$; we refer to Appendix B for a brief introduction to the proof principle. Consulting Table 2.6 we see that there are three non-vacuous cases:

The case [*ass*]. Then $\langle [x := a]^\ell, \sigma\rangle \rightarrow \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$ and we get:

$$\mathit{final}([x := a]^\ell) = \{\ell\} = \{\mathit{init}([x := a]^\ell)\}$$

The case [*skip*]. Then $\langle [\mathtt{skip}]^\ell, \sigma\rangle \rightarrow \sigma$ and we get:

$$\mathit{final}([\mathtt{skip}]^\ell) = \{\ell\} = \{\mathit{init}([\mathtt{skip}]^\ell)\}$$

The case [wh_2]. Then $\langle \mathtt{while}\ [b]^\ell\ \mathtt{do}\ S, \sigma\rangle \rightarrow \sigma$ because $\mathcal{B}[\![b]\!]\sigma = \mathit{false}$ and we get:

$$\mathit{final}(\mathtt{while}\ [b]^\ell\ \mathtt{do}\ S) = \{\ell\} = \{\mathit{init}(\mathtt{while}\ [b]^\ell\ \mathtt{do}\ S)\}$$

This completes the proof of (i).

The proof of (ii) is by induction on the shape of the inference tree used to establish $\langle S, \sigma\rangle \rightarrow \langle S', \sigma'\rangle$. There are five non-vacuous cases:

The case [seq_1]. Then $\langle S_1; S_2, \sigma\rangle \rightarrow \langle S_1'; S_2, \sigma'\rangle$ because $\langle S_1, \sigma\rangle \rightarrow \langle S_1', \sigma'\rangle$ and we get:

$$\mathit{final}(S_1; S_2) = \mathit{final}(S_2) = \mathit{final}(S_1'; S_2)$$

The case [seq_2]. Then $\langle S_1; S_2, \sigma\rangle \rightarrow \langle S_2, \sigma'\rangle$ because $\langle S_1, \sigma\rangle \rightarrow \sigma'$ and we get:

$$\mathit{final}(S_1; S_2) = \mathit{final}(S_2)$$

The case [if_1]. Then $\langle \mathtt{if}\ [b]^\ell\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, \sigma\rangle \rightarrow \langle S_1, \sigma\rangle$ because $\mathcal{B}[\![b]\!]\sigma = \mathit{true}$ and we get:

$$\mathit{final}(\mathtt{if}\ [b]^\ell\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2) = \mathit{final}(S_1) \cup \mathit{final}(S_2) \supseteq \mathit{final}(S_1)$$

The case [if_2] is similar to the previous case.

The case $[wh_1]$. Then $\langle \texttt{while } [b]^\ell \texttt{ do } S, \sigma\rangle \rightarrow \langle (S; \texttt{while } [b]^\ell \texttt{ do } S), \sigma\rangle$ because $\mathcal{B}[\![b]\!]\sigma = \mathit{true}$ and we get:

$$\mathit{final}(S; \texttt{while } [b]^\ell \texttt{ do } S) = \mathit{final}(\texttt{while } [b]^\ell \texttt{ do } S)$$

This completes the proof of (ii).

The proof of (iii) is by induction on the shape of the inference tree used to establish $\langle S, \sigma\rangle \rightarrow \langle S', \sigma'\rangle$. There are five non-vacuous cases:

The case $[seq_1]$. Then $\langle S_1; S_2, \sigma\rangle \rightarrow \langle S_1'; S_2, \sigma'\rangle$ because $\langle S_1, \sigma\rangle \rightarrow \langle S_1', \sigma'\rangle$ and we get

$$\begin{aligned}
\mathit{flow}(S_1; S_2) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{(\ell, \mathit{init}(S_2)) \mid \ell \in \mathit{final}(S_1)\} \\
&\supseteq \mathit{flow}(S_1') \cup \mathit{flow}(S_2) \cup \{(\ell, \mathit{init}(S_2)) \mid \ell \in \mathit{final}(S_1)\} \\
&\supseteq \mathit{flow}(S_1') \cup \mathit{flow}(S_2) \cup \{(\ell, \mathit{init}(S_2)) \mid \ell \in \mathit{final}(S_1')\} \\
&= \mathit{flow}(S_1'; S_2)
\end{aligned}$$

where we have used the induction hypothesis and (ii).

The case $[seq_2]$. Then $\langle S_1; S_2, \sigma\rangle \rightarrow \langle S_2, \sigma'\rangle$ because $\langle S_1, \sigma\rangle \rightarrow \sigma'$ and we get:

$$\begin{aligned}
\mathit{flow}(S_1; S_2) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{(\ell, \mathit{init}(S_2)) \mid \ell \in \mathit{final}(S_1)\} \\
&\supseteq \mathit{flow}(S_2)
\end{aligned}$$

The case $[if_1]$. Then $\langle \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2, \sigma\rangle \rightarrow \langle S_1, \sigma\rangle$ because $\mathcal{B}[\![b]\!]\sigma = \mathit{true}$ and we get:

$$\begin{aligned}
\mathit{flow}(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \\
&\quad \cup \{(\ell, \mathit{init}(S_1)), (\ell, \mathit{init}(S_2))\} \\
&\supseteq \mathit{flow}(S_1)
\end{aligned}$$

The case $[if_2]$ is similar to the previous case.

The case $[wh_2]$. Then $\langle \texttt{while } [b]^\ell \texttt{ do } S, \sigma\rangle \rightarrow \langle S; \texttt{while } [b]^\ell \texttt{ do } S, \sigma\rangle$ because $\mathcal{B}[\![b]\!]\sigma = \mathit{true}$ and we get:

$$\begin{aligned}
\mathit{flow}(S; \texttt{while } [b]^\ell \texttt{ do } S) &= \mathit{flow}(S) \cup \mathit{flow}(\texttt{while } [b]^\ell \texttt{ do } S) \\
&\quad \cup \{(\ell', \ell) \mid \ell' \in \mathit{final}(S)\} \\
&= \mathit{flow}(S) \cup \mathit{flow}(S) \cup \{(\ell, \mathit{init}(S))\} \\
&\quad \cup \{(\ell', \ell) \mid \ell' \in \mathit{final}(S)\} \cup \{(\ell', \ell) \mid \ell' \in \mathit{final}(S)\} \\
&= \mathit{flow}(S) \cup \{(\ell, \mathit{init}(S))\} \cup \{(\ell', \ell) \mid \ell' \in \mathit{final}(S)\} \\
&= \mathit{flow}(\texttt{while } [b]^\ell \texttt{ do } S)
\end{aligned}$$

This completes the proof of (iii).

The proof of (iv) is similar to that of (iii) and we omit the details. ∎

2.2.2 Correctness of Live Variables Analysis

Preservation of solutions. Subsection 2.1.4 shows how to define an *equation system* for a label consistent program $S_\star$ (with isolated exits); we will refer to this system as $\mathsf{LV}^{=}(S_\star)$. The construction of $\mathsf{LV}^{=}(S_\star)$ can be modified to give a *constraint system* $\mathsf{LV}^{\subseteq}(S_\star)$ of the form studied in Subsection 1.3.2:

$$\begin{array}{lll} \mathsf{LV}_{exit}(\ell) & \supseteq & \left\{ \begin{array}{ll} \emptyset & \text{if } \ell \in \mathit{final}(S_\star) \\ \bigcup\{\mathsf{LV}_{entry}(\ell') \mid (\ell',\ell) \in \mathit{flow}^R(S_\star)\} & \text{otherwise} \end{array} \right. \\ \mathsf{LV}_{entry}(\ell) & \supseteq & (\mathsf{LV}_{exit}(\ell) \backslash \mathit{kill}_{\mathsf{LV}}(B^\ell)) \cup \mathit{gen}_{\mathsf{LV}}(B^\ell) \\ & & \text{where } B^\ell \in \mathit{blocks}(S_\star) \end{array}$$

We make this definition because in the correctness proof we will want to use the *same* solution for all statements derived from $S_\star$; this will be possible for $\mathsf{LV}^{\subseteq}(S_\star)$ but not for $\mathsf{LV}^{=}(S_\star)$.

Now consider a collection *live* of functions:

$$\mathit{live}_{entry}, \mathit{live}_{exit} : \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{Var}_\star)$$

We say that *live* solves $\mathsf{LV}^{=}(S)$, and write

$$\mathit{live} \models \mathsf{LV}^{=}(S)$$

if the functions satisfy the equations; similarly we write

$$\mathit{live} \models \mathsf{LV}^{\subseteq}(S)$$

if *live* solves $\mathsf{LV}^{\subseteq}(S_\star)$. The following result shows that any solution of the equation system is also a solution of the constraint system and that the least solutions of the two systems coincide.

Lemma 2.15 Consider a label consistent program $S_\star$. If $\mathit{live} \models \mathsf{LV}^{=}(S_\star)$ then $\mathit{live} \models \mathsf{LV}^{\subseteq}(S_\star)$. The least solution of $\mathsf{LV}^{=}(S_\star)$ coincides with the least solution of $\mathsf{LV}^{\subseteq}(S_\star)$. ∎

Proof If $\mathit{live} \models \mathsf{LV}^{=}(S_\star)$ then clearly $\mathit{live} \models \mathsf{LV}^{\subseteq}(S_\star)$ because "$\supseteq$" includes the case of "$=$".

Next let us prove that $\mathsf{LV}^{\subseteq}(S_\star)$ and $\mathsf{LV}^{=}(S_\star)$ have the same least solution. We gave a constructive proof of a related result in Chapter 1 (under some assumptions about finiteness) so let us here give a more abstract proof using more advanced fixed point theory (as covered in Appendix A). In the manner of Chapter 1 we construct a function F^S_{LV} such that:

$$\mathit{live} \models \mathsf{LV}^{\subseteq}(S) \quad \text{iff} \quad \mathit{live} \sqsupseteq F^S_{\mathsf{LV}}(\mathit{live})$$

$$\mathit{live} \models \mathsf{LV}^{=}(S) \quad \text{iff} \quad \mathit{live} = F^S_{\mathsf{LV}}(\mathit{live})$$

$$\begin{array}{ccccccccc}
\langle S, \sigma_1 \rangle & \rightarrow & \langle S', \sigma_1' \rangle & \rightarrow & \cdots & \rightarrow & \langle S'', \sigma_1'' \rangle & \rightarrow & \sigma_1''' \\
\updownarrow \models \mathsf{LV}^{\subseteq} & & \updownarrow \models \mathsf{LV}^{\subseteq} & & & & \updownarrow \models \mathsf{LV}^{\subseteq} & & \\
\mathit{live} & & \mathit{live} & & \cdots & & \mathit{live} & &
\end{array}$$

Figure 2.4: Preservation of analysis result.

Using Tarski's Fixed Point Theorem (Proposition A.10) we now have that F^S_{LV} has a least fixed point $\mathit{lfp}(F^S_{\mathsf{LV}})$ such that

$$\mathit{lfp}(F^S_{\mathsf{LV}}) = \bigsqcap\{\mathit{live} \mid \mathit{live} \sqsupseteq F^S_{\mathsf{LV}}(\mathit{live})\} = \bigsqcap\{\mathit{live} \mid \mathit{live} = F^S_{\mathsf{LV}}(\mathit{live})\}$$

and since $\mathit{lfp}(F^S_{\mathsf{LV}}) = F^S_{\mathsf{LV}}(\mathit{lfp}(F^S_{\mathsf{LV}}))$ as well as $\mathit{lfp}(F^S_{\mathsf{LV}}) \sqsupseteq F^S_{\mathsf{LV}}(\mathit{lfp}(F^S_{\mathsf{LV}}))$ this proves the result. ■

The next result shows that if we have a solution to the constraint system corresponding to some statement S_1 then it will also be a solution to the constraint system obtained from a sub-statement S_2; this result will be essential in the proof of the correctness result.

Lemma 2.16 If $\mathit{live} \models \mathsf{LV}^{\subseteq}(S_1)$ (with S_1 being label consistent) and $\mathit{flow}(S_1) \supseteq \mathit{flow}(S_2)$ and $\mathit{blocks}(S_1) \supseteq \mathit{blocks}(S_2)$ then $\mathit{live} \models \mathsf{LV}^{\subseteq}(S_2)$ (with S_2 being label consistent). ■

Proof If S_1 is label consistent and $\mathit{blocks}(S_1) \supseteq \mathit{blocks}(S_2)$ then also S_2 is label consistent. If $\mathit{live} \models \mathsf{LV}^{\subseteq}(S_1)$ then live also satisfies each constraint in $\mathsf{LV}^{\subseteq}(S_2)$ and hence $\mathit{live} \models \mathsf{LV}^{\subseteq}(S_2)$. ■

We now have the following corollary expressing that the solution to the constraints of $\mathsf{LV}^{\subseteq}$ is preserved during computation; this is illustrated in Figure 2.4 for finite computations.

Corollary 2.17 If $\mathit{live} \models \mathsf{LV}^{\subseteq}(S)$ (for S being label consistent) and if $\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$ then also $\mathit{live} \models \mathsf{LV}^{\subseteq}(S')$. ■

Proof Follows from Lemma 2.14 and 2.16. ■

We also have an easy result relating entry and exit components of a solution.

Lemma 2.18 If $\mathit{live} \models \mathsf{LV}^{\subseteq}(S)$ (with S being label consistent) then for all $(\ell, \ell') \in \mathit{flow}(S)$ we have $\mathit{live}_{exit}(\ell) \supseteq \mathit{live}_{entry}(\ell')$. ■

Proof The result follows immediately from the construction of $\mathsf{LV}^{\subseteq}(S)$. ■

$$
\begin{array}{ccccccccc}
\langle S, \sigma_1 \rangle & \rightarrow & \langle S', \sigma_1' \rangle & \rightarrow & \cdots & \rightarrow & \langle S'', \sigma_1'' \rangle & \rightarrow & \sigma_1''' \\
\updownarrow \sim_V & & \updownarrow \sim_{V'} & & & & \updownarrow \sim_{V''} & & \updownarrow \sim_{X(\ell)} \\
\langle S, \sigma_2 \rangle & \rightarrow & \langle S', \sigma_2' \rangle & \rightarrow & \cdots & \rightarrow & \langle S'', \sigma_2'' \rangle & \rightarrow & \sigma_2''' \\
V = N(init(S)) & & V' = N(init(S')) & & & & V'' = N(init(S'')) & & \ell \in final(S)
\end{array}
$$

Figure 2.5: The correctness result for *live*.

Correctness relation. Intuitively, the correctness result for the Live Variables Analysis should express that the sets of live variables computed by the analysis are correct throughout the computation. Only the values of the live variables are of interest for the computation: if a variable is not live then its value in the state is irrelevant – its value cannot affect the interesting parts of the result of the computation. Assume now that V is a set of live variables and define the *correctness relation*:

$$\sigma_1 \sim_V \sigma_2 \quad \text{iff} \quad \forall x \in V : \sigma_1(x) = \sigma_2(x)$$

The relation expresses that for all practical purposes the two states σ_1 and σ_2 are equal: only the values of the live variables matters and here the two states are equal.

Example 2.19 Consider the statement $[\mathtt{x:=y+z}]^\ell$ and let $V_1 = \{\mathtt{y}, \mathtt{z}\}$ and $V_2 = \{\mathtt{x}\}$. Then $\sigma_1 \sim_{V_1} \sigma_2$ means $\sigma_1(\mathtt{y}) = \sigma_2(\mathtt{y}) \wedge \sigma_1(\mathtt{z}) = \sigma_2(\mathtt{z})$ and $\sigma_1 \sim_{V_2} \sigma_2$ means $\sigma_1(\mathtt{x}) = \sigma_2(\mathtt{x})$.

Next suppose that $\langle [\mathtt{x:=y+z}]^\ell, \sigma_1 \rangle \rightarrow \sigma_1'$ and $\langle [\mathtt{x:=y+z}]^\ell, \sigma_2 \rangle \rightarrow \sigma_2'$. Clearly $\sigma_1 \sim_{V_1} \sigma_2$ ensures that $\sigma_1' \sim_{V_2} \sigma_2'$. So if V_2 is the set of variables live after $[\mathtt{x:=y+z}]^\ell$ then V_1 is the set of variables live before $[\mathtt{x:=y+z}]^\ell$. ∎

The correctness result will express that the relation "$\sim$" is an invariant under the computation. This is illustrated in Figure 2.5 for finite computations and it is formally expressed by Corollary 2.22 below; to improve the legibility of formulae we write:

$$
\begin{aligned}
N(\ell) &= live_{entry}(\ell) \\
X(\ell) &= live_{exit}(\ell)
\end{aligned}
$$

Since the live variables at the exit from a label are defined to be (a superset of) the union of the live variables at the entries of all of it successor labels, we have the following result.

Lemma 2.20 Assume *live* $\models \mathsf{LV}^{\subseteq}(S)$ with S being label consistent. Then $\sigma_1 \sim_{X(\ell)} \sigma_2$ implies $\sigma_1 \sim_{N(\ell')} \sigma_2$ for all $(\ell, \ell') \in \mathit{flow}(S)$. ■

Proof Follows directly from Lemma 2.18 and the definition of $\sim_V$. ■

Correctness result. We are now ready for the main result. It states how semantically correct liveness information is preserved under each step of the execution: (i) in the case where we do not immediately terminate and (ii) in the case where we do immediately terminate.

Theorem 2.21
If *live* $\models \mathsf{LV}^{\subseteq}(S)$ (with S being label consistent) then:

(i) if $\langle S, \sigma_1 \rangle \rightarrow \langle S', \sigma_1' \rangle$ and $\sigma_1 \sim_{N(\mathit{init}(S))} \sigma_2$ then there exists σ_2' such that $\langle S, \sigma_2 \rangle \rightarrow \langle S', \sigma_2' \rangle$ and $\sigma_1' \sim_{N(\mathit{init}(S'))} \sigma_2'$, and

(ii) if $\langle S, \sigma_1 \rangle \rightarrow \sigma_1'$ and $\sigma_1 \sim_{N(\mathit{init}(S))} \sigma_2$ then there exists σ_2' such that $\langle S, \sigma_2 \rangle \rightarrow \sigma_2'$ and $\sigma_1' \sim_{X(\mathit{init}(S))} \sigma_2'$

Proof The proof is by induction on the shape of the inference tree used to establish $\langle S, \sigma_1 \rangle \rightarrow \langle S', \sigma_1' \rangle$ and $\langle S, \sigma_1 \rangle \rightarrow \sigma_1'$, respectively.

The case [*ass*]. Then $\langle [x := a]^\ell, \sigma_1 \rangle \rightarrow \sigma_1[x \mapsto \mathcal{A}[\![a]\!]\sigma_1]$ and from the specification of the constraint system we have

$$N(\ell) = \mathit{live}_{\mathit{entry}}(\ell) \supseteq (\mathit{live}_{\mathit{exit}}(\ell) \backslash \{x\}) \cup FV(a) = (X(\ell) \backslash \{x\}) \cup FV(a)$$

and thus

$$\sigma_1 \sim_{N(\ell)} \sigma_2 \text{ implies } \mathcal{A}[\![a]\!]\sigma_1 = \mathcal{A}[\![a]\!]\sigma_2$$

because the value of a is only affected by the variables occurring in it. Therefore, taking

$$\sigma_2' = \sigma_2[x \mapsto \mathcal{A}[\![a]\!]\sigma_2]$$

we have that $\sigma_1'(x) = \sigma_2'(x)$ and thus $\sigma_1' \sim_{X(\ell)} \sigma_2'$ as required.

The case [*skip*]. Then $\langle [\mathbf{skip}]^\ell, \sigma_1 \rangle \rightarrow \sigma_1$ and from the specification of the constraint system

$$N(\ell) = \mathit{live}_{\mathit{entry}}(\ell) \supseteq (\mathit{live}_{\mathit{exit}}(\ell) \backslash \emptyset) \cup \emptyset = \mathit{live}_{\mathit{exit}}(\ell) = X(\ell)$$

and we take σ_2' to be σ_2.

The case [*seq*$_1$]. Then $\langle S_1; S_2, \sigma_1 \rangle \rightarrow \langle S_1'; S_2, \sigma_1' \rangle$ because $\langle S_1, \sigma_1 \rangle \rightarrow \langle S_1', \sigma_1' \rangle$. By construction we have $\mathit{flow}(S_1; S_2) \supseteq \mathit{flow}(S_1)$ and also $\mathit{blocks}(S_1; S_2) \supseteq \mathit{blocks}(S_1)$. Thus by Lemma 2.16, *live* is a solution to $LV^{\subseteq}(S_1)$ and thus by the induction hypothesis there exists σ_2' such that

$$\langle S_1, \sigma_2 \rangle \rightarrow \langle S_1', \sigma_2' \rangle \text{ and } \sigma_1' \sim_{N(\mathit{init}(S_1'))} \sigma_2'$$

and the result follows.

The case [seq_2]. Then $\langle S_1; S_2, \sigma_1\rangle \rightarrow \langle S_2, \sigma_1'\rangle$ because $\langle S_1, \sigma_1\rangle \rightarrow \sigma_1'$. Once again by Lemma 2.16, *live* is a solution to $LV^{\subseteq}(S_1)$ and thus by the induction hypothesis there exists σ_2' such that:

$$\langle S_1, \sigma_2\rangle \rightarrow \sigma_2' \text{ and } \sigma_1' \sim_{X(init(S_1))} \sigma_2'$$

Now

$$\{(\ell, init(S_2)) \mid \ell \in final(S_1)\} \subseteq flow(S_1; S_2)$$

and by Lemma 2.14, $final(S_1) = \{init(S_1)\}$. Thus by Lemma 2.20

$$\sigma_1' \sim_{N(init(S_2))} \sigma_2'$$

and the result follows.

The case [if_1]. Then $\langle \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2, \sigma_1\rangle \rightarrow \langle S_1, \sigma_1\rangle$ because $\mathcal{B}[\![b]\!]\sigma_1 = true$. Since $\sigma_1 \sim_{N(\ell)} \sigma_2$ and $N(\ell) = live_{entry}(\ell) \supseteq FV(b)$, we also have that $\mathcal{B}[\![b]\!]\sigma_2 = true$ (the value of b is only affected by the variables occurring in it) and thus:

$$\langle \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2, \sigma_2\rangle \rightarrow \langle S_1, \sigma_2\rangle$$

From the specification of the constraint system, $N(\ell) = live_{entry}(\ell) \supseteq live_{exit}(\ell) = X(\ell)$ and hence $\sigma_1 \sim_{X(\ell)} \sigma_2$. Since $(\ell, init(S_1)) \in flow(S)$, Lemma 2.20 gives $\sigma_1 \sim_{N(init(S_1))} \sigma_2$ as required.

The case [if_2] is similar to the previous case.

The case [wh_1]. Then $\langle \texttt{while } [b]^\ell \texttt{ do } S, \sigma_1\rangle \rightarrow \langle S; \texttt{while } [b]^\ell \texttt{ do } S, \sigma_1\rangle$ because $\mathcal{B}[\![b]\!]\sigma_1 = true$. Since $\sigma_1 \sim_{N(\ell)} \sigma_2$ and $N(\ell) \supseteq FV(b)$, we also have that $\mathcal{B}[\![b]\!]\sigma_2 = true$ and thus

$$\langle \texttt{while } [b]^\ell \texttt{ do } S, \sigma_2\rangle \rightarrow \langle S; \texttt{while } [b]^\ell \texttt{ do } S, \sigma_2\rangle$$

and again, since $N(\ell) = live_{entry}(\ell) \supseteq live_{exit}(\ell) = X(\ell)$ we have $\sigma_1 \sim_{X(\ell)} \sigma_2$ and then

$$\sigma_1 \sim_{N(init(S))} \sigma_2$$

follows from Lemma 2.20 because $(\ell, init(S)) \in flow(\texttt{while } [b]^\ell \texttt{ do } S)$.

The case [wh_2]. Then $\langle \texttt{while } [b]^\ell \texttt{ do } S, \sigma_1\rangle \rightarrow \sigma_1$ because $\mathcal{B}[\![b]\!]\sigma_1 = false$. Since $\sigma_1 \sim_{N(\ell)} \sigma_2$ and $N(\ell) \supseteq FV(b)$, we also have that $\mathcal{B}[\![b]\!]\sigma_2 = false$ and thus:

$$\langle \texttt{while } [b]^\ell \texttt{ do } S, \sigma_2\rangle \rightarrow \sigma_2$$

From the specification of $\mathsf{LV}^{\subseteq}(S)$, we have $N(\ell) = live_{entry}(\ell) \supseteq live_{exit}(\ell) = X(\ell)$ and thus $\sigma_1 \sim_{X(\ell)} \sigma_2$.

This completes the proof. ∎

Finally, we have an important corollary which lifts the previous result to program executions: (i) in the case where the derivation sequence has not yet terminated and (ii) in the case it has terminated:

Corollary 2.22 If $live \models LV^{\subseteq}(S)$ (with S being label consistent) then:

(i) if $\langle S, \sigma_1\rangle \rightarrow^* \langle S', \sigma_1'\rangle$ and $\sigma_1 \sim_{N(init(S))} \sigma_2$ then there exists σ_2' such that $\langle S, \sigma_2\rangle \rightarrow^* \langle S', \sigma_2'\rangle$ and $\sigma_1' \sim_{N(init(S'))} \sigma_2'$, and

(ii) if $\langle S, \sigma_1 \rangle \rightarrow^* \sigma_1'$ and $\sigma_1 \sim_{N(init(S))} \sigma_2$ then there exists σ_2' such that $\langle S, \sigma_2 \rangle \rightarrow^* \sigma_2'$ and $\sigma_1' \sim_{X(\ell)} \sigma_2'$ for some $\ell \in final(S)$. ■

Proof The proof is by induction on the length of the derivation sequence and uses Theorem 2.21. ■

Remark. We have now proved the correctness of Live Variables Analysis with respect to a small step operational semantics. Obviously, the correctness of the analysis can also be proved with respect to other kinds of semantics. However, note that if one relies on, say, a big step (or natural) semantics then it is not so obvious how to express (and prove) the correctness of looping computations: in a big step semantics a looping computation is modelled by the absence of an inference tree – in contrast to the small step semantics where it is modelled by an infinite derivation sequence. ■

2.3 Monotone Frameworks

Despite the differences between the analyses presented in Section 2.1, there are sufficient similarities to make it plausible that there might be an underlying framework. The advantages that accrue from identifying such a framework include the possibility of designing generic algorithms for solving the data flow equations, as we will see in Section 2.4.

The overall pattern. Each of the four classical analyses (presented in Subsection 2.1.1 to 2.1.4) considers equations for a label consistent program $S_\star$ and they take the form

$$\begin{array}{lcl} Analysis_\circ(\ell) & = & \begin{cases} \iota & \text{if } \ell \in E \\ \bigsqcup\{Analysis_\bullet(\ell') \mid (\ell', \ell) \in F\} & \text{otherwise} \end{cases} \\ Analysis_\bullet(\ell) & = & f_\ell(Analysis_\circ(\ell)) \end{array}$$

where

- $\bigsqcup$ is $\bigcap$ or $\bigcup$ (and $\sqcup$ is $\cup$ or $\cap$),
- F is either $flow(S_\star)$ or $flow^R(S_\star)$,
- E is $\{init(S_\star)\}$ or $final(S_\star)$,
- ι specifies the initial or final analysis information, and
- f_ℓ is the transfer function associated with $B^\ell \in blocks(S_\star)$.

We now have the following characterisation:

- The *forward analyses* have F to be $\mathit{flow}(S_\star)$ and then $\mathit{Analysis}_\circ$ concerns entry conditions and $\mathit{Analysis}_\bullet$ concerns exit conditions; also the equation system presupposes that $S_\star$ has isolated entries.
- The *backward analyses* have F to be $\mathit{flow}^R(S_\star)$ and then $\mathit{Analysis}_\circ$ concerns exit conditions and $\mathit{Analysis}_\bullet$ concerns entry conditions; also the equation system presupposes that $S_\star$ has isolated exits.

The principle we have seen emerging in Section 2.1 is that:

- When $\bigsqcup$ is $\bigcap$ we require the *greatest* sets that solve the equations and we are able to detect properties satisfied by *all* paths of execution reaching (or leaving) the entry (or exit) of a label; these analyses are often called *must analyses*.
- When $\bigsqcup$ is $\bigcup$ we require the *least* sets that solve the equations and we are able to detect properties satisfied by *at least one* execution path to (or from) the entry (or exit) of a label; these analyses are often called *may analyses*.

Remark. Some authors propose a typology for Data Flow Analysis, characterising each analysis by a triple from

$$\{\bigcap, \bigcup\} \times \{\rightarrow, \leftarrow\} \times \{\uparrow, \downarrow\}$$

where $\rightarrow$ means forwards, $\leftarrow$ means backwards, $\downarrow$ means smallest and $\uparrow$ means largest. This leads to eight possible types of analysis – a cube. In fact, given our association of $\bigcap$ with $\uparrow$ and $\bigcup$ with $\downarrow$, the cube collapses to a square. We have presented analyses of the following four types: $(\bigcap, \rightarrow, \uparrow)$, $(\bigcup, \rightarrow, \downarrow)$, $(\bigcap, \leftarrow, \uparrow)$ and $(\bigcup, \leftarrow, \downarrow)$. ∎

It is occasionally awkward to have to assume that forward analyses have isolated entries and that backward analyses have isolated exits. This motivates reformulating the above equations to be of the form

$$\begin{aligned}
\mathit{Analysis}_\circ(\ell) &= \bigsqcup\{\mathit{Analysis}_\bullet(\ell') \mid (\ell', \ell) \in F\} \sqcup \iota_E^\ell \\
&\qquad \text{where } \iota_E^\ell = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases} \\
\mathit{Analysis}_\bullet(\ell) &= f_\ell(\mathit{Analysis}_\circ(\ell))
\end{aligned}$$

where $\bot$ satisfies $l \sqcup \bot = l$ (hence $\bot$ is not really there). This formulation makes sense also for analyses that do not have isolated entries and exits.

In this section, we present a more formal approach to defining data flow frameworks that exploits the similarities that we have identified above. Nothing that we present in this section is dependent on the definition of elementary

blocks, or the programming language constructs; however, the techniques do not directly apply to languages with procedures (which will be addressed in Section 2.5). The view that we take here is that a program is a *transition system*; the nodes represent blocks and each block has a *transfer function* associated with it that specifies how the block acts on the "input" state. (Note that for forward analyses, the input state is the entry state, and for backward analyses, it is the exit state.)

2.3.1 Basic Definitions

Property spaces. One important ingredient in the framework is the *property space*, L, used to represent the data flow information as well as the *combination operator*, $\bigsqcup : \mathcal{P}(L) \to L$, that combines information from different paths; as usual $\sqcup : L \times L \to L$ is defined by $l_1 \sqcup l_2 = \bigsqcup\{l_1, l_2\}$ and we write $\bot$ for $\bigsqcup \emptyset$. It is customary to demand that this property space is in fact a complete lattice; as discussed in Appendix A this just means that it is a partially ordered set, $(L, \sqsubseteq)$, such that each subset, Y, has a least upper bound, $\bigsqcup Y$. Looking ahead to the task of implementing the analysis one often requires that L satisfies the *Ascending Chain Condition*; as discussed in Appendix A this means that each ascending chain, $(l_n)_n$, i.e. $l_1 \sqsubseteq l_2 \sqsubseteq l_3 \sqsubseteq \cdots$, eventually stabilises, i.e. $\exists n : l_n = l_{n+1} = \cdots$.

Example 2.23 For Reaching Definitions we have $L = \mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star^?)$ and it is partially ordered by subset inclusion, i.e. "$\sqsubseteq$" is "$\subseteq$". Similarly, $\bigsqcup Y$ is $\bigcup Y$, $l_1 \sqcup l_2$ is $l_1 \cup l_2$, and $\bot$ is $\emptyset$. That L satisfies the Ascending Chain Condition, i.e. that $l_1 \subseteq l_2 \subseteq \cdots$ implies $\exists n : l_n = l_{n+1} = \cdots$, follows because $\mathbf{Var}_\star \times \mathbf{Lab}_\star^?$ is finite (unlike $\mathbf{Var} \times \mathbf{Lab}$). ∎

Example 2.24 For Available Expressions we have $L = \mathcal{P}(\mathbf{AExp}_\star)$ and it is partially ordered by superset inclusion, i.e. "$\sqsubseteq$" is "$\supseteq$". Similarly, $\bigsqcup Y$ is $\bigcap Y$, $l_1 \sqcup l_2$ is $l_1 \cap l_2$, and $\bot$ is $\mathbf{AExp}_\star$. That L satisfies the Ascending Chain Condition, i.e. that $l_1 \supseteq l_2 \supseteq \cdots$ implies $\exists n : l_n = l_{n+1} = \cdots$, follows because $\mathbf{AExp}_\star$ is finite (unlike $\mathbf{AExp}$). ∎

Remark. Historically, the demands on the property space, L, have often been expressed in a different way. A *join semi-lattice* is a non-empty set, L, with a binary *join* operation, $\sqcup$, which is idempotent, commutative and associative, i.e. $l \sqcup l = l$, $l_1 \sqcup l_2 = l_2 \sqcup l_1$ and $(l_1 \sqcup l_2) \sqcup l_3 = l_1 \sqcup (l_2 \sqcup l_3)$. The commutativity and associativity of the operation mean that it does not matter in which order we combine information from different paths. The join operation induces a partial ordering, $\sqsubseteq$, on the elements by taking $l_1 \sqsubseteq l_2$ if and only if $l_1 \sqcup l_2 = l_2$. It is not hard to show that this in fact defines a partial ordering and that $l_1 \sqcup l_2$ is the least upper bound (with respect to $\sqsubseteq$). A unit for the join operation is an element, $\bot$, such that $\bot \sqcup l = l$. It is

not hard to show that the unit is in fact the least element (with respect to $\sqsubseteq$). It has been customary to demand that the property space, L, is a join semi-lattice with a unit and that it satisfies the Ascending Chain Condition. As proved in Lemma A.8 of Appendix A this is equivalent to our assumption that the property space, L, is a complete lattice satisfying the Ascending Chain Condition. ∎

Some formulations of Monotone Frameworks are expressed in terms of property spaces satisfying a *Descending Chain Condition* and using a combination operator $\bigsqcap$. It follows from the principle of lattice duality (see the Concluding Remarks of Chapter 4) that this does not change the notion of Monotone Framework.

Transfer functions. Another important ingredient in the framework is the set of *transfer functions*, $f_\ell : L \to L$ for $\ell \in \mathbf{Lab}_\star$. It is natural to demand that each transfer function is monotone, i.e. $l \sqsubseteq l'$ implies $f_\ell(l) \sqsubseteq f_\ell(l')$. Intuitively, this says that an increase in our knowledge about the input must give rise to an increase in our knowledge about the output (or at the least that we know the same as before). Formally, we shall see that monotonicity is of importance for the algorithms we develop. To control the set of transfer functions we demand that there is a set $\mathcal{F}$ of monotone functions over L, fulfilling the following conditions:

- $\mathcal{F}$ contains all the transfer functions f_ℓ in question,
- $\mathcal{F}$ contains the identity function *id*, and
- $\mathcal{F}$ is closed under composition of functions.

The condition on the identity function is natural because of the `skip` statement and the condition on composition of functions is natural because of the sequencing of statements. Clearly one can take $\mathcal{F}$ to be the space of monotone functions over L but it is occasionally advantageous to consider a smaller set because it makes it easier to find compact representations of the functions.

Some formulations of Monotone Frameworks associate transfer functions with edges (or flows) rather than nodes (or labels). A similar effect can be obtained using the approach of Exercise 2.11.

Frameworks. In summary, a *Monotone Framework* consists of:

- a complete lattice, L, that satisfies the Ascending Chain Condition, and we write $\bigsqcup$ for the least upper bound operator; and
- a set $\mathcal{F}$ of monotone functions from L to L that contains the identity function and that is closed under function composition.

Note that we do *not* demand that $\mathcal{F}$ is a complete lattice or even a partially ordered set although this is the case for the set of all monotone functions from L to L (see Appendix A).

A somewhat stronger concept is that of a *Distributive Framework*. This is a Monotone Framework where additionally all functions f in $\mathcal{F}$ are required to be distributive:

$$f(l_1 \sqcup l_2) = f(l_1) \sqcup f(l_2)$$

Since $f(l_1 \sqcup l_2) \sqsupseteq f(l_1) \sqcup f(l_2)$ follows from monotonicity, the only additional demand is that $f(l_1 \sqcup l_2) \sqsubseteq f(l_1) \sqcup f(l_2)$. When this condition is fulfilled it is sometimes possible to get more efficient algorithms.

Instances. The data flow equations make it clear that more than just a Monotone (or Distributive) Framework is needed in order to specify an analysis. To this end we define an *instance*, Analysis, of a Monotone (or Distributive) Framework to consist of:

- the complete lattice, L, of the framework;
- the space of functions, $\mathcal{F}$, of the framework;
- a finite *flow*, F, that typically is $\mathit{flow}(S_\star)$ or $\mathit{flow}^R(S_\star)$;
- a finite set of so-called *extremal labels*, E, that typically is $\{\mathit{init}(S_\star)\}$ or $\mathit{final}(S_\star)$;
- an *extremal value*, $\iota \in L$, for the extremal labels; and
- a mapping, $f_\cdot$, from the labels $\mathbf{Lab}_\star$ of F and E to transfer functions in $\mathcal{F}$.

The instance then gives rise to a *set of equations*, $\mathsf{Analysis}^{=}$, of the form considered earlier:

$$\begin{aligned}
\mathit{Analysis}_\circ(\ell) &= \bigsqcup\{\mathit{Analysis}_\bullet(\ell') \mid (\ell',\ell) \in F\} \sqcup \iota_E^\ell \\
&\qquad \text{where } \iota_E^\ell = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases} \\
\mathit{Analysis}_\bullet(\ell) &= f_\ell(\mathit{Analysis}_\circ(\ell))
\end{aligned}$$

It also gives rise to a *set of constraints*, $\mathsf{Analysis}^{\sqsubseteq}$, defined by:

$$\begin{aligned}
\mathit{Analysis}_\circ(\ell) &\sqsupseteq \bigsqcup\{\mathit{Analysis}_\bullet(\ell') \mid (\ell',\ell) \in F\} \sqcup \iota_E^\ell \\
&\qquad \text{where } \iota_E^\ell = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases} \\
\mathit{Analysis}_\bullet(\ell) &\sqsupseteq f_\ell(\mathit{Analysis}_\circ(\ell))
\end{aligned}$$

	Available Expressions	Reaching Definitions	Very Busy Expressions	Live Variables
L	$\mathcal{P}(\mathbf{AExp}_\star)$	$\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star^?)$	$\mathcal{P}(\mathbf{AExp}_\star)$	$\mathcal{P}(\mathbf{Var}_\star)$
$\sqsubseteq$	$\supseteq$	$\subseteq$	$\supseteq$	$\subseteq$
$\bigsqcup$	$\bigcap$	$\bigcup$	$\bigcap$	$\bigcup$
$\bot$	$\mathbf{AExp}_\star$	$\emptyset$	$\mathbf{AExp}_\star$	$\emptyset$
ι	$\emptyset$	$\{(x,?) \mid x \in FV(S_\star)\}$	$\emptyset$	$\emptyset$
E	$\{init(S_\star)\}$	$\{init(S_\star)\}$	$final(S_\star)$	$final(S_\star)$
F	$flow(S_\star)$	$flow(S_\star)$	$flow^R(S_\star)$	$flow^R(S_\star)$
$\mathcal{F}$	$\{f : L \to L \mid \exists l_k, l_g : f(l) = (l \setminus l_k) \cup l_g\}$			
f_ℓ	$f_\ell(l) = (l \setminus kill([B]^\ell)) \cup gen([B]^\ell)$ where $[B]^\ell \in blocks(S_\star)$			

Figure 2.6: Instances for the four classical analyses.

2.3.2 The Examples Revisited

We now return to the four classical analyses from Section 2.1 and show how the analyses of a label consistent program, $S_\star$, can be recast as an instance of a Monotone (in fact Distributive) Framework. We refer to Figure 2.6 for all the data needed to specify the Monotone Framework as well as the instance.

It is immediate that the property space, L, is a complete lattice in all cases. Given the choice of a partial ordering, $\sqsubseteq$, the information about the least element, $\bot$, and the least upper bound operation, $\bigsqcup$, is immediate. Note that we define $\sqsubseteq$ to be $\subseteq$ for those analyses where we used $\bigcup$ (and required the least solution) in Section 2.1, and similarly, that we define $\sqsubseteq$ to be $\supseteq$ for those analyses where we used $\bigcap$ (and required the greatest solution) in Section 2.1. To ensure that L satisfies the Ascending Chain Condition we have restricted our attention to the finite sets of expressions, labels and variables occurring in the program, $S_\star$, under consideration.

The definition of the flow, F, is as one should expect: it is $flow(S_\star)$ for forward analyses and $flow^R(S_\star)$ for backward analyses. Similarly, the extremal labels, E are $\{init(S_\star)\}$ for forward analyses and $final(S_\star)$ for backward analyses. The only thing to note about the extremal value, ι, is that there seems to be no general pattern concerning how to define it: it is not always $\top_L$ (nor is it always $\bot_L$).

It remains to show that the conditions on the set $\mathcal{F}$ of transfer functions are satisfied.

Lemma 2.25 Each of the four data flow analyses in Figure 2.6 is a Monotone Framework as well as a Distributive Framework. ∎

Proof To prove that the analyses are Monotone Frameworks we just have to confirm that $\mathcal{F}$ has the necessary properties:

The functions of $\mathcal{F}$ are monotone: Assume that $l \sqsubseteq l'$. Then $(l \setminus l_k) \sqsubseteq (l' \setminus l_k)$ and, therefore $((l \setminus l_k) \cup l_g) \sqsubseteq ((l' \setminus l_k) \cup l_g)$ and thus $f(l) \sqsubseteq f(l')$ as required. Note that this calculation is valid regardless of whether $\sqsubseteq$ is $\subseteq$ or $\supseteq$.

The identity function is in $\mathcal{F}$: It is obtained by taking both l_k and l_g to be $\emptyset$.

The functions of $\mathcal{F}$ are closed under composition: Suppose $f(l) = (l \setminus l_k) \cup l_g$ and $f'(l) = (l \setminus l'_k) \cup l'_g$. Then we calculate:

$$\begin{aligned}(f \circ f')(l) &= (((l \setminus l'_k) \cup l'_g) \setminus l_k) \cup l_g \\ &= (l \setminus (l'_k \cup l_k)) \cup ((l'_g \setminus l_k) \cup l_g)\end{aligned}$$

So $(f \circ f')(l) = (l \setminus l''_k) \cup l''_g$ where $l''_k = l'_k \cup l_k$ and $l''_g = (l'_g \setminus l_k) \cup l_g$. This completes the proof of the first part of the lemma.

To prove that the analyses are Distributive Frameworks consider $f \in \mathcal{F}$ given by $f(l) = (l \setminus l_k) \cup l_g$. Then we have:

$$\begin{aligned}f(l \sqcup l') &= ((l \sqcup l') \setminus l_k) \cup l_g \\ &= ((l \setminus l_k) \sqcup (l' \setminus l_k)) \cup l_g \\ &= ((l \setminus l_k) \cup l_g) \sqcup ((l' \setminus l_k) \cup l_g) \\ &= f(l) \sqcup f(l')\end{aligned}$$

Note that the above calculation is valid regardless of whether $\sqcup$ is $\cup$ or $\cap$. This completes the proof. ∎

It is worth pointing out that in order to get this result we have made the frameworks dependent upon the actual program – this is needed to enforce that the Ascending Chain Condition is fulfilled.

Example 2.26 Let us return to the Available Expressions Analysis of the program

$$[\texttt{x:=a+b}]^1; [\texttt{y:=a*b}]^2; \texttt{while } [\texttt{y>a+b}]^3 \texttt{ do } ([\texttt{a:=a+1}]^4; [\texttt{x:=a+b}]^5)$$

of Examples 2.4 and 2.5 and let us specify it as an instance of the associated Monotone Framework. The complete lattice of interest is

$$(\mathcal{P}(\{\texttt{a+b}, \texttt{a*b}, \texttt{a+1}\}), \supseteq)$$

with least element $\{\texttt{a+b}, \texttt{a*b}, \texttt{a+1}\}$. The set of transfer functions has the form shown in Figure 2.6.

The instance of the framework additionally has the flow $\{(1,2), (2,3), (3,4), (4,5), (5,3)\}$ and the set of extremal labels is $\{1\}$. The extremal value is $\emptyset$

and the transfer functions associated with the labels are

$$\begin{aligned} f_1^{\mathsf{AE}}(Y) &= Y \cup \{\mathtt{a+b}\} \\ f_2^{\mathsf{AE}}(Y) &= Y \cup \{\mathtt{a*b}\} \\ f_3^{\mathsf{AE}}(Y) &= Y \cup \{\mathtt{a+b}\} \\ f_4^{\mathsf{AE}}(Y) &= Y \setminus \{\mathtt{a+b}, \mathtt{a*b}, \mathtt{a+1}\} \\ f_5^{\mathsf{AE}}(Y) &= Y \cup \{\mathtt{a+b}\} \end{aligned}$$

for $Y \subseteq \{\mathtt{a+b}, \mathtt{a*b}, \mathtt{a+1}\}$. ■

2.3.3 A Non-distributive Example

Lest the reader should imagine that all Monotone Frameworks are Distributive Frameworks, here we present one that is not. The *Constant Propagation Analysis* will determine:

> For each program point, whether or not a variable has a constant value whenever execution reaches that point.

Such information can be used as the basis for an optimisation known as *Constant Folding*: all uses of the variable may be replaced by the constant value.

The Constant Propagation framework. The complete lattice used for Constant Propagation Analysis of a program, $S_\star$, is

$$\widehat{\mathbf{State}}_{\mathsf{CP}} = ((\mathbf{Var}_\star \to \mathbf{Z}^\top)_\bot, \sqsubseteq, \sqcup, \sqcap, \bot, \lambda x.\top)$$

where $\mathbf{Var}_\star$ is the set of variables appearing in the program and $\mathbf{Z}^\top = \mathbf{Z} \cup \{\top\}$ is partially ordered as follows:

$$\begin{aligned} &\forall z \in \mathbf{Z}^\top : z \sqsubseteq \top \\ &\forall z_1, z_2 \in \mathbf{Z} : (z_1 \sqsubseteq z_2) \Leftrightarrow (z_1 = z_2) \end{aligned}$$

The top element of $\mathbf{Z}^\top$ is used to indicate that a variable is non-constant and all other elements indicate that the value is that particular constant. The idea is that an element $\widehat{\sigma}$ of $\mathbf{Var}_\star \to \mathbf{Z}^\top$ is a property state: for each variable x, $\widehat{\sigma}(x)$ will give information about whether or not x is a constant and in the latter case which constant.

To capture the case where no information is available we extend $\mathbf{Var}_\star \to \mathbf{Z}^\top$ with a least element $\bot$, written $(\mathbf{Var}_\star \to \mathbf{Z}^\top)_\bot$. The partial ordering $\sqsubseteq$ on $\widehat{\mathbf{State}}_{\mathsf{CP}} = (\mathbf{Var}_\star \to \mathbf{Z}^\top)_\bot$ is defined by

$$\begin{aligned} &\forall \widehat{\sigma} \in (\mathbf{Var}_\star \to \mathbf{Z}^\top)_\bot : \quad \bot \sqsubseteq \widehat{\sigma} \\ &\forall \widehat{\sigma}_1, \widehat{\sigma}_2 \in \mathbf{Var}_\star \to \mathbf{Z}^\top : \quad \widehat{\sigma}_1 \sqsubseteq \widehat{\sigma}_2 \ \text{ iff } \ \forall x : \widehat{\sigma}_1(x) \sqsubseteq \widehat{\sigma}_2(x) \end{aligned}$$

$$\mathcal{A}_{\mathsf{CP}} : \mathbf{AExp} \to (\widehat{\mathbf{State}}_{\mathsf{CP}} \to \mathbf{Z}_{\perp}^{\top})$$

$$\begin{array}{rcl}
\mathcal{A}_{\mathsf{CP}}[\![x]\!]\widehat{\sigma} & = & \begin{cases} \perp & \text{if } \widehat{\sigma} = \perp \\ \widehat{\sigma}(x) & \text{otherwise} \end{cases} \\
\mathcal{A}_{\mathsf{CP}}[\![n]\!]\widehat{\sigma} & = & \begin{cases} \perp & \text{if } \widehat{\sigma} = \perp \\ n & \text{otherwise} \end{cases} \\
\mathcal{A}_{\mathsf{CP}}[\![a_1 \ op_a \ a_2]\!]\widehat{\sigma} & = & \mathcal{A}_{\mathsf{CP}}[\![a_1]\!]\widehat{\sigma} \ \widehat{\mathsf{op}}_a \ \mathcal{A}_{\mathsf{CP}}[\![a_2]\!]\widehat{\sigma}
\end{array}$$

transfer functions: f_ℓ^{CP}

$$\begin{array}{lrcl}
[x := a]^\ell : & f_\ell^{\mathsf{CP}}(\widehat{\sigma}) & = & \begin{cases} \perp & \text{if } \widehat{\sigma} = \perp \\ \widehat{\sigma}[x \mapsto \mathcal{A}_{\mathsf{CP}}[\![a]\!]\widehat{\sigma}] & \text{otherwise} \end{cases} \\
[\mathtt{skip}]^\ell : & f_\ell^{\mathsf{CP}}(\widehat{\sigma}) & = & \widehat{\sigma} \\
[b]^\ell : & f_\ell^{\mathsf{CP}}(\widehat{\sigma}) & = & \widehat{\sigma}
\end{array}$$

Table 2.7: Constant Propagation Analysis.

and the binary least upper bound operation is then:

$$\begin{array}{ll}
\forall \widehat{\sigma} \in (\mathbf{Var}_\star \to \mathbf{Z}^{\top})_{\perp} : & \widehat{\sigma} \sqcup \perp = \widehat{\sigma} = \perp \sqcup \widehat{\sigma} \\
\forall \widehat{\sigma}_1, \widehat{\sigma}_2 \in \mathbf{Var}_\star \to \mathbf{Z}^{\top} : & \forall x : (\widehat{\sigma}_1 \sqcup \widehat{\sigma}_2)(x) = \widehat{\sigma}_1(x) \sqcup \widehat{\sigma}_2(x)
\end{array}$$

In contrast to the earlier examples, we define the transfer functions as follows:

$$\mathcal{F}_{\mathsf{CP}} = \{f \mid f \text{ is a monotone function on } \widehat{\mathbf{State}}_{\mathsf{CP}}\}$$

It is easy to verify that $\widehat{\mathbf{State}}_{\mathsf{CP}}$ and $\mathcal{F}_{\mathsf{CP}}$ satisfy the requirements of being a Monotone Framework (see Exercise 2.8).

Constant Propagation is a forward analysis, so for the program $S_\star$ we take the flow, F, to be $\mathit{flow}(S_\star)$, the extremal labels, E, to be $\{\mathit{init}(S_\star)\}$, the extremal value, ι_{CP}, to be $\lambda x.\top$, and the mapping, $f_\cdot^{\mathsf{CP}}$, of labels to transfer functions is given in Table 2.7. The specification of the transfer functions uses the function

$$\mathcal{A}_{\mathsf{CP}} : \mathbf{AExp} \to (\widehat{\mathbf{State}}_{\mathsf{CP}} \to \mathbf{Z}_{\perp}^{\top})$$

for analysing expressions. Here the operations on $\mathbf{Z}$ are lifted to $\mathbf{Z}_{\perp}^{\top} = \mathbf{Z} \cup \{\perp, \top\}$ by taking $z_1 \ \widehat{\mathsf{op}}_a \ z_2 = z_1 \ \mathbf{op}_a \ z_2$ if $z_1, z_2 \in \mathbf{Z}$ (and where $\mathbf{op}_a$ is the corresponding arithmetic operation on $\mathbf{Z}$), $z_1 \ \widehat{\mathsf{op}}_a \ z_2 = \perp$ if $z_1 = \perp$ or $z_2 = \perp$ and $z_1 \ \widehat{\mathsf{op}}_a \ z_2 = \top$ otherwise.

Lemma 2.27 Constant Propagation is a Monotone Framework that is *not* a Distributive Framework. ∎

Proof The proof that Constant Propagation is a Monotone Framework is left for Exercise 2.8. To show that it is not a Distributive Framework consider the transfer

function f_ℓ^{CP} for $[\mathtt{y:=x*x}]^\ell$ and let $\widehat{\sigma}_1$ and $\widehat{\sigma}_2$ be such that $\widehat{\sigma}_1(\mathtt{x}) = 1$ and $\widehat{\sigma}_2(\mathtt{x}) = -1$. Then $\widehat{\sigma}_1 \sqcup \widehat{\sigma}_2$ maps x to $\top$ and thus $f_\ell^{\mathsf{CP}}(\widehat{\sigma}_1 \sqcup \widehat{\sigma}_2)$ maps y to $\top$ and hence fails to record that y has the constant value 1. However, both $f_\ell^{\mathsf{CP}}(\widehat{\sigma}_1)$ and $f_\ell^{\mathsf{CP}}(\widehat{\sigma}_2)$ map y to 1 and so does $f_\ell^{\mathsf{CP}}(\widehat{\sigma}_1) \sqcup f_\ell^{\mathsf{CP}}(\widehat{\sigma}_2)$. ∎

Correctness of the analysis will be established in Section 4.5.

2.4 Equation Solving

Having set up a framework, there remains the question of how to *use* the framework to obtain an analysis result. In this section we shall consider two approaches. One is an iterative algorithm in the spirit of Chaotic Iteration as presented in Section 1.7. The other more directly propagates analysis information along paths in the program.

2.4.1 The MFP Solution

We first present a general iterative algorithm for Monotone Frameworks that computes the least solution to the data flow equations. Historically, this is called the *MFP solution* (for *Maximal Fixed Point*) although it in fact computes the least fixed point; the reason is that the classical literature tends to focus on analyses where $\sqcup$ is $\cap$ (and because the least fixed point with respect to $\sqsubseteq$ or $\supseteq$ then equals the greatest fixed point with respect to $\subseteq$).

The algorithm, written in pseudo-code in Table 2.8, takes as input an instance of a Monotone Framework. It uses an array, Analysis, which contains the $Analysis_\circ$ information for each elementary block; the array is indexed by labels. It also uses a *worklist* W which is a list of pairs; each pair is an element of the flow relation F. The presence of a pair in the worklist indicates that the analysis has changed at the exit of (or entry to – for backward analyses) the block labelled by the first component and so must be recomputed at the entry to (or exit from) the block labelled by the second component. As a final stage the algorithm presents the result $(MFP_\circ, MFP_\bullet)$ of the analysis in a form close to the formulation of the data flow equations.

Example 2.28 To illustrate how the algorithms works let us return to Example 2.26 where we consider the program

$$[\mathtt{x:=a+b}]^1; [\mathtt{y:=a*b}]^2; \mathtt{while}\ [\mathtt{y>a+b}]^3\ \mathtt{do}\ ([\mathtt{a:=a+1}]^4; [\mathtt{x:=a+b}]^5)$$

Writing W for the list ((2,3),(3,4),(4,5),(5,3)) and U for the set {a+b, a*b, a+1}, step 1 of the algorithm will initialise the data structures as in the first row in Table 2.9. Step 2 will inspect the first element of the worklist

INPUT: An instance of a Monotone Framework: $(L, \mathcal{F}, F, E, \iota, f_\cdot)$

OUTPUT: $MFP_\circ, MFP_\bullet$

METHOD:

```
Step 1:  Initialisation (of W and Analysis)
         W := nil;
         for all (ℓ, ℓ') in F do
            W := cons((ℓ, ℓ'),W);
         for all ℓ in F or E do
            if ℓ ∈ E then Analysis[ℓ] := ι
                      else Analysis[ℓ] := ⊥_L;

Step 2:  Iteration (updating W and Analysis)
         while W ≠ nil do
            ℓ := fst(head(W)); ℓ' = snd(head(W));
            W := tail(W);
            if f_ℓ(Analysis[ℓ]) ⋢ Analysis[ℓ'] then
               Analysis[ℓ'] := Analysis[ℓ'] ⊔ f_ℓ(Analysis[ℓ]);
               for all ℓ'' with (ℓ', ℓ'') in F do
                  W := cons((ℓ', ℓ''),W);

Step 3:  Presenting the result (MFP_∘ and MFP_•)
         for all ℓ in F or E do
            MFP_∘(ℓ) := Analysis[ℓ];
            MFP_•(ℓ) := f_ℓ(Analysis[ℓ])
```

Table 2.8: Algorithm for solving data flow equations.

and rows 2–7 represent cases where there is a change in the array Analysis and hence a new pair is placed on top of the worklist; it is inspected in the next iteration. Rows 8–12 represent cases where no modification is made in the array and hence the worklist is getting smaller – the elements of W are merely inspected. Step 3 will then produce the solution we already saw in Example 2.5. ∎

Properties of the algorithm. We shall first show that the algorithm computes the expected solution to the equation system.

Lemma 2.29 The worklist algorithm in Table 2.8 always terminates and it computes the least (or MFP) solution to the instance of the framework given as input. ∎

Proof First we prove the termination result. Step 1 and 3 are bounded loops over finite sets and thus trivially terminate. Next consider step 2. Assume that there

		Analysis[ℓ] for ℓ being				
	W	1	2	3	4	5
1	((1,2),W)	$\emptyset$	U	U	U	U
2	((2,3),W)	$\emptyset$	{a+b}	U	U	U
3	((3,4),W)	$\emptyset$	{a+b}	{a+b,a*b}	U	U
4	((4,5),W)	$\emptyset$	{a+b}	{a+b,a*b}	{a+b,a*b}	U
5	((5,3),W)	$\emptyset$	{a+b}	{a+b,a*b}	{a+b,a*b}	$\emptyset$
6	((3,4),W)	$\emptyset$	{a+b}	{a+b}	{a+b,a*b}	$\emptyset$
7	((4,5),W)	$\emptyset$	{a+b}	{a+b}	{a+b}	$\emptyset$
8	((2,3),$\cdots$)	$\emptyset$	{a+b}	{a+b}	{a+b}	$\emptyset$
9	((3,4),$\cdots$)	$\emptyset$	{a+b}	{a+b}	{a+b}	$\emptyset$
10	((4,5),$\cdots$)	$\emptyset$	{a+b}	{a+b}	{a+b}	$\emptyset$
11	((5,3))	$\emptyset$	{a+b}	{a+b}	{a+b}	$\emptyset$
12	()	$\emptyset$	{a+b}	{a+b}	{a+b}	$\emptyset$

Table 2.9: Iteration steps of the worklist algorithm.

are b labels in the program. Then the worklist initially has at most b^2 elements; the worst case is that F associates every label to every label. Each iteration either deletes an element from the worklist or adds up to b new elements. New elements are added if for the pair selected in this iteration, (ℓ, ℓ'), we have $f_\ell(\mathsf{Analysis}[\ell]) \not\sqsubseteq \mathsf{Analysis}[\ell']$; that is, $f_\ell(\mathsf{Analysis}[\ell]) \sqsupset \mathsf{Analysis}[\ell']$ or they are incomparable. In either case, the new value of $\mathsf{Analysis}[\ell']$ is strictly greater than the previous one. Since the set of values satisfies the Ascending Chain Condition, this can only happen a finite number of times. Thus the worklist will eventually be exhausted.

Next we prove the correctness result. Let $Analysis_\circ$ and $Analysis_\bullet$ be the least solution to the instance given as input to the algorithm. The proof is now in three parts: (i) first we show that on each iteration the values in Analysis are approximations to the corresponding values of $Analysis_\circ$, (ii) then we show that $Analysis_\circ$ is an approximation to Analysis at the termination of step 2 of the algorithm, and (iii) finally we combine these results.

Part (i). We show that

$$\forall \ell : \mathsf{Analysis}[\ell] \sqsubseteq Analysis_\circ(\ell)$$

is an invariant of the loop of step 2. After step 1 we have $\mathsf{Analysis}[\ell] \sqsubseteq Analysis_\circ(\ell)$ for all ℓ because $Analysis_\circ(\ell) \sqsupseteq \iota$ whenever $\ell \in E$. After each iteration through the loop either there is no change because the iteration just deletes an element from the worklist or else $\mathsf{Analysis}[\ell'']$ is unchanged for all ℓ'' except for some ℓ'. In that case there is some ℓ such that $(\ell, \ell') \in F$ and

$$\begin{aligned} new\mathsf{Analysis}[\ell'] &= old\mathsf{Analysis}[\ell'] \sqcup f_\ell(old\mathsf{Analysis}[\ell]) \\ &\sqsubseteq Analysis_\circ(\ell') \sqcup f_\ell(Analysis_\circ(\ell)) \\ &= Analysis_\circ(\ell') \end{aligned}$$

The inequality follows since f_ℓ is monotone and the last equation follows from $(Analysis_\circ, Analysis_\bullet)$ being a solution to the instance.

Part (ii). On termination of the loop, the worklist is empty. We show that

$$\forall \ell, \ell' : (\ell, \ell') \in F \Rightarrow \mathsf{Analysis}[\ell'] \sqsupseteq f_\ell(\mathsf{Analysis}[\ell])$$

by contradiction. So suppose that $\mathsf{Analysis}[\ell'] \not\sqsupseteq f_\ell(\mathsf{Analysis}[\ell])$ for some $(\ell, \ell') \in F$ and let us obtain a contradiction. Consider the last time that $\mathsf{Analysis}[\ell]$ was updated. If this was in step 1 we considered (ℓ, ℓ') in step 2 and ensured that

$$\mathsf{Analysis}[\ell'] \sqsupseteq f_\ell(\mathsf{Analysis}[\ell])$$

and this invariant has been maintained ever since; hence this case cannot apply. It follows that $\mathsf{Analysis}[\ell]$ was last updated in step 2. But at that time (ℓ, ℓ') was placed in the worklist once again. When considering (ℓ, ℓ') in step 2 we then ensured that

$$\mathsf{Analysis}[\ell'] \sqsupseteq f_\ell(\mathsf{Analysis}[\ell])$$

and this invariant has been maintained ever since; hence this case cannot apply either. This completes the proof by contradiction.

On termination of the loop we have:

$$\forall \ell \in E : \mathsf{Analysis}[\ell] \sqsupseteq \iota$$

This follows because it was established in step 1 and it is maintained ever since. Thus it follows that at termination of step 2:

$$\forall \ell : \mathsf{Analysis}[\ell] \sqsupseteq (\bigsqcup\{f_{\ell'}(\mathsf{Analysis}[\ell']) \mid (\ell', \ell) \in F\}) \sqcup \iota_E^\ell$$

Part (iii). By our assumptions and Proposition A.10 we have

$$\forall \ell : MFP_\circ(\ell) \sqsupseteq Analysis_\circ(\ell)$$

since $Analysis_\circ(\ell)$ is the least solution to the above constraint system and $MFP_\circ$ equals the final value of Analysis. Together with part (i) this proves that

$$\forall \ell : MFP_\circ[\ell] = Analysis_\circ(\ell)$$

upon termination of step 2. ∎

Based on the proof of termination in Lemma 2.29 we can determine an upper bound on the number of basic operations (for example an application of f_ℓ, an application of $\sqcup$, or an update of Analysis) performed by the algorithm. For this we shall assume that the flow F is represented in such a way (for example an array of lists) that all (ℓ', ℓ'') emanating from ℓ' can be found in time proportional to their number. Suppose that E and F contain at most $b \geq 1$ distinct labels, that F contains at most $e \geq b$ pairs, and that L has finite height at most $h \geq 1$. Then steps 1 and 3 perform at most $O(b + e)$ basic operations. Concerning step 2 a pair is placed on the worklist at most $O(h)$ times, and each time it takes only a constant number of basic steps to

process it – not counting the time needed to add new pairs to W; this yields at most $O(e \cdot h)$ basic operations for step 2. Since $h \geq 1$ and $e \geq b$ this gives at most $O(e \cdot h)$ basic operations for the algorithm. (Since $e \leq b^2$ a potentially coarser bound is $O(b^2 \cdot h)$.)

Example 2.30 Consider the Reaching Definitions Analysis and suppose that there are at most $v \geq 1$ variables and $b \geq 1$ labels in the program, $S_\star$, being analysed. Since $L = \mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star^?)$, it follows that $h \leq v \cdot b$ and thus we have an $O(v \cdot b^3)$ upper bound on the number of basic operations.

Actually we can do better. If $S_\star$ is label consistent then the variable of the pairs (x, ℓ) of $\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star^?)$ will always be uniquely determined by the label ℓ so we get an $O(b^3)$ upper bound on the number of basic operations. Furthermore, F is *flow*$(S_\star)$ and inspection of the equations for *flow*$(S_\star)$ shows that for each label ℓ we construct at most two pairs with ℓ in the first component. This means that $e \leq 2 \cdot b$ and we get an $O(b^2)$ upper bound on the number of basic operations. ∎

2.4.2 The MOP Solution

Let us now consider the other solution method for Monotone Frameworks where we more directly propagate analysis information along paths in the program. Historically, this is called the *MOP solution* (for *Meet Over all Paths*) although we do in fact take the join (or least upper bound) over all paths leading to an elementary block; once again the reason is that the classical literature tends to focus on analyses where $\sqcup$ is $\cap$.

Paths. For the moment, we adopt the informal notion of a *path* to the entry of a block as the list of blocks traversed from the start of the program up to that block (but not including it); analogously, we can define a path from an exit of the block. Data Flow Analyses determine properties of such paths. Forward analyses concern paths from the initial block to the entry of a block; backward analyses concern paths from the exit of a block to a final block. The effect of a path on the state can be computed by composing the transfer functions associated with the individual blocks in the path. In the forward case we collect information about the state of affairs before the block is executed and in the backward case we collect information about the state of affairs immediately after the block has been executed. This informal description contrasts with the approach taken in Section 2.1 and earlier in this section; there we presented equations which were defined in terms of the immediate predecessors (successors) of a block (as defined by the *flow* and *flow*^{R} functions). We will see later that, for a large class of analyses, these two approaches coincide.

For the formal development let us consider an instance $(L, \mathcal{F}, F, E, \iota, f_\cdot)$ of a Monotone Framework. We shall use the notation $\vec{\ell} = [\ell_1, \cdots, \ell_n]$ for a

sequence of $n \geq 0$ labels. We then define two sets of paths. The paths up to *but not* including ℓ are

$$path_\circ(\ell) = \{[\ell_1, \cdots, \ell_{n-1}] \mid n \geq 1 \wedge \forall i < n : (\ell_i, \ell_{i+1}) \in F \wedge \ell_n = \ell \wedge \ell_1 \in E\}$$

and the paths up to *and* including ℓ are:

$$path_\bullet(\ell) = \{[\ell_1, \cdots, \ell_n] \mid n \geq 1 \wedge \forall i < n : (\ell_i, \ell_{i+1}) \in F \wedge \ell_n = \ell \wedge \ell_1 \in E\}$$

For a path $\vec{\ell} = [\ell_1, \cdots, \ell_n]$ we define the transfer function

$$f_{\vec{\ell}} = f_{\ell_n} \circ \cdots \circ f_{\ell_1} \circ id$$

so that for the empty path we have $f_{[\,]} = id$ where id is the identity function. By analogy with the definition of solutions to the equation system, in particular $MFP_\circ(\ell)$ and $MFP_\bullet(\ell)$, we now define two components of the MOP solution. The solution up to *but not* including ℓ is

$$MOP_\circ(\ell) = \bigsqcup\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in path_\circ(\ell)\}$$

and the solution up to *and* including ℓ is:

$$MOP_\bullet(\ell) = \bigsqcup\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in path_\bullet(\ell)\}$$

Unfortunately, the MOP solution is sometimes uncomputable (meaning that it is undecidable) even though the MFP solution is always easily computable (because of the property space satisfying the Ascending Chain Condition); the following result establishes one such result:

Lemma 2.31 The MOP solution for Constant Propagation is undecidable. ■

Proof Let $u_1, \cdots, u_n$ and $v_1, \cdots, v_n$ be strings over the alphabet $\{1, \cdots, 9\}$ (see Appendix C). The *Modified Post Correspondence Problem* is to determine whether or not there exists a sequence $i_1, \cdots, i_m$ with $i_1 = 1$ such that $u_{i_1} \cdots u_{i_m} = v_{i_1} \cdots v_{i_n}$. Let $\mid u \mid$ denote the length of the string u and let $[\![u]\!]$ be its value interpreted as a natural number. Consider the program (omitting most labels)

```
x:=[[u1]]; y:=[[v1]];
while [···] do
      (if [···] then x:=x * 10^|u1| + [[u1]]; y:=y * 10^|v1| + [[v1]] else
       ⋮
       if [···] then x:=x * 10^|un| + [[un]]; y:=y * 10^|vn| + [[vn]] else skip)
[z:=sign((x-y)*(x-y))]^ℓ
```

where **sign** gives the sign (which is 1 for a positive argument and 0 or -1 otherwise) and where the details of $[\cdots]$ are of no concern to us.

Then $MOP_\bullet(\ell)$ will map **z** to 1 if and only if the Modified Post Correspondence Problem has no solution. Since the Modified Post Correspondence problem is undecidable [76] so is the MOP solution for Constant Propagation (assuming that our selection of arithmetic operations does indeed allow those used to be defined). ∎

MOP versus MFP solutions. We shall shortly prove that the MFP solution safely approximates the MOP solution (informally, MFP $\sqsupseteq$ MOP). In the case of a $(\bigcap, \rightarrow, \uparrow)$ or $(\bigcap, \leftarrow, \uparrow)$ analysis, the MFP solution is a subset of the MOP solution ($\sqsupseteq$ is $\subseteq$); in the case of a $(\bigcup, \rightarrow, \downarrow)$ or $(\bigcup, \leftarrow, \downarrow)$ analysis, the MFP solution is a superset of the MOP solution. We can also show that, in the case of Distributive Frameworks, the MOP and MFP solutions coincide.

Lemma 2.32 Consider the MFP and MOP solutions to an instance $(L, \mathcal{F}, F, B, \iota, f_\cdot)$ of a Monotone Framework; then:

$$MFP_\circ \sqsupseteq MOP_\circ \text{ and } MFP_\bullet \sqsupseteq MOP_\bullet$$

If the framework is distributive and if $path_\circ(\ell) \neq \emptyset$ for all ℓ in E and F then:

$$MFP_\circ = MOP_\circ \text{ and } MFP_\bullet = MOP_\bullet$$

∎

Proof It is straightforward to show that:

$$\forall \ell : MOP_\bullet(\ell) \sqsubseteq f_\ell(MOP_\circ(\ell))$$
$$\forall \ell : MFP_\bullet(\ell) = f_\ell(MFP_\circ(\ell))$$

For the first part of the lemma it therefore suffices to prove that:

$$\forall \ell : MOP_\circ(\ell) \sqsubseteq MFP_\circ(\ell)$$

Note that $MFP_\circ$ is the least fixed point of the functional F defined by:

$$F(A_\circ)(\ell) = (\bigsqcup\{f_{\ell'}(A_\circ(\ell')) \mid (\ell', \ell) \in F\}) \sqcup \iota_E^\ell$$

Next let us restrict the length of the paths used to compute $MOP_\circ$; for $n \geq 0$ define:

$$MOP_\circ^n(\ell) = \bigsqcup\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in path_\circ(\ell), |\vec{\ell}| < n\}$$

Clearly, $MOP_\circ(\ell) = \bigsqcup_n MOP_\circ^n(\ell)$ and to prove $MFP_\circ \sqsupseteq MOP_\circ$ is therefore suffices to prove

$$\forall n : MFP_\circ \sqsupseteq MOP_\circ^n$$

and we do so by numerical induction. The basis, $MFP_\circ \sqsupseteq MOP_\circ^0$, is trivial. The inductive step proceeds as follows:

$$MFP_\circ(\ell) \quad = \quad F(MFP_\circ)(\ell)$$

$$
\begin{aligned}
&= (\bigsqcup\{f_{\ell'}(MFP_\circ(\ell')) \mid (\ell',\ell) \in F\}) \sqcup \iota_E^\ell \\
&\sqsupseteq (\bigsqcup\{f_{\ell'}(MOP_\circ^n(\ell')) \mid (\ell',\ell) \in F\}) \sqcup \iota_E^\ell \\
&= (\bigsqcup\{f_{\ell'}(\bigsqcup\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in path_\circ(\ell'), |\vec{\ell}| < n\}) \mid (\ell',\ell) \in F\}) \sqcup \iota_E^\ell \\
&\sqsupseteq (\bigsqcup(\{\bigsqcup\{f_{\ell'}(f_{\vec{\ell}}(\iota)) \mid \vec{\ell} \in path_\circ(\ell'), |\vec{\ell}| < n\} \mid (\ell',\ell) \in F\}) \sqcup \iota_E^\ell \\
&= \bigsqcup(\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in path_\circ(\ell), 1 \leq |\vec{\ell}| \leq n\}) \sqcup \iota_E^\ell \\
&= MOP_\circ^{n+1}(\ell)
\end{aligned}
$$

where we have used the induction hypothesis to get the first inequality. This completes the proof of $MFP_\circ \sqsupseteq MOP_\circ$ and $MFP_\bullet \sqsupseteq MOP_\bullet$.

To prove the second part of the lemma we now assume that the framework is distributive. Consider ℓ in E or F. By assumption f_ℓ is distributive, that is $f_\ell(l_1 \sqcup l_2) = f_\ell(l_1) \sqcup f_\ell(l_2)$, and from Lemma A.9 of Appendix A it follows that

$$f_\ell(\bigsqcup Y) = \bigsqcup\{f_\ell(l) \mid l \in Y\}$$

whenever Y is non-empty. By assumption we also have $path_\circ(\ell) \neq \emptyset$ and it follows that

$$
\begin{aligned}
f_\ell(\bigsqcup\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in path_\circ(\ell)\}) &= \bigsqcup\{f_\ell(f_{\vec{\ell}}(\iota)) \mid \vec{\ell} \in path_\circ(\ell)\} \\
&= \bigsqcup\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in path_\bullet(\ell)\}
\end{aligned}
$$

and this shows that:

$$\forall \ell : f_\ell(MOP_\circ(\ell)) = MOP_\bullet(\ell)$$

Next we calculate:

$$
\begin{aligned}
MOP_\circ(\ell) &= \bigsqcup\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in path_\circ(\ell)\} \\
&= \bigsqcup\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in \bigcup\{path_\bullet(\ell') \mid (\ell',\ell) \in F\} \cup \{[\,] \mid \ell \in E\}\} \\
&= \bigsqcup(\{f_{\ell'}(f_{\vec{\ell}}(\iota)) \mid \vec{\ell} \in path_\circ(\ell'), (\ell',\ell) \in F\} \cup \{\iota \mid \ell \in E\}) \\
&= (\bigsqcup\{f_{\ell'}(\bigsqcup\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in path_\circ(\ell')\} \mid (\ell',\ell) \in F\}) \sqcup \iota_E^\ell \\
&= (\bigsqcup\{f_{\ell'}(MOP_\circ(\ell')) \mid (\ell',\ell) \in F\}) \sqcup \iota_E^\ell
\end{aligned}
$$

Together this shows that $(MOP_\circ, MOP_\bullet)$ is a solution to the data flow equations. Using Proposition A.10 of Appendix A and the fact that $(MFP_\circ, MFP_\bullet)$ is the least solution we get $MOP_\circ \sqsupseteq MFP_\circ$ and $MOP_\bullet \sqsupseteq MFP_\bullet$. Together with the results of the first part of the lemma we get $MOP_\circ = MFP_\circ$ and $MOP_\bullet = MFP_\bullet$. ∎

We shall leave it to Exercise 2.13 to show that the condition that $path_\circ(\ell) \neq \emptyset$ (for ℓ in E and F) does hold when the Monotone Framework is constructed from a program $S_\star$ in the manner of the earlier sections.

It is sometimes stated that the MOP solution is the desired solution and that one only uses the MFP solution because the MOP solution might not be

computable. In order to validate this belief we would need to prove that the MOP solution is semantically correct as was proved for the MFP solution in Section 2.2 in the case of Live Variables Analysis – in the case of Live Variables this is of course immediate since it is a Distributive Framework. We shall not do so because it is always possible to formulate the MOP solution as an MFP solution over a different property space (like $\mathcal{P}(L)$) and therefore little is lost by focusing on the fixed point approach to Monotone Frameworks. (Also note that $\mathcal{P}(L)$ satisfies the Ascending Chain Condition when L is finite.)

2.5 Interprocedural Analysis

The Data Flow Analysis techniques that have been presented in the previous sections are called *intraprocedural analyses* because they deal with simple languages without functions or procedures. It is somewhat more demanding to perform *interprocedural analyses* where functions and procedures are taken into account. Complications arise when ensuring that calls and returns match one another, when dealing with parameter mechanisms (and the aliasing that may result from call-by-reference) and when allowing procedures as parameters.

In this section we shall introduce some of the key techniques of interprocedural analysis. To keep things simple we just extend the WHILE language with top-level declarations of global mutually recursive procedures having a *call-by-value* parameter and a *call-by-result* parameter. The extension of the techniques to a language where procedures may have multiple call-by-value, call-by-result and call-by-value-result parameters is straightforward and so is the extension with local variable declarations (see Exercise 2.20); we shall freely use these extensions in examples.

Syntax of the procedure language. A program, $P_\star$, in the extended WHILE-language has the form

$$\texttt{begin}\ D_\star\ S_\star\ \texttt{end}$$

where $D_\star$ is a sequence of procedure declarations:

$$D ::= \texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x} \mid D\ D$$

Procedure names (denoted p) are syntactically distinct from variables (denoted x and y). The label ℓ_n of `is` marks the entry to the procedure body and the label ℓ_x of `end` marks the exit from the procedure body. The syntax of statements is extended with:

$$S ::= \cdots \mid [\texttt{call}\ p(a, z)]^{\ell_c}_{\ell_r}$$

The call statement has two labels: ℓ_c will be used for the call of the procedure and ℓ_r will be used for the associated return; the actual parameters are a and z.

The language is statically scoped, the parameter mechanism is call-by-value for the first parameter and call-by-result for the second parameter and the procedures may be mutually recursive. We shall *assume* throughout that the program is uniquely labelled (and hence label consistent); also we shall *assume* that only procedures that have been declared in $D_\star$ are ever called and that $D_\star$ does not contain two definitions of the same procedure name.

Example 2.33 Consider the following program calculating the Fibonacci number of the positive integer stored in `x` and returning it in `y`:

```
begin   proc fib(val z, u, res v) is¹
                  if [z<3]² then [v:=u+1]³
                  else ([call fib(z-1,u,v)]⁴₅; [call fib(z-2,v,v)]⁶₇)
        end⁸;
        [call fib(x,0,y)]⁹₁₀
end
```

It uses the procedure `fib` that returns in `v` the Fibonacci number of `z` plus the value of `u`. Both `x` and `y` are global variables whereas `z`, `u` and `v` are formal parameters and hence local variables. ∎

Flow graphs for statements. The next step is to extend the definitions of the functions *init*, *final*, *blocks*, *labels*, and *flow* to specify the flow graphs also for the procedure language. For the new statement we take:

$$
\begin{array}{rcl}
\mathit{init}([\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}) & = & \ell_c \\
\mathit{final}([\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}) & = & \{\ell_r\} \\
\mathit{blocks}([\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}) & = & \{[\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}\} \\
\mathit{labels}([\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}) & = & \{\ell_c, \ell_r\} \\
\mathit{flow}([\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}) & = & \{(\ell_c;\ell_n), (\ell_x;\ell_r)\} \\
 & & \text{if } \texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x} \\
 & & \text{is in } D_\star
\end{array}
$$

Here $(\ell_c;\ell_n)$ and $(\ell_x;\ell_r)$ are new kinds of flows:

- $(\ell_c;\ell_n)$ is the flow corresponding to *calling* a procedure at ℓ_c and with ℓ_n being the entry point for the procedure body, and
- $(\ell_x;\ell_r)$ is the flow corresponding to exiting a procedure body at ℓ_x and *returning* to the call at ℓ_r.

The definition of $\mathit{flow}([\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r})$ exploits the fact that the syntax of procedure calls only allows us to use the (constant) name of a procedure

defined in the program; had we been allowed to use a variable that denotes a procedure (e.g. because it was a formal parameter to some procedure or because it was a variable being assigned some procedure) then it would be much harder to define $\mathit{flow}([\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r})$. This is often called the *dynamic dispatch problem* and we shall deal with it in Chapter 3.

Flow graphs for programs. Next consider the program $P_\star$ of the form $\texttt{begin}\ D_\star\ S_\star\ \texttt{end}$. For each procedure declaration $\texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x}$ we set

$$\begin{aligned}
\mathit{init}(p) &= \ell_n \\
\mathit{final}(p) &= \{\ell_x\} \\
\mathit{blocks}(p) &= \{\texttt{is}^{\ell_n}, \texttt{end}^{\ell_x}\} \cup \mathit{blocks}(S) \\
\mathit{labels}(p) &= \{\ell_n, \ell_x\} \cup \mathit{labels}(S) \\
\mathit{flow}(p) &= \{(\ell_n, \mathit{init}(S))\} \cup \mathit{flow}(S) \cup \{(\ell, \ell_x) \mid \ell \in \mathit{final}(S)\}
\end{aligned}$$

and for the entire program $P_\star$ we set

$$\begin{aligned}
\mathit{init}_\star &= \mathit{init}(S_\star) \\
\mathit{final}_\star &= \mathit{final}(S_\star) \\
\mathit{blocks}_\star &= \bigcup\{\mathit{blocks}(p) \mid \texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x} \text{ is in } D_\star\} \\
&\quad \cup\ \mathit{blocks}(S_\star) \\
\mathit{labels}_\star &= \bigcup\{\mathit{labels}(p) \mid \texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x} \text{ is in } D_\star\} \\
&\quad \cup\ \mathit{labels}(S_\star) \\
\mathit{flow}_\star &= \bigcup\{\mathit{flow}(p) \mid \texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x} \text{ is in } D_\star\} \\
&\quad \cup\ \mathit{flow}(S_\star)
\end{aligned}$$

as well as $\mathbf{Lab}_\star = \mathit{labels}_\star$.

We shall also need to define a notion of *interprocedural flow*

$$\begin{aligned}
\mathit{inter\text{-}flow}_\star &= \{(\ell_c, \ell_n, \ell_x, \ell_r) \mid P_\star \text{ contains } [\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r} \\
&\qquad \text{as well as } \texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x}\}
\end{aligned}$$

that clearly indicates the relationship between the labels of a procedure call and the corresponding procedure body. This information will be used later to analyse procedure calls and returns more precisely than is otherwise possible. Indeed, suppose that $\mathit{inter\text{-}flow}_\star$ contains $(\ell^i_c, \ell_n, \ell_x, \ell^i_r)$ for $i = 1,2$ in which case $\mathit{flow}_\star$ contains $(\ell^i_c; \ell_n)$ and $(\ell_x; \ell^i_r)$ for $i = 1,2$. But this "gives rise to" the four tuples $(\ell^i_c, \ell_n, \ell_x, \ell^j_r)$ for $i = 1,2$ and $j = 1,2$ and only the tuples with $i = j$ match the return with the call: these tuples are exactly the ones in $\mathit{inter\text{-}flow}_\star$.

Figure 2.7: Flow graph for the Fibonacci program.

Example 2.34 For the Fibonacci program considered in Example 2.33 we have

$$\begin{aligned} \mathit{flow}_\star &= \{(1,2),(2,3),(3,8),\\ &\quad (2,4),(4;1),(8;5),(5,6),(6;1),(8;7),(7,8),\\ &\quad (9;1),(8;10)\} \\ \mathit{inter\text{-}flow}_\star &= \{(9,1,8,10),(4,1,8,5),(6,1,8,7)\} \end{aligned}$$

and $\mathit{init}_\star = 9$ and $\mathit{final}_\star = \{10\}$. The corresponding flow graph is illustrated in Figure 2.7. ■

For a *forward analysis* we use $F = \mathit{flow}_\star$ and $E = \{\mathit{init}_\star\}$ much as before and we introduce a new "metavariable" $IF = \mathit{inter\text{-}flow}_\star$ for the interprocedural flow; for a *backward analysis* we use $F = \mathit{flow}_\star^R$, $E = \mathit{final}_\star$ and $IF = \mathit{inter\text{-}flow}_\star^R$. Most of the explanations in the sequel will focus on forward analyses.

2.5.1 Structural Operational Semantics

We shall now show how the semantics of WHILE can be extended to cope with the new constructs. To ensure that the language allows *local data* in

procedures we shall need to distinguish between the values assigned to different incarnations of the same variable and for this we introduce an infinite set of *locations* (or addresses):

$$\xi \in \mathbf{Loc} \qquad \text{locations}$$

An *environment*, ρ, will map the variables in the current scope to their locations, and a *store*, ς, will then specify the values of these locations:

$$\begin{array}{lllll} \rho & \in & \mathbf{Env} & = & \mathbf{Var}_\star \to \mathbf{Loc} \qquad \text{environments} \\ \varsigma & \in & \mathbf{Store} & = & \mathbf{Loc} \to_{\text{fin}} \mathbf{Z} \qquad \text{stores} \end{array}$$

Here $\mathbf{Var}_\star$ is the (finite) set of variables occurring in the program and $\mathbf{Loc} \to_{\text{fin}} \mathbf{Z}$ denotes the set of *partial* functions from $\mathbf{Loc}$ to $\mathbf{Z}$ that have a *finite* domain. Thus the previously used states $\sigma \in \mathbf{State} = \mathbf{Var}_\star \to \mathbf{Z}$ have been replaced by the two mappings ρ and ς and can be reconstructed as $\sigma = \varsigma \circ \rho$: to determine the value of a variable x we first determine its location $\xi = \rho(x)$ and next the value $\varsigma(\xi)$ stored in that location. For this to work it is essential that $\varsigma \circ \rho : \mathbf{Var}_\star \to \mathbf{Z}$ is a *total* function rather than a partial function; in other words, we demand that $ran(\rho) \subseteq dom(\varsigma)$ where $ran(\rho) = \{\rho(x) \mid x \in \mathbf{Var}_\star\}$ and $dom(\varsigma) = \{\xi \mid \varsigma \text{ is defined on } \xi\}$.

The locations of the global variables of the program $P_\star$ are given by a *top-level environment* denoted $\rho_\star$; we shall *assume* that it maps all variables to unique locations. The semantics of statements is now given relative to modifications of this environment. The transitions have the general form

$$\rho \vdash_\star \langle S, \varsigma \rangle \to \langle S', \varsigma' \rangle$$

in case that the computation does not terminate in one step, and the form

$$\rho \vdash_\star \langle S, \varsigma \rangle \to \varsigma'$$

in case that it does terminate in one step. It is fairly straightforward to rewrite the semantics of WHILE given in Table 2.6 to have this form; as an example the clause [*ass*] for assignments becomes:

$$\rho \vdash_\star \langle x := a, \varsigma \rangle \to \varsigma[\rho(x) \mapsto \mathcal{A}[\![a]\!](\varsigma \circ \rho)] \quad \text{if } \varsigma \circ \rho \text{ is total}$$

Note that there is no need to modify the semantics of arithmetic and boolean expressions.

For procedure calls we make use of the top-level environment, $\rho_\star$, and we take:

$$\rho \vdash_\star \langle [\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}, \varsigma\rangle \rightarrow$$
$$\langle \texttt{bind}\ \rho_\star[x \mapsto \xi_1, y \mapsto \xi_2]\ \texttt{in}\ S\ \texttt{then}\ z{:=}y, \varsigma[\xi_1 \mapsto \mathcal{A}[\![a]\!](\varsigma \circ \rho), \xi_2 \mapsto v]\rangle$$
$$\text{where } \xi_1, \xi_2 \notin dom(\varsigma), v \in \mathbf{Z}$$
$$\text{and } \texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x} \text{ is in } D_\star$$

The idea is that we allocate new locations ξ_1 and ξ_2 for the formal parameters x and y, and we then make use of a bind-construct to combine the procedure body S with the environment $\rho_\star[x \mapsto \xi_1, y \mapsto \xi_2]$ in which it must be executed and we also record that the final value of y must be returned in the actual parameter z. At the same time the store is updated such that the new location for x is mapped to the value of the actual parameter a whereas we do not control the initial value, v, of the new location for y. The bind-construct is only needed to ensure that we have static scope rules and its semantics is as follows:

$$\frac{\rho' \vdash_\star \langle S, \varsigma\rangle \rightarrow \langle S', \varsigma'\rangle}{\rho \vdash_\star \langle \texttt{bind}\ \rho'\ \texttt{in}\ S\ \texttt{then}\ z{:=}y, \varsigma\rangle \rightarrow \langle \texttt{bind}\ \rho'\ \texttt{in}\ S'\ \texttt{then}\ z{:=}y, \varsigma'\rangle}$$

$$\frac{\rho' \vdash_\star \langle S, \varsigma\rangle \rightarrow \varsigma'}{\rho \vdash_\star \langle \texttt{bind}\ \rho'\ \texttt{in}\ S\ \texttt{then}\ z{:=}y, \varsigma\rangle \rightarrow \varsigma'[\rho(z) \mapsto \varsigma'(\rho'(y))]}$$

The first rule expresses that executing one step of the body of the construct amounts to executing one step of the construct itself; note that we use the local environment when executing the body. The second rule expresses that when the execution of the body finishes then so does execution of the construct itself and we update the value of the global variable z to be that of the local variable y; furthermore, there is no need for the local environment ρ' to be retained as subsequent computations will use the previous environment ρ.

Remark. Although the semantics works with two mappings, an environment and a store, it is often the case that the analysis abstracts the state, i.e. the composition of the environment and the store. The correctness of the analysis will then have to relate the abstract state both to the environment and the store.

The correctness result will often be expressed in the style of Section 2.2: information obtained by analysing the original program will remain correct under execution of the program. The semantics presented above deviates from that of the WHILE-language in that it introduces the bind-construct which is only used in the intermediate configurations. So in order to prove the correctness result we will also need to specify how to analyse the bind-construct. We refer to Chapter 3 for an illustration of how to do this. ∎

2.5.2 Intraprocedural versus Interprocedural Analysis

To appreciate why interprocedural analysis is harder than intraprocedural analysis let us begin by just naively using the techniques from the previous sections. For this we suppose that:

- for each procedure call $[\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}$ we have two transfer functions f_{ℓ_c} and f_{ℓ_r} corresponding to calling the procedure and returning from the call, and
- for each procedure definition $\texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x}$ we have two transfer functions f_{ℓ_n} and f_{ℓ_x} corresponding to entering and exiting the procedure body.

A naive formulation. Given an instance $(L, \mathcal{F}, F, E, \iota, f_\cdot)$ of a Monotone Framework we shall now treat the two kinds of flow $((\ell_1, \ell_2)$ versus $(\ell_c; \ell_n)$ and $(\ell_x; \ell_r))$ in the same way: we interpret the semi-colon as standing for a comma. While a Monotone Framework is allowed to interpret all transfer functions freely, we shall for now naively assume that the two transfer functions associated with procedure definitions are the identity functions, and that the two transfer functions associated with each procedure call are also the identity functions, thus effectively ignoring the parameter-passing.

We now obtain an equation system of the form considered in the previous sections:

$$\begin{aligned} A_\bullet(\ell) &= f_\ell(A_\circ(\ell)) \\ A_\circ(\ell) &= \bigsqcup\{A_\bullet(\ell') \mid (\ell', \ell) \in F \text{ or } (\ell'; \ell) \in F\} \sqcup \iota^\ell_E \end{aligned}$$

Here ι^ℓ_E is as in Section 2.3:

$$\iota^\ell_E = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases}$$

When inspecting this equation system is should be apparent that both procedure calls $(\ell_c; \ell_n)$ and procedure returns $(\ell_x; \ell_r)$ are treated like goto's: there is no mechanism for ensuring that information flowing along $(\ell_c; \ell_n)$ from a call to a procedure only flows back along $(\ell_x; \ell_r)$ from the procedure to the *same* call. (Indeed, nowhere does the formulation consult the interprocedural flow, *IF*.) Expressed in terms of the flow graph in Figure 2.7, there is nothing preventing us from considering a path like $[9, 1, 2, 4, 1, 2, 3, 8, 10]$ that does not correspond to a run of the program. Intuitively, the equation system considers a much too large set of "paths" through the program and hence will be grossly imprecise (although formally on the safe side).

Valid paths. A natural way to overcome this shortcoming is to somehow restrict the attention to paths that have the proper nesting of procedure calls and exits. We shall explore this idea in the context of redefining the MOP solution of Section 2.4 to only take the proper set of paths into account, thereby defining an *MVP solution* (for *Meet over all Valid Paths*).

So consider a program $P_\star$ of the form **begin** $D_\star$ $S_\star$ **end**. A path is said to be a *complete path* from ℓ_1 to ℓ_2 in $P_\star$ if it is has proper nesting of procedure entries and exits and such that a procedure returns to the point where it was called. These paths are generated by the nonterminal CP_{ℓ_1,ℓ_2} according to the following productions:

$CP_{\ell_1,\ell_2} \longrightarrow \ell_1$	whenever $\ell_1 = \ell_2$
$CP_{\ell_1,\ell_3} \longrightarrow \ell_1, CP_{\ell_2,\ell_3}$	whenever $(\ell_1, \ell_2) \in F$; for a forward analysis this means that $(\ell_1, \ell_2) \in \mathit{flow}_\star$
$CP_{\ell_c,\ell} \longrightarrow \ell_c, CP_{\ell_n,\ell_x}, CP_{\ell_r,\ell}$	whenever $(\ell_c, \ell_n, \ell_x, \ell_r) \in IF$; for a forward analysis this means that $P_\star$ contains $[\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}$ and $\texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x}$

The matching of calls and returns is ensured by the last kind of productions: the flows $(\ell_c; \ell_n)$ and $(\ell_x; \ell_r)$ are forced to obey a parenthesis structure in that ℓ_c, ℓ_n will be in the generated path only if there is a matching occurrence of ℓ_x, ℓ_r – and vice versa. Hence for a forward analysis, a terminating computation will give rise to a complete path from $\mathit{init}_\star$ to one of the labels of $\mathit{final}_\star$. Note that the grammar constructed above will only have a finite set of nonterminals because there is only a finite set of labels in $P_\star$.

Example 2.35 For the Fibonacci program of Example 2.33 we obtain the following grammar (using forward flow and ignoring the parts not reachable from $CP_{9,10}$):

$$
\begin{array}{rcl@{\qquad}rcl}
CP_{9,10} & \longrightarrow & 9, CP_{1,8}, CP_{10,10} & CP_{3,8} & \longrightarrow & 3, CP_{8,8} \\
CP_{10,10} & \longrightarrow & 10 & CP_{8,8} & \longrightarrow & 8 \\
CP_{1,8} & \longrightarrow & 1, CP_{2,8} & CP_{4,8} & \longrightarrow & 4, CP_{1,8}, CP_{5,8} \\
CP_{2,8} & \longrightarrow & 2, CP_{3,8} & CP_{5,8} & \longrightarrow & 5, CP_{6,8} \\
CP_{2,8} & \longrightarrow & 2, CP_{4,8} & CP_{6,8} & \longrightarrow & 6, CP_{1,8}, CP_{7,8} \\
 & & & CP_{7,8} & \longrightarrow & 7, CP_{8,8}
\end{array}
$$

It is now easy to verify that the path $[9, 1, 2, 4, 1, 2, 3, 8, 5, 6, 1, 2, 3, 8, 7, 8, 10]$ is generated by $CP_{9,10}$ whereas the path $[9, 1, 2, 4, 1, 2, 3, 8, 10]$ is not. ∎

A path is said to be a *valid path* if it starts at an extremal node of $P_\star$ and if all the procedure exits match the procedure entries but it is possible that

some procedures are entered but not yet exited. This will obviously include all prefixes of the complete paths starting in E but we also have to take into account prefixes of computations that might not terminate. To specify the valid paths we therefore construct another grammar with productions:

$$
\begin{array}{ll}
VP_\star \longrightarrow VP_{\ell_1,\ell_2} & \text{whenever } \ell_1 \in E \text{ and } \ell_2 \in \mathbf{Lab}_\star \\
VP_{\ell_1,\ell_2} \longrightarrow \ell_1 & \text{whenever } \ell_1 = \ell_2 \\
VP_{\ell_1,\ell_3} \longrightarrow \ell_1, VP_{\ell_2,\ell_3} & \text{whenever } (\ell_1,\ell_2) \in F \\
VP_{\ell_c,\ell} \longrightarrow \ell_c, CP_{\ell_n,\ell_x}, VP_{\ell_r,\ell} & \text{whenever } (\ell_c,\ell_n,\ell_x,\ell_r) \in IF \\
VP_{\ell_c,\ell} \longrightarrow \ell_c, VP_{\ell_n,\ell} & \text{whenever } (\ell_c,\ell_n,\ell_x,\ell_r) \in IF
\end{array}
$$

The valid paths will then be generated by the nonterminal $VP_\star$. For a forward analysis, to go from the label ℓ_c of a procedure call to the program point ℓ there are two possibilities. One is that the call initiated at ℓ_c terminates before reaching ℓ and this corresponds to the second last kind of production where we use the nonterminal CP_{ℓ_n,ℓ_x} to generate the complete path corresponding to executing the procedure body. The other possibility is that ℓ is reached before the call terminates and this corresponds to the last kind of production where we simply use $VP_{\ell_n,\ell}$ to generate a valid path in the procedure body.

We can now modify the two sets of paths defined in Section 2.3 as follows (keeping in mind that the definitions are implicitly parameterised on F, E and IF):

$$
\begin{array}{rcl}
vpath_\circ(\ell) & = & \{[\ell_1,\cdots,\ell_{n-1}] \mid n \geq 1 \wedge \ell_n = \ell \wedge [\ell_1,\cdots,\ell_n] \text{ is a valid path}\} \\
vpath_\bullet(\ell) & = & \{[\ell_1,\cdots,\ell_n] \mid n \geq 1 \wedge \ell_n = \ell \wedge [\ell_1,\cdots,\ell_n] \text{ is a valid path}\}
\end{array}
$$

Clearly the sets of paths are smaller than what would have resulted if we had merely regarded $(\ell_1;\ell_2)$ as standing for (ℓ_1,ℓ_2) and had used the notions $path_\circ(\ell)$ and $path_\bullet(\ell)$ of Subsection 2.4.2.

Using valid paths we now define the *MVP solution* as follows:

$$MVP_\circ(\ell) = \bigsqcup\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in vpath_\circ(\ell)\}$$

$$MVP_\bullet(\ell) = \bigsqcup\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in vpath_\bullet(\ell)\}$$

Since the sets of paths are smaller than in the similar definitions in Subsection 2.4.2, we clearly have $MVP_\circ(\ell) \sqsubseteq MOP_\circ(\ell)$ and $MVP_\bullet(\ell) \sqsubseteq MOP_\bullet(\ell)$ for all ℓ.

2.5.3 Making Context Explicit

The MVP solution may be undecidable for lattices of finite height, just as was the case for the MOP solution, so we now have to reconsider the MFP solution

and how to avoid taking too many invalid paths. An obvious approach is to encode information about the paths taken into the data flow properties themselves; to this end we introduce *context information*:

$$\delta \in \Delta \qquad \text{context information}$$

The context may simply be an encoding of the path taken but we shall see in Subsection 2.5.5 that there are other possibilities. We shall now show how an instance of a Monotone Framework (as introduced in Section 2.3) can be extended to take context into account.

The intraprocedural fragment. Consider an instance $(L, \mathcal{F}, F, E, \iota, f_\cdot)$ of a Monotone Framework. We shall now construct an instance

$$(\widehat{L}, \widehat{\mathcal{F}}, F, E, \widehat{\iota}, \widehat{f}_\cdot)$$

of an *embellished Monotone Framework* that takes context into account. We begin with the parts of its definition that are independent of the actual choice of Δ, i.e. the parts that correspond to the intraprocedural analysis:

- $\widehat{L} = \Delta \to L$;
- the transfer functions in $\widehat{\mathcal{F}}$ are monotone; and
- each transfer function $\widehat{f_\ell}$ is given by $\widehat{f_\ell}(\widehat{l})(\delta) = f_\ell(\widehat{l}\ (\delta))$.

In other words, the new instance applies the transfer functions of the original instance in a pointwise fashion.

Ignoring procedures, the data flow equations will take the form displayed earlier:

$$\begin{array}{lcl} A_\bullet(\ell) & = & \widehat{f_\ell}(A_\circ(\ell)) \\ & & \text{for all labels that do not label a procedure call} \\ & & \text{(i.e. that do not occur as first or fourth components} \\ & & \text{of a tuple in } IF) \\ A_\circ(\ell) & = & \bigsqcup\{A_\bullet(\ell') \mid (\ell', \ell) \in F \text{ or } (\ell'; \ell) \in F\} \sqcup \widehat{\iota_E^\ell} \\ & & \text{for all labels (including those that label procedure calls)} \end{array}$$

Example 2.36 Let $(L_{\mathsf{sign}}, \mathcal{F}_{\mathsf{sign}}, F, E, \iota_{\mathsf{sign}}, f_\cdot^{\mathsf{sign}})$ be an instance of a Monotone Framework specifying a *Detection of Signs Analysis* (see Exercise 2.15) and assume that

$$L_{\mathsf{sign}} = \mathcal{P}(\mathbf{Var}_\star \to \mathbf{Sign})$$

where $\mathbf{Sign} = \{-, 0, +\}$. Thus L_{sign} describes sets of abstract states σ^{sign} mapping variables to their possible signs. The transfer function f_ℓ^{sign} associated with the assignment $[x := a]^\ell$ will now be written as

$$f_\ell^{\mathsf{sign}}(Y) = \bigcup\{\phi_\ell^{\mathsf{sign}}(\sigma^{\mathsf{sign}}) \mid \sigma^{\mathsf{sign}} \in Y\}$$

where $Y \subseteq \mathbf{Var}_\star \to \mathbf{Sign}$ and

$$\phi_\ell^{\mathsf{sign}}(\sigma^{\mathsf{sign}}) = \{\sigma^{\mathsf{sign}}[x \mapsto s] \mid s \in \mathcal{A}_{\mathsf{sign}}[\![a]\!](\sigma^{\mathsf{sign}})\}$$

Here $\mathcal{A}_{\mathsf{sign}} : \mathbf{AExp} \to (\mathbf{Var}_\star \to \mathbf{Sign}) \to \mathcal{P}(\mathbf{Sign})$ specifies the analysis of arithmetic expressions. The transfer functions for tests and `skip`-statements are the identity functions.

Given a set Δ of contexts, the embellished Monotone Framework will have

$$\widehat{L_{\mathsf{sign}}} = \Delta \to L_{\mathsf{sign}}$$

but we shall prefer the following isomorphic definition

$$\widehat{L_{\mathsf{sign}}} = \mathcal{P}(\Delta \times (\mathbf{Var}_\star \to \mathbf{Sign}))$$

Thus $\widehat{L_{\mathsf{sign}}}$ describes sets of pairs of context and abstract states. The transfer function associated with the assignment $[x := a]^\ell$ will now be:

$$\widehat{f_\ell^{\mathsf{sign}}}(Z) = \bigcup\{\{\delta\} \times \phi_\ell^{\mathsf{sign}}(\sigma^{\mathsf{sign}}) \mid (\delta, \sigma^{\mathsf{sign}}) \in Z\}$$

In subsequent examples we shall further develop this analysis. ∎

The interprocedural fragment. It remains to formulate the data flow equations corresponding to procedures.

For a procedure definition `proc` p(`val` x, `res` y) `is`$^{\ell_n}$ S `end`$^{\ell_x}$ we have two transfer functions:

$$\widehat{f_{\ell_n}}, \widehat{f_{\ell_x}} : (\Delta \to L) \to (\Delta \to L)$$

In the case of our simple language we shall prefer to take both of these transfer functions to be the identity function; i.e.

$$\begin{aligned} \widehat{f_{\ell_n}}(\widehat{l}) &= \widehat{l} \\ \widehat{f_{\ell_x}}(\widehat{l}) &= \widehat{l} \end{aligned}$$

for all $\widehat{l} \in \widehat{L}$. Hence the effect of procedure entry is handled by the transfer function for procedure call (considered below) and similarly the effect of procedure exit is handled by the transfer function for procedure return (also

Figure 2.8: Analysis of procedure call: the forward case.

considered below). For more advanced languages where many semantic actions take place at procedure entry or exit it may be preferable to reconsider this decision.

For a procedure call $(\ell_c, \ell_n, \ell_x, \ell_r) \in IF$ we shall define two transfer functions. In our explanation we shall concentrate on the case of forward analyses where $P_\star$ contains $[\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}$ as well as $\texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x}$. Corresponding to the actual call we have the transfer function

$$\widehat{f^1_{\ell_c}} : (\Delta \to L) \to (\Delta \to L)$$

and it is used in the equation:

$$A_\bullet(\ell_c) \quad = \quad \widehat{f^1_{\ell_c}}(A_\circ(\ell_c)) \ \text{ for all } (\ell_c, \ell_n, \ell_x, \ell_r) \in IF$$

In other words, the transfer function modifies the data flow properties (and the context) as required for passing to the procedure entry.

Corresponding to the return we have the transfer function

$$\widehat{f^2_{\ell_c,\ell_r}} : (\Delta \to L) \times (\Delta \to L) \to (\Delta \to L)$$

and it is used in the equation:

$$A_\bullet(\ell_r) \quad = \quad \widehat{f^2_{\ell_c,\ell_r}}(A_\circ(\ell_c), A_\circ(\ell_r)) \ \text{ for all } (\ell_c, \ell_n, \ell_x, \ell_r) \in IF$$

The first parameter of $\widehat{f^2_{\ell_c,\ell_r}}$ describes the data flow properties at the call point for the procedure and the second parameter describes the properties at the exit from the procedure body. Ignoring the first parameter, the transfer function modifies the data flow properties (and the context) as required for passing back from the procedure exit. The purpose of the first parameter is

Figure 2.9: Analysis of procedure call: ignoring calling context.

to recover some of the information (data flow properties as well as context information) that was available before the actual call; how this is done depends on the actual choice of the set, Δ, of context information and we shall return to this shortly. Figure 2.8 illustrates the flow of data in the analysis of the procedure call.

Variations. The functionality and use of $\widehat{f^2_{\ell_c,\ell_r}}$ (as well as Figure 2.8) is sufficiently general that it allows us to deal with most of the scenarios found in the literature. A simple example being the possibility to define

$$\widehat{f^2_{\ell_c,\ell_r}}\,(\widehat{l},\widehat{l'}) = \widehat{f^2_{\ell_r}}\,(\widehat{l'})$$

thereby completely ignoring the information before the call; this is illustrated in Figure 2.9.

A somewhat more interesting example is the ability to define

$$\widehat{f^2_{\ell_c,\ell_r}}\,(\widehat{l},\widehat{l'}) = \widehat{f^{2A}_{\ell_c,\ell_r}}\,(\widehat{l}) \sqcup \widehat{f^{2B}_{\ell_c,\ell_r}}\,(\widehat{l'})$$

thereby allowing a simple combination of the information coming back from the call with the information pertaining before the call. This form is illustrated in Figure 2.10 and is often motivated on the grounds that $\widehat{f^{2A}_{\ell_c,\ell_r}}$ copies data that is local to the calling procedure whereas $\widehat{f^{2B}_{\ell_c,\ell_r}}$ copies information that is global. (It may be worth noticing that the function $\widehat{f^2_{\ell_c,\ell_r}}$ is completely additive if and only if it can be written in this form with $\widehat{f^{2A}_{\ell_c,\ell_r}}$ and $\widehat{f^{2B}_{\ell_c,\ell_r}}$ being completely additive.)

Context-sensitive versus context-insensitive. So far we have criticised the naive approach because it was unable to maintain the proper

Figure 2.10: Analysis of procedure call: merging of context.

relationship between procedure calls and procedure returns. A related criticism of the naive approach is that it cannot distinguish between the different calls of a procedure. The information about calling states is combined for all call sites, the procedure body is analysed only once using this combined information, and the resulting information about the set of return states is used at all return points. The phrase *context-insensitive* is often used to refer to this shortcoming.

The use of non-trivial context information not only helps to avoid the first criticism but also the second: if there are two different calls but they are reached with different contexts, δ_1 and δ_2, then all information obtained from the procedure will be clearly related to δ_1 or δ_2 and no undesired combination or "cross-over" will take place. The phrase *context-sensitive* is often used to refer to this ability.

Clearly a context-sensitive analysis is more precise than a context-insensitive analysis but at the same time it is also likely to be more costly. The choice between which technique to use amounts to a careful balance between precision and efficiency.

2.5.4 Call Strings as Context

To complete the design of the analysis of the program we must choose the set, Δ, of context information and also specify the extremal value, $\widehat{\iota}$, and define the two transfer functions associated with procedure calls. In this subsection we shall consider two approaches based on call strings and our explanation will be in terms of forward analyses.

Call strings of unbounded length. As the first possibility we simply encode the path taken; however, since our main interest is with pro-

cedure calls we shall only record flows of the form $(\ell_c; \ell_n)$ corresponding to a procedure call. Formally we take

$$\Delta = \mathbf{Lab}^*$$

where the most recent label ℓ_c of a procedure call is at the right end (just as was the case for valid paths and paths); elements of Δ are called *call strings*. We then define the extremal value $\widehat{\iota}$ by the formula

$$\widehat{\iota}(\delta) = \begin{cases} \iota & \text{if } \delta = \Lambda \\ \bot & \text{otherwise} \end{cases}$$

where Λ is the empty sequence corresponding to the fact that there are no pending procedure calls when the program starts execution; ι is the extremal value available from the underlying Monotone Framework.

Example 2.37 For the Fibonacci program of Example 2.33 the following call strings will be of interest:

$$\Lambda, [9], [9,4], [9,6], [9,4,4], [9,4,6], [9,6,4], [9,6,6], \cdots$$

corresponding to the cases with $0, 1, 2, 3, \cdots$ pending procedure calls. ∎

For a procedure call $(\ell_c, \ell_n, \ell_x, \ell_r) \in \mathit{IF}$, amounting to $[\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}$ in the case of a forward analysis, we define the transfer function $\widehat{f^1_{\ell_c}}$ such that $\widehat{f^1_{\ell_c}}(\widehat{l})([\delta, \ell_c]) = f^1_{\ell_c}(\widehat{l}(\delta))$ where $[\delta, \ell_c]$ denotes the path obtained by appending ℓ_c to δ (so as to reflect that now we enter the body of the procedure) and the function $f^1_{\ell_c} : L \to L$ describes how the property is modified. This is achieved by setting

$$\widehat{f^1_{\ell_c}}(\widehat{l})(\delta') = \begin{cases} f^1_{\ell_c}(\widehat{l}(\delta)) & \text{when } \delta' = [\delta, \ell_c] \\ \bot & \text{otherwise} \end{cases}$$

which takes care of the special case of empty paths.

Next we define the transfer function $\widehat{f^2_{\ell_c,\ell_r}}$ corresponding to returning from the procedure call:

$$\widehat{f^2_{\ell_c,\ell_r}}(\widehat{l}, \widehat{l'})(\delta) = f^2_{\ell_c,\ell_r}(\widehat{l}(\delta), \widehat{l'}([\delta, \ell_c]))$$

Here the information $\widehat{l}$ from the original call is combined with information $\widehat{l'}$ from the procedure exit using the function $f^2_{\ell_c,\ell_r} : L \times L \to L$. However, only information corresponding to the *same* contexts for call point ℓ_c is combined: this is ensured by the two occurrences of δ in the right hand side of the above formula.

Example 2.38 Let us return to the Detection of Signs Analysis of Example 2.36. For a procedure call $[\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}$ where p is declared by $\texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x}$ we may take:

$$\widehat{f^{\mathsf{sign1}}_{\ell_c}}(Z) = \bigcup\{\{\delta'\} \times \phi^{\mathsf{sign1}}_{\ell_c}(\sigma^{\mathsf{sign}}) \mid (\delta, \sigma^{\mathsf{sign}}) \in Z \ \wedge\ \delta' = [\delta, \ell_c]\}$$

$$\phi^{\mathsf{sign1}}_{\ell_c}(\sigma^{\mathsf{sign}}) = \{\sigma^{\mathsf{sign}}[x \mapsto s][y \mapsto s'] \mid s \in \mathcal{A}_{\mathsf{sign}}[\![a]\!](\sigma^{\mathsf{sign}}) \ \wedge\ s' \in \{-, 0, +\}\}$$

When returning from the procedure call we take:

$$\widehat{f^{\mathsf{sign2}}_{\ell_c,\ell_r}}(Z, Z') = \bigcup\{\{\delta\} \times \phi^{\mathsf{sign2}}_{\ell_c,\ell_r}(\sigma^{\mathsf{sign}}_1, \sigma^{\mathsf{sign}}_2) \mid (\delta, \sigma^{\mathsf{sign}}_1) \in Z \ \wedge$$
$$\wedge\ (\delta', \sigma^{\mathsf{sign}}_2) \in Z' \ \wedge\ \delta' = [\delta, \ell_c]\}$$

$$\phi^{\mathsf{sign2}}_{\ell_c,\ell_r}(\sigma^{\mathsf{sign}}_1, \sigma^{\mathsf{sign}}_2) = \{\sigma^{\mathsf{sign}}_2[x \mapsto \sigma^{\mathsf{sign}}_1(x); y \mapsto \sigma^{\mathsf{sign}}_1(y); z \mapsto \sigma^{\mathsf{sign}}_2(y)]\}$$

Thus we extract all the information from the procedure body except for the information about the formal parameters x and y and the actual parameter z. For the formal parameters we rely on the information available before the current call which is still correct and for the actual parameter we perform the required update of the information. Note that to facilitate this definition it is crucial that the transfer function $\widehat{f^2_{\ell_c,\ell_r}}$ takes two arguments: information from the call point as well as from the procedure exit. ∎

Call strings of bounded length. Clearly the call strings can become arbitrarily long because the procedures may be recursive. It is therefore customary to restrict their length to be at most k for some number $k \geq 0$; the idea being that only the last k calls are recorded. We write this as

$$\Delta = \mathbf{Lab}^{\leq k}$$

and we still take the extremal value $\widehat{\iota}$ to be defined by the formula

$$\widehat{\iota}(\delta) = \begin{cases} \iota & \text{if } \delta = \Lambda \\ \bot & \text{otherwise} \end{cases}$$

Note that in the case $k = 0$ we have $\Delta = \{\Lambda\}$ which is equivalent to having no context information.

Example 2.39 Consider the Fibonacci program of Example 2.33 and assume that we are only interested in recording the last call, i.e. $k = 1$. Then the call strings of interest are:

$$\Lambda, [9], [4], [6]$$

Alternatively, we may choose to record the last two calls, i.e. $k = 2$, in which case the following call strings are of interest:

$$\Lambda, [9], [9,4], [9,6], [4,4], [4,6], [6,4], [6,6]$$

In general, we would expect an analysis using these 8 contexts to be more precise than one using the 4 different contexts displayed above. ■

We shall now present the transfer functions for the general case where call strings have length at most k. The transfer function $\widehat{f^1_{\ell_c}}$ for procedure call is redefined by

$$\widehat{f^1_{\ell_c}}(\widehat{l})(\delta') = \bigsqcup\{f^1_{\ell_c}(\widehat{l}(\delta)) \mid \delta' = \lceil \delta, \ell_c \rceil_k\}$$

where $\lceil \delta, \ell_c \rceil_k$ denotes the call string $[\delta, \ell_c]$ but possibly truncated (by omitting elements on the left) so as to have length at most k. Since the function mapping δ to $\lceil \delta, \ell_c \rceil_k$ is not injective (unlike the one mapping δ to $[\delta, \ell_c]$) we need to take the least upper bound over all δ that can be mapped to the relevant context δ'.

Similarly, the transfer function $\widehat{f^2_{\ell_c,\ell_r}}$ for procedure return is redefined by

$$\widehat{f^2_{\ell_c,\ell_r}}(\widehat{l}, \widehat{l'})(\delta) = f^2_{\ell_c,\ell_r}(\widehat{l}(\delta), \widehat{l'}(\lceil \delta, \ell_c \rceil_k))$$

as should be expected.

Example 2.40 Let us consider Detection of Signs Analysis in the special case where $k = 0$, i.e. where $\Delta = \{\Lambda\}$ and hence $\Delta \times (\mathbf{Var}_\star \to \mathbf{Sign})$ is isomorphic to $\mathbf{Var}_\star \to \mathbf{Sign}$. Using this isomorphism the formulae defining the transfer functions for procedure call can be simplified to

$$\begin{aligned}
\widehat{f^{\mathsf{sign1}}_{\ell_c}}(Y) &= \bigcup\{\phi^{\mathsf{sign1}}_{\ell_c}(\sigma^{\mathsf{sign}}) \mid \sigma^{\mathsf{sign}} \in Y\} \\
\widehat{f^{\mathsf{sign2}}_{\ell_c,\ell_r}}(Y, Y') &= \bigcup\{\phi^{\mathsf{sign2}}_{\ell_c,\ell_r}(\sigma^{\mathsf{sign}}_1, \sigma^{\mathsf{sign}}_2) \mid \sigma^{\mathsf{sign}}_1 \in Y \ \wedge \ \sigma^{\mathsf{sign}}_2 \in Y'\}
\end{aligned}$$

where $Y, Y' \subseteq \mathbf{Var}_\star \to \mathbf{Sign}$. It is now easy to see that the analysis is context-insensitive: at procedure return it is not possible to distinguish between the different call points.

Let us next consider the case where $k = 1$. Here $\Delta = \mathbf{Lab} \cup \{\Lambda\}$ and the transfer functions for procedure call are:

$$\begin{aligned}
\widehat{f^{\mathsf{sign1}}_{\ell_c}}(Z) &= \bigcup\{\{\ell_c\} \times \phi^{\mathsf{sign1}}_{\ell_c}(\sigma^{\mathsf{sign}}) \mid (\delta, \sigma^{\mathsf{sign}}) \in Z\} \\
\widehat{f^{\mathsf{sign2}}_{\ell_c,\ell_r}}(Z, Z') &= \bigcup\{\{\delta\} \times \phi^{\mathsf{sign2}}_{\ell_c,\ell_r}(\sigma^{\mathsf{sign}}_1, \sigma^{\mathsf{sign}}_2) \mid (\delta, \sigma^{\mathsf{sign}}_1) \in Z \\
&\qquad \wedge \ (\ell_c, \sigma^{\mathsf{sign}}_2) \in Z'\}
\end{aligned}$$

Now the transfer function $\widehat{f^{\mathsf{sign1}}_{\ell_c}}$ will mark all data from the call point ℓ_c with that label. Thus it does not harm that the information $\widehat{f^{\mathsf{sign1}}_{\ell_c}}(Z)$ is merged

with similar information $\widehat{f^{\mathsf{sign1}}_{\ell_{c'}}}(Z)$ from another procedure call. At the return from the call the transfer function $\widehat{f^{\mathsf{sign2}}_{\ell_c,\ell_r}}$ selects those pairs $(\ell_c, \sigma_2^{\mathsf{sign}}) \in Z'$ that are relevant for the current call and combines them with those pairs $(\delta, \sigma_1^{\mathsf{sign}}) \in Z$ that describe the situation before the call; in particular, this allows us to reset the context to be that of the call point. ∎

2.5.5 Assumption Sets as Context

An alternative to describing a path directly in terms of the calls being performed is to record information about the state in which the call was made; these methods can clearly be combined but in the interest of simplicity we shall abstain from doing so.

Large assumption sets. Throughout this subsection we shall make the simplifying assumption that

$$L = \mathcal{P}(D)$$

as is the case for the Detection of Signs Analysis. Much as in Examples 2.36 and 2.38 the property space $\widehat{L} = \Delta \to L$ is then isomorphic to

$$\widehat{L} = \mathcal{P}(\Delta \times D)$$

and we shall use this definition throughout this subsection. Restricting the attention to only recording information about the last call (corresponding to taking $k = 1$ above), one possibility is to take

$$\Delta = \mathcal{P}(D)$$

and we then take the extremal value to be

$$\widehat{\iota} = \{(\{\iota\}, \iota)\}$$

meaning that the initial context is described by the initial abstract state. This kind of context information is often called an *assumption set* and expresses a dependency on data (as opposed to a dependency on control as in the case of call strings).

Example 2.41 Assume that we want to perform a Detection of Signs Analysis (Example 2.36) of the Fibonacci program of Example 2.33 and that the extremal value ι_{sign} is the singleton $[\mathsf{x} \mapsto +, \mathsf{y} \mapsto -, \mathsf{z} \mapsto -]$. Then the contexts of primary interest will be sets consisting of some of the following abstract states

$$\begin{array}{lll} [\mathsf{x} \mapsto +, \mathsf{y} \mapsto 0, \mathsf{z} \mapsto -], & [\mathsf{x} \mapsto +, \mathsf{y} \mapsto 0, \mathsf{z} \mapsto 0], & [\mathsf{x} \mapsto +, \mathsf{y} \mapsto 0, \mathsf{z} \mapsto +], \\ [\mathsf{x} \mapsto +, \mathsf{y} \mapsto +, \mathsf{z} \mapsto -], & [\mathsf{x} \mapsto +, \mathsf{y} \mapsto +, \mathsf{z} \mapsto 0], & [\mathsf{x} \mapsto +, \mathsf{y} \mapsto +, \mathsf{z} \mapsto +] \end{array}$$

corresponding to the states in which the `call`-statements may be encountered. ∎

For a procedure call $(\ell_c, \ell_n, \ell_x, \ell_r) \in IF$, i.e. $[\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}$ in the case of forward analysis, we define the transfer function $\widehat{f^1_{\ell_c}}$ for procedure call by:

$$\widehat{f^1_{\ell_c}}(Z) = \bigcup\{\{\delta'\} \times \phi^1_{\ell_c}(d) \mid \begin{array}{l}(\delta, d) \in Z \wedge \\ \delta' = \{d'' \mid (\delta, d'') \in Z\}\}\end{array}$$

where $\phi^1_{\ell_c} : D \to \mathcal{P}(D)$. The idea is as follows: a pair $(\delta, d) \in Z$ describes a context and an abstract state for the current call. We now have to modify the context to take the call into account, i.e. we have to determine the set of possible abstract states in which the call could happen in the current context and this is $\delta' = \{d'' \mid (\delta, d'') \in Z\}$. Given this context we proceed as in the call string formulations presented above and mark the data flow property with this context.

Next we shall consider the transfer function $\widehat{f^2_{\ell_c,\ell_r}}$ for procedure return

$$\widehat{f^2_{\ell_c,\ell_r}}(Z, Z') = \bigcup\{\{\delta\} \times \phi^2_{\ell_c,\ell_r}(d, d') \mid \begin{array}{l}(\delta, d) \in Z \ \wedge (\delta', d') \in Z' \wedge \\ \delta' = \{d'' \mid (\delta, d'') \in Z\}\}\end{array}$$

where $\phi^2_{\ell_c,\ell_r} : D \times D \to \mathcal{P}(D)$. Here $(\delta, d) \in Z$ describes the situation before the call and $(\delta', d') \in Z'$ describes the situation at the procedure exit. From the definition of $\widehat{f^1_{\ell_c}}$ we know that the context matching (δ, d) will be $\delta' = \{d'' \mid (\delta, d'') \in Z\}$ so we impose that condition. We can now combine information from before the call with that at the procedure exit much as in the call string approach; in particular, we can reset the context to be that of the call point.

There is one important snag with the definitions of the transfer functions $\widehat{f^1_{\ell_c}}$ and $\widehat{f^2_{\ell_c,\ell_r}}$: they are in general not monotone! One way to overcome this problem is to consider more general techniques for solving systems of equations where the transfer functions satisfy a weaker condition than monotonicity; we provide references to this approach in the Concluding Remarks. Another way to overcome the problem is to use more approximate definitions that are indeed monotone; one possibility is to replace the condition $\delta' = \{d'' \mid (\delta, d'') \in Z\}$ by $\delta' \subseteq \{d'' \mid (\delta, d'') \in Z\}$. An even more approximate, but computationally more tractable, solution is to use smaller assumption sets as detailed below.

Small assumption sets. As a simpler version of using assumption sets one may take

$$\Delta = D$$

and then use $\hat{\iota} = \{(\iota, \iota)\}$ as the extremal value. So rather than basing the embellished Monotone Framework on $\mathcal{P}(D) \times D$ as above we now base it on $D \times D$. Of course, this is much less precise but, on the positive side, the size of the data flow properties has been reduced dramatically.

For a procedure call $(\ell_c, \ell_n, \ell_x, \ell_r) \in IF$, i.e. $[\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}$ for forward analyses, the transfer function $\widehat{f^1_{\ell_c}}$ is now defined by

$$\widehat{f^1_{\ell_c}}(Z) = \bigcup\{\{d\} \times \phi^1_{\ell_c}(d) \mid (\delta, d) \in Z\}$$

where, as before, $\phi^1_{\ell_c} : D \to \mathcal{P}(D)$. Here the individual pieces of information concerning the abstract state of the call have their own local contexts; we have no way of grouping the abstract states corresponding to δ as we did in the approach with large assumption sets.

The corresponding definition of the transfer function $\widehat{f^2_{\ell_c,\ell_r}}$ for procedure return then is

$$\widehat{f^2_{\ell_c,\ell_r}}(Z, Z') = \bigcup\{\{\delta\} \times \phi^2_{\ell_c,\ell_r}(d, d') \mid \ (\delta, d) \in Z \ \wedge \ (d, d') \in Z'\}$$

where again $\phi^2_{\ell_c,\ell_r} : D \times D \to \mathcal{P}(D)$. Examples of how to use assumption sets will be considered in the exercises.

2.5.6 Flow-Sensitivity versus Flow-Insensitivity

All of the data flow analyses we have considered so far have been *flow-sensitive*: this just means that in general we would expect the analysis of a program $S_1; S_2$ to differ from the analysis of the program $S_2; S_1$ where the statements come in a different order.

Sometimes one considers *flow-insensitive* analyses where the order of statements is of no importance for the analysis being performed. This may sound weird at first, but suppose that the analysis being performed is like the ones considered in Section 2.1 except that for simplicity all kill components are empty sets. Given these assumptions one might expect that the programs $S_1; S_2$ and $S_2; S_1$ give rise to the same analysis. Clearly a flow-insensitive analysis may be much less precise than its flow-sensitive analogue but also it is likely to be much cheaper; since interprocedural data flow analyses tend to be very costly, it is therefore useful to have a repertoire of techniques for reducing the cost.

Sets of assigned variables. We shall now present an example of a flow-insensitive analysis. Consider a program $P_\star$ of the form **begin** $D_\star$ $S_\star$ **end**. For each procedure

$$\texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x}$$

in $D_\star$, the aim is to determine the set $IAV(p)$ of global variables that might be assigned directly or indirectly when p is called.

To compute these sets we need two auxiliary notions. The set $AV(S)$ of directly *assigned variables* gives for each statement S the set of variables

that could be assigned in S – but ignoring the effect of procedure calls. It is defined inductively upon the structure of S:

$$\begin{array}{rcl} AV([\texttt{skip}]^\ell) & = & \emptyset \\ AV([x := a]^\ell) & = & \{x\} \\ AV(S_1; S_2) & = & AV(S_1) \cup AV(S_2) \\ AV(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2) & = & AV(S_1) \cup AV(S_2) \\ AV(\texttt{while } [b]^\ell \texttt{ do } S) & = & AV(S) \\ AV([\texttt{call } p(a, z)]^{\ell_c}_{\ell_r}) & = & \{z\} \end{array}$$

Similarly we shall need the set $CP(S)$ of immediately *called procedures* that gives for each statement S the set of procedure names that could be directly called in S – but ignoring the effect of procedure calls. It is defined inductively upon the structure of S:

$$\begin{array}{rcl} CP([\texttt{skip}]^\ell) & = & \emptyset \\ CP([x := a]^\ell) & = & \emptyset \\ CP(S_1; S_2) & = & CP(S_1) \cup CP(S_2) \\ CP(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2) & = & CP(S_1) \cup CP(S_2) \\ CP(\texttt{while } [b]^\ell \texttt{ do } S) & = & CP(S) \\ CP([\texttt{call } p(a, z)]^{\ell_c}_{\ell_r}) & = & \{p\} \end{array}$$

Both the sets $AV(S)$ and $CP(S)$ are well-defined by induction on the structure of S; also it should be clear that they are context-insensitive in the sense that any rearrangement of the statements inside S would have given the same result. The information in $CP(\cdots)$ can be presented graphically: let the graph have a node for each procedure name as well as a node called $\texttt{main}_\star$ for the program itself, and let the graph have an edge from p (respectively $\texttt{main}_\star$) to p' whenever the procedure body S of p has $p' \in CP(S)$ (respectively $p' \in CP(S_\star)$). This graph is usually called the *procedure call graph.*

We can now formulate a system of data flow equations that specifies how to obtain the desired sets $IAV(p)$:

$$\begin{array}{rcl} IAV(p) & = & (AV(S) \setminus \{x\}) \cup \bigcup \{IAV(p') \mid p' \in CP(S)\} \\ & & \text{where } \texttt{proc } p(\texttt{val } x, \texttt{res } y) \texttt{ is}^{\ell_n} S \texttt{ end}^{\ell_x} \text{ is in } D_\star \end{array}$$

By analogy with the considerations in Section 2.1 we want the least solution of this system of equations.

Figure **2.11**: Procedure call graph for example program.

Example 2.42 Let us now consider the following version of the Fibonacci program (omitting labels):

```
begin proc fib(val z) is if z<3 then call add(1)
                         else (call fib(z-1); call fib(z-2))
      end;
      proc add(val u) is (y:=y+u; u:=0)
      end;
      y:=0; call fib(x)
end
```

We then get the following equations

$$\begin{aligned} IAV(\texttt{fib}) &= (\emptyset \setminus \{\texttt{z}\}) \cup IAV(\texttt{fib}) \cup IAV(\texttt{add}) \\ IAV(\texttt{add}) &= \{\texttt{y},\texttt{u}\} \setminus \{\texttt{u}\} \end{aligned}$$

The associated procedure call graph is shown in Figure 2.11. The least solution to the equation system is

$$IAV(\texttt{fib}) = IAV(\texttt{add}) = \{\texttt{y}\}$$

showing that only the variable y will be assigned by the procedure calls. (Had we instead taken the greatest solution to the equations we would have $IAV(\texttt{fib}) = IAV(\texttt{add}) = \mathbf{Var}_\star$ for any set $\mathbf{Var}_\star$ of variables that contains those used in the program and this would be completely unusable.) ■

Note that the formulation of the example analysis did not associate information with entries and exits of blocks but rather with the blocks (or more generally the statements) themselves. This is a rather natural space saving approach for a context-insensitive analysis. It also relates to the discussion of Type and Effect Systems in Section 1.6: the "annotated base types" in Table 1.2 versus the "annotated type constructors" in Table 1.3.

2.6 Shape Analysis

We shall now study an extension of the WHILE-language with heap allocated data structures and an intraprocedural *Shape Analysis* that gives a *finite characterisation* of the *shapes* of these data structures. So while the aim of the previous sections has been to present the basic techniques of Data Flow Analysis, the aim of this section is to show how the techniques can be used to specify a rather complex analysis.

Shape analysis information is not only useful for classical compiler optimisations but also for software development tools: the Shape Analysis will allow us to statically detect errors like dereferencing a `nil`-pointer – this is guaranteed to give rise to a dynamic error and a warning can be issued. Perhaps more surprisingly, the analysis allows us to validate certain properties of the shape of the data structures manipulated by the program; we can for example validate that a program for in-situ list reversal does indeed transform a non-cyclic list into a non-cyclic list.

Syntax of the pointer language. We shall study an extension of WHILE that allows us to create cells in the heap; the cells are structured and may contain values as well as pointers to other cells. The data stored in a cell is accessed via selectors so we assume that a finite and non-empty set **Sel** of *selector names* are given:

$$sel \in \mathbf{Sel} \qquad \text{selector names}$$

As an example **Sel** may include the Lisp-like selectors `car` and `cdr` for selecting the first and second components of pairs. The cells of the heap can be addressed by expressions like `x.cdr`: this will first determine the cell pointed to by the variable `x` and then return the value of the `cdr` field. For the sake of simplicity we shall only allow one level of selectors although the development generalises to several levels. Formally the *pointer expressions*

$$p \in \mathbf{PExp}$$

are given by:

$$p ::= x \mid x.sel$$

The syntax of the WHILE-language is now extended to have:

$$\begin{array}{lcl} a & ::= & p \mid n \mid a_1\ op_a\ a_2 \mid \texttt{nil} \\ b & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{not}\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \mid op_p\ p \\ S & ::= & [p\texttt{:=}a]^\ell \mid [\texttt{skip}]^\ell \mid S_1;\ S_2 \mid \\ & & \texttt{if}\ [b]^\ell\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ [b]^\ell\ \texttt{do}\ S \mid \\ & & [\texttt{malloc}\ p]^\ell \end{array}$$

Arithmetic expressions are extended to use pointer expressions rather than just variables, and an arithmetic expression can also be the constant `nil`.

The binary operations op_a are as before, that is, they are the standard arithmetic operations and in particular they do *not* allow pointer arithmetic. The boolean expressions are extended such that the relational operators op_r now allow testing for the *equality of pointers* and also we shall allow unary operations op_p on pointers (as for example `is-nil` and `has-`*sel* for each *sel* $\in$ **Sel**). Note that arithmetic as well as boolean expressions can only access cells in the heap, they cannot create new cells nor update existing cells.

The assignment statement takes the general form p`:=`a where p is a pointer expression. In the case where p is just a variable we have an extension of the ordinary assignment of the WHILE language and in the case where p contains a selector we have a destructive update of the heap. The statements of the extended language also contain a statement `malloc` p for creating a new cell pointed to by p.

Example 2.43 The following program reverses the list pointed to by `x` and leaves the result in `y`:

```
[y:=nil]^1;
while [not is-nil(x)]^2 do
      ([z:=y]^3; [y:=x]^4; [x:=x.cdr]^5; [y.cdr:=z]^6);
[z:=nil]^7
```

Figure 2.12 illustrates the effect of the program when `x` points to a five element list and `y` and `z` are initially undefined. Row 0 shows the heap just before entering the `while`-loop: `x` points to the list and `y` is `nil` (denoted by $\diamond$); to avoid cluttering the figure we do not draw the `car`-pointers. After having executed the statements of the body of the loop the situation is as in row 1: `x` now points to the tail of the list, `y` points to the head of the list and `z` is `nil`. In general the n'th row illustrates the situation just before entering the loop the $n+1$'th time so in row 5 we see that `x` points to `nil` and the execution of the loop terminates and `y` points to the reversed list. The final statement `z:=nil` simply removes the pointer from `z` to ξ_4 and sets it to the `nil`-value. ∎

2.6.1 Structural Operational Semantics

To model the scenario described above we shall introduce an infinite set **Loc** of *locations* (or addresses) for the heap cells:

$$\xi \in \mathbf{Loc} \qquad \text{locations}$$

The value of a variable will now either be an integer (as before), a location (i.e. a pointer) or the special constant $\diamond$ reflecting that it is the `nil` value. Thus the *states* are given by

$$\sigma \in \mathbf{State} = \mathbf{Var}_\star \to (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$

Figure 2.12: Reversal of a list of five elements.

where as usual $\mathbf{Var}_\star$ is the (finite) set of variables occurring in the program of interest. As mentioned above the cells of the *heap* have multiple fields and they are accessed using the selectors. Each field can either be an integer, a pointer to another cell or it can be `nil`. We formalise this by taking

$$\varkappa \in \mathbf{Heap} = (\mathbf{Loc} \times \mathbf{Sel}) \rightarrow_{\text{fin}} (\mathbf{Z} + \mathbf{Loc} + \{\diamond\})$$

where the use of *partial* functions with finite domain reflects that not all selector fields need to be defined; as we shall see later, a newly created cell with location ξ will have all its fields to be uninitialised and hence the corresponding heap $\mathcal{H}$ will have $\mathcal{H}(\xi, sel)$ to be undefined for all $sel \in \mathbf{Sel}$.

Pointer expressions. Given a state and a heap we need to determine the value of a pointer expression p as an element of $\mathbf{Z} + \mathbf{Loc} + \{\diamond\}$. For this we introduce the function

$$\wp \ : \ \mathbf{PExp}_\star \to (\mathbf{State} \times \mathbf{Heap}) \to_{\mathsf{fin}} (\mathbf{Z} + \{\diamond\} + \mathbf{Loc})$$

where $\mathbf{PExp}_\star$ denotes pointer expressions with variables in $\mathbf{Var}_\star$. It is defined by:

$$\begin{array}{rcl} \wp[\![x]\!](\sigma, \mathcal{H}) & = & \sigma(x) \\ \wp[\![x.sel]\!](\sigma, \mathcal{H}) & = & \left\{ \begin{array}{l} \mathcal{H}(\sigma(x), sel) \\ \quad \text{if } \sigma(x) \in \mathbf{Loc} \text{ and } \mathcal{H} \text{ is defined on } (\sigma(x), sel) \\ \mathit{undef} \\ \quad \text{if } \sigma(x) \notin \mathbf{Loc} \text{ or } \mathcal{H} \text{ is undefined on } (\sigma(x), sel) \end{array} \right. \end{array}$$

The first clause takes care of the situation where p is a simple variable and using the state we determine its value – note that this may be an integer, a location or the special `nil`-value $\diamond$. The second clause takes care of the case where the pointer expression has the form $x.sel$. Here we first have to determine the value of x; it only makes sense to inspect the *sel*-field in the case x evaluates to a location that has a *sel*-field and hence the clause is split into two cases. In the case where x evaluates to a location we simply inspect the heap $\mathcal{H}$ to determine the value of the *sel*-field – again we may note that this can be an integer, a location or the special value $\diamond$.

Example 2.44 In Figure 2.12 the *oval nodes* model the cells of the heap $\mathcal{H}$ and they are labelled with their location (or address). The *unlabelled edges* denote the state σ: an edge from a variable x to some node labelled ξ means that $\sigma(x) = \xi$; an edge from x to the symbol $\diamond$ means that $\sigma(x) = \diamond$. The *labelled edges* model the heap $\mathcal{H}$: an edge labelled *sel* from a node labelled ξ to a node labelled ξ' means that there is a *sel* pointer between the two cells, that is $\mathcal{H}(\xi, sel) = \xi'$; an edge labelled *sel* from a node labelled ξ to the symbol $\diamond$ means that the pointer is a `nil`-pointer, that is $\mathcal{H}(\xi, sel) = \diamond$.

Consider the pointer expression `x.cdr` and assume that σ and $\mathcal{H}$ are as in row 0 of Figure 2.12, that is $\sigma(\mathtt{x}) = \xi_1$ and $\mathcal{H}(\xi_1, \mathtt{cdr}) = \xi_2$. Then $\wp[\![\mathtt{x.cdr}]\!](\sigma, \mathcal{H}) = \xi_2$. ∎

Arithmetic and boolean expressions. It is now straightforward to extend the semantics of arithmetic and boolean expressions to handle pointer expressions and the `nil`-constant. Obviously the functionality of the

semantic functions $\mathcal{A}$ and $\mathcal{B}$ has to be changed to take the heap into account:

$$\begin{array}{lll} \mathcal{A} & : & \mathbf{AExp} \to (\mathbf{State} \times \mathbf{Heap}) \to_{\text{fin}} (\mathbf{Z} + \mathbf{Loc} + \{\diamond\}) \\ \mathcal{B} & : & \mathbf{BExp} \to (\mathbf{State} \times \mathbf{Heap}) \to_{\text{fin}} \mathbf{T} \end{array}$$

The clauses for arithmetic expressions are

$$\begin{array}{rcl} \mathcal{A}[\![p]\!](\sigma, \mathcal{H}) & = & \wp[\![p]\!](\sigma, \mathcal{H}) \\ \mathcal{A}[\![n]\!](\sigma, \mathcal{H}) & = & \mathcal{N}[\![n]\!] \\ \mathcal{A}[\![a_1 \ op_a \ a_2]\!](\sigma, \mathcal{H}) & = & \mathcal{A}[\![a_1]\!](\sigma, \mathcal{H}) \ \mathbf{op}_a \ \mathcal{A}[\![a_2]\!](\sigma, \mathcal{H}) \\ \mathcal{A}[\![\texttt{nil}]\!](\sigma, \mathcal{H}) & = & \diamond \end{array}$$

where we use $\wp$ to determine the value of pointer expressions and we explicitly write that the meaning of `nil` is $\diamond$. Also the meaning $\mathbf{op}_a$ of the binary operation op_a has to be suitably modified to be undefined unless both arguments are integers in which case the results are as for the WHILE-language.

The definition of the semantics of boolean expressions is similar so we only give two of the clauses:

$$\begin{array}{rcl} \mathcal{B}[\![a_1 \ op_r \ a_2]\!](\sigma, \mathcal{H}) & = & \mathcal{A}[\![a_1]\!](\sigma, \mathcal{H}) \ \mathbf{op}_r \ \mathcal{A}[\![a_2]\!](\sigma, \mathcal{H}) \\ \mathcal{B}[\![op_p \ p]\!](\sigma, \mathcal{H}) & = & \mathbf{op}_p \ (\wp[\![p]\!](\sigma, \mathcal{H})) \end{array}$$

Analogously to above, the meaning $\mathbf{op}_r$ of the binary relation operator op_r has to be suitably modified to give undefined in case the arguments are not both integers or both pointers (in which case the equality operation tests for the equality of the pointers). The meaning of the unary operation op_p is defined by $\mathbf{op}_p$; as an example:

$$\textbf{is-nil}(v) \quad = \quad \begin{cases} \mathsf{tt} & \text{if } v = \diamond \\ \mathsf{ff} & \text{otherwise} \end{cases}$$

Statements. Finally, the semantics of statements is extended to cope with the heap component. The configurations will now contain a state as well as a heap so we have

$$\begin{array}{c} \langle [x := a]^\ell, \sigma, \mathcal{H} \rangle \to \langle \sigma[x \mapsto \mathcal{A}[\![a]\!](\sigma, \mathcal{H})], \mathcal{H} \rangle \\ \text{if } \mathcal{A}[\![a]\!](\sigma, \mathcal{H}) \text{ is defined} \end{array}$$

reflecting that for the assignment $x := a$ the state is updated as usual and the heap is left unchanged. In the case where we assign to a pointer expression containing a selector field we shall leave the state unchanged and update the heap as follows:

$$\begin{array}{c} \langle [x.sel := a]^\ell, \sigma, \mathcal{H} \rangle \to \langle \sigma, \mathcal{H}[(\sigma(x), sel) \mapsto \mathcal{A}[\![a]\!](\sigma, \mathcal{H})] \rangle \\ \text{if } \sigma(x) \in \mathbf{Loc} \text{ and } \mathcal{A}[\![a]\!](\sigma, \mathcal{H}) \text{ is defined} \end{array}$$

Here the side condition ensures that the left hand side of the assignment does indeed evaluate to a location.

The construct `malloc` p is responsible for creating a new cell. We have two clauses depending on the form of p:

$$\langle [\texttt{malloc}\ x]^\ell, \sigma, \mathcal{H}\rangle \rightarrow \langle \sigma[x \mapsto \xi], \mathcal{H}\rangle$$
$$\text{where } \xi \text{ does not occur in } \sigma \text{ or } \mathcal{H}$$

$$\langle [\texttt{malloc}\ (x.sel)]^\ell, \sigma, \mathcal{H}\rangle \rightarrow \langle \sigma, \mathcal{H}[(\sigma(x), sel) \mapsto \xi]\rangle$$
$$\text{where } \xi \text{ does not occur in } \sigma \text{ or } \mathcal{H} \text{ and } \sigma(x) \in \mathbf{Loc}$$

Note that in both cases we introduce a fresh location ξ but we do not specify any values for $\mathcal{H}(\xi, sel)$ – as discussed before we have settled for a semantics where the fields of ξ are undefined; obviously other choices are possible. Also note that in the last clause the side condition ensures that we already have a location corresponding to x and hence can create an edge to the new location.

Remark. The semantics only allows a limited reuse of garbage locations. For a statement like $[\texttt{malloc x}]^1; [\texttt{x:=nil}]^2; [\texttt{malloc y}]^3$ we will assign some location to `x` at the statement with label 1 and since it neither occurs in the state nor the heap after the assignment labelled 2 we are free to reuse it in the statement labelled 3 (but we do not have to). For a statement like $[\texttt{malloc x}]^1; [\texttt{x.cdr:=nil}]^2; [\texttt{x:=nil}]^3; [\texttt{malloc y}]^4$ we would not be able to reuse the location allocated at 1 although it will be unreachable (and hence garbage) after the statement labelled 3. ∎

2.6.2 Shape Graphs

It should be evident that there are programs for which the heap can grow arbitrarily large. Therefore the aim of the analysis will be to come up with *finite* representations of it. To do so we shall introduce a method for combining the locations of the semantics into a finite number of *abstract locations.* We then introduce an *abstract state* S mapping variables to abstract locations (rather than locations) and an *abstract heap* H specifying the links between the abstract locations (rather than the locations). More precisely, the analysis will operate on *shape graphs* $(\mathsf{S}, \mathsf{H}, \mathsf{is})$ consisting of:

- an abstract state, S,
- an abstract heap, H, and
- sharing information, is, for the abstract locations.

The last component allows us to recover some of the imprecision introduced by combining many locations into one abstract location. We shall now describe how a given state σ and heap $\mathcal{H}$ give rise to a shape graph $(\mathsf{S}, \mathsf{H}, \mathsf{is})$; in doing so we shall specify the functionality of S, H and is in detail as well as formulate a total of five invariants.

Abstract locations. The *abstract locations* have the form n_X where X is a subset of the variables of $\mathbf{Var}_\star$:

$$\mathbf{ALoc} = \{n_X \mid X \subseteq \mathbf{Var}_\star\} \qquad \text{abstract locations}$$

Since $\mathbf{Var}_\star$ is finite it is clear that **ALoc** is finite and a given shape graph will contain a subset of the abstract locations of **ALoc**.

The idea is that if $x \in X$ then the abstract location n_X will represent the location $\sigma(x)$. The abstract location $n_\emptyset$ is called the *abstract summary location* and will represent all the locations that cannot be reached directly from the state without consulting the heap. Clearly n_X and $n_\emptyset$ will represent disjoint sets of locations when $X \neq \emptyset$.

In general, we shall enforce the invariant that two distinct abstract locations n_X and n_Y always represent disjoint sets of locations. As a consequence, for any two abstract locations n_X and n_Y it is either the case that $X = Y$ or that $X \cap Y = \emptyset$. To prove this assume by way of contradiction that $X \neq Y$ and that $z \in X \cap Y$. From $z \in X$ we get that $\sigma(z)$ is represented by n_X and similarly $z \in Y$ gives that $\sigma(z)$ is represented by n_Y. But then $\sigma(z)$ must be distinct from $\sigma(z)$ and we have the desired contradiction.

The invariant can be formulated as follows:

> **Invariant 1.** If two abstract locations n_X and n_Y occur in the same shape graph then either $X = Y$ or $X \cap Y = \emptyset$.

Example 2.45 Consider the state and heap in row 2 of Figure 2.12. The variables **x**, **y** and **z** point to different locations (ξ_3, ξ_2, and ξ_1, respectively) so in the shape graph they will be represented by different abstract locations named $n_{\{\mathtt{x}\}}$, $n_{\{\mathtt{y}\}}$ and $n_{\{\mathtt{z}\}}$. The two locations ξ_4 and ξ_5 cannot be reached directly from the state so they will be represented by the abstract summary location $n_\emptyset$. ∎

Abstract states. One of the components of a shape graph is the *abstract state*, S, that maps variables to abstract locations. To maintain the naming convention for abstract locations we shall ensure that:

> **Invariant 2.** If x is mapped to n_X by the abstract state then $x \in X$.

From Invariant 1 it follows that there will be at most one abstract location in the shape graph containing a given variable.

We shall only be interested in the shape of the heap so we shall not distinguish between integer values, `nil`-pointers and uninitialised fields; hence we can view the abstract state as an element of

$$\mathsf{S} \in \mathbf{AState} = \mathcal{P}(\mathbf{Var}_\star \times \mathbf{ALoc})$$

where we have chosen to use powersets so as to simplify the notation in later parts of the development. We shall write $ALoc(\mathsf{S}) = \{n_X \mid \exists x : (x, n_X) \in \mathsf{S}\}$ for the set of abstract locations occurring in S. (Note that **AState** is too large in the sense that it contains elements that do not satisfy the invariants.)

Abstract heaps. Another component of the shape graph is the *abstract heap*, H, that specifies the links between the abstract locations (just as the heap specifies the links between the locations in the semantics). The links will be specified by triples (n_V, sel, n_W) and formally we take the abstract heap as an element of

$$\mathsf{H} \in \mathbf{AHeap} = \mathcal{P}(\mathbf{ALoc} \times \mathbf{Sel} \times \mathbf{ALoc})$$

where we again do not distinguish between integers, `nil`-pointers and uninitialised fields. We shall write $ALoc(\mathsf{H}) = \{n_V, n_W \mid \exists sel : (n_V, sel, n_W) \in \mathsf{H}\}$ for the set of abstract locations occurring in H.

The intention is that if $\varkappa(\xi_1, sel) = \xi_2$ and ξ_1 and ξ_2 are represented by n_V and n_W, respectively, then $(n_V, sel, n_W) \in \mathsf{H}$.

In the heap $\varkappa$ there will be *at most* one location ξ_2 such that $\varkappa(\xi_1, sel) = \xi_2$. The abstract heap only partly shares this property because the abstract location $n_\emptyset$ can represent several locations pointing to different locations. However, the abstract heap must satisfy:

> **Invariant 3.** Whenever (n_V, sel, n_W) and $(n_V, sel, n_{W'})$ are in the abstract heap then either $V = \emptyset$ or $W = W'$.

Thus the target of a selector field will be uniquely determined by the source unless the source is the abstract summary location $n_\emptyset$.

Example 2.46 Continuing Example 2.45 we can now see that the abstract state S_2 corresponding to the state of row 2 of Figure 2.12 will be

$$\mathsf{S}_2 = \{(\mathtt{x}, n_{\{\mathtt{x}\}}), (\mathtt{y}, n_{\{\mathtt{y}\}}), (\mathtt{z}, n_{\{\mathtt{z}\}})\}$$

The abstract heap H_2 corresponding to row 2 has

$$\mathsf{H}_2 = \{(n_{\{\mathtt{x}\}}, \mathtt{cdr}, n_\emptyset), (n_\emptyset, \mathtt{cdr}, n_\emptyset), (n_{\{\mathtt{y}\}}, \mathtt{cdr}, n_{\{\mathtt{z}\}})\}$$

The first triple reflects that the heap maps ξ_3 and `cdr` to ξ_4, ξ_3 is represented by $n_{\{\mathtt{x}\}}$ and ξ_4 is represented by $n_\emptyset$. The second triple reflects that the heap maps ξ_4 and `cdr` to ξ_5 and both ξ_4 and ξ_5 are represented by $n_\emptyset$. The final triple reflects that the heap maps ξ_2 and `cdr` to ξ_1, ξ_2 is represented by $n_{\{\mathtt{y}\}}$ and ξ_1 is represented by $n_{\{\mathtt{z}\}}$. Note that there is no triple $(n_{\{\mathtt{z}\}}, \mathtt{cdr}, n_\emptyset)$ because the heap maps ξ_1 and `cdr` to $\diamond$ rather than a location.

The resulting abstract state and abstract heap is illustrated in Figure 2.13 together with similar information for the other states and heaps of Figure

Figure 2.13: Shape graphs corresponding to Figure 2.12.

2.12. The *square nodes* model abstract locations; the *unlabelled edges* from variables to square nodes model the abstract state and the *labelled edges* between square nodes model the abstract heap. If the abstract state does not associate an abstract location with some variable then that variable does not occur in the picture.

Note that even if the semantics uses the same locations throughout the computation it need not be the case that the locations are associated with the same abstract locations at all points in the analysis. Consider Figures 2.12 and 2.13: the abstract location $n_\emptyset$ will in turn represent the locations $\{\xi_2, \xi_3, \xi_4, \xi_5\}$, $\{\xi_3, \xi_4, \xi_5\}$, $\{\xi_4, \xi_5\}$, $\{\xi_1, \xi_5\}$, $\{\xi_1, \xi_2\}$ and $\{\xi_1, \xi_2, \xi_3\}$. ■

Sharing information. We are now ready to introduce the third and final component of the shape graphs. Consider the top row of Figure 2.14. The abstract state and abstract heap to the right represent the state and the heap to the left but they also represent the state and the heap shown in the second row. We shall now show how to distinguish between these two cases.

The idea is to specify a subset, is, of the abstract locations that represent locations that are shared due to pointers in the heap: an abstract location n_X will be included in is if it does represent a location that is the target of more than one pointer in the heap. In the top row of Figure 2.14, the abstract location $n_{\{y\}}$ represents the location ξ_5 and it *is not* shared (by two or more *heap* pointers) so $n_{\{y\}} \notin$ is; the fat box indicates that the abstract location is unshared. On the other hand, in the second row ξ_5 *is* shared (both ξ_3 and ξ_4 point to it) so $n_{\{y\}} \in$ is; the double box indicates that the abstract location might be shared.

Figure 2.14: Sharing information.

Obviously, the abstract heaps themselves also contain some implicit sharing information: this is illustrated in the bottom row of Figure 2.14 where there are two distinct edges with target $n_{\{y\}}$. We shall ensure that this implicit sharing information is consistent with the explicit sharing information (as given by is) by imposing two invariants. The first ensures that information in the sharing component is also reflected in the abstract heap:

Invariant 4. If $n_X \in$ is then either

(a) $(n_\emptyset, sel, n_X)$ is in the abstract heap for some sel, or

(b) there exists two distinct triples (n_V, sel_1, n_X) and (n_W, sel_2, n_X) in the abstract heap (that is either $sel_1 \neq sel_2$ or $V \neq W$).

Case 4(a) takes care of the situation where there might be several locations represented by $n_\emptyset$ that point to n_X (as in the second and third rows of Figure 2.14). Case 4(b) takes care of the case where two distinct pointers (with different source or different selectors) point to n_X (as in the bottom row of Figure 2.14).

The second invariant ensures that sharing information present in the abstract heap is also reflected in the sharing component:

Invariant 5. Whenever there are two distinct triples (n_V, sel_1, n_X) and (n_W, sel_2, n_X) in the abstract heap and $n_X \neq n_\emptyset$ then $n_X \in$ is.

This takes care of the case where n_X represents a single location being the target of two or more heap pointers (as in the bottom row of Figure 2.14). Note that invariant 5 is the "inverse" of invariant 4(b) and that we have no "inverse" of invariant 4(a) – the presence of a pointer from $n_\emptyset$ to n_X gives no information about sharing properties of n_X.

In the case of the abstract summary location the explicit sharing information clearly gives extra information: if $n_\emptyset \in \mathsf{is}$ then there might be a location represented by $n_\emptyset$ that is the target of two or more heap pointers, whereas if $n_\emptyset \notin \mathsf{is}$ then all the locations represented by $n_\emptyset$ will be the target of at most one heap pointer. The explicit sharing information may also give extra information for abstract locations n_X where $X \neq \emptyset$: from 4(a) alone we cannot deduce that n_X is shared – this is clearly illustrated for the node $n_{\{\mathsf{y}\}}$ by the top two rows of Figure 2.14.

The complete lattice of shape graphs. To summarise, a *shape graph* is a triple consisting of an abstract state S, an abstract heap H, and a set is of abstract locations that are shared:

$$\begin{array}{lcll} \mathsf{S} & \in & \mathbf{AState} & = \mathcal{P}(\mathbf{Var}_\star \times \mathbf{ALoc}) \\ \mathsf{H} & \in & \mathbf{AHeap} & = \mathcal{P}(\mathbf{ALoc} \times \mathbf{Sel} \times \mathbf{ALoc}) \\ \mathsf{is} & \in & \mathbf{IsShared} & = \mathcal{P}(\mathbf{ALoc}) \end{array}$$

where $\mathbf{ALoc} = \{n_Z \mid Z \subseteq \mathbf{Var}_\star\}$. A shape graph $(\mathsf{S}, \mathsf{H}, \mathsf{is})$ is a *compatible shape graph* if it fulfils the five invariants presented above:

1. $\forall n_V, n_W \in ALoc(\mathsf{S}) \cup ALoc(\mathsf{H}) \cup \mathsf{is} : (V = W) \ \vee \ (V \cap W = \emptyset)$
2. $\forall (x, n_X) \in \mathsf{S} : x \in X$
3. $\forall (n_V, sel, n_W), (n_V, sel, n_{W'}) \in \mathsf{H} : (V = \emptyset) \ \vee \ (W = W')$
4. $\forall n_X \in \mathsf{is} : \ (\exists sel : (n_\emptyset, sel, n_X) \in \mathsf{H}) \vee$
   $(\exists (n_V, sel_1, n_X), (n_W, sel_2, n_X) \in \mathsf{H} :$
   $sel_1 \neq sel_2 \vee V \neq W)$
5. $\forall (n_V, sel_1, n_X), (n_W, sel_2, n_X) \in \mathsf{H} :$
   $((sel_1 \neq sel_2 \vee V \neq W) \wedge X \neq \emptyset) \Rightarrow n_X \in \mathsf{is}$

The set of compatible shape graphs is denoted

$$\mathbf{SG} = \{(\mathsf{S}, \mathsf{H}, \mathsf{is}) \mid (\mathsf{S}, \mathsf{H}, \mathsf{is}) \text{ is compatible}\}$$

and the analysis, to be called *Shape*, will operate over *sets* of compatible shape graphs, i.e. elements of $\mathcal{P}(\mathbf{SG})$. Since $\mathcal{P}(\mathbf{SG})$ is a powerset, it is trivially a complete lattice with $\sqcup$ being $\cup$ and $\sqsubseteq$ being $\subseteq$. Furthermore, $\mathcal{P}(\mathbf{SG})$ is finite because $\mathbf{SG} \subseteq \mathbf{AState} \times \mathbf{AHeap} \times \mathbf{IsShared}$ and all of **AState**, **AHeap** and **IsShared** are finite.

Figure 2.15: The single shape graph in the extremal value ι for the list reversal program.

2.6.3 The Analysis

The analysis will be specified as an instance of a Monotone Framework with the complete lattice of properties being $\mathcal{P}(\mathbf{SG})$. For each label consistent program $S_\star$ with isolated entries we obtain a set of equations of the form

$$\begin{array}{rcl} \mathit{Shape}_\circ(\ell) & = & \begin{cases} \iota & \text{if } \ell = \mathit{init}(S_\star) \\ \bigcup\{\mathit{Shape}_\bullet(\ell') \mid (\ell',\ell) \in \mathit{flow}(S_\star)\} & \text{otherwise} \end{cases} \\ \mathit{Shape}_\bullet(\ell) & = & f_\ell^{\mathsf{SA}}(\mathit{Shape}_\circ(\ell)) \end{array}$$

where $\iota \in \mathcal{P}(\mathbf{SG})$ is the extremal value holding at entry to $S_\star$ and f_ℓ^{SA} are the transfer functions to be developed below.

The analysis is a *forward analysis* since it is defined in terms of the set $\mathit{flow}(S_\star)$, and it is a *may analysis* since we are using $\bigcup$ as the combination operation. However, there are also aspects of a *must analysis* because each individual shape graph must not contain any superfluous information. This will be useful for achieving *strong update* and *strong nullification*; here "strong" means that an update or nullification of a pointer expression allows one to *remove* the existing binding before adding a new one. This in turn leads to a very powerful analysis.

Example 2.47 Consider again the list reversal program of Example 2.43 and assume that **x** initially points to an unshared list with at least two elements and that **y** and **z** are initially undefined; the singleton shape graph corresponding to this state and heap is illustrated in Figure 2.15 and will be the extremal value ι used throughout this development.

The Shape Analysis computes the sets $\mathit{Shape}_\circ(\ell)$ and $\mathit{Shape}_\bullet(\ell)$ of shape graphs describing the state and heap before and after executing the elementary block labelled ℓ. The equations for $\mathit{Shape}_\bullet(\ell)$ are

$$\begin{array}{rclcl} \mathit{Shape}_\bullet(1) & = & f_1^{\mathsf{SA}}(\mathit{Shape}_\circ(1)) & = & f_1^{\mathsf{SA}}(\iota) \\ \mathit{Shape}_\bullet(2) & = & f_2^{\mathsf{SA}}(\mathit{Shape}_\circ(2)) & = & f_2^{\mathsf{SA}}(\mathit{Shape}_\bullet(1) \cup \mathit{Shape}_\bullet(6)) \\ \mathit{Shape}_\bullet(3) & = & f_3^{\mathsf{SA}}(\mathit{Shape}_\circ(3)) & = & f_3^{\mathsf{SA}}(\mathit{Shape}_\bullet(2)) \\ \mathit{Shape}_\bullet(4) & = & f_4^{\mathsf{SA}}(\mathit{Shape}_\circ(4)) & = & f_4^{\mathsf{SA}}(\mathit{Shape}_\bullet(3)) \\ \mathit{Shape}_\bullet(5) & = & f_5^{\mathsf{SA}}(\mathit{Shape}_\circ(5)) & = & f_5^{\mathsf{SA}}(\mathit{Shape}_\bullet(4)) \end{array}$$

Figure 2.16: The effect of $[x\texttt{:=nil}]^\ell$.

$$\begin{aligned} Shape_\bullet(6) &= f_6^{\mathsf{SA}}(Shape_\circ(6)) = f_6^{\mathsf{SA}}(Shape_\bullet(5)) \\ Shape_\bullet(7) &= f_7^{\mathsf{SA}}(Shape_\circ(7)) = f_7^{\mathsf{SA}}(Shape_\bullet(2)) \end{aligned}$$

where the transfer functions f_ℓ^{SA} will be specified below. In the examples that follow we will provide further information about the equations but the computation of $Shape_\bullet(1), \cdots, Shape_\bullet(7)$ is left to Exercise 2.21. ■

The transfer function $f_\ell^{\mathsf{SA}} : \mathcal{P}(\mathbf{SG}) \to \mathcal{P}(\mathbf{SG})$ associated with a label, ℓ, has the form:

$$f_\ell^{\mathsf{SA}}(SG) = \bigcup\{\phi_\ell^{\mathsf{SA}}((\mathsf{S},\mathsf{H},\mathsf{is})) \mid (\mathsf{S},\mathsf{H},\mathsf{is}) \in SG\}$$

where $\phi_\ell^{\mathsf{SA}} : \mathbf{SG} \to \mathcal{P}(\mathbf{SG})$ specifies how a *single* shape graph (in $Shape_\circ(\ell)$) may be transformed into a *set* of shape graphs (in $Shape_\bullet(\ell)$) by the elementary block labelled ℓ. We shall now inspect the various forms of elementary block and specify ϕ_ℓ^{SA} in each case. We shall first consider the boolean expressions and the `skip`-statement, then the different forms of assignments and finally the `malloc`-statement.

Transfer function for $[b]^\ell$ and $[\texttt{skip}]^\ell$. We are only interested in the shape of the heap and the boolean tests do not modify the heap. Hence we take

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S},\mathsf{H},\mathsf{is})\}$$

so that the transfer function f_ℓ^{SA} will be the identity function. Similarly for the `skip`-statement.

Example 2.48 This case is illustrated by the test $[\texttt{not is-nil(x)}]^2$ of the list reversal program of Example 2.43: the transfer function f_2^{SA} is the identity function. Hence $Shape_\bullet(2) = Shape_\bullet(1) \cup Shape_\bullet(6)$ as shown in Example 2.47. ■

Transfer function for $[x{:=}a]^\ell$ where a is of the form n, $a_1\ op_a\ a_2$ or `nil`. The effect of this assignment will be to remove the binding to x, and to rename all abstract locations so that they do not include x in their name. The renaming of abstract locations is specified by the function

$$k_x(n_Z) = n_{Z\setminus\{x\}}$$

and we then take

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S},\mathsf{H},\mathsf{is})) = \{kill_x((\mathsf{S},\mathsf{H},\mathsf{is}))\}$$

where $kill_x((\mathsf{S},\mathsf{H},\mathsf{is})) = (\mathsf{S}',\mathsf{H}',\mathsf{is}')$ is given by

$$\begin{array}{rcl} \mathsf{S}' & = & \{(z, k_x(n_Z)) \mid (z, n_Z) \in S \ \wedge\ z \neq x\} \\ \mathsf{H}' & = & \{(k_x(n_V), sel, k_x(n_W)) \mid (n_V, sel, n_W) \in \mathsf{H}\} \\ \mathsf{is}' & = & \{k_x(n_X) \mid n_X \in \mathsf{is}\} \end{array}$$

so that we obtain strong nullification. It is easy to check that if $(\mathsf{S},\mathsf{H},\mathsf{is})$ is compatible then so is $(\mathsf{S}',\mathsf{H}',\mathsf{is}')$.

Example 2.49 The statement $[\mathtt{y{:=}nil}]^1$ of the list reversal program of Example 2.43 is of the form considered here. Since there is no occurrence of y in the single shape graph in ι of Figure 2.15, the shape graph $Shape_\bullet(1)$ in Example 2.47 is equal to ι. ■

An interesting case is when $(x, n_{\{x\}}) \in \mathsf{S}$ since this will cause the two abstract locations $n_{\{x\}}$ and $n_\emptyset$ to be merged. The sharing information is then updated to capture that we can only be sure that $n_\emptyset$ is unshared in the updated shape graph if both $n_\emptyset$ and $n_{\{x\}}$ were unshared in the original shape graph. This is illustrated in Figure 2.16: the left hand picture shows the interesting parts of the shape graph $(\mathsf{S},\mathsf{H},\mathsf{is})$ and the right hand picture shows the corresponding parts of $(\mathsf{S}',\mathsf{H}',\mathsf{is}')$. We shall assume that the square boxes represent distinct abstract locations so in particular V, $\{x\}$, W and $\emptyset$ are all distinct sets. The fat boxes represent unshared abstract locations as before, the thin boxes represent abstract locations whose sharing information is not affected by the transfer function, and unlabelled edges between abstract locations represent pointers that are unaffected by the transfer function.

Example 2.50 The statement $[\mathtt{z{:=}nil}]^7$ of the list reversal program of Example 2.43 illustrates this case: for each of the shape graphs of $Shape_\bullet(2)$ the abstract location $n_{\{\mathtt{z}\}}$ is merged with $n_\emptyset$ to produce one of the shape graphs of $Shape_\bullet(7)$. ■

Remark. The analysis does *not* perform garbage collection: it might be the case that there are no heap pointers to $n_{\{x\}}$ and then the corresponding location in the heap will be unreachable after the assignment. Nonetheless the analysis will merge the two abstract locations $n_{\{x\}}$ and $n_\emptyset$ and insist on a pointer from $n_\emptyset$ to any abstract location that $n_{\{x\}}$ might point to. ■

Figure 2.17: The effect of $[x{:=}y]^\ell$ when $x{\neq}y$.

Transfer function for $[x{:=}y]^\ell$. If $x = y$ then the transfer function f_ℓ^{SA} is just the identity.

Next suppose that $x \neq y$. The first effect of the assignment is to remove the old bindings to x; for this we use the kill_x operation introduced above. Then the new binding to x is recorded; this includes renaming the abstract location that includes y in its variable set to also include x. The renaming of the abstract locations is specified by the function:

$$g_x^y(n_Z) = \begin{cases} n_{Z\cup\{x\}} & \text{if } y \in Z \\ n_Z & \text{otherwise} \end{cases}$$

We shall then take

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')\}$$

where $(\mathsf{S}',\mathsf{H}',\mathsf{is}') = \mathit{kill}_x((\mathsf{S},\mathsf{H},\mathsf{is}))$ and

$$\begin{aligned} \mathsf{S}'' &= \{(z, g_x^y(n_Z)) \mid (z, n_Z) \in \mathsf{S}'\} \\ &\quad \cup \{(x, g_x^y(n_Y)) \mid (y', n_Y) \in \mathsf{S}' \wedge y' = y\} \\ \mathsf{H}'' &= \{(g_x^y(n_V), \mathit{sel}, g_x^y(n_W)) \mid (n_V, \mathit{sel}, n_W) \in \mathsf{H}'\} \\ \mathsf{is}'' &= \{g_x^y(n_Z) \mid n_Z \in \mathsf{is}'\} \end{aligned}$$

so that we obtain strong update. Here the second clause in the formula for S'' adds the new binding to x. Again we note that if $(\mathsf{S},\mathsf{H},\mathsf{is})$ is compatible then so is $(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')$.

The clause is illustrated in Figure 2.17 where we assume that nodes represent distinct abstract locations; it follows from the invariants that $y \in Y$ but $y \notin V$ and $y \notin W$. Note that $n_{Y\cup\{x\}}$ inherits the sharing properties of n_Y although

both x and y will point to the same cell; the reason is that the sharing information only records sharing in the heap – not sharing via the state.

Example 2.51 The statement $[\mathtt{y:=x}]^4$ of the list reversal program of Example 2.43 is of the form considered here: each of the shape graphs of $Shape_\bullet(3)$ in Example 2.47 is transformed into one of the shape graphs of $Shape_\bullet(4)$.

Also the statement $[\mathtt{z:=y}]^3$ is of the form considered here: each of the shape graphs of $Shape_\bullet(2)$ is transformed into one of those of $Shape_\bullet(3)$. ■

Transfer function for $[x:=y.sel]^\ell$. First assume that $x = y$; then the assignment is semantically equivalent to the following sequence of assignments

$$[t:=y.sel]^{\ell_1}; [x:=t]^{\ell_2}; [t:=\mathtt{nil}]^{\ell_3}$$

where t is a fresh variable and ℓ_1, ℓ_2 and ℓ_3 are fresh labels. The transfer function f_ℓ^{SA} can therefore be obtained as

$$f_\ell^{\mathsf{SA}} = f_{\ell_3}^{\mathsf{SA}} \circ f_{\ell_2}^{\mathsf{SA}} \circ f_{\ell_1}^{\mathsf{SA}}$$

where the transfer functions $f_{\ell_2}^{\mathsf{SA}}$ and $f_{\ell_3}^{\mathsf{SA}}$ follow the pattern described above. We shall therefore concentrate on the transfer function $f_{\ell_1}^{\mathsf{SA}}$, or equivalently, f_ℓ^{SA} in the case where $x \neq y$.

Example 2.52 The statement $[\mathtt{x:=x.cdr}]^5$ of the list reversal program of Example 2.43 is transformed into $[\mathtt{t:=x.cdr}]^{51}; [\mathtt{x:=t}]^{52}; [\mathtt{t:=nil}]^{53}$. We shall return to the analysis of $[\mathtt{t:=x.cdr}]^{51}$ later. ■

So assume that $x \neq y$ and let $(\mathsf{S}, \mathsf{H}, \mathsf{is})$ be a compatible shape graph before the analysis of the statement. As in the previous case, the first step will be to remove the old binding for x and again we use the auxiliary function $kill_x$:

$$(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$$

The next step will be to rename the abstract location corresponding to $y.sel$ to include x in its name and to establish the binding of x to that abstract location. We can now identify three possibilities:

1. There is no abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ or there is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ but no n_Z such that $(n_Y, sel, n_Z) \in \mathsf{H}'$; in this case the shape graph will represent a state and a heap where y or $y.sel$ is an integer, `nil` or undefined.

2. There is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is an abstract location $n_U \neq n_\emptyset$ such that $(n_Y, sel, n_U) \in \mathsf{H}'$; in this case the shape graph will represent a state and a heap where the location pointed to by $y.sel$ will also be pointed to by some other variable (in U).

Figure 2.18: The effect of $[x{:=}y.sel]^\ell$ in Case 2 when $x \neq y$.

3. There is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$; in this case the shape graph will represent a state and a heap where no other variable points to the location pointed to by $y.sel$.

Case 1. First consider the statement $[x{:=}y.sel]^\ell$ (where $x \neq y$) in the case where there is no abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$. Then there is no abstract location for $y.sel$ and hence no abstract location to rename and no binding to establish. Thus we take:

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))\}$$

Note that this situation captures the case where an attempt is made to dereference a `nil`-pointer.

Alternatively, there is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ but there is no abstract location n such that $(n_Y, sel, n) \in \mathsf{H}'$. From the invariants it follows that n_Y is unique but still there is no abstract location to rename and no binding to establish. So again we take:

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))\}$$

This situation captures the case where an attempt is made to dereference a non-existing selector field of pointer.

Case 2. We consider the statement $[x{:=}y.sel]^\ell$ (where $x \neq y$) in the case where there is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and there is an abstract location $n_U \neq n_\emptyset$ such that $(n_Y, sel, n_U) \in \mathsf{H}'$. Both n_Y and n_U will be uniquely determined because of the invariants (and they might be equal). The abstract location n_U will be renamed to include the variable x using the function:

$$h_x^U(n_Z) = \begin{cases} n_{U\cup\{x\}} & \text{if } Z = U \\ n_Z & \text{otherwise} \end{cases}$$

We shall then take

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')\}$$

where $(\mathsf{S}',\mathsf{H}',\mathsf{is}') = \mathit{kill}_x((\mathsf{S},\mathsf{H},\mathsf{is}))$ and

$$\begin{array}{rcl}
\mathsf{S}'' & = & \{(z, h_x^U(n_Z)) \mid (z, n_Z) \in \mathsf{S}'\} \cup \{(x, h_x^U(n_U))\} \\
\mathsf{H}'' & = & \{(h_x^U(n_V), \mathit{sel}', h_x^U(n_W)) \mid (n_V, \mathit{sel}', n_W) \in \mathsf{H}'\} \\
\mathsf{is}'' & = & \{h_x^U(n_Z) \mid n_Z \in \mathsf{is}'\}
\end{array}$$

The inclusion of $(x, h_x^U(n_U))$ in S'' reflects the assignment. The definition of is'' ensures that sharing is preserved by the operation; in particular, $n_{U\cup\{x\}}$ is shared in H'' if and only if n_U is shared in H'.

The effect of the assignment is illustrated in Figure 2.18 in the case where $n_U \in \mathsf{is}$. As before we assume that the abstract locations shown on the figure are distinct so in particular Y, V and W are all distinct from U.

Case 3. We now consider the statement $[x\!:\!=\!y.\mathit{sel}]^\ell$ (where $x \neq y$) in the case where there is an abstract location n_Y such that $(y, n_Y) \in \mathsf{S}'$ and furthermore $(n_Y, \mathit{sel}, n_\emptyset) \in \mathsf{H}'$. As before the invariants ensure that n_Y is uniquely determined. The location $n_\emptyset$ describes the location for $y.\mathit{sel}$ as well as a (possibly empty) set of other locations. We now have to *materialise* a new abstract location $n_{\{x\}}$ from $n_\emptyset$; then $n_{\{x\}}$ will describe the location for $y.\mathit{sel}$ and $n_\emptyset$ will continue to represent the remaining locations. Having introduced a new abstract location we will have to modify the abstract heap accordingly.

This is a potentially difficult operation, so let us consider the following sequence of assignments:

$$\begin{array}{ccccccc}
 & & [x\!:\!=\!\mathtt{nil}]^{\cdots}; & & [x\!:\!=\!y.\mathit{sel}]^\ell; & & [x\!:\!=\!\mathtt{nil}]^{\cdots} \\
 & \uparrow & & \uparrow & & \uparrow & \\
(\mathsf{S},\mathsf{H},\mathsf{is}) & & & (\mathsf{S}',\mathsf{H}',\mathsf{is}') & & (\mathsf{S}'',\mathsf{H}'',\mathsf{is}'') & \\
 & & & & & & (\mathsf{S}''',\mathsf{H}''',\mathsf{is}''')
\end{array}$$

Clearly $[x\!:\!=\!\mathtt{nil}]^{\cdots}; [x\!:\!=\!y.\mathit{sel}]^\ell$ is equivalent to $[x\!:\!=\!y.\mathit{sel}]^\ell$ both in terms of the analysis and the semantics. Indeed, $(\mathsf{S}',\mathsf{H}',\mathsf{is}') = \mathit{kill}_x((\mathsf{S},\mathsf{H},\mathsf{is}))$ represents the effect of removing the binding to x. We are trying to determine candidate shape graphs $(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')$ holding after the assignment $[x\!:\!=\!y.\mathit{sel}]^\ell$ (where $x \neq y$) but let us first study our expectations of $(\mathsf{S}''',\mathsf{H}''',\mathsf{is}''')$. It is immediate that $(\mathsf{S}''',\mathsf{H}''',\mathsf{is}''') = \mathit{kill}_x((\mathsf{S}'',\mathsf{H}'',\mathsf{is}''))$. Furthermore, the states and heaps possible at the point described by $(\mathsf{S}',\mathsf{H}',\mathsf{is}')$ should be the same as those possible at the point described by $(\mathsf{S}''',\mathsf{H}''',\mathsf{is}''')$. This suggests demanding that

$$(\mathsf{S}''',\mathsf{H}''',\mathsf{is}''') = (\mathsf{S}',\mathsf{H}',\mathsf{is}')$$

which means that $\mathit{kill}_x((\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')) = (\mathsf{S}', \mathsf{H}', \mathsf{is}')$. It is also immediate that $(x, n_{\{x\}}) \in \mathsf{S}''$ and that $(n_Y, \mathit{sel}, n_{\{x\}}) \in \mathsf{H}''$.

We shall then take

$$\begin{array}{rl} \phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'') \mid & (\mathsf{S}'', \mathsf{H}'', \mathsf{is}'') \text{ is compatible} \wedge \\ & \mathit{kill}_x((\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')) = (\mathsf{S}', \mathsf{H}', \mathsf{is}') \wedge \\ & (x, n_{\{x\}}) \in \mathsf{S}'' \ \wedge \ (n_Y, \mathit{sel}, n_{\{x\}}) \in \mathsf{H}''\} \end{array}$$

where $(\mathsf{S}', \mathsf{H}', \mathsf{is}') = \mathit{kill}_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$.

It is hopefully clear that we have not missed any shape graphs $(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')$ that might be the result of the assignment. What might be a worry is that we have included an excessive amount of irrelevant shape graphs. (Indeed producing all compatible shape graphs would be trivially sound but also utterly useless.) Although it is possible to do slightly better (see Exercise 2.23) we shall now argue that amount of imprecision in the above definition is not excessive.

We first establish that

$$\mathsf{S}'' = \mathsf{S}' \cup \{(x, n_{\{x\}})\}$$

showing that the abstract state is fully determined. Consider $(z, n_Z) \in \mathsf{S}''$. If $z = x$ it follows from the compatibility of $(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')$ that $n_Z = n_{\{x\}}$. If $z \neq x$ it follows from $(x, n_{\{x\}}) \in \mathsf{S}''$ and the compatibility of $(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')$ that $x \notin Z$ and hence $(z, n_Z) = (z, k_x(n_Z))$ (where k_x is the auxiliary function used to define the kill_x operation). This establishes that $\mathsf{S}'' \subseteq \mathsf{S}' \cup \{(x, n_{\{x\}})\}$. Next consider $(u, n_U) \in \mathsf{S}'$. We know that $u \neq x$ and $x \notin U$ from the definition of S' and from compatibility of $(\mathsf{S}', \mathsf{H}', \mathsf{is}')$. There must exist $(u, n'_U) \in \mathsf{S}''$ such that $k_x(n'_U) = n_U$ but since $x \neq u$ this gives $n'_U = n_U$. It follows that $\mathsf{S}'' \supseteq \mathsf{S}' \cup \{(x, n_{\{x\}})\}$ and we have proved the required equality.

We next establish that

$$\begin{array}{rcl} \mathsf{is}' \setminus \{n_\emptyset\} & = & \mathsf{is}'' \setminus \{n_\emptyset, n_{\{x\}}\} \\ n_\emptyset \in \mathsf{is}' & \text{iff} & n_\emptyset \in \mathsf{is}'' \ \vee \ n_{\{x\}} \in \mathsf{is}'' \end{array}$$

showing that

- abstract locations apart from $n_\emptyset$ retain their sharing information,
- if $n_\emptyset$ is shared then that sharing cannot go away but must give rise to sharing of at least one of $n_\emptyset$ or $n_{\{x\}}$, and
- if $n_\emptyset$ is not shared then no sharing can be introduced for $n_\emptyset$ or $n_{\{x\}}$.

Since both $(\mathsf{S}', \mathsf{H}', \mathsf{is}')$ and $(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')$ are compatible shape graphs it follows that if $n_U \in \mathsf{is}'$ then $x \notin U$ and if $n_U \in \mathsf{is}''$ then $x \notin U \ \vee \ \{x\} = U$. Hence $\mathsf{is}' = \{k_x(n_U) \mid n_U \in \mathsf{is}''\}$ establishes $\mathsf{is}' \setminus \{n_\emptyset\} = \mathsf{is}'' \setminus \{n_\emptyset, n_{\{x\}}\}$

(S, H, is)

Figure 2.19: The effect of $[x{:=}y.sel]^\ell$ in a special case (part 1).

because $k_x(n_U) = n_U \neq n_\emptyset$ for all $n_U \in \mathsf{is}'' \setminus \{n_\emptyset, n_{\{x\}}\}$. Furthermore, $n_\emptyset \in \mathsf{is}'' \vee n_{\{x\}} \in \mathsf{is}''$ gives $n_\emptyset \in \mathsf{is}'$, and $n_\emptyset \notin \mathsf{is}'' \wedge n_{\{x\}} \notin \mathsf{is}''$ gives $n_\emptyset \notin \mathsf{is}'$. Thus we have established the required relationship.

We now turn to the abstract heap. We shall classify the labelled edges (n_V, sel', n_W) into four groups depending on whether or not the source or target may be one of the nodes $n_\emptyset$ or $n_{\{x\}}$:

$$
\begin{array}{rll}
(n_V, sel', n_W) \text{ is } \textit{external} & \text{iff} & \{n_V, n_W\} \cap \{n_\emptyset, n_{\{x\}}\} = \emptyset \\
(n_V, sel', n_W) \text{ is } \textit{internal} & \text{iff} & \{n_V, n_W\} \subseteq \{n_\emptyset, n_{\{x\}}\} \\
(n_V, sel', n_W) \text{ is } \textit{going-out} & \text{iff} & n_V \in \{n_\emptyset, n_{\{x\}}\} \wedge n_W \notin \{n_\emptyset, n_{\{x\}}\} \\
(n_V, sel', n_W) \text{ is } \textit{going-in} & \text{iff} & n_V \notin \{n_\emptyset, n_{\{x\}}\} \wedge n_W \in \{n_\emptyset, n_{\{x\}}\}
\end{array}
$$

We shall also say that two edges (n_V, sel', n_W) and (n'_V, sel'', n'_W) are *related* if and only if $k_x(n_V) = k_x(n'_V)$, $sel' = sel''$ and $k_x(n_W) = k_x(n'_W)$. Clearly an external edge is related only to itself.

Reasoning as above one can show that:

- H' and H'' have the same external edges,
- each internal edge in H' is related to an internal edge in H'' and vice versa,
- each edge going-out in H' is related to an edge going-out in H'' and vice versa, and
- each edge going-in in H' is related to an edge going-in in H'' and vice versa.

Figure 2.20: The effect of $[x{:=}y.sel]^\ell$ in a special case (part 2).

One consideration is that the going-in edge $(n_Y, sel, n_\emptyset) \in \mathsf{H}'$ should be *changed* into the going-in edge $(n_Y, sel, n_{\{x\}}) \in \mathsf{H}''$. We clearly demanded that $(n_Y, sel, n_{\{x\}}) \in \mathsf{H}''$ and because $(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')$ is compatible it follows that $(n_Y, sel, n_\emptyset) \notin \mathsf{H}''$.

As a more concrete illustration consider the scenario in Figure 2.19. Here neither $n_\emptyset$ nor n_W is shared and we assume that both n_V and n_W are distinct from $n_\emptyset$; we also assume that $x \neq y$ and $sel_2 \neq sel_3$. The result of the

transfer function is shown in Figure 2.20. First note that the going-in edge $(n_Y, sel, n_\emptyset) \in \mathsf{H}$ is changed to $(n_Y, sel, n_{\{x\}}) \in \mathsf{H}''_i$ in all shape graphs. Next note that the going-in edge labelled sel_1 can only point to $n_\emptyset$ because $n_{\{x\}}$ is not shared (as $n_\emptyset$ is not) and n_Y points to $n_{\{x\}}$. The going-out edge labelled sel_2 can start at both $n_\emptyset$ and $n_{\{x\}}$ but it cannot do so simultaneously because n_W is not shared. The internal edge labelled sel_3 can only point to $n_\emptyset$ because $n_{\{x\}}$ is not shared and n_Y points to $n_{\{x\}}$; but it can start at both $n_\emptyset$ and $n_{\{x\}}$ and can even do so simultaneously. This explains why there are six shape graphs in $\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is}))$, all of which are clearly needed.

Example 2.53 The statement $[\texttt{t:=x.cdr}]^{51}$ introduced in Example 2.52 is of the form considered here: the transfer function will transform each of the shape graphs of $Shape_\bullet(4)$ and subsequent transformations will produce $Shape_\bullet(5)$. ■

Transfer function for $[x.sel\texttt{:=}a]^\ell$ where a is of the form n, $a_1\ op_a\ a_2$ or `nil`. Again we consider a compatible shape graph $(\mathsf{S}, \mathsf{H}, \mathsf{is})$. First assume that there is no n_X such that $(x, n_X) \in \mathsf{S}$; then x will not point to a cell in the heap and the statement will have no effect on the shape of the heap so the transfer function f_ℓ^{SA} is just the identity. Next assume that there is a (necessarily unique) n_X such that $(x, n_X) \in \mathsf{S}$ but that there is no n_U such that $(n_X, sel, n_U) \in \mathsf{H}$; then the cell pointed to by sel does not point to another cell so the statement will not change the shape of the heap and also in this case the transfer function f_ℓ^{SA} will be the identity.

The interesting case is when there are abstract locations n_X and n_U such that $(x, n_X) \in \mathsf{S}$ and $(n_X, sel, n_U) \in \mathsf{H}$; these abstract locations will be unique because of the invariants. The effect of the assignment will be to remove the triple (n_X, sel, n_U) from H:

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{kill_{x.sel}((\mathsf{S}, \mathsf{H}, \mathsf{is}))\}$$

where $kill_{x.sel}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = (\mathsf{S}', \mathsf{H}', \mathsf{is}')$ is given by:

$$\begin{array}{rcl}
\mathsf{S}' & = & \mathsf{S} \\
\mathsf{H}' & = & \{(n_V, sel', n_W) \mid (n_V, sel', n_W) \in \mathsf{H} \ \wedge\ \neg(X = V \ \wedge\ sel = sel')\} \\
\mathsf{is}' & = & \begin{cases} \mathsf{is}\backslash\{n_U\} & \text{if } \ n_U \in \mathsf{is} \ \wedge\ \#into(n_U, \mathsf{H}') \leq 1 \ \wedge \\ & \qquad \neg\exists sel' : (n_\emptyset, sel', n_U) \in \mathsf{H}' \\ \mathsf{is} & \text{otherwise} \end{cases}
\end{array}$$

The sharing information is as before except that we may be able to do better for the node n_U – we have removed one of the pointers to it and in the case where there is at most one pointer left and it does not have source $n_\emptyset$ the corresponding location will be unshared. This is yet another aspect of strong update. Here we write $\#into(n_U, \mathsf{H}')$ for the number of pointers to n_U in H'. This clause is illustrated in Figure 2.21.

Figure 2.21: The effect of $[x.sel{:=}\texttt{nil}]^\ell$ when $\#into(n_U, \mathsf{H}') \leq 1$.

Remark. Again we shall note that the analysis does *not* incorporate garbage collection: it might be the case that there is only one pointer to the abstract location n_U and that after the assignment $x.sel{:=}\texttt{nil}$ the corresponding location will be unreachable. However, the abstract location may still be part of the shape graph. ■

Transfer function for $[x.sel{:=}y]^\ell$. First assume that $x = y$. The statement is then semantically equivalent to

$$[t{:=}y]^{\ell_1}; [x.sel{:=}t]^{\ell_2}; [t{:=}\texttt{nil}]^{\ell_3}$$

where t is a fresh variable and ℓ_1, ℓ_2 and ℓ_3 are fresh labels. The transfer function f_ℓ^{SA} is then given by

$$f_\ell^{\mathsf{SA}} = f_{\ell_3}^{\mathsf{SA}} \circ f_{\ell_2}^{\mathsf{SA}} \circ f_{\ell_1}^{\mathsf{SA}}$$

The transfer functions $f_{\ell_1}^{\mathsf{SA}}$ and $f_{\ell_3}^{\mathsf{SA}}$ follow the pattern we have seen before so we shall concentrate on the clause for $f_{\ell_2}^{\mathsf{SA}}$, or equivalently, f_ℓ^{SA} in the case where $x \neq y$.

So assume that $x \neq y$ and that $(\mathsf{S}, \mathsf{H}, \mathsf{is})$ is a compatible shape graph. It may be the case that there is no n_X such that $(x, n_X) \in \mathsf{S}$ and in that case the transfer function will be the identity since the statement cannot affect the shape of the heap.

So assume that n_X satisfies $(x, n_X) \in \mathsf{S}$. The case where there is no n_Y such that $(y, n_Y) \in \mathsf{S}$ corresponds to a situation where the value of y is an integer, the `nil`-value or undefined and is therefore similar to the case $[x.sel{:=}\texttt{nil}]^\ell$:

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{kill_{x.sel}((\mathsf{S}, \mathsf{H}, \mathsf{is}))\}$$

The interesting case is when $x \neq y$, $(x, n_X) \in \mathsf{S}$ and $(y, n_Y) \in \mathsf{S}$. The first step will be to remove the binding for $x.sel$ and for this we can use the $kill_{x.sel}$ function. The second step will be to establish the new binding. So we take

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')\}$$

Figure 2.22: The effect of $[x.sel{:=}y]^\ell$ when $\#into(n_Y, \mathsf{H}') < 1$.

where $(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_{x.sel}((\mathsf{S}, \mathsf{H}, \mathsf{is}))$ and

$$\begin{array}{rcl} \mathsf{S}'' & = & \mathsf{S}' \quad (= \mathsf{S}) \\ \mathsf{H}'' & = & \mathsf{H}' \cup \{(n_X, sel, n_Y)\} \\ \mathsf{is}'' & = & \begin{cases} \mathsf{is}' \cup \{n_Y\} & \text{if } \#into(n_Y, \mathsf{H}') \geq 1 \\ \mathsf{is}' & \text{otherwise} \end{cases} \end{array}$$

Note that the node n_Y might become shared when we add a new pointer to it. The effect of the transfer function is illustrated in Figure 2.22.

Example 2.54 This transfer function is illustrated by the assignment $[\mathtt{y.cdr{:=}z}]^6$ of the list reversal program of Example 2.43: each of the shape graphs of $Shape_\bullet(5)$ in Example 2.47 are transformed into one of the shape graphs of $Shape_\bullet(6)$. ■

Transfer function for $[x.sel{:=}y.sel']^\ell$. This statement is equivalent to the sequence of statement

$$[t{:=}y.sel']^{\ell_1}; [x.sel{:=}t]^{\ell_2}; [t{:=}\mathtt{nil}]^{\ell_3}$$

for t being a fresh variable and ℓ_1, ℓ_2 and ℓ_3 being fresh labels. Thus the transfer function f_ℓ^{SA} satisfies

$$f_\ell^{\mathsf{SA}} = f_{\ell_3}^{\mathsf{SA}} \circ f_{\ell_2}^{\mathsf{SA}} \circ f_{\ell_1}^{\mathsf{SA}}$$

The transfer functions $f_{\ell_1}^{\mathsf{SA}}$, $f_{\ell_2}^{\mathsf{SA}}$ and $f_{\ell_3}^{\mathsf{SA}}$ all follow the patterns we have seen before so this completes the specification of the transfer function.

Transfer function for $[\mathtt{malloc}\ p]^\ell$. We first consider the statement $[\mathtt{malloc}\ x]^\ell$ where we have to remove the binding for x and then introduce a new (unshared) location pointed to by x. Thus we define

$$\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) = \{(\mathsf{S}' \cup \{(x, n_{\{x\}})\}, \mathsf{H}', \mathsf{is}')\}$$

where $(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x(\mathsf{S}, \mathsf{H}, \mathsf{is})$.

The statement $[\texttt{malloc}\ (x.sel)]^{\ell}$ is equivalent to the sequence

$$[\texttt{malloc}\ t]^{\ell_1}; [x.sel\texttt{:=}t]^{\ell_2}; [t\texttt{:=nil}]^{\ell_3}$$

where t is a fresh variable and ℓ_1, ℓ_2 and ℓ_3 are fresh labels. The transfer function f_{ℓ}^{SA} is then

$$f_{\ell}^{\mathsf{SA}} = f_{\ell_3}^{\mathsf{SA}} \circ f_{\ell_2}^{\mathsf{SA}} \circ f_{\ell_1}^{\mathsf{SA}}$$

The transfer functions $f_{\ell_1}^{\mathsf{SA}}$, $f_{\ell_2}^{\mathsf{SA}}$ and $f_{\ell_3}^{\mathsf{SA}}$ all follow the patterns we have seen before so this completes the specification of the transfer function.

Concluding Remarks

Data Flow Analysis for imperative languages. As mentioned in the beginning of this chapter, Data Flow Analysis has a long tradition. Most compiler textbooks contain sections on optimisation which mainly discuss Data Flow Analyses and their implementation [5, 55, 181]. The emphasis in these books is often on practical implementations of data flow analyses. A classic textbook which provides a more theoretical treatment of the subject is by Hecht [69]; the book contains a detailed discussion of the four example Data Flow Analyses in Section 2.1, and also presents a more traditional treatment of Monotone Frameworks based on the use of semi-lattices as well as a number of algorithms (see Chapter 6 for a more thorough treatment of algorithms). Marlowe and Ryder [103] provide a survey of data flow frameworks. Steffen [164] and Schmidt [151] express data flow analyses using modal logic (rather than equations) thereby opening up the possibility of using model checking techniques for program analysis.

The examples presented in Section 2.1 are fairly standard. Alternative treatments of this material can be found in any of the books already cited. The examples may all be represented as Bit Vector Frameworks (see Exercise 2.9): the lattice elements may be represented by a vector of bits and the lattice operations efficiently implemented as boolean operations. The method used in Section 2.2 to prove the correctness of the Live Variables Analysis is adapted from [112] and is expressed by means of an operational semantics [140, 130]. The notion of faint variables, introduced in Exercise 2.4, was first introduced by Giegerich, Möncke and Wilhelm [65].

The use of semi-lattices in Data Flow Analysis was first proposed in [96]. The notion of Monotone Frameworks is due to Kam and Ullman [93]. These early papers, and much of the later literature, use the dual notions (meets and maximal fixed points) to our presentation. Kam and Ullman [93] prove that the existence of a general algorithm to compute MOP solutions would imply the decidability of the Modified Post Correspondence Problem [76]. Cousot and Cousot [37] model abstract program properties by complete semi-lattices in their paper on Abstract Interpretation (see Chapter 4).

We have associated transfer functions with elementary blocks. It would be possible to associate transfer functions with flows instead as e.g. in [147]. These two approaches have equal power: to go from the first to the second, the transfer functions may be moved from the blocks to their outgoing flows; to go from the second to the first, we can introduce artificial blocks. In fact artificial blocks can be avoided as shown in Exercise 2.11.

Most of the papers that we have cited so far concentrate on intraprocedural analysis. An early, and influential, paper on interprocedural analysis is [155] that studies two approaches to establishing context. One is based on call strings and expresses aspects of the dynamic calling context; our presentation is inspired by [178]. The other is the "functional approach" that is based on data and that shares some of the aims of assumption sets [99, 138, 145]; the technical formulation is different because [155] obtains the effect by calculating the transfer functions for the call statement. Most of the subsequent papers in the literature can be seen as variations and combinations over this theme; a substantial effort in this direction may be found in [44]. As mentioned in Section 2.5.5, the use of large assumption sets may lead to equation systems where the transfer functions are not monotone; we refer to [51, 52] for a modern presentation of techniques that allow the solution of so-called *weakly monotonic* systems of equations.

Pointer analysis. There is an extensive literature on the analysis of alias problems for languages with pointers. Following [62] we can distinguish between analyses of pointers to (1) *statically* allocated data (typically on the stack) and (2) *dynamically* allocated data (typically in the heap). The analysis of pointers to statically allocated data is the simplest: typically the data will have compile-time names and the analysis result can be presented as a set of points-to pairs of the form (p, x) meaning that the pointer p points to the data x or as alias pairs of the form $(*p, x)$ meaning that $*p$ and x are aliased. Analyses in this category include [47, 100, 145, 182, 162, 153].

The analysis of dynamically allocated data is more complicated since the objects of interest are inherently anonymous. The simplest analyses [38, 61] study the *connectivity* of the heap: they attempt to split the heap into disjoint parts and do not keep any information about the internal structure of the individual parts. These analyses have been found quite useful for many applications.

The more complex analyses of dynamically allocated data give more precise information about the *shape* of the heap. A number of approaches use *graphs* to represent the heap. A main distinction between these approaches is how they map a heap of potentially unbounded size to a graph of bounded size: some bound the length of paths in the heap [84, 165], others merge heap cells created at the same program point [85, 30], and yet others merge heap cells that cannot be kept apart by the set of pointer variables pointing to them [148, 149]. Another group of analyses obtain information about the shape

of the heap by more directly approximating the *access paths*. Here a main distinction is the kind of properties of the access paths that are recorded: some focus on simple connectivity properties [62], others use some limited form of regular expressions [101], and yet others use monomial relations [45].

The analysis presented in Section 2.6 is based on the work of Sagiv, Reps and Wilhelm [148, 149]. In contrast to [148, 149] it uses *sets* of compatible shape graphs; [148, 149] merge sets of compatible shape graphs into a *single* summary shape graph and then use various mechanisms for extracting parts of the individual compatible shape graphs and in this way an exponential factor in the cost of the analysis can be avoided. The sharing component of the shape graphs is designed to detect list-like properties; it can be replaced by other components detecting other shape properties [150].

Static Single Assignments. Some program analyses can be performed more efficiently or more accurately when the program is transformed into an intermediate form called *static single assignment* (SSA) form [42]. The benefit of SSA form is that the definition-use chains of the program are explicit in the representation: each variable has at most one definition (meaning that it is assigned at most once). As a consequence some optimisations can be performed more efficiently using this representation [12, 180, 109, 110].

The transformation to SSA form amounts to renaming the variables and introducing special assignments at the points where flow of control might join. The assignments use so-called ϕ-functions; each argument position of the ϕ-function identifies one of the program points where flow of control might come from. The special statements have the form $x := \phi(x_1, \cdots, x_n)$ and the idea is that the value of x will equal the value of x_i whenever control comes from the i'th predecessor. The algorithms for transforming to SSA form often proceed in two stages: the first stage identifies the points where flow of control might join and where the special assignments are to be inserted, and the second stage renames the variables to ensure that each of them is assigned at most once. To obtain a compact representation of the program one wants to minimise the number of extra statements (and variables) and there are techniques based on additional data flow information for achieving this.

Data Flow Analysis for other language paradigms. The analysis techniques that we have studied assume the existence of some representation of the flow of control in the program. For the class of imperative languages that we have studied, it is relatively easy to determine this control flow information. For many languages, for example functional programming languages, this is not the case. The next chapter presents techniques for determining control flow information for such languages and shows how Data Flow Analysis can be integrated with control flow analysis.

The techniques we have presented can be applied directly to other language paradigms. Two examples are in object-oriented programming and a communicating processes language. Vitek, Horspool and Uhl [178] present an analysis for object-oriented languages which determines classes of objects and their lifetimes. Their analysis is an interprocedural analysis that uses a graph-based representation of the memory as data flow values. Reif and Smolka [142] apply Data Flow Analysis techniques to distributed communicating processes to detect unreachable code and to determine the values of program expressions. They apply their analysis to a language with asynchronous communication. Their reachability analysis is based on an algorithm that builds a spanning tree for each process flow graph and links matching transmits and receives between processes. They construct a Monotone Framework for determining value sets.

Intraprocedural control flow analysis. Many compilers transform the source program into an intermediate form consisting of sequences of fairly low-level instructions (like three address code) and then perform the optimisations based on this representation. For this to work more information is needed about the flow of control within the individual program segments; in our terminology we need the *flow* (or $flow^R$) relation in order to apply the data flow analysis techniques. It is the task of intraprocedural control flow analysis to provide this information.

More refined forms of intraprocedural control flow analysis include *structural analysis* [154, 110] that aims at discovering a wider variety of control structures in the code – control structures like conditionals, `while`-loops and `repeat`-loops that resemble those of the source language. The starting point for structural analysis is the flow graph; it is examined to identify instances of the various control structures, the instances are replaced by abstract nodes and the connecting edges are collapsed; this process is repeated until the flow graph has been collapsed into a single abstract node. The approach described above is a refinement of classical techniques based on identifying natural loops (or interval analysis [69, 5, 110]) intended to provide more meaningful program structure.

We refer to the Concluding Remarks of Chapter 6 for a discussion of systems implementing data flow analysers.

Figure 2.23: *du*-and *ud*-chains.

Mini Projects

Mini Project 2.1 *ud*- and *du*-chains

The aim of this mini project is to develop a more thorough understanding of the concepts of *ud*- and *du*-chains introduced in Subsection 2.1.5.

1. The function *ud* is specified in terms of definition clear paths, whilst UD re-uses functions introduced for the Reaching Definitions and Live Variables Analyses. Prove that the two functions compute the same information.

2. DU can be defined by analogy with UD. Starting from the definition of *du*, develop an equational definition of DU and verify its correctness.

3. A *Constant Propagation Analysis* is presented in Subsection 2.3.3; an alternative approach would be to use *du*- and *ud*-chains. Suppose there is a block $[x := n]^{\ell}$ that assigns a constant n to a variable x. By following the *du*-chain it is possible to find all blocks using the variable x. It is only safe to replace a use of x by the constant n in a block ℓ' if all other definitions that reach ℓ' also assign the same constant n to x. This can be determined by using the *ud*-chain. This is illustrated in Figure 2.23. Considering the program of Example 2.12, *Constant Folding* (followed by *Dead Code Elimination*) can be used to produce the following program:

   $$(\texttt{if } [\texttt{z=3}]^3 \texttt{ then } [\texttt{z:=0}]^4 \texttt{ else } [\texttt{z:=3}]^5); [\texttt{y:=3}]^6; [\texttt{x:=3+z}]^7$$

   Develop a formal description of this analysis. ■

$[ass]$	$\langle [x := a]^\ell, \sigma, tr\rangle \to \langle \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma], tr : (x, \ell)\rangle$	
$[skip]$	$\langle [\texttt{skip}]^\ell, \sigma, tr\rangle \to \langle \sigma, tr\rangle$	
$[seq_1]$	$\dfrac{\langle S_1, \sigma, tr\rangle \to \langle S_1', \sigma', tr'\rangle}{\langle S_1; S_2, \sigma, tr\rangle \to \langle S_1'; S_2, \sigma', tr'\rangle}$	
$[seq_2]$	$\dfrac{\langle S_1, \sigma, tr\rangle \to \langle \sigma', tr'\rangle}{\langle S_1; S_2, \sigma, tr\rangle \to \langle S_2, \sigma', tr'\rangle}$	
$[if_1]$	$\langle \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2, \sigma, tr\rangle \to \langle S_1, \sigma, tr\rangle$	if $\mathcal{B}[\![b]\!]\sigma = \mathit{true}$
$[if_2]$	$\langle \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2, \sigma, tr\rangle \to \langle S_2, \sigma, tr\rangle$	if $\mathcal{B}[\![b]\!]\sigma = \mathit{false}$
$[wh_1]$	$\langle \texttt{while } [b]^\ell \texttt{ do } S, \sigma, tr\rangle \to \langle (S; \texttt{while } [b]^\ell \texttt{ do } S), \sigma, tr\rangle$	if $\mathcal{B}[\![b]\!]\sigma = \mathit{true}$
$[wh_2]$	$\langle \texttt{while } [b]^\ell \texttt{ do } S, \sigma, tr\rangle \to \langle \sigma, tr\rangle$	if $\mathcal{B}[\![b]\!]\sigma = \mathit{false}$

Table 2.10: The instrumented semantics of WHILE.

Mini Project 2.2 Correctness of Reaching Definitions

The aim of this mini project is to prove the correctness of Reaching Definitions with respect to the notion of semantic reaching definitions introduced in Section 1.5. To get a precise definition of the set of traces of interest we shall begin by introducing a so-called *instrumented semantics*: an extension of a more traditional semantics that keeps track of additional information that is mainly of interest for the program analysis.

The instrumented semantics has transitions of the forms:

$$\langle S, \sigma, tr\rangle \to \langle \sigma', tr'\rangle \quad \text{and} \quad \langle S, \sigma, tr\rangle \to \langle S', \sigma', tr'\rangle$$

All configurations include a *trace* $tr \in \mathbf{Trace} = (\mathbf{Var} \times \mathbf{Lab})^*$ that records the elementary block in which a variable is being assigned. The detailed definition of the instrumented semantics is given in Table 2.10.

Given a program $S_\star$ and an initial state $\sigma_\star \in \mathbf{State}$ it is natural to construct the trace

$$tr_\star = ((x_1, ?), \cdots, (x_n, ?))$$

where $x_1, \cdots, x_n$ are the variables in $\mathbf{Var}_\star$ and to consider the finite derivation

sequence:

$$\langle S_\star, \sigma_\star, tr_\star \rangle \rightarrow^* \langle \sigma', tr' \rangle$$

It should be intuitively obvious (and can be proved formally) that

$$tr' \in \mathbf{TrVar}_\star^? = \{tr \in \mathbf{Trace} \mid \forall x \in \mathbf{Var}_\star : \exists \ell \in \mathbf{Lab}_\star^? : (x, \ell) \text{ occurs in } tr\}$$

Intuitively (and as can be proved formally), there should be a similar derivation sequence $\langle S_\star, \sigma_\star \rangle \rightarrow^* \sigma'$ in the Structural Operational Semantics. Similar remarks apply to infinite derivation sequences.

As in Section 2.2 we shall study the constraint system $\mathsf{RD}^{\subseteq}(S_\star)$ corresponding to the equation system $\mathsf{RD}^{=}(S_\star)$. Let *reach* be a collection of functions:

$$reach_{entry}, reach_{exit} : \mathbf{Lab}_\star \rightarrow \mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star^?)$$

We say that *reach* solves $\mathsf{RD}^{\subseteq}(S)$, and write

$$reach \models \mathsf{RD}^{\subseteq}(S)$$

if the functions satisfy the constraints; similarly for $reach \models \mathsf{RD}^{=}(S)$.

1. Formulate and prove results corresponding to Lemmas 2.15, 2.16 and 2.18.

The correctness relation $\sim$ will relate traces $tr \in \mathbf{Trace}$ to the information obtained by the analysis. Let $Y \subseteq \mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star^?)$ and define

$$tr \sim Y \quad \text{iff} \quad \forall x \in \mathsf{DOM}(tr) : (x, \mathsf{SRD}(tr)(x)) \in Y$$

meaning that Y contains at least the semantically reaching definitions obtained from the trace tr by the function SRD introduced in Section 1.5. (Note that $\mathsf{DOM}(tr) = \mathbf{Var}_\star$ whenever $tr \in \mathbf{TrVar}_\star^?$.)

2. Formulate and prove results corresponding to Lemma 2.20, Theorem 2.21 and Corollary 2.22. ■

Mini Project 2.3 A Prototype Implementation

In this mini project we shall implement one of the program analyses considered in Section 2.1. As implementation language we shall choose a functional language such as Standard ML or Haskell. We can then define a suitable

data type for WHILE programs as follows:

```
type var        =  string
type label      =  int
datatype aexp   =  Var of var | Const of int
                |  Op of string * aexp * aexp
and bexp        =  True | False
                |  Not of bexp | Boolop of string * bexp * bexp
                |  Relop of string * aexp * aexp
datatype stat   =  Assign of var * aexp * label | Skip of label
                |  Seq of stat * stat | If of bexp * label * stat * stat
                |  While of bexp * label * stat
```

Now proceed as follows:

1. Implement the operations *init*, *final*, *flow*, $flow^R$ and *blocks*.
2. Generate the data flow equations for the Live Variables Analysis of Subsection 2.1.4.
3. Solve the data flow equations; the function should be based on the algorithm of Section 2.4.

For the more ambitious: generalise your program to accept an instance of a Monotone Framework as input. ■

Exercises

Exercise 2.1 Formulate data flow equations for the Reaching Definitions Analysis of the program studied in Example 1.1 of Chapter 1 and in particular define the appropriate *gen* and *kill* functions. ■

Exercise 2.2 Consider the following program:

$$[\texttt{x:=1}]^1; (\texttt{while } [\texttt{y>0}]^2 \texttt{ do } [\texttt{x:=x-1}]^3); [\texttt{x:=2}]^4$$

Perform a Live Variables Analysis for this program using the equations of Section 2.1.4. ■

Exercise 2.3 A modification of the Available Expressions Analysis detects when an expression is available *in a particular variable*: a non-trivial expression a is available in x at a label ℓ if it has been evaluated and assigned to x on all paths leading to ℓ and if the values of x and the variables in the expression have not changed since then. Write down the data flow equations and any auxiliary functions for this analysis. ■

Exercise 2.4 Consider the following program:

$$[\texttt{x:=1}]^1; [\texttt{x:=x-1}]^2; [\texttt{x:=2}]^3$$

Clearly x is dead at the exits from 2 and 3. But x is live at the exit of 1 even though its only use is to calculate a new value for a variable that turns out to be dead. We shall say that a variable is a *faint variable* if it is dead or if it is only used to calculate new values for faint variables; otherwise it is *strongly live*. In the example x is faint at the exits from 1, 2 and 3. Define a Data Flow Analysis that detects strongly live variables. (Hint: For an assignment $[x := a]^\ell$ the definition of $f_\ell(l)$ should be by cases on whether x is in l or not.) ■

Exercise 2.5 A *basic block* is often taken to be a maximal group of statements such that all transfers to the block are to the first statement in the group and, once the block has been entered, all statements in the group are executed sequentially. In this exercise we shall consider basic blocks of the form

$$[x_1 := a_1; \cdots; x_n := a_n; B]^\ell$$

where $n \geq 0$ and B is $x := a$, `skip` or b. Reformulate the analyses of Section 2.1 for this more general notion of basic block. ■

Exercise 2.6 Consider the analyses Available Expressions and Reaching Definitions. Which of the equations make sense for programs that do not have isolated entries (and how can this be improved)? Similarly, which of the equations for Very Busy Expressions and Live Variables make sense for programs that do not have isolated exits (and how can this be improved)? (Hint: See the beginning of Section 2.3.) ■

Exercise 2.7 Consider the correctness proof for the Live Variables Analysis in Section 2.2. Give a compositional definition of $\mathsf{LV}^=(\cdots)$ for a label consistent statement using

$$\mathsf{LV}^=([\texttt{skip}]^\ell) = \{\mathsf{LV}_{exit}(\ell) = \mathsf{LV}_{entry}(\ell)\}$$

as one of the clauses and observe that a similar development is possible for $\mathsf{LV}^\subseteq(\cdots)$. Give a formal definition of $live \models C$ where C is a set of equalities or inclusions as might have been produced by $\mathsf{LV}^=(S)$ or $\mathsf{LV}^\subseteq(S)$.

Prove that $\{live \mid live \models \mathsf{LV}^\subseteq(S)\}$ is a *Moore family* in the sense of Appendix A (with $\sqcap$ being $\bigcap$ and determine whether or not a similar result holds for $\{live \mid live \models \mathsf{LV}^=(S)\}$. ■

Exercise 2.8 Show that Constant Propagation is a Monotone Framework with the set $\mathcal{F}_{\mathsf{CP}}$ as defined in Section 2.3.3. ■

Exercise 2.9 A *Bit Vector Framework* is a special instance of a Monotone Framework where

- $L = (\mathcal{P}(D), \sqsubseteq)$ for some finite set D and where $\sqsubseteq$ is either $\subseteq$ or $\supseteq$, and
- $\mathcal{F} = \{f : \mathcal{P}(D) \to \mathcal{P}(D) \mid \exists Y_f^1, Y_f^2 \subseteq D : \forall Y \subseteq D : f(Y) = (Y \cap Y_f^1) \cup Y_f^2\}$.

Show that the four classical analyses of Section 2.1 are Bit Vector Frameworks. Show that all Bit Vector Frameworks are indeed Distributive Frameworks. Devise a Distributive Framework that is not also a Bit Vector Framework. ■

Exercise 2.10 Consider the Constant Propagation Analysis of Section 2.3.3 and the program

$$(\texttt{if } [\cdots]^1 \texttt{ then } [\texttt{x:=-1}]^2; [\texttt{y:=1}]^3 \texttt{ else } [\texttt{x:=1}]^4; [\texttt{y:=-1}]^5); [\texttt{z:=x*y}]^6$$

Show that $MFP_\bullet(6)$ differs from $MOP_\bullet(6)$. ■

Exercise 2.11 In our formulation of Monotone Frameworks we associate transfer functions with basic blocks. In a statement of the form

$$\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2$$

this prevents us from using the result of the test to pass different information to S_1 and S_2; as an example suppose that `x` is known to be positive or negative and that b is the test `x>0`, then `x` is always positive at the entry to S_1 and always negative at the entry to S_2. To remedy this deficiency consider writing $[b]^\ell$ as $[b]^{\ell_1,\ell_2}$ where ℓ_1 corresponds to b evaluating to true and ℓ_2 corresponds to b evaluating to false. Make the necessary changes to the development in Sections 2.1 and 2.3. (Begin by considering forward analyses.) ■

Exercise 2.12 Consider one of the analyses Available Expressions, Very Busy Expressions and Live Variables Analysis and perform a complexity analysis in the manner of Example 2.30. ■

Exercise 2.13 Let F be $flow(S_\star)$ and E be $\{init(S_\star)\}$ for a label consistent program $S_\star$. Show that

$$\forall \ell \in \mathbf{Lab}_\star : path_\circ(\ell) \neq \emptyset$$

Prove a similar result when F is $flow^R(S_\star)$ and E is $final(S_\star)$. ■

Exercise 2.14 In a Detection of Signs Analysis one models all negative numbers by the symbol -, zero by the symbol 0, and all positive numbers by the symbol +. As an example, the set $\{-2, -1, 1\}$ is modelled by the set $\{\texttt{-}, \texttt{+}\}$, that is an element of the powerset $\mathcal{P}(\{\texttt{-}, \texttt{0}, \texttt{+}\})$.

Let $S_\star$ be a program and $\mathbf{Var}_\star$ be the finite set of variables in $S_\star$. Take L to be $\mathbf{Var}_\star \to \mathcal{P}(\{\texttt{-}, \texttt{0}, \texttt{+}\})$ and define an instance $(L, \mathcal{F}, F, E, \iota, f_\cdot)$ of a Monotone Framework for performing Detection of Signs Analysis.

Similarly, take L' to be $\mathcal{P}(\mathbf{Var}_\star \times \{\texttt{-}, \texttt{0}, \texttt{+}\})$ and define an instance $(L', \mathcal{F}', F', E', \iota', f'_\cdot)$ of a Monotone Framework for Detection of Signs Analysis. Is there any difference in the precision obtained by the two approaches? ■

Exercise 2.15 In the previous exercise we defined a Detection of Signs Analysis that could *not* record the interdependencies between signs of variables (e.g. that two variables x and y always will have the same sign); this is sometimes called an *independent attribute analysis*. In this exercise we shall consider a variant of the analysis that is able to record the interdependencies between signs of variables; this is sometimes called a *relational analysis*. To do so take L to be $\mathcal{P}(\mathbf{Var}_\star \to \{\texttt{-}, \texttt{0}, \texttt{+}\})$ and define an instance $(L, \mathcal{F}, F, E, \iota, f_\cdot)$ of a Monotone Framework for performing Detection of Signs Analysis. Construct an example showing that the result of this relational analysis may be more informative than that of the independent attribute analysis. The distinction between independent attribute methods and relational methods is further discussed in Chapter 4. ■

Exercise 2.16 The interprocedural analysis using bounded call strings uses contexts to record the last k call sites. Reformulate the analysis for a notion of context that records the last k *distinct* call sites. Discuss whether or not this analysis is useful for distinguishing between the call of a procedure and subsequent recursive calls. ■

Exercise 2.17 Consider the Fibonacci program of Example 2.33 and the Detection of Signs Analysis of Exercise 2.15 and Example 2.36. Construct the data flow equations corresponding to using large and small assumption sets, respectively. ■

Exercise 2.18 Choose one of the four classical analyses from Section 2.1 and formulate it as an interprocedural analysis based on call strings. (Hint: Some may be easier than others.) ■

Exercise 2.19 Extend the syntax of programs to have the form

$$\texttt{begin}\ D_\star;\ \texttt{input}\ x;\ S_\star;\ \texttt{output}\ y\ \texttt{end}$$

so that it maps integers to integers rather than states to states. Consider the Detection of Signs Analysis and define the transfer functions for the input and output statements. ■

Exercise 2.20 Consider extending the procedure language such that procedures can have multiple call-by-value, call-by-result and call-by-value-result parameters as well as local variables and reconsider the Detection of Signs Analysis. How should one define the transfer functions associated with procedure call, procedure entry, procedure exit, and procedure return? ■

Exercise* 2.21 Compute the shape graphs $Shape_\bullet(1), \cdots, Shape_\bullet(7)$ of Example 2.47 using the information supplied in Examples 2.48, $\cdots$, 2.54. (Warning: there will be more than 50 shape graphs.) ■

Exercise 2.22 In the Shape Analysis of Section 2.6 work out direct definitions of the transfer functions for elementary statements of the forms $[x{:=}x.sel]^\ell$, $[x.sel{:=}x]^\ell$, $[x.sel{:=}x.sel']^\ell$ and $[\texttt{malloc}\ (x.sel)]^\ell$. ■

Exercise* 2.23 Consider Case 3 in the definition of the transfer function for $[x{:=}y.sel]^\ell$ (where $x \neq y$) in the Shape Analysis. Make a careful analysis of internal, going-in and going-out edges and determine whether or not some of the shape graphs $(\mathsf{S}'', \mathsf{H}'', \mathsf{is}'')$ in $\phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is}))$ can be removed by placing stronger demands on the edges in H'' compared to those in H' (where $(\mathsf{S}', \mathsf{H}', \mathsf{is}') = kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is})))$. ■

Exercise* 2.24 The Shape Analysis as presented in Section 2.6 does not take garbage collection into account. Modify the Structural Operational Semantics of the pointer language to perform garbage collection and subsequently modify the analysis to reflect this. ■

Exercise* 2.25 The use of a single abstract summary location leads to a certain amount of inaccuracy in the Shape Analysis. A more accurate analysis could associate allocation sites with the abstract locations. An abstract location would then have the form $n_{\ell,X}$ where ℓ is an allocation site (a label of a `malloc`-statement) and X is a set of variables as before. Develop the transfer functions for the new analysis. ■

Chapter 3

Constraint Based Analysis

In this chapter we present the technique of Constraint Based Analysis using a simple functional language, FUN. We begin by presenting an abstract specification of a Control Flow Analysis and then study its theoretical properties: it is correct with respect to a Structural Operational Semantics and it can be used to analyse all programs. This specification of the analysis does not immediately lend itself to an efficient algorithm for computing a solution so we proceed by developing first a syntax directed specification and then a constraint based formulation and finally we show how the constraints can be solved. We conclude by illustrating how the precision of the analysis can be improved by combining it with Data Flow Analysis and by incorporating context information thereby linking up with the development of the previous chapter.

3.1 Abstract 0-CFA Analysis

In Chapter 2 we saw how properties of data could be propagated through a program. In developing the specification we relied on the ability to identify for each program fragment all the possible successor (and predecessor) fragments via the operator *flow* (and flow^R) and the interprocedural flow $\mathit{inter\text{-}flow}_\star$ (and $\mathit{inter\text{-}flow}_\star^R$). The usefulness of the resulting specification was due to the number of successors and predecessors being small (usually just one or two except for procedure exits). This is a typical feature of imperative programs without procedures but it usually fails for more general languages, whether imperative languages with procedures as parameters, functional languages, or object-oriented languages. In particular, the interprocedural techniques of Section 2.5 provide a solution for the simpler cases where the program text allows one to limit the number of successors, as is the case when a proce-

dure call is performed by explicitly mentioning the name of the procedure. However, these techniques are not powerful enough to handle the *dynamic dispatch problem* where variables can denote procedures. In Section 1.4 we illustrated this by the functional program

```
let f = fn x => x 1;
    g = fn y => y+2;
    h = fn z => z+3
in (f g) + (f h)
```

where the function application `x 1` in the body of `f` will transfer control to the body of the function `x`, and here it is not so obvious what program fragment this actually is, since `x` is the formal parameter of `f`. The Control Flow Analysis of the present chapter will provide a solution to the dynamic dispatch problem by determining for each subexpression a hopefully small number of functions that it may evaluate to; thereby it will determine where the flow of control may be transferred to in the case where the subexpression is the operator of a function application. In short, Control Flow Analysis will determine the *interprocedural flow* information (*inter-flow*$_\star$ or *IF*) upon which the development of Section 2.5 is based.

Syntax of the FUN language. For the main part of this chapter we shall concentrate on a small functional language: the untyped lambda calculus extended with explicit operators for recursion, conditional and local definitions. The purpose of the Control Flow Analysis will be to compute for *each* subexpression the set of functions that it could evaluate to, and to express this it is important that we are able to label *all* program fragments. We shall be very explicit about this: a program fragment with a label is called an *expression* whereas a program fragment without a label is called a *term*. So we use the following syntactic categories:

e	$\in$	**Exp**	expressions (or labelled terms)
t	$\in$	**Term**	terms (or unlabelled expressions)

We assume that a countable set of variables is given and that constants (including the truth values), binary operators (including the usual arithmetic, boolean and relational operators) and labels are left unspecified:

f, x	$\in$	**Var**	variables
c	$\in$	**Const**	constants
op	$\in$	**Op**	binary operators
ℓ	$\in$	**Lab**	labels

The *abstract syntax* of the language is now given by:

$$
\begin{array}{rcl}
e & ::= & t^\ell \\
t & ::= & c \mid x \mid \texttt{fn}\ x\ \texttt{=>}\ e_0 \mid \texttt{fun}\ f\ x\ \texttt{=>}\ e_0 \mid e_1\ e_2 \\
 & \mid & \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mid e_1\ op\ e_2
\end{array}
$$

Here `fn` x `=>` e_0 is a function definition (or function abstraction) whereas `fun` f x `=>` e_0 is a recursive variant of `fn` x `=>` e_0 where all free occurrences of f in e_0 refer to `fun` f x `=>` e_0 itself. The construct `let` x `=` e_1 `in` e_2 is a non-recursive local definition that is semantically equivalent to (`fn` x `=>` e_2)(e_1). As usual we shall use parentheses to disambiguate the parsing whenever needed. Also we shall *assume* throughout that in all occurrences of `fun` f x `=>` e_0, f and x are *distinct* variables.

We shall need the notion of *free variables* of expressions and terms so we define the function

$$FV : (\mathbf{Term} \cup \mathbf{Exp}) \to \mathcal{P}(\mathbf{Var})$$

in the following standard way. The abstractions `fn` x `=>` e_0 and `fun` f x `=>` e_0 contain binding occurrences of variables so $FV($`fn` x `=>` $e_0) = FV(e_0) \setminus \{x\}$ and similarly $FV($`fun` f x `=>` $e_0) = FV(e_0) \setminus \{f, x\}$. Similarly, `let` x `=` e_1 `in` e_2 contains a binding occurrence of x so we have $FV($`let` x `=` e_1 `in` $e_2) = FV(e_1) \cup (FV(e_2) \setminus \{x\})$ reflecting that free occurrences of x in e_1 are bound outside the construct. The remaining clauses for FV are straightforward.

Example 3.1 The functional program (`fn x => x`) (`fn y => y`) considered in Section 1.4 is now written as:

$$((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5$$

Compared with the notation of Example 1.2 we have omitted the square brackets. ■

Example 3.2 Consider the following expression, loop, of FUN:

$$\begin{array}{l}(\texttt{let g = (fun f x => (f}^1\ (\texttt{fn y => y}^2)^3)^4)^5\\ \ \texttt{in (g}^6\ (\texttt{fn z => z}^7)^8)^9)^{10}\end{array}$$

It defines a function `g` that is applied to the identity function `fn z => z`^{7}. The function `g` is defined recursively: `f` is its local name and `x` is the formal parameter. Hence the function will ignore its actual parameter and call itself recursively with the argument `fn y => y`^{2}. This will happen again and again so the program loops. ■

3.1.1 The Analysis

Abstract domains. We shall now show how to specify 0-CFA analyses. These may be regarded as the simplest possible form of Control Flow Analysis in that *no* context information is taken into account. (As will become clear in Section 3.6, this is what the number 0 is indicating.)

The result of a 0-CFA analysis is a pair $(\widehat{\mathsf{C}}, \widehat{\rho})$ where:

- $\widehat{\mathsf{C}}$ is the *abstract cache* associating abstract values with each labelled program point.
- $\widehat{\rho}$ is the *abstract environment* associating abstract values with each variable.

This is made precise by:

$$\begin{array}{lcll} \widehat{v} & \in & \widehat{\mathbf{Val}} & = \mathcal{P}(\mathbf{Term}) & \text{abstract values} \\ \widehat{\rho} & \in & \widehat{\mathbf{Env}} & = \mathbf{Var} \to \widehat{\mathbf{Val}} & \text{abstract environments} \\ \widehat{\mathsf{C}} & \in & \widehat{\mathbf{Cache}} & = \mathbf{Lab} \to \widehat{\mathbf{Val}} & \text{abstract caches} \end{array}$$

Here an *abstract value* $\widehat{v}$ is an abstraction of a set of functions: it is a set of terms of the form `fn` x `=>` e_0 or `fun` f x `=>` e_0. We will not be recording any constants in the abstract values because the analysis we shall specify is a pure Control Flow Analysis with no Data Flow Analysis component; in Section 3.5 we shall show how to extend it with Data Flow Analysis components. Furthermore, we do *not* need to assume that all bound variables are distinct and that all labels are distinct, but clearly, greater precision is achieved if this is the case; this means that semantically equivalent programs can have different analysis results and this is a common feature of all approaches to program analysis. As we shall see an *abstract environment* is an abstraction of a set of environments occurring in closures at run-time (see the semantics in Subsection 3.2.1). In a similar way an *abstract cache* might be considered as an abstraction of a set of execution profiles: as discussed below some texts prefer to combine the abstract environment with the abstract cache.

Example 3.3 Consider the expression $((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5$ of Example 3.1. The following table contains three guesses of a 0-CFA analysis:

	$(\widehat{\mathsf{C}}_e, \widehat{\rho}_e)$	$(\widehat{\mathsf{C}}'_e, \widehat{\rho}'_e)$	$(\widehat{\mathsf{C}}''_e, \widehat{\rho}''_e)$
1	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn x => x}^1, \texttt{fn y => y}^3\}$
2	$\{\texttt{fn x => x}^1\}$	$\{\texttt{fn x => x}^1\}$	$\{\texttt{fn x => x}^1, \texttt{fn y => y}^3\}$
3	$\emptyset$	$\emptyset$	$\{\texttt{fn x => x}^1, \texttt{fn y => y}^3\}$
4	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn x => x}^1, \texttt{fn y => y}^3\}$
5	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn x => x}^1, \texttt{fn y => y}^3\}$
x	$\{\texttt{fn y => y}^3\}$	$\emptyset$	$\{\texttt{fn x => x}^1, \texttt{fn y => y}^3\}$
y	$\emptyset$	$\emptyset$	$\{\texttt{fn x => x}^1, \texttt{fn y => y}^3\}$

Intuitively, the guess $(\widehat{\mathsf{C}}_e, \widehat{\rho}_e)$ of the first column is acceptable whereas the guess $(\widehat{\mathsf{C}}'_e, \widehat{\rho}'_e)$ of the second column is wrong: it would seem that `fn x =>` $\texttt{x}^1$ is never called since $\widehat{\rho}'_e(\texttt{x}) = \emptyset$ indicates that `x` will never be bound to any closures. Also the guess $(\widehat{\mathsf{C}}''_e, \widehat{\rho}''_e)$ of the third column would seem to be acceptable although clearly more imprecise than $(\widehat{\mathsf{C}}_e, \widehat{\rho}_e)$. ∎

Example 3.4 Let us consider the expression, loop, of Example 3.2 and introduce the following abbreviations for abstract values:

$$\begin{array}{rcl} \mathsf{f} & = & \texttt{fun f x => (f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4 \\ \mathsf{id}_y & = & \texttt{fn y => y}^2 \\ \mathsf{id}_z & = & \texttt{fn z => z}^7 \end{array}$$

One guess of a 0-CFA analysis for this program is $(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}})$ defined by:

$$\begin{array}{rclrclrcl} \widehat{\mathsf{C}}_{\mathsf{lp}}(1) & = & \{\mathsf{f}\} & \widehat{\mathsf{C}}_{\mathsf{lp}}(6) & = & \{\mathsf{f}\} & \widehat{\rho}_{\mathsf{lp}}(\mathtt{f}) & = & \{\mathsf{f}\} \\ \widehat{\mathsf{C}}_{\mathsf{lp}}(2) & = & \emptyset & \widehat{\mathsf{C}}_{\mathsf{lp}}(7) & = & \emptyset & \widehat{\rho}_{\mathsf{lp}}(\mathtt{g}) & = & \{\mathsf{f}\} \\ \widehat{\mathsf{C}}_{\mathsf{lp}}(3) & = & \{\mathsf{id}_y\} & \widehat{\mathsf{C}}_{\mathsf{lp}}(8) & = & \{\mathsf{id}_z\} & \widehat{\rho}_{\mathsf{lp}}(\mathtt{x}) & = & \{\mathsf{id}_y, \mathsf{id}_z\} \\ \widehat{\mathsf{C}}_{\mathsf{lp}}(4) & = & \emptyset & \widehat{\mathsf{C}}_{\mathsf{lp}}(9) & = & \emptyset & \widehat{\rho}_{\mathsf{lp}}(\mathtt{y}) & = & \emptyset \\ \widehat{\mathsf{C}}_{\mathsf{lp}}(5) & = & \{\mathsf{f}\} & \widehat{\mathsf{C}}_{\mathsf{lp}}(10) & = & \emptyset & \widehat{\rho}_{\mathsf{lp}}(\mathtt{z}) & = & \emptyset \end{array}$$

Intuitively, this is an acceptable guess. The choice of $\widehat{\rho}_{\mathsf{lp}}(\mathtt{g}) = \{\mathsf{f}\}$ reflects that g will evaluate to a closure constructed from that abstraction. The choice of $\widehat{\rho}_{\mathsf{lp}}(\mathtt{x}) = \{\mathsf{id}_y, \mathsf{id}_z\}$ reflects that x will be bound to closures constructed from both abstractions in the course of the evaluation. The choice of $\widehat{\mathsf{C}}_{\mathsf{lp}}(10) = \emptyset$ reflects that the evaluation of the expression will never terminate. ∎

We have already said that Control Flow Analysis computes the interprocedural flow information used in Section 2.5. It is also instructive to point out the similarity between Control Flow Analysis and *Use-Definition chains* (*ud*-chains) for imperative languages (see Subsection 2.1.5): in both cases we attempt to trace how definition points reach points of use. In the case of Control Flow Analysis the *definition points* are the points where the function abstractions are created, and the *use points* are the points where functions are applied; in the case of Use-Definition chains the *definition points* are the points where variables are assigned a value, and the *use points* are the points where values of variables are accessed.

Remark. Clearly an abstract cache $\widehat{\mathsf{C}} : \mathbf{Lab} \to \widehat{\mathbf{Val}}$ and an abstract environment $\widehat{\rho} : \mathbf{Var} \to \widehat{\mathbf{Val}}$ can be combined into an entity of type $(\mathbf{Var} \cup \mathbf{Lab}) \to \widehat{\mathbf{Val}}$. Some texts dispense with the labels altogether, simply using an abstract environment and no abstract cache, by ensuring that *all* subterms are properly "labelled" by variables. This type of expression frequently occurs in the internals of compilers in the form of "continuation passing style", "A-normal form" or "three address code". We have abstained from doing so in order to illustrate that the techniques not only work for compiler intermediate forms but also for general programming languages and calculi for computation; this flexibility is useful when dealing with non-standard applications (as discussed in the Concluding Remarks). ∎

$$
\begin{array}{ll}
[con] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models c^{\ell} \text{ always} \\
[var] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models x^{\ell} \text{ iff } \widehat{\rho}(x) \subseteq \widehat{\mathsf{C}}(\ell) \\
[fn] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models (\texttt{fn}\ x \texttt{ => } e_0)^{\ell} \text{ iff } \{\texttt{fn}\ x \texttt{ => } e_0\} \subseteq \widehat{\mathsf{C}}(\ell) \\
[fun] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models (\texttt{fun}\ f\ x \texttt{ => } e_0)^{\ell} \text{ iff } \{\texttt{fun}\ f\ x \texttt{ => } e_0\} \subseteq \widehat{\mathsf{C}}(\ell) \\
[app] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models (t_1^{\ell_1}\ t_2^{\ell_2})^{\ell} \\
& \quad \text{iff } (\widehat{\mathsf{C}}, \widehat{\rho}) \models t_1^{\ell_1} \wedge (\widehat{\mathsf{C}}, \widehat{\rho}) \models t_2^{\ell_2} \wedge \\
& \qquad (\forall (\texttt{fn}\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\mathsf{C}}(\ell_1) : \\
& \qquad\qquad (\widehat{\mathsf{C}}, \widehat{\rho}) \models t_0^{\ell_0} \wedge \\
& \qquad\qquad \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\rho}(x) \wedge \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell)) \wedge \\
& \qquad (\forall (\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\mathsf{C}}(\ell_1) : \\
& \qquad\qquad (\widehat{\mathsf{C}}, \widehat{\rho}) \models t_0^{\ell_0} \wedge \\
& \qquad\qquad \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\rho}(x) \wedge \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell) \wedge \\
& \qquad\qquad \{\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}\} \subseteq \widehat{\rho}(f)) \\
[if] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models (\texttt{if}\ t_0^{\ell_0}\ \texttt{then}\ t_1^{\ell_1}\ \texttt{else}\ t_2^{\ell_2})^{\ell} \\
& \quad \text{iff } (\widehat{\mathsf{C}}, \widehat{\rho}) \models t_0^{\ell_0} \wedge \\
& \qquad (\widehat{\mathsf{C}}, \widehat{\rho}) \models t_1^{\ell_1} \wedge (\widehat{\mathsf{C}}, \widehat{\rho}) \models t_2^{\ell_2} \wedge \\
& \qquad \widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\mathsf{C}}(\ell) \wedge \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\mathsf{C}}(\ell) \\
[let] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models (\texttt{let}\ x = t_1^{\ell_1}\ \texttt{in}\ t_2^{\ell_2})^{\ell} \\
& \quad \text{iff } (\widehat{\mathsf{C}}, \widehat{\rho}) \models t_1^{\ell_1} \wedge (\widehat{\mathsf{C}}, \widehat{\rho}) \models t_2^{\ell_2} \wedge \\
& \qquad \widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\rho}(x) \wedge \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\mathsf{C}}(\ell) \\
[op] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models (t_1^{\ell_1}\ op\ t_2^{\ell_2})^{\ell} \text{ iff } (\widehat{\mathsf{C}}, \widehat{\rho}) \models t_1^{\ell_1} \wedge (\widehat{\mathsf{C}}, \widehat{\rho}) \models t_2^{\ell_2}
\end{array}
$$

Table 3.1: Abstract Control Flow Analysis (Subsections 3.1.1 and 3.1.2).

Acceptability relation. It remains to determine whether or not a proposed guess $(\widehat{\mathsf{C}}, \widehat{\rho})$ of an analysis results is in fact an *acceptable 0-CFA* analysis for the program considered. We shall give an *abstract specification* of what this means; having studied its theoretical properties (in Section 3.2) we then consider how to compute the desired analysis (in Sections 3.3 and 3.4).

It is instructive to point out that the abstract specification corresponds to an *implicit* formulation of the data flow equations of Chapter 2; it will be used to determine whether or not a a guess is indeed an acceptable solution to the analysis problem. The syntax directed and constraint based formulations (of Sections 3.3 and 3.4) correspond to *explicit* formulations of the data flow equations from which an iterative algorithm in the spirit of Chaotic Iteration (Section 1.7) can be used to compute an analysis result.

For the formulation of the *abstract 0-CFA* analysis we shall write

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models e$$

for when $(\widehat{\mathsf{C}}, \widehat{\rho})$ is an acceptable Control Flow Analysis of the expression e. Thus the relation "$\models$" has functionality

$$\models : (\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Env}} \times \mathbf{Exp}) \to \{\mathit{true}, \mathit{false}\}$$

and its defining clauses are given in Table 3.1 (writing "always" for "iff *true*"); the clauses are explained below but the relation $\models$ will not be formally defined until Subsection 3.1.2.

The clause [*con*] places no demands on $\widehat{\mathsf{C}}(\ell)$ because we are not tracking any data values in the pure 0-CFA analysis considered here and because we assume that there are no functions among the constants; the clause can be reformulated as

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models c^{\ell} \text{ iff } \emptyset \subseteq \widehat{\mathsf{C}}(\ell)$$

thereby highlighting this point.

The clause [*var*] is responsible for linking the abstract environment into the abstract cache: so in order for $(\widehat{\mathsf{C}}, \widehat{\rho})$ to be an acceptable analysis, everything the variable x can evaluate to has to be included in what may be observed at the program point ℓ: $\widehat{\rho}(x) \subseteq \widehat{\mathsf{C}}(\ell)$.

The clauses [*fn*] and [*fun*] simply demand that in order for $(\widehat{\mathsf{C}}, \widehat{\rho})$ to be an acceptable analysis, the functional term (`fn` x `=>` e_0 or `fun` f x `=>` e_0) must be included in $\widehat{\mathsf{C}}(\ell)$; this says that the term is part of a closure that can arise at program point ℓ during evaluation. Note that these clauses do not demand that $(\widehat{\mathsf{C}}, \widehat{\rho})$ is an acceptable analysis result for the bodies of the functions; the clause for function application will take care of that.

Before turning to the more complicated clause [*app*] let us consider the clauses [*if*] and [*let*]. They contain "recursive calls" demanding that subexpressions must be analysed in consistent ways using $(\widehat{\mathsf{C}}, \widehat{\rho})$; additionally, the clauses explicitly link the values produced by subexpressions to the value of the overall expression, and in the case of [*let*] also the abstract cache is linked into the abstract environment. The interplay between the clauses [*var*] and [*let*] is illustrated in Figure 3.1; as in Chapter 2 an arrow indicates a flow of information. The clause [*op*] follows the same overall pattern.

Clause [*app*] also contains "recursive calls" demanding that the operator $t_1^{\ell_1}$ and the operand $t_2^{\ell_2}$ can be analysed using $(\widehat{\mathsf{C}}, \widehat{\rho})$. For each term `fn` x `=>` $t_0^{\ell_0}$ that may reach the operator position (ℓ_1), i.e. where

$$(\texttt{fn}\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\mathsf{C}}(\ell_1)$$

it further demands that the actual parameter (labelled ℓ_2) is linked to the formal parameter (x)

$$\widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\rho}(x)$$

Figure 3.1: Pictorial illustration of the clauses [*let*] and [*var*].

and that the result of the function evaluation (labelled ℓ_0) is linked to the result of the application itself (labelled ℓ)

$$\widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell)$$

and finally, that the function body itself can be analysed using $(\widehat{\mathsf{C}}, \widehat{\rho})$:

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models t_0^{\ell_0}$$

This is illustrated in Figure 3.2. For terms $\texttt{fun}\ f\ x\ \texttt{=>}\ t_0^{\ell_0}$ the demands are much the same except that the term itself additionally needs to be included in $\widehat{\rho}(f)$ in order to reflect the recursive nature of $\texttt{fun}\ f\ x\ \texttt{=>}\ t_0^{\ell_0}$.

Example 3.5 Consider Example 3.3 and the guesses of a 0-CFA analysis for the expression. First we show that $(\widehat{\mathsf{C}}_{\mathsf{e}}, \widehat{\rho}_{\mathsf{e}})$ is an acceptable guess:

$$(\widehat{\mathsf{C}}_{\mathsf{e}}, \widehat{\rho}_{\mathsf{e}}) \models ((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5$$

Using clause [*app*] and $\widehat{\mathsf{C}}_{\mathsf{e}}(2) = \{\texttt{fn x => x}^1\}$ we must check:

$$\begin{aligned}
&(\widehat{\mathsf{C}}_{\mathsf{e}}, \widehat{\rho}_{\mathsf{e}}) \models (\texttt{fn x => x}^1)^2 \\
&(\widehat{\mathsf{C}}_{\mathsf{e}}, \widehat{\rho}_{\mathsf{e}}) \models (\texttt{fn y => y}^3)^4 \\
&(\widehat{\mathsf{C}}_{\mathsf{e}}, \widehat{\rho}_{\mathsf{e}}) \models \texttt{x}^1 \\
&\widehat{\mathsf{C}}_{\mathsf{e}}(4) \subseteq \widehat{\rho}_{\mathsf{e}}(\texttt{x}) \\
&\widehat{\mathsf{C}}_{\mathsf{e}}(1) \subseteq \widehat{\mathsf{C}}_{\mathsf{e}}(5)
\end{aligned}$$

Figure 3.2: Pictorial illustration of the clauses [*app*], [*fn*] and [*var*].

All of these are easily checked using the clauses [*fn*] and [*var*].

Next we show that $(\widehat{\mathsf{C}}'_{\mathsf{e}}, \widehat{\rho}'_{\mathsf{e}})$ is *not* an acceptable guess:

$$(\widehat{\mathsf{C}}'_{\mathsf{e}}, \widehat{\rho}'_{\mathsf{e}}) \not\models ((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5$$

We do so by proceeding as above and observing that $\widehat{\mathsf{C}}'_{\mathsf{e}}(4) \not\subseteq \widehat{\rho}'_{\mathsf{e}}(\mathbf{x})$. ∎

Note that the clauses contain a number of inclusions of the form

$$\mathit{lhs} \subseteq \mathit{rhs}$$

where *rhs* is of the form $\widehat{\mathsf{C}}(\ell)$ or $\widehat{\rho}(x)$ and where *lhs* is of the form $\widehat{\mathsf{C}}(\ell)$, $\widehat{\rho}(x)$, or $\{t\}$. These inclusions express how the higher-order entities may flow through the expression.

It is important to observe that the clauses [*fn*] and [*fun*] do *not* contain "recursive calls" demanding that subexpressions must be analysed. Instead one relies on the clause [*app*] demanding this for all "subexpressions" that may eventually be applied. This is a phenomenon common in program analysis, where one does *not* want to analyse *unreachable* program fragments: occasionally results obtained from these parts of the program can suppress transformations in the reachable part of the program. It also allows us to deal with *open systems* where functions may be supplied by the environment; this is particularly important for languages involving concurrency. However,

note that this perspective is different from that of type inference, where even unreachable fragments must be correctly typed.

In the terminology of Section 2.5 the analysis is *flow-insensitive* because FUN contains no side effects and because we analyse the operand to a function call even when the operator cannot evaluate to any function; see Exercise 3.3 for how to improve on this. Also the analysis is *context-insensitive* because it treats all function calls in the same way; we refer to Section 3.6 for how to improve on this.

3.1.2 Well-definedness of the Analysis

Finally, we need to clarify that the clauses of Table 3.1 do indeed define a relation. The difficulty here is that the clause [*app*] is *not* in a form that allows us to define $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e$ by structural induction in the expression e – it requires checking the acceptability of $(\widehat{\mathsf{C}}, \widehat{\rho})$ for an expression $t_0^{\ell_0}$ that is not a subexpression of the application $(t_1^{\ell_1}\ t_2^{\ell_2})^{\ell}$. This leads to defining the relation "$\models$" of Table 3.1 by *coinduction*, that is as the *greatest fixed point* of a certain functional. An alternative will be to define the analysis as the least fixed point of the functional but, as we shall see in Example 3.6 and more formally in Proposition 3.16, this is not always appropriate.

The formal definition of $\models$. Following the approach of Appendix B we shall view Table 3.1 as defining a function:

$$\begin{aligned} \mathcal{Q} : ((\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Env}} \times \mathbf{Exp}) &\to \{\mathit{true}, \mathit{false}\}) \\ &\to ((\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Env}} \times \mathbf{Exp}) \to \{\mathit{true}, \mathit{false}\}) \end{aligned}$$

As an example we have:

$$\begin{aligned} &\mathcal{Q}(Q)(\widehat{\mathsf{C}}, \widehat{\rho}, (\texttt{let}\ x = t_1^{\ell_1}\ \texttt{in}\ t_2^{\ell_2})^{\ell}) \\ &\quad = Q(\widehat{\mathsf{C}}, \widehat{\rho}, t_1^{\ell_1}) \ \wedge\ Q(\widehat{\mathsf{C}}, \widehat{\rho}, t_2^{\ell_2}) \ \wedge\ \widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\rho}(x) \ \wedge\ \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\mathsf{C}}(\ell) \end{aligned}$$

We frequently refer to $\mathcal{Q}$ as a *functional* because its argument and result are themselves functions.

Now by inspecting Table 3.1 it is easy to verify that the functional $\mathcal{Q}$ constructed this way is a monotone function on the complete lattice

$$((\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Env}} \times \mathbf{Exp}) \to \{\mathit{true}, \mathit{false}\}, \sqsubseteq)$$

where the ordering $\sqsubseteq$ is defined by:

$$Q_1 \sqsubseteq Q_2 \ \text{ iff } \ \forall(\widehat{\mathsf{C}}, \widehat{\rho}, e) : (Q_1(\widehat{\mathsf{C}}, \widehat{\rho}, e) = \mathit{true}) \ \Rightarrow\ (Q_2(\widehat{\mathsf{C}}, \widehat{\rho}, e) = \mathit{true})$$

Hence $\mathcal{Q}$ has fixed points and we shall define "$\models$" coinductively:

$$\models \text{ is the } \textit{greatest} \text{ fixed point of } \mathcal{Q}$$

The following example intuitively motivates the use of a coinductive (i.e. a greatest fixed point) definition as opposed to an inductive (i.e. a least fixed point) definition; a more formal explanation will be given in Subsection 3.2.4.

Example 3.6 Consider the expression loop of Example 3.4

$$\begin{array}{l}(\texttt{let g = (fun f x => (f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4\texttt{)}^5\\ \ \texttt{in (g}^6\ \texttt{(fn z => z}^7\texttt{)}^8\texttt{)}^9\texttt{)}^{10}\end{array}$$

and the suggested analysis result $(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}})$. To show $(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models \mathsf{loop}$ it is, according to the clause [*let*], sufficient to verify that

$$\begin{array}{l}(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models (\texttt{fun f x => (f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4\texttt{)}^5\\ (\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models (\texttt{g}^6\ \texttt{(fn z => z}^7\texttt{)}^8\texttt{)}^9\end{array}$$

because $\widehat{\mathsf{C}}_{\mathsf{lp}}(5) \subseteq \widehat{\rho}_{\mathsf{lp}}(\mathsf{g})$ and $\widehat{\mathsf{C}}_{\mathsf{lp}}(9) \subseteq \widehat{\mathsf{C}}_{\mathsf{lp}}(10)$. The first clause follows from [*fun*] and for the second clause we use that $\widehat{\mathsf{C}}_{\mathsf{lp}}(6) = \{\mathsf{f}\}$ so it is, according to [*app*], sufficient to verify that

$$\begin{array}{l}(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models \texttt{g}^6\\ (\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models (\texttt{fn z => z}^7\texttt{)}^8\\ (\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models (\texttt{f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4\end{array}$$

because $\widehat{\mathsf{C}}_{\mathsf{lp}}(8) \subseteq \widehat{\rho}_{\mathsf{lp}}(\mathsf{x})$, $\widehat{\mathsf{C}}_{\mathsf{lp}}(4) \subseteq \widehat{\mathsf{C}}_{\mathsf{lp}}(9)$ and $\mathsf{f} \in \widehat{\rho}_{\mathsf{lp}}(\mathsf{f})$. The first two clauses now follow from [*var*] and [*fn*]. For the third clause we proceed as above and since $\widehat{\mathsf{C}}_{\mathsf{lp}}(1) = \{\mathsf{f}\}$ it is sufficient to verify

$$\begin{array}{l}(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models \texttt{f}^1\\ (\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models (\texttt{fn y => y}^2\texttt{)}^3\\ (\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models (\texttt{f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4\end{array}$$

because $\widehat{\mathsf{C}}_{\mathsf{lp}}(3) \subseteq \widehat{\rho}_{\mathsf{lp}}(\mathsf{x})$, $\widehat{\mathsf{C}}_{\mathsf{lp}}(4) \subseteq \widehat{\mathsf{C}}_{\mathsf{lp}}(4)$ and $\mathsf{f} \in \widehat{\rho}_{\mathsf{lp}}(\mathsf{f})$.

Again the first two clauses are straightforward but in the last clause we encounter a *circularity*: to verify $(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models (\texttt{f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4$ we have to verify $(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models (\texttt{f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4$!

To solve this we use coinduction: basically this amounts to assuming that $(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models (\texttt{f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4$ holds at the "inner level" and proving that it also holds at the "outer level". This will give us the required proof. ∎

Analogy. The use of coinduction and induction, or greatest and least fixed points, may be confusing at first. We therefore offer the following analogy. Imagine a company that is about to buy a large number of computers. To comply with national and international rules for commerce the deal must be offered in such a way that all vendors have a chance to make a bid. To this effect the company first makes a specification of the requirements to the computers (like the ability to execute certain benchmarks efficiently); however, the specification is necessarily loose meaning that there might be many ways of fulfilling the requirements. Among the bids made the company then chooses the bid believed to be best (in the sense of offering the required functionality as cheaply as possible or offering as much functionality as possible). Then suppose that when the actual computers are delivered the company suspects that they are not fulfilling the promises. It then starts arbitration in order to persuade the vendor that they have not fulfilled their obligations (and must take back the computers, or upgrade them or sell them at reduced price). In order for the company to be successful it must be able to use the details of the specification, and not any other information about what other vendors would have been able to provide, in order to prove that the computers delivered fail to fulfil one or more demands explicitly put forward in the specification. If the specification is too loose, like just demanding an Intel-based PC, there is no way that the inability to run Linux (or some other operating system) can be used against the vendor. — Coming back to the analysis in Table 3.1 its primary purpose is to provide a specification of when a potential analysis result is acceptable. The specification takes the form of defining a relation

$$\models : (\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Env}} \times \mathbf{Exp}) \rightarrow \{\mathit{true}, \mathit{false}\}$$

and since the specification is intended to be loose it must be defined coinductively: something only fails to live up to the specification if it can be demonstrated to follow from the clauses; this line of reasoning is in fact a fundamental ingredient in algebraic specification theory. As an example, suppose we were silly and merely defined

$$(C, \rho) \models P \quad \text{iff} \quad (C, \rho) \models P$$

then any solution (C, ρ) would in fact live up to the specification; it is a simple mathematical fact that only the coinductive interpretation of the definition of $\models$ will live up to this demand (and in particular the inductive definition will not as it will not accept any solutions at all). Having clarified the meaning of the specification we can then look for the best solution (C, ρ): this is typically the least solution that satisfies the specification. So to summarise, the actual specification itself is defined by a greatest fixed point (namely coinductively) whereas algorithms for computing the intended solution are normally defined by least fixed points.

3.2 Theoretical Properties

In this section we shall investigate some more theoretical properties of the Control Flow Analysis, namely:

- semantic correctness, and
- the existence of least solutions.

The semantic correctness result is important since it ensures that the information from the analysis is indeed a safe description of what will happen during the evaluation of the program. The result about the existence of least solutions ensures that all programs can be analysed and furthermore that there is a "best" or "most precise" analysis result.

As in Section 2.2, the material of this section may be skimmed through on a first reading; however, we reiterate that it is frequently when conducting the correctness proof that the final and subtle errors in the analysis are found and corrected!

3.2.1 Structural Operational Semantics

Configurations. We shall equip the language FUN with a *Structural Operational Semantics.* We shall choose an approach based on *explicit* environments rather than substitutions because (as discussed in the Concluding Remarks) a substitution based semantics does not preserve the identity of functions (and hence abstract values) during evaluation. So a function definition will evaluate to a *closure* containing the syntax of the function definition together with an *environment* mapping its free variables to their *values.* For this we introduce the following categories

$$\begin{array}{lcll} v & \in & \mathbf{Val} & \text{values} \\ \rho & \in & \mathbf{Env} & \text{environments} \end{array}$$

defined by:

$$\begin{array}{lcl} v & ::= & c \mid \texttt{close}\ t\ \texttt{in}\ \rho \\ \rho & ::= & [\,] \mid \rho[x \mapsto v] \end{array}$$

A function abstraction `fn` x `=>` e_0 will then evaluate to a closure, written `close` (`fn` x `=>` e_0) `in` ρ; similarly, the abstraction `fun` f x `=>` e_0 will evaluate to `close` (`fun` f x `=>` e_0) `in` ρ. Our definitions do not demand that all terms t occurring in some `close` t `in` ρ in the semantics will be of the form `fn` x `=>` e_0 or `fun` f x `=>` e_0; however, it will be the case for the semantics presented below.

As in Section 2.5 we shall need intermediate configurations to handle the binding of local variables. We therefore introduce syntactic categories for *intermediate expressions* and *intermediate terms*

$$\begin{array}{lll} ie & \in & \mathbf{IExp} \quad \text{intermediate expressions} \\ it & \in & \mathbf{ITerm} \quad \text{intermediate terms} \end{array}$$

that extend the syntax of expressions and terms as follows:

$$\begin{array}{lll} ie & ::= & it^{\ell} \\ \\ it & ::= & c \mid x \mid \texttt{fn}\ x\ \texttt{=>}\ e_0 \mid \texttt{fun}\ f\ x\ \texttt{=>}\ e_0 \mid ie_1\ ie_2 \\ & \mid & \texttt{if}\ ie_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x\ \texttt{=}\ ie_1\ \texttt{in}\ e_2 \mid ie_1\ op\ ie_2 \\ & \mid & \texttt{bind}\ \rho\ \texttt{in}\ ie \mid \texttt{close}\ t\ \texttt{in}\ \rho \end{array}$$

The role of the `bind`-construct is much as in Section 2.5: `bind` ρ `in` ie records that the intermediate expression ie has to be evaluated in an environment with the bindings ρ. (The sequence of environments of nested `bind`-constructs may be viewed as an encoding of the frames of a run-time stack.) So while `close` t `in` ρ is a fully evaluated value this is not the case for `bind` ρ `in` ie. We shall need these intermediate terms because we define a small step semantics; only the `close`-constructs will be needed for a big step variant of the semantics.

Alternatively, the definitions of **Val** and **Env** could have been written $\mathbf{Val} = \mathbf{Const} + (\mathbf{Term} \times \mathbf{Env})$ and $\mathbf{Env} = \mathbf{Var} \rightarrow_{\text{fin}} \mathbf{Val}$ (for a finite mapping) but it is important to stress that all entities are defined mutually recursively in the manner of context-free grammars. Formally, we defined an environment ρ as a list but nevertheless we shall feel free to regard it as a finite mapping: we write $dom(\rho)$ for $\{x \mid \rho \text{ contains } [x \mapsto \cdots]\}$; we write $\rho(x) = v$ if $x \in dom(\rho)$ and the rightmost occurrence of $[x \mapsto \cdots]$ in ρ is $[x \mapsto v]$, and we write $\rho \mid X$ for the environment obtained from ρ by removing all occurrences of $[x \mapsto \cdots]$ with $x \notin X$. For the sake of readability we shall write $[x \mapsto v]$ for $[\,][x \mapsto v]$.

We have been very deliberate in when to use intermediate expressions and when to use expressions although it is evident that all expressions are also intermediate expressions. Since we do not evaluate the body of a function before it is applied we continue to let the body be an expression rather than an intermediate expression. Similar remarks apply to the branches of the conditional and the body of the local definitions. Note that although an environment only records the *terms* `fn` x `=>` e_0 and `fun` f x `=>` e_0 in the closures bound into it, we do not lose the identity of the function abstractions as e_0 will be of the form $t_0^{\ell_0}$ and hence ℓ_0 may be used as the *"unique" identification* of the function abstraction.

Transitions. We are now ready to define the transition rules of the Structural Operational Semantics by means of judgements of the form

$$\rho \vdash ie_1 \rightarrow ie_2$$

[*var*]	$\rho \vdash x^\ell \rightarrow v^\ell$ if $x \in dom(\rho)$ and $v = \rho(x)$
[*fn*]	$\rho \vdash (\texttt{fn}\ x\ \texttt{=>}\ e_0)^\ell \rightarrow (\texttt{close}\ (\texttt{fn}\ x\ \texttt{=>}\ e_0)\ \texttt{in}\ \rho_0)^\ell$ where $\rho_0 = \rho \mid FV(\texttt{fn}\ x\ \texttt{=>}\ e_0)$
[*fun*]	$\rho \vdash (\texttt{fun}\ f\ x\ \texttt{=>}\ e_0)^\ell \rightarrow (\texttt{close}\ (\texttt{fun}\ f\ x\ \texttt{=>}\ e_0)\ \texttt{in}\ \rho_0)^\ell$ where $\rho_0 = \rho \mid FV(\texttt{fun}\ f\ x\ \texttt{=>}\ e_0)$
[app_1]	$\dfrac{\rho \vdash ie_1 \rightarrow ie_1'}{\rho \vdash (ie_1\ ie_2)^\ell \rightarrow (ie_1'\ ie_2)^\ell}$
[app_2]	$\dfrac{\rho \vdash ie_2 \rightarrow ie_2'}{\rho \vdash (v_1^{\ell_1}\ ie_2)^\ell \rightarrow (v_1^{\ell_1}\ ie_2')^\ell}$
[app_{fn}]	$\rho \vdash ((\texttt{close}\ (\texttt{fn}\ x\ \texttt{=>}\ e_1)\ \texttt{in}\ \rho_1)^{\ell_1}\ v_2^{\ell_2})^\ell \rightarrow (\texttt{bind}\ \rho_1[x \mapsto v_2]\ \texttt{in}\ e_1)^\ell$
[app_{fun}]	$\rho \vdash ((\texttt{close}\ (\texttt{fun}\ f\ x\ \texttt{=>}\ e_1)\ \texttt{in}\ \rho_1)^{\ell_1}\ v_2^{\ell_2})^\ell \rightarrow (\texttt{bind}\ \rho_2[x \mapsto v_2]\ \texttt{in}\ e_1)^\ell$ where $\rho_2 = \rho_1[f \mapsto \texttt{close}\ (\texttt{fun}\ f\ x\ \texttt{=>}\ e_1)\ \texttt{in}\ \rho_1]$
[$bind_1$]	$\dfrac{\rho_1 \vdash ie_1 \rightarrow ie_1'}{\rho \vdash (\texttt{bind}\ \rho_1\ \texttt{in}\ ie_1)^\ell \rightarrow (\texttt{bind}\ \rho_1\ \texttt{in}\ ie_1')^\ell}$
[$bind_2$]	$\rho \vdash (\texttt{bind}\ \rho_1\ \texttt{in}\ v_1^{\ell_1})^\ell \rightarrow v_1^\ell$

Table 3.2: The Structural Operational Semantics of FUN (part 1).

given by the axioms and inference rules of Tables 3.2 and 3.3; they are explained below. The idea is that *one step* of computation of the expression ie_1 in the environment ρ will transform it into ie_2.

The value of a variable is obtained from the environment as expressed by the axiom [*var*]. The axioms [*fn*] and [*fun*] construct the appropriate closures; they restrict the environment ρ to the free variables of the abstraction. Note that in [*fun*] it is only recorded that we have a recursively defined function; the unfolding of the recursion will not happen until it is called.

The clauses for application shows that the semantics is a *call-by-value* semantics: In an application we first evaluate the operator in a number of steps using the rule [app_1] and then we evaluate the operand in a number of steps using the rule [app_2]. The next stage is to use one of the rules [app_{fn}] or [app_{fun}] to bind the actual parameter to the formal parameter and, in the case of [app_{fun}], to unfold the recursive function so that subsequent recursive calls will be bound correctly. We shall use a bind-construct to contain the body of the function together with the appropriate environment. Finally, we

$$[if_1] \quad \frac{\rho \vdash ie_0 \rightarrow ie_0'}{\rho \vdash (\texttt{if}\ ie_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2)^\ell \rightarrow (\texttt{if}\ ie_0'\ \texttt{then}\ e_1\ \texttt{else}\ e_2)^\ell}$$

$$[if_2] \quad \rho \vdash (\texttt{if}\ \texttt{true}^{\ell_0}\ \texttt{then}\ t_1^{\ell_1}\ \texttt{else}\ t_2^{\ell_2})^\ell \rightarrow t_1^\ell$$

$$[if_3] \quad \rho \vdash (\texttt{if}\ \texttt{false}^{\ell_0}\ \texttt{then}\ t_1^{\ell_1}\ \texttt{else}\ t_2^{\ell_2})^\ell \rightarrow t_2^\ell$$

$$[let_1] \quad \frac{\rho \vdash ie_1 \rightarrow ie_1'}{\rho \vdash (\texttt{let}\ x = ie_1\ \texttt{in}\ e_2)^\ell \rightarrow (\texttt{let}\ x = ie_1'\ \texttt{in}\ e_2)^\ell}$$

$$[let_2] \quad \rho \vdash (\texttt{let}\ x = v^{\ell_1}\ \texttt{in}\ e_2)^\ell \rightarrow (\texttt{bind}\ \rho_0[x \mapsto v]\ \texttt{in}\ e_2)^\ell \quad \text{where } \rho_0 = \rho \mid FV(e_2)$$

$$[op_1] \quad \frac{\rho \vdash ie_1 \rightarrow ie_1'}{\rho \vdash (ie_1\ op\ ie_2)^\ell \rightarrow (ie_1'\ op\ ie_2)^\ell}$$

$$[op_2] \quad \frac{\rho \vdash ie_2 \rightarrow ie_2'}{\rho \vdash (v_1^{\ell_1}\ op\ ie_2)^\ell \rightarrow (v_1^{\ell_1}\ op\ ie_2')^\ell}$$

$$[op_3] \quad \rho \vdash (v_1^{\ell_1}\ op\ v_2^{\ell_2})^\ell \rightarrow v^\ell \qquad \text{if } v = v_1\ \mathbf{op}\ v_2$$

Table 3.3: The Structural Operational Semantics of FUN (part 2).

evaluate the bind-construct using rule $[bind_1]$ a number of times, and we get the result of the application by using rule $[bind_2]$. The interplay between these rules is illustrated by the following example.

Example 3.7 Consider the expression $((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5$ of Example 3.1. It has the following derivation sequence (explained below):

$$\begin{array}{lll}
[\,] & \vdash & ((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5 \\
\rightarrow & & ((\texttt{close (fn x => x}^1)\ \texttt{in}\ [\,])^2\ (\texttt{fn y => y}^3)^4)^5 \\
\rightarrow & & ((\texttt{close (fn x => x}^1)\ \texttt{in}\ [\,])^2\ (\texttt{close (fn y => y}^3)\ \texttt{in}\ [\,])^4)^5 \\
\rightarrow & & (\texttt{bind}\ [\texttt{x} \mapsto (\texttt{close (fn y => y}^3)\ \texttt{in}\ [\,])]\ \texttt{in x}^1)^5 \\
\rightarrow & & (\texttt{bind}\ [\texttt{x} \mapsto (\texttt{close (fn y => y}^3)\ \texttt{in}\ [\,])]\ \texttt{in} \\
 & & \quad (\texttt{close (fn y => y}^3)\ \texttt{in}\ [\,])^1)^5 \\
\rightarrow & & (\texttt{close (fn y => y}^3)\ \texttt{in}\ [\,])^5
\end{array}$$

First $[app_1]$ and $[fn]$ are used to evaluate the operator, then $[app_2]$ and $[fn]$ are used to evaluate the operand and $[app_{fn}]$ introduces the bind-construct containing the local environment $[\texttt{x} \mapsto (\texttt{close (fn y => y}^3)\ \texttt{in}\ [\,])]$ needed to evaluate its body. So $\texttt{x}^1$ is evaluated using $[bind_1]$ and $[var]$, and finally $[bind_2]$ is used to get rid of the local environment. ■

The semantics of the conditional is the usual one: first the condition is evaluated in a number of steps using rule $[if_1]$ and then the appropriate branch is selected by rules $[if_2]$ and $[if_3]$. For the local definitions we first compute the value of the bound variable in a number of steps using rule $[let_1]$ and then we introduce a `bind`-construct using rule $[let_2]$ reflecting that the body of the `let`-construct has to be evaluated in an extended environment. The rules $[bind_1]$ and $[bind_2]$ are now used to compute the result. For binary expressions we first evaluate the arguments using $[op_1]$ and $[op_2]$ and then the operation itself, denoted **op**, is performed using $[op_3]$.

As in Chapter 2 the labels have no impact on the semantics but are merely carried along. It is important to note that the outermost label never changes while inner labels may disappear; see for example the rules $[if_2]$ and $[bind_2]$. This is an important property of the semantics that is exploited by the 0-CFA analysis.

Example 3.8 Let us consider the expression, loop

$$\begin{array}{l}(\texttt{let g = (fun f x => (f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4\texttt{)}^5\\ \ \texttt{in (g}^6\ \texttt{(fn z => z}^7\texttt{)}^8\texttt{)}^9\texttt{)}^{10}\end{array}$$

of Example 3.2 and see how the informal explanation of its semantics is captured in the formal semantics. First we introduce abbreviations for three closures:

$$\begin{array}{rcl}\mathbf{f} & = & \texttt{close (fun f x => (f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4\texttt{) in [ ]}\\ \mathbf{id}_y & = & \texttt{close (fn y => y}^2\texttt{) in [ ]}\\ \mathbf{id}_z & = & \texttt{close (fn z => z}^7\texttt{) in [ ]}\end{array}$$

Then we have the following derivation sequence

$$\begin{array}{lll}[\,] & \vdash & \mathsf{loop}\\ \rightarrow & & (\texttt{let g} = \mathbf{f}^5\ \texttt{in (g}^6\ \texttt{(fn z => z}^7\texttt{)}^8\texttt{)}^9\texttt{)}^{10}\\ \rightarrow & & (\texttt{bind [g} \mapsto \mathbf{f}\texttt{] in (g}^6\ \texttt{(fn z => z}^7\texttt{)}^8\texttt{)}^9\texttt{)}^{10}\\ \rightarrow & & (\texttt{bind [g} \mapsto \mathbf{f}\texttt{] in (f}^6\ \texttt{(fn z => z}^7\texttt{)}^8\texttt{)}^9\texttt{)}^{10}\\ \rightarrow & & (\texttt{bind [g} \mapsto \mathbf{f}\texttt{] in (f}^6\ \mathbf{id}_z^8\texttt{)}^9\texttt{)}^{10}\\ \rightarrow & & (\texttt{bind [g} \mapsto \mathbf{f}\texttt{] in}\\ & & \quad (\texttt{bind [f} \mapsto \mathbf{f}\texttt{][x} \mapsto \mathbf{id}_z\texttt{] in (f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4\texttt{)}^9\texttt{)}^{10}\\ \rightarrow^* & & (\texttt{bind [g} \mapsto \mathbf{f}\texttt{] in}\\ & & \quad (\texttt{bind [f} \mapsto \mathbf{f}\texttt{][x} \mapsto \mathbf{id}_z\texttt{] in}\\ & & \quad\quad (\texttt{bind [f} \mapsto \mathbf{f}\texttt{][x} \mapsto \mathbf{id}_y\texttt{] in (f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4\texttt{)}^4\texttt{)}^9\texttt{)}^{10}\\ \rightarrow^* & & \ldots\end{array}$$

showing that the program does indeed loop. ∎

$$[\mathit{bind}] \quad (\widehat{\mathsf{C}}, \widehat{\rho}) \models (\texttt{bind}\ \rho\ \texttt{in}\ it_0^{\ell_0})^{\ell}$$
$$\text{iff} \quad (\widehat{\mathsf{C}}, \widehat{\rho}) \models it_0^{\ell_0} \ \wedge\ \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell) \ \wedge\ \rho\ \mathcal{R}\ \widehat{\rho}$$

$$[\mathit{close}] \quad (\widehat{\mathsf{C}}, \widehat{\rho}) \models (\texttt{close}\ t_0\ \texttt{in}\ \rho)^{\ell}$$
$$\text{iff} \quad \{t_0\} \subseteq \widehat{\mathsf{C}}(\ell) \ \wedge\ \rho\ \mathcal{R}\ \widehat{\rho}$$

Table 3.4: Abstract Control Flow Analysis for intermediate expressions.

3.2.2 Semantic Correctness

We shall formulate semantic correctness of the Control Flow Analysis as a *subject reduction result*; this is an approach borrowed from type theory and merely says that an acceptable result of the analysis remains acceptable under evaluation. However, in order to do that we need to extend the analysis to intermediate expressions.

Analysis of intermediate expressions. The clauses for the constructs `bind` ρ `in` ie and `close` t_0 `in` ρ are given in Table 3.4; the remaining clauses are as in Table 3.1 (with the obvious replacements of expressions with intermediate expressions).

The clause [*bind*] reflects that its body will be executed and hence whatever it evaluates to will also be a possible value for the construct. Additionally, it expresses that there is a certain relationship $\mathcal{R}$ between the local environment (of the semantics) and the abstract environment (of the analysis). The clause [*close*] is similar in spirit to the clauses for function abstraction: the term of the closure is a possible value of the construct. Additionally, there has to be a relationship $\mathcal{R}$ between the two environments.

Correctness relation. The purpose of the global abstract environment, $\widehat{\rho}$, is to model *all* of the local environments arising during evaluation. We formalise this by defining the *correctness relation*

$$\mathcal{R} : (\mathbf{Env} \times \widehat{\mathbf{Env}}) \rightarrow \{\mathit{true}, \mathit{false}\}$$

and demanding that $\rho\ \mathcal{R}\ \widehat{\rho}$ for all local environments, ρ, occurring in the intermediate expressions. We then define:

$$\rho\ \mathcal{R}\ \widehat{\rho} \quad \text{iff} \quad \mathit{dom}(\rho) \subseteq \mathit{dom}(\widehat{\rho}) \wedge \forall x \in \mathit{dom}(\rho)\ \forall t_x\ \forall \rho_x :$$
$$(\rho(x) = \texttt{close}\ t_x\ \texttt{in}\ \rho_x) \ \Rightarrow\ (t_x \in \widehat{\rho}(x) \wedge \rho_x\ \mathcal{R}\ \widehat{\rho})$$

This clearly demands that the function abstraction, t_x, in $\rho(x)$ must be an element of $\widehat{\rho}(x)$. It also shows that all local environments reachable from ρ, e.g. ρ_x, must be modelled by $\widehat{\rho}$ as well. Note that the relation $\mathcal{R}$ is well-defined because each recursive call is performed on a local environment that

$$
\begin{array}{ccccccccccc}
\rho & \vdash & ie & \rightarrow & ie' & \rightarrow & ie'' & \rightarrow & \cdots & \rightarrow & v^{\ell} \\
\updownarrow \mathcal{R} & & \updownarrow \models & & \updownarrow \models & & \updownarrow \models & & & & \updownarrow \models \\
\widehat{\rho} & & (\widehat{\mathsf{C}}, \widehat{\rho}) & & (\widehat{\mathsf{C}}, \widehat{\rho}) & & (\widehat{\mathsf{C}}, \widehat{\rho}) & & \cdots & & (\widehat{\mathsf{C}}, \widehat{\rho})
\end{array}
$$

Figure 3.3: Preservation of analysis result.

is *strictly smaller* than that of the call itself; thus a simple proof by well-founded induction (Appendix B) suffices for showing the well-definedness of $\mathcal{R}$.

Example 3.9 Suppose that:

$$
\begin{array}{rcl}
\rho & = & [x \mapsto \texttt{close}\ t_1\ \texttt{in}\ \rho_1][y \mapsto \texttt{close}\ t_2\ \texttt{in}\ \rho_2] \\
\rho_1 & = & [\,] \\
\rho_2 & = & [x \mapsto \texttt{close}\ t_3\ \texttt{in}\ \rho_3] \\
\rho_3 & = & [\,]
\end{array}
$$

Then $\rho\ \mathcal{R}\ \widehat{\rho}$ amounts to $\{t_1, t_3\} \subseteq \widehat{\rho}(x) \wedge \{t_2\} \subseteq \widehat{\rho}(y)$. ■

We shall sometimes find it helpful to split the definition of $\mathcal{R}$ into two components. For this we make use of the auxiliary relation

$$\mathcal{V} : (\mathbf{Val} \times (\widehat{\mathbf{Env}} \times \widehat{\mathbf{Val}})) \rightarrow \{\mathit{true}, \mathit{false}\}$$

and define $\mathcal{V}$ and $\mathcal{R}$ by mutual recursion:

$$
\begin{array}{rcl}
v\ \mathcal{V}\ (\widehat{\rho}, \widehat{v}) & \text{iff} & \forall t\ \forall \rho : (v = \texttt{close}\ t\ \texttt{in}\ \rho) \Rightarrow (t \in \widehat{v} \wedge \rho\ \mathcal{R}\ \widehat{\rho}) \\
\rho\ \mathcal{R}\ \widehat{\rho} & \text{iff} & \mathit{dom}(\rho) \subseteq \mathit{dom}(\widehat{\rho}) \wedge \forall x \in \mathit{dom}(\rho) : \rho(x)\ \mathcal{V}\ (\widehat{\rho}, \widehat{\rho}(x))
\end{array}
$$

Clearly the two definitions of $\mathcal{R}$ are equivalent.

Correctness result. The correctness result is now expressed by:

Theorem 3.10
If $\rho\ \mathcal{R}\ \widehat{\rho}$ and $\rho \vdash ie \rightarrow ie'$ then $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie$ implies $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie'$.

This is illustrated in Figure 3.3 for a terminating evaluation sequence $\rho \vdash ie \rightarrow^* v^{\ell}$; note that the result is analogous to that of Corollary 2.17 for the Live Variables Analysis in Chapter 2.

The intuitive content of the result is as follows:

> If there is a possible evaluation of the program such that the function at a call point evaluates to some abstraction, then this abstraction has to be in the set of possible abstractions computed by the analysis.

To see this assume that $\rho \vdash t^\ell \rightarrow^* (\texttt{close}\ t_0\ \texttt{in}\ \rho_0)^\ell$ and that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models t^\ell$ as well as $\rho\ \mathcal{R}\ \widehat{\rho}$. Then Theorem 3.10 (and an immediate numerical induction) gives that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models (\texttt{close}\ t_0\ \texttt{in}\ \rho_0)^\ell$. Now from the clause [*close*] of Table 3.4 we get that $t_0 \in \widehat{\mathsf{C}}(\ell)$ as was claimed. It is worth noticing that if the program is closed, i.e. if it does not contain free variables, then ρ will be [ ] and the condition $\rho\ \mathcal{R}\ \widehat{\rho}$ is trivially fulfilled.

Note that the theorem expresses that *all* acceptable analysis results remain acceptable under evaluation. One advantage of this is that we do not need to rely on the existence of a least or "best" solution (to be proved in Subsection 3.2.3) in order to formulate the result. Indeed the result does not say that the "best" solution remains "best" – merely that it remains acceptable. More importantly, the result opens up the possibility that the efficient realisation of Sections 3.3 and 3.4 computes a more approximate solution than the least (perhaps using the techniques of Chapter 4). Finally, note that the formulation of the theorem crucially depends on having defined the analysis for all intermediate expressions rather than just all ordinary expressions.

We shall now turn to the proof of Theorem 3.10. We first state an important observation:

Fact 3.11 If $(\widehat{\mathsf{C}}, \widehat{\rho}) \models it^{\ell_1}$ and $\widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\mathsf{C}}(\ell_2)$ then $(\widehat{\mathsf{C}}, \widehat{\rho}) \models it^{\ell_2}$. ∎

Proof By cases on the clauses for "$\models$". ∎

We then prove Theorem 3.10:

Proof We assume that $\rho\ \mathcal{R}\ \widehat{\rho}$ and $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie$ and prove $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie'$ by induction on the structure of the inference tree for $\rho \vdash ie \rightarrow ie'$. Most cases simply amount to inspecting the defining clause for $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie$; note that this method of proof applies to all fixed points of a recursive definition and in particular also to the greatest fixed point. We only give the proofs for some of the more interesting cases.

The case [*var*]. Here $\rho \vdash ie \rightarrow ie'$ is:

$$\rho \vdash x^\ell \rightarrow v^\ell \quad \text{because} \quad x \in dom(\rho) \text{ and } v = \rho(x)$$

If $v = c$ there is nothing to prove so suppose that $v = \texttt{close}\ t_0\ \texttt{in}\ \rho_0$. From $\rho\ \mathcal{R}\ \widehat{\rho}$ we get $v\ \mathcal{V}\ (\widehat{\rho}, \widehat{\rho}(x))$ and hence $t_0 \in \widehat{\rho}(x)$ and $\rho_0\ \mathcal{R}\ \widehat{\rho}$. From $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie$ we get $\widehat{\rho}(x) \subseteq \widehat{\mathsf{C}}(\ell)$, and hence $t_0 \in \widehat{\mathsf{C}}(\ell)$. Since $t_0 \in \widehat{\mathsf{C}}(\ell)$ and $\rho_0\ \mathcal{R}\ \widehat{\rho}$ we have established $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie'$.

The case [*fn*]. Here $\rho \vdash ie \to ie'$ is:

$$\begin{array}{l}\rho \vdash (\texttt{fn}\ x\ \texttt{=>}\ e_0)^\ell \to (\texttt{close}\ (\texttt{fn}\ x\ \texttt{=>}\ e_0)\ \texttt{in}\ \rho_0)^\ell \\ \text{where } \rho_0 = \rho \mid FV(\texttt{fn}\ x\ \texttt{=>}\ e_0)\end{array}$$

From $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie$ we get $(\texttt{fn}\ x\ \texttt{=>}\ e_0) \in \widehat{\mathsf{C}}(\ell)$; from $\rho\ \mathcal{R}\ \widehat{\rho}$ it is immediate to get $\rho_0\ \mathcal{R}\ \widehat{\rho}$; this then establishes $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie'$.

The case [app_1]. Here $\rho \vdash ie \to ie'$ is:

$$\rho \vdash (ie_1\ ie_2)^\ell \to (ie_1'\ ie_2)^\ell \quad \text{because} \quad \rho \vdash ie_1 \to ie_1'$$

The defining clauses of $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie$ and $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie'$ are equal except that the former has $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie_1$ where the latter has $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie_1'$. From the induction hypothesis applied to

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie_1, \quad \rho\ \mathcal{R}\ \widehat{\rho}, \quad \text{and} \quad \rho \vdash ie_1 \to ie_1'$$

we get $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie_1'$ and the desired result then follows.

The case [app_{fn}]. Here $\rho \vdash ie \to ie'$ is:

$$\rho \vdash ((\texttt{close}\ (\texttt{fn}\ x\ \texttt{=>}\ t_0^{\ell_0})\ \texttt{in}\ \rho_1)^{\ell_1}\ v_2^{\ell_2})^\ell \to (\texttt{bind}\ \rho_1[x \mapsto v_2]\ \texttt{in}\ t_0^{\ell_0})^\ell$$

From $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie$ we have $(\widehat{\mathsf{C}}, \widehat{\rho}) \models (\texttt{close}\ (\texttt{fn}\ x\ \texttt{=>}\ t_0^{\ell_0})\ \texttt{in}\ \rho_1)^{\ell_1}$ which yields:

$$(\texttt{fn}\ x\ \texttt{=>}\ t_0^{\ell_0}) \in \widehat{\mathsf{C}}(\ell_1) \quad \text{and} \quad \rho_1\ \mathcal{R}\ \widehat{\rho}$$

Further we have $(\widehat{\mathsf{C}}, \widehat{\rho}) \models v_2^{\ell_2}$; in the case where $v_2 = c$, it is immediate that

$$v_2\ \mathcal{V}\ (\widehat{\rho}, \widehat{\mathsf{C}}(\ell_2))$$

and in the case where $v_2 = \texttt{close}\ t_2\ \texttt{in}\ \rho_2$ it follows from the definition of $(\widehat{\mathsf{C}}, \widehat{\rho}) \models v_2^{\ell_2}$. Finally, the first universally quantified formula of the definition of $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie$ gives:

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models t_0^{\ell_0}, \quad \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\rho}(x), \quad \text{and} \quad \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell)$$

Now observe that $v_2\ \mathcal{V}\ (\widehat{\rho}, \widehat{\rho}(x))$ since $\widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\rho}(x)$ follows from the clause (app_{fn}). Since $\rho_1\ \mathcal{R}\ \widehat{\rho}$ we now have

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models t_0^{\ell_0}, \quad \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell), \quad \text{and} \quad (\rho_1[x \mapsto v_2])\ \mathcal{R}\ \widehat{\rho}$$

and this establishes the desired $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie'$.

The case [$bind_2$]. Here $\rho \vdash ie \to ie'$ is:

$$\rho \vdash (\texttt{bind}\ \rho_1\ \texttt{in}\ v_1^{\ell_1})^\ell \to v_1^\ell$$

From $(\widehat{\mathsf{C}}, \widehat{\rho}) \models ie$ we have $(\widehat{\mathsf{C}}, \widehat{\rho}) \models v_1^{\ell_1}$ as well as $\widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\mathsf{C}}(\ell)$ and the desired $(\widehat{\mathsf{C}}, \widehat{\rho}) \models v_1^\ell$ follows from Fact 3.11.

This completes the proof. ∎

Example 3.12 From Example 3.7 we have:

$$[\,] \vdash ((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5 \rightarrow^* (\texttt{close (fn y => y}^3)\ \texttt{in}\ [\,])^5$$

Next let $(\widehat{\mathsf{C}}_{\mathsf{e}}, \widehat{\rho}_{\mathsf{e}})$ be as in Example 3.3. Clearly $[\,]\ \mathcal{R}\ \widehat{\rho}_{\mathsf{e}}$ and from Example 3.5 we have:

$$(\widehat{\mathsf{C}}_{\mathsf{e}}, \widehat{\rho}_{\mathsf{e}}) \models ((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5$$

According to Theorem 3.10 we can now conclude:

$$(\widehat{\mathsf{C}}_{\mathsf{e}}, \widehat{\rho}_{\mathsf{e}}) \models (\texttt{close (fn y => y}^3)\ \texttt{in}\ [\,])^5$$

Using Table 3.4 it is easy to check that this is indeed the case. ■

3.2.3 Existence of Solutions

Having defined the analysis in Table 3.1 it is natural to ask the following question: Does each expression e admit a Control Flow Analysis, i.e. does there exist $(\widehat{\mathsf{C}}, \widehat{\rho})$ such that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e$? We shall show that the answer to this question is *yes*.

However, this does not exclude the possibility of having many different analyses for the same expression so an additional question is: Does each expression e have a "least" Control Flow Analysis, i.e. does there exists $(\widehat{\mathsf{C}}_0, \widehat{\rho}_0)$ such that $(\widehat{\mathsf{C}}_0, \widehat{\rho}_0) \models e$ and such that whenever $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e$ then $(\widehat{\mathsf{C}}_0, \widehat{\rho}_0)$ is "less than" $(\widehat{\mathsf{C}}, \widehat{\rho})$? Again, the answer will be *yes*.

Here "least" is with respect to the partial order defined by:

$$\begin{array}{lll} (\widehat{\mathsf{C}}_1, \widehat{\rho}_1) \sqsubseteq (\widehat{\mathsf{C}}_2, \widehat{\rho}_2) & \text{iff} & (\forall \ell \in \mathbf{Lab} : \widehat{\mathsf{C}}_1(\ell) \subseteq \widehat{\mathsf{C}}_2(\ell)) \ \wedge \\ & & (\forall x \in \mathbf{Var} : \widehat{\rho}_1(x) \subseteq \widehat{\rho}_2(x)) \end{array}$$

It will be the topic of Sections 3.3 and 3.4 (and Mini Project 3.1) to show that the least solution can be computed efficiently for all expressions. However, it may be instructive to give a general proof for the existence of least solutions also for intermediate expressions. To this end we recall the notion of a Moore family (see Appendix A and Exercise 2.7):

> A subset Y of a complete lattice $L = (L, \sqsubseteq)$ is a *Moore family* if and only if $(\bigsqcap Y') \in Y$ for all $Y' \subseteq Y$.

This property is also called the *model intersection property* because whenever we take the "intersection" of a number of "models" we still get a "model".

Proposition 3.13
For all $ie \in \mathbf{IExp}$ the set $\{(\widehat{\mathsf{C}}, \widehat{\rho}) \mid (\widehat{\mathsf{C}}, \widehat{\rho}) \models ie\}$ is a Moore family.

It is an immediate corollary that all intermediate expressions ie admit a Control Flow Analysis: Let Y' be the empty set; then $\bigsqcap Y'$ is an element of $\{(\widehat{\mathsf{C}}, \widehat{\rho}) \mid (\widehat{\mathsf{C}}, \widehat{\rho}) \models ie\}$ showing that there exists at least one analysis of ie.

It is also an immediate corollary that all intermediate expressions have a least Control Flow Analysis: Let Y' be the set $\{(\widehat{\mathsf{C}}, \widehat{\rho}) \mid (\widehat{\mathsf{C}}, \widehat{\rho}) \models ie\}$; then $\bigsqcap Y'$ is an element of $\{(\widehat{\mathsf{C}}, \widehat{\rho}) \mid (\widehat{\mathsf{C}}, \widehat{\rho}) \models ie\}$ so it will also be an analysis of ie. Clearly $\bigsqcap Y' \sqsubseteq (\widehat{\mathsf{C}}, \widehat{\rho})$ for all other analyses $(\widehat{\mathsf{C}}, \widehat{\rho})$ of ie so it is the least analysis result.

In preparation for the proof of Proposition 3.13 we shall first establish an auxiliary result for $\mathcal{R}$ and $\mathcal{V}$:

Lemma 3.14

(i) For all $\rho \in \mathbf{Env}$ the set $\{\widehat{\rho} \mid \rho \; \mathcal{R} \; \widehat{\rho}\}$ is a Moore family.
(ii) For all $v \in \mathbf{Val}$ the set $\{(\widehat{\rho}, \widehat{v}) \mid v \; \mathcal{V} \; (\widehat{\rho}, \widehat{v})\}$ is a Moore family. ∎

Proof To prove (i) we proceed by well-founded induction on ρ (which is also the manner in which the existence of the predicate was proved). Now assume that

$$\forall i \in I : \rho \; \mathcal{R} \; \widehat{\rho}_i$$

for some index set I and let us show that $\rho \; \mathcal{R} \; (\bigsqcap_i \widehat{\rho}_i)$. For this consider x, t_x, and ρ_x such that:

$$\rho(x) = \texttt{close } t_x \texttt{ in } \rho_x$$

We then know

$$\forall i \in I : t_x \in \widehat{\rho}_i(x) \;\wedge\; \rho_x \; \mathcal{R} \; \widehat{\rho}_i$$

and using the induction hypothesis it follows that

$$t_x \in (\bigsqcap_i \widehat{\rho}_i)(x) \;\wedge\; \rho_x \; \mathcal{R} \; (\bigsqcap_i \widehat{\rho}_i)$$

(taking care when $I = \emptyset$).

To prove (ii) we simply expand the definition of $\mathcal{V}$ and note that the result then follows from (i). ∎

We now prove Proposition 3.13 using coinduction (see Appendix B):

Proof The ternary relation $\models$ of Tables 3.1 and 3.4 is the greatest fixed point of a function $\mathcal{Q}$ as explained in Section 3.1. Now assume that

$$\forall i \in I : (\widehat{\mathsf{C}}_i, \widehat{\rho}_i) \models ie$$

and let us prove that $\bigsqcap_i (\widehat{\mathsf{C}}_i, \widehat{\rho}_i) \models ie$. We shall proceed by coinduction (see Appendix B) so we start by defining the ternary relation Q' by:

$$(\widehat{\mathsf{C}}', \widehat{\rho}') \; Q' \; ie' \;\text{ iff }\; (\widehat{\mathsf{C}}', \widehat{\rho}') = \bigsqcap_i (\widehat{\mathsf{C}}_i, \widehat{\rho}_i) \;\wedge\; \forall i \in I : (\widehat{\mathsf{C}}_i, \widehat{\rho}_i) \models ie'$$

It is then immediate that we have:

$$\bigsqcap_i(\widehat{\mathsf{C}}_i, \widehat{\rho}_i)\ Q'\ ie$$

The coinduction proof principle requires that we prove

$$Q' \sqsubseteq \mathcal{Q}(Q')$$

and this amounts to assuming $(\widehat{\mathsf{C}}', \widehat{\rho}')\ Q'\ ie'$ and proving that $(\widehat{\mathsf{C}}', \widehat{\rho}')\ (\mathcal{Q}(Q'))\ ie'$. So let us assume that

$$\forall i \in I : (\widehat{\mathsf{C}}_i, \widehat{\rho}_i) \models ie'$$

and let us show that:

$$\bigsqcap_i(\widehat{\mathsf{C}}_i, \widehat{\rho}_i)\ (\mathcal{Q}(Q'))\ ie'$$

For this we consider each of the clauses for ie' in turn.

Here we shall only deal with the more complicated choice $ie' = (it_1^{\ell_1}\ it_2^{\ell_2})^{\ell}$. From

$$\forall i \in I : (\widehat{\mathsf{C}}_i, \widehat{\rho}_i) \models (it_1^{\ell_1}\ it_2^{\ell_2})^{\ell}$$

we get $\forall i \in I : (\widehat{\mathsf{C}}_i, \widehat{\rho}_i) \models it_1^{\ell_1}$ and hence:

$$\bigsqcap_i(\widehat{\mathsf{C}}_i, \widehat{\rho}_i)\ Q'\ it_1^{\ell_1}$$

Similarly we get:

$$\bigsqcap_i(\widehat{\mathsf{C}}_i, \widehat{\rho}_i)\ Q'\ it_2^{\ell_2}$$

Next consider $(\texttt{fn}\ x \texttt{ => } t_0^{\ell_0}) \in \bigcap_i(\widehat{\mathsf{C}}_i(\ell_1))$ and let us prove that:

$$\bigcap_i(\widehat{\mathsf{C}}_i(\ell_2)) \subseteq \bigcap_i(\widehat{\rho}_i(x)),\quad \bigcap_i(\widehat{\mathsf{C}}_i(\ell_0)) \subseteq \bigcap_i(\widehat{\mathsf{C}}_i(\ell)),\quad \bigsqcap_i(\widehat{\mathsf{C}}_i, \widehat{\rho}_i)\ Q'\ t_0^{\ell_0} \qquad (3.1)$$

For all $i \in I$ we have that $(\widehat{\mathsf{C}}_i, \widehat{\rho}_i) \models ie'$ and since $(\texttt{fn}\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\mathsf{C}}_i(\ell_1)$ we have

$$\widehat{\mathsf{C}}_i(\ell_2) \subseteq \widehat{\rho}_i(x),\quad \widehat{\mathsf{C}}_i(\ell_0) \subseteq \widehat{\mathsf{C}}_i(\ell),\quad \text{and}\quad (\widehat{\mathsf{C}}_i, \widehat{\rho}_i) \models t_0^{\ell_0}$$

and this then gives (3.1) as desired (taking care when $I = \emptyset$). The case of $(\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}) \in \bigcap_i(\widehat{\mathsf{C}}_i(\ell_1))$ is similar. This completes the proof. ∎

Example 3.15 Let us return to Example 3.5 and consider the following potential analysis results for $((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5$:

	$(\widehat{\mathsf{C}}_e, \widehat{\rho}_e)$	$(\widehat{\mathsf{C}}'_e, \widehat{\rho}'_e)$	$(\widehat{\mathsf{C}}''_e, \widehat{\rho}''_e)$
1	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn y => y}^3\}$
2	$\{\texttt{fn x => x}^1\}$	$\{\texttt{fn x => x}^1\}$	$\{\texttt{fn x => x}^1\}$
3	$\emptyset$	$\{\texttt{fn x => x}^1\}$	$\{\texttt{fn y => y}^3\}$
4	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn y => y}^3\}$
5	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn y => y}^3\}$
x	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn y => y}^3\}$	$\{\texttt{fn y => y}^3\}$
y	$\emptyset$	$\{\texttt{fn x => x}^1\}$	$\{\texttt{fn y => y}^3\}$

It is straightforward to verify that

$$(\widehat{\mathsf{C}}'_{\mathsf{e}}, \widehat{\rho}'_{\mathsf{e}}) \models ((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5$$
$$(\widehat{\mathsf{C}}''_{\mathsf{e}}, \widehat{\rho}''_{\mathsf{e}}) \models ((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5$$

Now Proposition 3.13 ensures that also:

$$(\widehat{\mathsf{C}}'_{\mathsf{e}} \sqcap \widehat{\mathsf{C}}''_{\mathsf{e}}, \widehat{\rho}'_{\mathsf{e}} \sqcap \widehat{\rho}''_{\mathsf{e}}) \models ((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5$$

Neither $(\widehat{\mathsf{C}}'_{\mathsf{e}}, \widehat{\rho}'_{\mathsf{e}})$ nor $(\widehat{\mathsf{C}}''_{\mathsf{e}}, \widehat{\rho}''_{\mathsf{e}})$ is a least solution. Their "intersection" $(\widehat{\mathsf{C}}'_{\mathsf{e}} \sqcap \widehat{\mathsf{C}}''_{\mathsf{e}}, \widehat{\rho}'_{\mathsf{e}} \sqcap \widehat{\rho}''_{\mathsf{e}})$ is smaller and equals $(\widehat{\mathsf{C}}_{\mathsf{e}}, \widehat{\rho}_{\mathsf{e}})$ which turns out to be the least analysis result for the expression. ■

3.2.4 Coinduction versus Induction

One of the important aspects of the development of the abstract Control Flow Analysis in Table 3.1 is the *coinductive definition* of the acceptability relation:

$$\models \text{ as the } \textit{greatest} \text{ fixed point of a functional } \mathcal{Q}$$

An alternative might be an *inductive definition* of an acceptability relation:

$$\models' \text{ as the } \textit{least} \text{ fixed point of the functional } \mathcal{Q}.$$

However, in Example 3.6 we argued that this might be inappropriate and here we are going to demonstrate that an important part of the development of the previous subsection actually fails for the least fixed point ($\models'$) of $\mathcal{Q}$.

Proposition 3.16
There exists $e_\star \in \mathbf{Exp}$ such that $\{(\widehat{\mathsf{C}}, \widehat{\rho}) \mid (\widehat{\mathsf{C}}, \widehat{\rho}) \models' e_\star\}$ is *not* a Moore family.

Proof (sketch) This proof is rather demanding and is best omitted on a first reading. To make the proof tractable we consider

$$\begin{array}{rcl} \mathsf{e}_\star & = & \mathsf{t}_\star^\ell \\ \mathsf{t}_\star & = & (\texttt{fn x => } (\mathsf{x}^\ell\ \mathsf{x}^\ell)^\ell)^\ell\ (\texttt{fn x => } (\mathsf{x}^\ell\ \mathsf{x}^\ell)^\ell)^\ell \end{array}$$

and take:

$$\begin{array}{rcl} \mathbf{Lab}_{\mathsf{e}} & = & \{\ell\} \\ \mathbf{Var}_{\mathsf{e}} & = & \{\mathsf{x}\} \\ \mathbf{Term}_{\mathsf{e}} & = & \{\mathsf{t}_\star, \texttt{fn x => } (\mathsf{x}^\ell\ \mathsf{x}^\ell)^\ell, \mathsf{x}^\ell\ \mathsf{x}^\ell, \mathsf{x}\} \\ \mathbf{IExp}_{\mathsf{e}} & = & \{t^\ell \mid t \in \mathbf{Term}_{\mathsf{e}}\} \\ \widehat{\mathbf{Val}}_{\mathsf{e}} & = & \mathcal{P}(\{\texttt{fn x => } (\mathsf{x}^\ell\ \mathsf{x}^\ell)^\ell\}) = \{\emptyset, \{\texttt{fn x => } (\mathsf{x}^\ell\ \mathsf{x}^\ell)^\ell\}\} \end{array}$$

This is in line with Exercise 3.2 (and the development of Subsection 3.3.2) and as we shall see the proof of Proposition 3.13 is not invalidated.

Next let $\mathcal{Q}$ be the functional defined by Table 3.1 and let Q be in the domain of $\mathcal{Q}$. The condition

$$Q = \mathcal{Q}(Q)$$

is equivalent to:

$$\forall t \in \mathbf{Term}_e : \forall(\widehat{\mathsf{C}}, \widehat{\rho}) : ((\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ t^\ell \ \text{ iff } \ (\widehat{\mathsf{C}}, \widehat{\rho})\ \mathcal{Q}(Q)\ t^\ell)$$

By considering the four possibilities of $t \in \mathbf{Term}_e$ this is then equivalent to the conjunction of the following four conditions (where $(\widehat{\mathsf{C}}, \widehat{\rho})$ is universally quantified):

$$\begin{array}{rcl}
(\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ \mathtt{x}^\ell & \text{iff} & \widehat{\rho}(\mathtt{x}) \subseteq \widehat{\mathsf{C}}(\ell) \\
(\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ (\mathtt{fn\ x\ =>\ (x}^\ell\ \mathtt{x}^\ell)^\ell)^\ell & \text{iff} & \{\mathtt{fn\ x\ =>\ (x}^\ell\ \mathtt{x}^\ell)^\ell\} \subseteq \widehat{\mathsf{C}}(\ell) \\
(\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell & \text{iff} & (\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ \mathtt{x}^\ell\ \wedge \\
 & & \widehat{\mathsf{C}}(\ell) \neq \emptyset\ \Rightarrow\ ((\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell\ \wedge \\
 & & \qquad \widehat{\mathsf{C}}(\ell) \subseteq \widehat{\rho}(\mathtt{x})) \\
(\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ t_*^\ell & \text{iff} & (\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ (\mathtt{fn\ x\ =>\ (x}^\ell\ \mathtt{x}^\ell)^\ell)^\ell\ \wedge \\
 & & \widehat{\mathsf{C}}(\ell) \neq \emptyset\ \Rightarrow\ ((\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell\ \wedge \\
 & & \qquad \widehat{\mathsf{C}}(\ell) \subseteq \widehat{\rho}(\mathtt{x}))
\end{array}$$

Here we have used that $\widehat{\mathsf{C}}(\ell) \neq \emptyset$ implies that $\widehat{\mathsf{C}}(\ell) = \{\mathtt{fn\ x\ =>\ (x}^\ell\ \mathtt{x}^\ell)^\ell\}$ as follows from the definition of $\widehat{\mathbf{Val}}_e$ in the beginning of this proof.

The conjunction of the above four conditions implies the conjunction of the following four conditions:

$$\begin{array}{rcl}
(\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ \mathtt{x}^\ell & \text{iff} & \widehat{\rho}(\mathtt{x}) \subseteq \widehat{\mathsf{C}}(\ell) \\
(\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ (\mathtt{fn\ x\ =>\ (x}^\ell\ \mathtt{x}^\ell)^\ell)^\ell & \text{iff} & \{\mathtt{fn\ x\ =>\ (x}^\ell\ \mathtt{x}^\ell)^\ell\} \subseteq \widehat{\mathsf{C}}(\ell) \\
(\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell & \text{iff} & \widehat{\rho}(\mathtt{x}) \subseteq \widehat{\mathsf{C}}(\ell)\ \wedge \\
 & & (\widehat{\mathsf{C}}(\ell) \neq \emptyset\ \Rightarrow\ (\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell)\ \wedge \\
 & & \widehat{\mathsf{C}}(\ell) \subseteq \widehat{\rho}(\mathtt{x}) \\
(\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ t_*^\ell & \text{iff} & \{\mathtt{fn\ x\ =>\ (x}^\ell\ \mathtt{x}^\ell)^\ell\} \subseteq \widehat{\mathsf{C}}(\ell)\ \wedge \\
 & & (\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell\ \wedge \\
 & & \widehat{\mathsf{C}}(\ell) \subseteq \widehat{\rho}(\mathtt{x})
\end{array}$$

This implication can be reversed and this shows that also the conjunct of the above four conditions is equivalent to $Q = \mathcal{Q}(Q)$.

Using that $\widehat{\rho}(\mathtt{x})$ can only be $\emptyset$ or $\{\mathtt{fn\ x\ =>\ (x}^\ell\ \mathtt{x}^\ell)^\ell\}$, and similarly for $\widehat{\mathsf{C}}(\ell)$, the above four conditions are equivalent to the following:

$$(\widehat{\mathsf{C}}, \widehat{\rho})\ Q\ \mathtt{x}^\ell \ \text{ iff } \ \widehat{\rho}(\mathtt{x}) \subseteq \widehat{\mathsf{C}}(\ell)$$

$$
\begin{array}{rcl}
(\widehat{\mathsf{C}},\widehat{\rho})\ Q\ (\mathtt{fn\ x\ =>}\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell)^\ell & \text{iff} & \{\mathtt{fn\ x\ =>}\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell\} = \widehat{\mathsf{C}}(\ell) \\
(\widehat{\mathsf{C}},\widehat{\rho})\ Q\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell & \text{iff} & \widehat{\rho}(\mathtt{x}) = \widehat{\mathsf{C}}(\ell)\ \wedge \\
 & & (\widehat{\mathsf{C}}(\ell) \neq \emptyset\ \Rightarrow\ (\widehat{\mathsf{C}},\widehat{\rho})\ Q\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell) \\
(\widehat{\mathsf{C}},\widehat{\rho})\ Q\ t_*^\ell & \text{iff} & \{\mathtt{fn\ x\ =>}\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell\} = \widehat{\mathsf{C}}(\ell) = \widehat{\rho}(\mathtt{x})\ \wedge \\
 & & (\widehat{\mathsf{C}},\widehat{\rho})\ Q\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell
\end{array}
$$

It follows that the conjunct of the above four conditions is once more equivalent to $Q = \mathcal{Q}(Q)$.

The crucial case in the definition of $(\widehat{\mathsf{C}},\widehat{\rho})\ Q\ e$ is for $e = (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell$ as this determines the truth or falsity of all other cases. We shall now try to get a handle on the candidates $Q_1, \cdots, Q_n$ for satisfying $Q_i = \mathcal{Q}(Q_i)$. Concentrating on the condition for $(\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell$ it follows that $(\widehat{\mathsf{C}},\widehat{\rho})\ Q_i\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell$ must demand that $\widehat{\mathsf{C}}(\ell) = \widehat{\rho}(\mathtt{x})$. Since each of $\widehat{\mathsf{C}}(\ell)$ and $\widehat{\rho}(\mathtt{x})$ can only be $\{\mathtt{fn\ x\ =>}\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell\}$ or $\emptyset$ there are at most the following four candidates for Q_i:

$$
\begin{array}{rcl}
(\widehat{\mathsf{C}},\widehat{\rho})\ Q_1\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell & \text{iff} & \widehat{\mathsf{C}}(\ell) = \widehat{\rho}(\mathtt{x}) \\
(\widehat{\mathsf{C}},\widehat{\rho})\ Q_2\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell & \text{iff} & \widehat{\mathsf{C}}(\ell) = \widehat{\rho}(\mathtt{x}) = \emptyset \\
(\widehat{\mathsf{C}},\widehat{\rho})\ Q_3\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell & \text{iff} & \widehat{\mathsf{C}}(\ell) = \widehat{\rho}(\mathtt{x}) \neq \emptyset \\
(\widehat{\mathsf{C}},\widehat{\rho})\ Q_4\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell & \text{iff} & \mathit{false}
\end{array}
$$

Verifying the condition

$$
\forall(\widehat{\mathsf{C}},\widehat{\rho}) : \left(\begin{array}{rcl} (\widehat{\mathsf{C}},\widehat{\rho})\ Q_i\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell & \text{iff} & \widehat{\rho}(\mathtt{x}) = \widehat{\mathsf{C}}(\ell)\ \wedge \\ & & (\widehat{\mathsf{C}}(\ell) \neq \emptyset\ \Rightarrow\ (\widehat{\mathsf{C}},\widehat{\rho})\ Q_i\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell) \end{array} \right)
$$

for $i \in \{1,2,3,4\}$ it follows that Q_1 and Q_2 satisfy the condition whereas Q_3 and Q_4 do not.

It is now straightforward to verify also the remaining three conditions and it follows that:

$$Q_i = \mathcal{Q}(Q_i) \quad \text{for } i = 1,2$$

This means that Q_1 equals $\models$ (the greatest fixed point of $\mathcal{Q}$) and that Q_2 equals $\models'$ (the least fixed point of $\mathcal{Q}$). One can then calculate that

$$
\begin{array}{rcl}
(\widehat{\mathsf{C}},\widehat{\rho})\ Q_1\ t_*^\ell & \text{iff} & \widehat{\mathsf{C}}(\ell) = \widehat{\rho}(\mathtt{x}) \neq \emptyset \\
(\widehat{\mathsf{C}},\widehat{\rho})\ Q_2\ t_*^\ell & \text{iff} & \mathit{false}
\end{array}
$$

and this shows that

$$\{(\widehat{\mathsf{C}},\widehat{\rho}) \mid (\widehat{\mathsf{C}},\widehat{\rho})\ Q_1\ e_*\} = \{(\widehat{\mathsf{C}},\widehat{\rho}) \mid \widehat{\mathsf{C}}(\ell) = \widehat{\rho}(\mathtt{x}) = \{\mathtt{fn\ x\ =>}\ (\mathtt{x}^\ell\ \mathtt{x}^\ell)^\ell\}\}$$

which is a singleton set and in fact a Moore family, whereas

$$\{(\widehat{\mathsf{C}},\widehat{\rho}) \mid (\widehat{\mathsf{C}},\widehat{\rho})\ Q_2\ e_*\} = \emptyset$$

which cannot be a Moore family (since a Moore family is never empty). This completes the proof. ∎

3.3 Syntax Directed 0-CFA Analysis

We shall now show how to obtain efficient realisations of 0-CFA analyses. So assume throughout this section that $e_\star \in \mathbf{Exp}$ is the expression of interest and that we want to find a "good" solution $(\widehat{\mathsf{C}}, \widehat{\rho})$ satisfying $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$. This entails finding a solution that is *as small as possible* with respect to the partial order $\sqsubseteq$ defined in Section 3.2 by:

$$(\widehat{\mathsf{C}}_1, \widehat{\rho}_1) \sqsubseteq (\widehat{\mathsf{C}}_2, \widehat{\rho}_2) \text{ iff } (\forall \ell : \widehat{\mathsf{C}}_1(\ell) \subseteq \widehat{\mathsf{C}}_2(\ell)) \wedge (\forall x : \widehat{\rho}_1(x) \subseteq \widehat{\rho}_2(x))$$

Proposition 3.13 shows that a least solution does exist; however, the algorithm that is implicit in the proof does not have tractable (i.e. polynomial) complexity: it involves enumerating all candidate solutions, determining if they are indeed solutions, and if so taking the greatest lower bound with respect to the others found so far.

An alternative approach is somehow to obtain a finite set of constraints, say of the form *lhs* $\subseteq$ *rhs* (where *lhs* and *rhs* are much as described in Section 3.1), and then take the least solution to this system of constraints. The most obvious method is to expand the formula $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$ by unfolding all "recursive calls", using memoisation to keep track of all the expansions that have been performed so far, and stopping the expansion whenever a previously expanded call is re-encountered.

Three phases. We shall take a more direct route motivated by the above considerations; it has three phases:

(i) The specification of Table 3.1 is reformulated in a syntax directed manner (Subsection 3.3.1).

(ii) The syntax directed specification is turned into an algorithm for constructing a finite set of constraints (Subsection 3.4.1).

(iii) The least solution of this set of constraints is computed (Subsection 3.4.2).

This is indeed a *common phenomenon*: a specification "$\models_A$" is reformulated into a specification "$\models_B$" ensuring that

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_A e_\star \;\Leftarrow\; (\widehat{\mathsf{C}}, \widehat{\rho}) \models_B e_\star$$

so that "$\models_B$" is a *safe approximation* to "$\models_A$" and in particular the best (i.e. least) solution to "$\models_B e_\star$" is also a solution to "$\models_A e_\star$". This also ensures that all solutions to "$\models_B$" are semantically correct (assuming that this has already been established for all solutions to "$\models_A$"); however, we do not claim that a subject reduction result holds for $\models_B$ (even though it has been established for $\models_A$).

If additionally

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_A e_\star \;\Rightarrow\; (\widehat{\mathsf{C}}, \widehat{\rho}) \models_B e_\star$$

then we can be assured that *no solutions are lost* and hence the best (i.e. least) solution to "$\models_B e_\star$" will also be the best (i.e. least) solution to "$\models_A e_\star$". As we shall see, it may be necessary to *restrict attention* to only solutions $(\widehat{\mathsf{C}}, \widehat{\rho})$ satisfying some additional properties (e.g. that only program fragments of $e_\star$ appear in the range of $\widehat{\mathsf{C}}$ and $\widehat{\rho}$).

3.3.1 Syntax Directed Specification

In reformulating the specification of "$\models e_\star$" into a more computationally oriented specification "$\models_s e_\star$" we shall ensure that each function body is analysed *at most once* rather than each time the function could be applied. One way to achieve this is to analyse each function body *exactly once* as is done in the *syntax directed 0-CFA* analysis of Table 3.5; a better alternative would be to analyse only reachable function bodies and we refer to Mini Project 3.1 for how to achieve this. In Table 3.5 each function body is therefore analysed in the relevant clause for function abstraction rather than in the clause for function application; thus we now risk analysing unreachable program fragments.

The formal definition of $\models_s$. Since semantic correctness was dealt with in Section 3.2 there is no longer any need to consider intermediate expressions and consequently our specification of

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e$$

in Table 3.5 considers ordinary expressions only. We shall take "$\models_s$" to be the *largest* relation that satisfies the specification; however, given the syntax directed nature of the specification there is in fact only one relation that satisfies the specification (see Exercise 3.9). Hence it would be technically correct, but intuitively misleading, to claim that we take the least relation that satisfies the specification. In other words, whether or not $\models_s$ is defined inductively or coinductively, the same relation is defined; this is a common phenomenon whenever the clauses are defined in a syntax directed manner.

Example 3.17 Consider the expression loop

$$\begin{array}{l}(\texttt{let g = (fun f x => (f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4\texttt{)}^5\\ \texttt{in (g}^6\ \texttt{(fn z => z}^7\texttt{)}^8\texttt{)}^9\texttt{)}^{10}\end{array}$$

of Example 3.4. We shall verify that $(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models_s \mathsf{loop}$ where $\widehat{\mathsf{C}}_{\mathsf{lp}}$ and $\widehat{\rho}_{\mathsf{lp}}$ are as in Example 3.4. Using the clause [*let*], it is sufficient to show

$$(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \;\models_s\; (\texttt{fun f x => (f}^1\ \texttt{(fn y => y}^2\texttt{)}^3\texttt{)}^4\texttt{)}^5 \tag{3.2}$$

$$(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \;\models_s\; (\texttt{g}^6\ \texttt{(fn z => z}^7\texttt{)}^8\texttt{)}^9 \tag{3.3}$$

$[con]$ $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s c^\ell$ always

$[var]$ $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s x^\ell$ iff $\widehat{\rho}(x) \subseteq \widehat{\mathsf{C}}(\ell)$

$[fn]$ $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s (\texttt{fn}\ x \texttt{ => } e_0)^\ell$
iff $\{\texttt{fn}\ x \texttt{ => } e_0\} \subseteq \widehat{\mathsf{C}}(\ell) \wedge$
$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_0$

$[fun]$ $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s (\texttt{fun}\ f\ x \texttt{ => } e_0)^\ell$
iff $\{\texttt{fun}\ f\ x \texttt{ => } e_0\} \subseteq \widehat{\mathsf{C}}(\ell) \wedge$
$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_0 \wedge \{\texttt{fun}\ f\ x \texttt{ => } e_0\} \subseteq \widehat{\rho}(f)$

$[app]$ $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s (t_1^{\ell_1}\ t_2^{\ell_2})^\ell$
iff $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s t_1^{\ell_1} \wedge (\widehat{\mathsf{C}}, \widehat{\rho}) \models_s t_2^{\ell_2} \wedge$
$(\forall(\texttt{fn}\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\mathsf{C}}(\ell_1):$
$\widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\rho}(x) \wedge \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell)) \wedge$
$(\forall(\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\mathsf{C}}(\ell_1):$
$\widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\rho}(x) \wedge \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell))$

$[if]$ $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s (\texttt{if}\ t_0^{\ell_0}\ \texttt{then}\ t_1^{\ell_1}\ \texttt{else}\ t_2^{\ell_2})^\ell$
iff $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s t_0^{\ell_0} \wedge$
$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s t_1^{\ell_1} \wedge (\widehat{\mathsf{C}}, \widehat{\rho}) \models_s t_2^{\ell_2} \wedge$
$\widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\mathsf{C}}(\ell) \wedge \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\mathsf{C}}(\ell)$

$[let]$ $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s (\texttt{let}\ x = t_1^{\ell_1}\ \texttt{in}\ t_2^{\ell_2})^\ell$
iff $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s t_1^{\ell_1} \wedge (\widehat{\mathsf{C}}, \widehat{\rho}) \models_s t_2^{\ell_2} \wedge$
$\widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\rho}(x) \wedge \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\mathsf{C}}(\ell)$

$[op]$ $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s (t_1^{\ell_1}\ op\ t_2^{\ell_2})^\ell$ iff $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s t_1^{\ell_1} \wedge (\widehat{\mathsf{C}}, \widehat{\rho}) \models_s t_2^{\ell_2}$

Table 3.5: Syntax directed Control Flow Analysis.

since we have $\widehat{\mathsf{C}}_{\mathsf{lp}}(5) \subseteq \widehat{\rho}_{\mathsf{lp}}(\mathsf{g})$ and $\widehat{\mathsf{C}}_{\mathsf{lp}}(9) \subseteq \widehat{\mathsf{C}}_{\mathsf{lp}}(10)$. To show (3.2) we use the clause $[fun]$ and it is sufficient to show

$$(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models_s (\mathtt{f}^1\ (\texttt{fn}\ \mathtt{y} \texttt{ => } \mathtt{y}^2)^3)^4$$

since $\mathsf{f} \in \widehat{\mathsf{C}}_{\mathsf{lp}}(5)$ and $\mathsf{f} \in \widehat{\rho}_{\mathsf{lp}}(\mathtt{f})$. Now $\widehat{\mathsf{C}}_{\mathsf{lp}}(1) = \{\mathsf{f}\}$ so, according to clause $[app]$ this follows from

$$(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models_s \mathtt{f}^1$$
$$(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models_s (\texttt{fn}\ \mathtt{y} \texttt{ => } \mathtt{y}^2)^3$$

since $\widehat{\mathsf{C}}_{\mathsf{lp}}(3) \subseteq \widehat{\rho}_{\mathsf{lp}}(\mathtt{x})$ and $\widehat{\mathsf{C}}_{\mathsf{lp}}(4) \subseteq \widehat{\mathsf{C}}_{\mathsf{lp}}(4)$. The first clause follows from [*var*] since $\widehat{\rho}_{\mathsf{lp}}(\mathtt{f}) \subseteq \widehat{\mathsf{C}}_{\mathsf{lp}}(1)$ and for the last clause we observe that $\mathsf{id}_y \in \widehat{\mathsf{C}}_{\mathsf{lp}}(3)$ and $(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models_s \mathtt{y}^2$ as follows from $\widehat{\rho}_{\mathsf{lp}}(\mathtt{y}) \subseteq \widehat{\mathsf{C}}_{\mathsf{lp}}(2)$.

To show (3.3) we observe that $\widehat{\mathsf{C}}_{\mathsf{lp}}(6) = \{\mathtt{f}\}$ so using [*app*] it is sufficient to show

$$(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models_s \mathtt{g}^6$$
$$(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models_s (\mathtt{fn\ z=>\ z}^7)^8$$

since $\widehat{\mathsf{C}}_{\mathsf{lp}}(8) \subseteq \widehat{\rho}_{\mathsf{lp}}(\mathtt{x})$ and $\widehat{\mathsf{C}}_{\mathsf{lp}}(4) \subseteq \widehat{\mathsf{C}}_{\mathsf{lp}}(9)$. This is straightforward except for the last clause where we observe that $\mathsf{id}_z \in \widehat{\mathsf{C}}_{\mathsf{lp}}(8)$ and $(\widehat{\mathsf{C}}_{\mathsf{lp}}, \widehat{\rho}_{\mathsf{lp}}) \models_s \mathtt{z}^7$ as follows from $\widehat{\rho}_{\mathsf{lp}}(\mathtt{z}) \subseteq \widehat{\mathsf{C}}_{\mathsf{lp}}(7)$.

Note that because the analysis is syntax directed we have had *no need for coinduction*, unlike what was the case in Example 3.6. ∎

3.3.2 Preservation of Solutions

The specification of the analysis in Table 3.5 uses potentially infinite value spaces although this is not really necessary (as Exercise 3.2 demonstrates for Table 3.1). We can easily restrict ourselves to entities occurring in the original expression and this forms the basis for relating the results of the analysis of Table 3.5 to those of the analysis of Table 3.1.

So let $\mathbf{Lab}_\star \subseteq \mathbf{Lab}$ be the finite set of labels occurring in the program $e_\star$ of interest, let $\mathbf{Var}_\star \subseteq \mathbf{Var}$ be the finite set of variables occurring in $e_\star$ and let $\mathbf{Term}_\star$ be the finite set of subterms occurring in $e_\star$. Define $(\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top)$ by:

$$\widehat{\mathsf{C}}_\star^\top(\ell) = \begin{cases} \emptyset & \text{if } \ell \notin \mathbf{Lab}_\star \\ \mathbf{Term}_\star & \text{if } \ell \in \mathbf{Lab}_\star \end{cases}$$

$$\widehat{\rho}_\star^\top(x) = \begin{cases} \emptyset & \text{if } x \notin \mathbf{Var}_\star \\ \mathbf{Term}_\star & \text{if } x \in \mathbf{Var}_\star \end{cases}$$

Then the claim

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \sqsubseteq (\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top)$$

intuitively expresses that $(\widehat{\mathsf{C}}, \widehat{\rho})$ is concerned only with subterms occurring in the expression $e_\star$; obviously, we are primarily interested in analysis results with that property. Actually, this condition can be "reformulated" as the technically more manageable

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \in \widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Env}}_\star$$

where we define $\widehat{\mathbf{Cache}}_\star = \mathbf{Lab}_\star \to \widehat{\mathbf{Val}}_\star$, $\widehat{\mathbf{Env}}_\star = \mathbf{Var}_\star \to \widehat{\mathbf{Val}}_\star$ and $\widehat{\mathbf{Val}}_\star = \mathcal{P}(\mathbf{Term}_\star)$.

We can now show that all the solutions to "$\models_s e_\star$" that are "less than" $(\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top)$ are solutions to "$\models e_\star$" as well:

Proposition 3.18
If $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star$ and $(\widehat{\mathsf{C}}, \widehat{\rho}) \sqsubseteq (\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top)$ then $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$.

Proof Assume that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star$ and that $(\widehat{\mathsf{C}}, \widehat{\rho}) \sqsubseteq (\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top)$. Furthermore let $\mathbf{Exp}_\star$ be the set of expressions occurring in $e_\star$ and note that

$$\forall e \in \mathbf{Exp}_\star : (\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e \tag{3.4}$$

is an immediate consequence of the syntax directed nature of the definition of $\models_s$.

To show that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$ we proceed by coinduction. We know that "$\models$" is defined coinductively by the specification of Table 3.1, i.e. "$\models = gfp(\mathcal{Q})$" where $\mathcal{Q}$ is the function (implicitly) defined by Table 3.1. Similarly, we know that "$\models_s = gfp(\mathcal{Q}_s)$" where $\mathcal{Q}_s$ is the function (implicitly) defined by Table 3.5.

Next write $(\widehat{\mathsf{C}}', \widehat{\rho}') \models^\star e'$ for $(\widehat{\mathsf{C}}', \widehat{\rho}') = (\widehat{\mathsf{C}}, \widehat{\rho}) \wedge e' \in \mathbf{Exp}_\star$. It now suffices to show

$$(\mathcal{Q}_s(\models_s) \cap \models^\star) \subseteq \mathcal{Q}(\models_s \cap \models^\star) \tag{3.5}$$

because then "$(\models_s \cap \models^\star) \subseteq \mathcal{Q}(\models_s \cap \models^\star)$" follows and hence by coinduction "$(\models_s \cap \models^\star) \subseteq \models$" and since $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star$ as well as $(\widehat{\mathsf{C}}, \widehat{\rho}) \models^\star e_\star$ we then have the required $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$.

The proof of (3.5) amounts to a comparison of the right hand sides of Table 3.5 and Table 3.1: for each clause we shall assume that the right hand side of Table 3.5 holds for $(\widehat{\mathsf{C}}, \widehat{\rho}, e)$ and that $e \in \mathbf{Exp}_\star$ and we shall show that the corresponding right hand side of Table 3.1 holds when all occurrences of "$\models$" are replaced by "$\models_s \cap \models^\star$".

The clauses [*con*], [*var*], [*if*], [*let*] and [*op*] are trivial as the right hand sides of Tables 3.5 and 3.1 are similar and the subterms will all be in $\mathbf{Exp}_\star$. The clauses [*fn*] and [*fun*] are straightforward as the right hand sides of Table 3.5 imply the right-hand sides of Table 3.1. Finally, we consider the clause [*app*]. For $(\texttt{fn}\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\mathsf{C}}(\ell_1)$ we need to show that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s t_0^{\ell_0}$; but since $(\widehat{\mathsf{C}}, \widehat{\rho}) \sqsubseteq (\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top)$ this follows from (3.4). For $(\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\mathsf{C}}(\ell_1)$ we need to show that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s t_0^{\ell_0}$ and that $(\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\rho}(f)$; the first follows from (3.4) and the second is an immediate consequence of $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s (\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0})^\ell$ (for some ℓ) that again follows from (3.4). ∎

We show an analogue of Proposition 3.13 for the syntax directed analysis:

Proposition 3.19
$\{(\widehat{\mathsf{C}}, \widehat{\rho}) \in \widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Env}}_\star \mid (\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star\}$ is a *Moore family*.

This result has as immediate corollaries that:

- each expression $e_\star$ has a Control Flow Analysis that is "less than" $(\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top)$, and
- each expression $e_\star$ has a "least" Control Flow Analysis that is "less than" $(\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top)$.

This means that the properties obtained for the analysis of Table 3.1 in Subsection 3.2.3 also hold for the analysis of Table 3.5 with the additional restriction on the range of the analysis functions. In particular, any analysis result that is acceptable with respect to Table 3.5 (and properly restricted to $\widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Env}}_\star$) is also an acceptable analysis result with respect to Table 3.1. The converse relationship is studied in Exercise 3.11 and Mini Project 3.1.

Proof We shall write $(\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top)$ also for the greatest element of $\widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Env}}_\star$. It is immediate to show that

(a) $(\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top) \models_s e$

(b) if $(\widehat{\mathsf{C}}_1, \widehat{\rho}_1) \models_s e$ and $(\widehat{\mathsf{C}}_2, \widehat{\rho}_2) \models_s e$ then $((\widehat{\mathsf{C}}_1, \widehat{\rho}_1) \sqcap (\widehat{\mathsf{C}}_2, \widehat{\rho}_2)) \models_s e$.

for all subexpressions e of $e_\star$ by means of structural induction on e. This establishes (a) and (b) also for $e = e_\star$. Next consider some

$$Y \subseteq \{(\widehat{\mathsf{C}}, \widehat{\rho}) \in \widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Env}}_\star \mid (\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star\}$$

and note than one can write $Y = \{(\widehat{\mathsf{C}}_i, \widehat{\rho}_i) \mid i \in \{1, \cdots, n\}\}$ for some $n \geq 0$ since $\widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Env}}_\star$ is finite. That

$$\bigsqcap Y \in \{(\widehat{\mathsf{C}}, \widehat{\rho}) \in \widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Env}}_\star \mid (\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star\}$$

then follows from (a) and (b) because $\bigsqcap Y = (\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top) \sqcap (\widehat{\mathsf{C}}_1, \widehat{\rho}_1) \sqcap \cdots \sqcap (\widehat{\mathsf{C}}_n, \widehat{\rho}_n)$. ∎

3.4 Constraint Based 0-CFA Analysis

We are now ready to consider efficient ways of finding the least solution $(\widehat{\mathsf{C}}, \widehat{\rho})$ such that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star$. To do so we first construct a finite set $\mathcal{C}_\star[\![e_\star]\!]$ of *constraints* and *conditional constraints* of the form

$$lhs \subseteq rhs \tag{3.6}$$

$$\{t\} \subseteq rhs' \Rightarrow lhs \subseteq rhs \tag{3.7}$$

where *rhs* is of the form $\mathsf{C}(\ell)$ or $\mathsf{r}(x)$, and *lhs* is of the form $\mathsf{C}(\ell), \mathsf{r}(x)$, or $\{t\}$, and all occurrences of t are of the form `fn` x `=>` e_0 or `fun` f x `=>` e_0. To simplify the technical development we shall read (3.7) as

$$(\{t\} \subseteq rhs' \Rightarrow lhs) \subseteq rhs$$

and we shall write *ls* for *lhs* as well as $\{t\} \subseteq rhs' \Rightarrow lhs$.

$$
\begin{array}{lll}
[con] & \mathcal{C}_\star[\![c^\ell]\!] = \emptyset & \\
[var] & \mathcal{C}_\star[\![x^\ell]\!] = \{\mathsf{r}(x) \subseteq \mathsf{C}(\ell)\} & \\
[fn] & \mathcal{C}_\star[\![(\texttt{fn}\ x \texttt{ => } e_0)^\ell]\!] & = \{\{\texttt{fn}\ x \texttt{ => } e_0\} \subseteq \mathsf{C}(\ell)\} \\
 & & \cup\ \mathcal{C}_\star[\![e_0]\!] \\
[fun] & \mathcal{C}_\star[\![(\texttt{fun}\ f\ x \texttt{ => } e_0)^\ell]\!] & = \{\{\texttt{fun}\ f\ x \texttt{ => } e_0\} \subseteq \mathsf{C}(\ell)\} \\
 & & \cup\ \mathcal{C}_\star[\![e_0]\!]\ \cup\ \{\{\texttt{fun}\ f\ x \texttt{ => } e_0\} \subseteq \mathsf{r}(f)\} \\
[app] & \mathcal{C}_\star[\![(t_1^{\ell_1}\ t_2^{\ell_2})^\ell]\!] & = \mathcal{C}_\star[\![t_1^{\ell_1}]\!] \cup \mathcal{C}_\star[\![t_2^{\ell_2}]\!] \\
 & & \cup\ \{\{t\} \subseteq \mathsf{C}(\ell_1) \Rightarrow \mathsf{C}(\ell_2) \subseteq \mathsf{r}(x) \\
 & & \qquad \mid t = (\texttt{fn}\ x \texttt{ => } t_0^{\ell_0}) \in \mathbf{Term}_\star\} \\
 & & \cup\ \{\{t\} \subseteq \mathsf{C}(\ell_1) \Rightarrow \mathsf{C}(\ell_0) \subseteq \mathsf{C}(\ell) \\
 & & \qquad \mid t = (\texttt{fn}\ x \texttt{ => } t_0^{\ell_0}) \in \mathbf{Term}_\star\} \\
 & & \cup\ \{\{t\} \subseteq \mathsf{C}(\ell_1) \Rightarrow \mathsf{C}(\ell_2) \subseteq \mathsf{r}(x) \\
 & & \qquad \mid t = (\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}) \in \mathbf{Term}_\star\} \\
 & & \cup\ \{\{t\} \subseteq \mathsf{C}(\ell_1) \Rightarrow \mathsf{C}(\ell_0) \subseteq \mathsf{C}(\ell) \\
 & & \qquad \mid t = (\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}) \in \mathbf{Term}_\star\} \\
[if] & \mathcal{C}_\star[\![(\texttt{if}\ t_0^{\ell_0}\ \texttt{then}\ t_1^{\ell_1}\ \texttt{else}\ t_2^{\ell_2})^\ell]\!] & = \mathcal{C}_\star[\![t_0^{\ell_0}]\!] \cup \mathcal{C}_\star[\![t_1^{\ell_1}]\!] \cup \mathcal{C}_\star[\![t_2^{\ell_2}]\!] \\
 & & \cup\ \{\mathsf{C}(\ell_1) \subseteq \mathsf{C}(\ell)\} \\
 & & \cup \{\mathsf{C}(\ell_2) \subseteq \mathsf{C}(\ell)\} \\
[let] & \mathcal{C}_\star[\![(\texttt{let}\ x = t_1^{\ell_1}\ \texttt{in}\ t_2^{\ell_2})^\ell]\!] & = \mathcal{C}_\star[\![t_1^{\ell_1}]\!] \cup \mathcal{C}_\star[\![t_2^{\ell_2}]\!] \\
 & & \cup\ \{\mathsf{C}(\ell_1) \subseteq \mathsf{r}(x)\} \cup \{\mathsf{C}(\ell_2) \subseteq \mathsf{C}(\ell)\} \\
[op] & \mathcal{C}_\star[\![(t_1^{\ell_1}\ op\ t_2^{\ell_2})^\ell]\!] & = \mathcal{C}_\star[\![t_1^{\ell_1}]\!] \cup \mathcal{C}_\star[\![t_2^{\ell_2}]\!]
\end{array}
$$

Table 3.6: Constraint based Control Flow Analysis.

Informally, the constraints are obtained by expanding the clauses defining $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star$ into a finite set of constraints of the above form and then letting $\mathcal{C}_\star[\![e_\star]\!]$ be the set of individual conjuncts. One caveat is that all occurrences of "$\widehat{\mathsf{C}}$" are changed into "C" and that all occurrences of "$\widehat{\rho}$" are changed into "r" to avoid confusion: $\widehat{\mathsf{C}}(\ell)$ will be a *set of terms* whereas $\mathsf{C}(\ell)$ is *pure syntax* and similarly for $\widehat{\rho}(x)$ and $\mathsf{r}(x)$.

Formally, the *constraint based 0-CFA* analysis is defined by the function $\mathcal{C}_\star$ of Table 3.6: it makes use of the set $\mathbf{Term}_\star$ of subterms occurring in the expression $e_\star$ in order to generate only a finite number of constraints in the clause for application; this is justified by Propositions 3.18 and 3.19.

If the size of the expression $e_\star$ is n then it might seem that there could be $O(n^2)$ constraints of the form (3.6) and $O(n^4)$ constraints of the form (3.7).

However, inspection of the definition of $\mathcal{C}_\star$ ensures that at most $O(n)$ constraints of the form (3.6) and $O(n^2)$ constraints of the form (3.7) are ever generated: each of the $O(n)$ constituents only generate $O(1)$ constraints of the form (3.6) and $O(n)$ constraints of the form (3.7).

Example 3.20 Consider the expression

$$((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5$$

of Example 3.7. We generate the following set of constraints

$$\begin{array}{l}
\mathcal{C}_\star[\![((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5]\!] = \\
\quad \{\ \{\texttt{fn x => x}^1\} \subseteq \mathsf{C}(2), \\
\quad\ \ \mathsf{r}(\mathbf{x}) \subseteq \mathsf{C}(1), \\
\quad\ \ \{\texttt{fn y => y}^3\} \subseteq \mathsf{C}(4), \\
\quad\ \ \mathsf{r}(\mathbf{y}) \subseteq \mathsf{C}(3), \\
\quad\ \ \{\texttt{fn x => x}^1\} \subseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(4) \subseteq \mathsf{r}(\mathbf{x}), \\
\quad\ \ \{\texttt{fn x => x}^1\} \subseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(1) \subseteq \mathsf{C}(5), \\
\quad\ \ \{\texttt{fn y => y}^3\} \subseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(4) \subseteq \mathsf{r}(\mathbf{y}), \\
\quad\ \ \{\texttt{fn y => y}^3\} \subseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(3) \subseteq \mathsf{C}(5)\ \}
\end{array}$$

where we use that $\texttt{fn x => x}^1$ and $\texttt{fn y => y}^3$ are the only abstractions in $\mathbf{Term}_\star$. ∎

3.4.1 Preservation of Solutions

It is important to stress that while $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star$ is a logical formula, $\mathcal{C}_\star[\![e_\star]\!]$ is a set of syntactic entities. To give meaning to the syntax we first translate the "C" and "r" symbols into the sets "$\widehat{\mathsf{C}}$" and "$\widehat{\rho}$":

$$\begin{array}{rcl}
(\widehat{\mathsf{C}}, \widehat{\rho})[\![\mathsf{C}(\ell)]\!] & = & \widehat{\mathsf{C}}(\ell) \\
(\widehat{\mathsf{C}}, \widehat{\rho})[\![\mathsf{r}(x)]\!] & = & \widehat{\rho}(x)
\end{array}$$

To deal with the possible forms of *ls* we additionally take:

$$\begin{array}{rcl}
(\widehat{\mathsf{C}}, \widehat{\rho})[\![\{t\}]\!] & = & \{t\} \\
(\widehat{\mathsf{C}}, \widehat{\rho})[\![\{t\} \subseteq rhs' \Rightarrow lhs]\!] & = & \begin{cases} (\widehat{\mathsf{C}}, \widehat{\rho})[\![lhs]\!] & \text{if } \{t\} \subseteq (\widehat{\mathsf{C}}, \widehat{\rho})[\![rhs']\!] \\ \emptyset & \text{otherwise} \end{cases}
\end{array}$$

Next we define a satisfaction relation $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c (ls \subseteq rhs)$ on the individual constraints:

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c (ls \subseteq rhs) \quad \text{iff} \quad (\widehat{\mathsf{C}}, \widehat{\rho})[\![ls]\!] \subseteq (\widehat{\mathsf{C}}, \widehat{\rho})[\![rhs]\!]$$

This definition can be lifted to a set C of constraints by:

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c C \text{ iff } \forall (ls \subseteq rhs) \in C : (\widehat{\mathsf{C}}, \widehat{\rho}) \models_c (ls \subseteq rhs)$$

We then have the following result showing that all solutions to the set $\mathcal{C}_\star[\![e_\star]\!]$ of constraints also satisfy the syntax directed specification of the Control Flow Analysis and vice versa:

Proposition 3.21
If $(\widehat{\mathsf{C}}, \widehat{\rho}) \sqsubseteq (\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top)$ then

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star \text{ if and only if } (\widehat{\mathsf{C}}, \widehat{\rho}) \models_c \mathcal{C}_\star[\![e_\star]\!]$$

Thus the least solution $(\widehat{\mathsf{C}}, \widehat{\rho})$ to $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star$ equals the least solution to $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c \mathcal{C}_\star[\![e_\star]\!]$.

Proof A simple structural induction on e shows that

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e \text{ iff } (\widehat{\mathsf{C}}, \widehat{\rho}) \models_c \mathcal{C}_\star[\![e]\!]$$

for all subexpressions e of $e_\star$; in the case of the function application $(t_1^{\ell_1}\ t_2^{\ell_2})^\ell$ the assumption $(\widehat{\mathsf{C}}, \widehat{\rho}) \sqsubseteq (\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top)$ is used to ensure that $\widehat{\mathsf{C}}(\ell_1) \subseteq \mathbf{Term}_\star$. ∎

3.4.2 Solving the Constraints

We shall present two approaches to solving the set of constraints $\mathcal{C}_\star[\![e_\star]\!]$. First we shall show that finding the least solution to $\mathcal{C}_\star[\![e_\star]\!]$ is equivalent to finding the least fixed point of a certain function; straightforward techniques allow us to compute the least fixed point in time $O(n^5)$ when the size of the expression $e_\star$ is n. Improvements upon this are possible, but to obtain the best known result we shall consider a graph representation of the problem; this will give us a $O(n^3)$ algorithm. This is indeed a *common phenomenon* in program analysis: syntax directed specifications are appropriate for correctness considerations but often they need to be "massaged" in order to obtain efficient implementations.

Fixed point formulation. To show that finding the solution of the set $\mathcal{C}_\star[\![e_\star]\!]$ of constraints is a *fixed point problem* we shall define a function

$$F_\star : \widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Env}}_\star \to \widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Env}}_\star$$

and show that it has a least fixed point $lfp(F_\star)$ that is indeed the least solution whose existence is guaranteed by Propositions 3.18 and 3.21.

We define the function $F_\star$ by

$$F_\star(\widehat{\mathsf{C}},\widehat{\rho}) = (F_1(\widehat{\mathsf{C}},\widehat{\rho}), F_2(\widehat{\mathsf{C}},\widehat{\rho}))$$

where:

$$\begin{aligned} F_1(\widehat{\mathsf{C}},\widehat{\rho})(\ell) &= \bigcup\{(\widehat{\mathsf{C}},\widehat{\rho})[\![ls]\!] \mid (ls \subseteq \mathsf{C}(\ell)) \in \mathcal{C}_\star[\![e_\star]\!]\} \\ F_2(\widehat{\mathsf{C}},\widehat{\rho})(x) &= \bigcup\{(\widehat{\mathsf{C}},\widehat{\rho})[\![ls]\!] \mid (ls \subseteq \mathsf{r}(x)) \in \mathcal{C}_\star[\![e_\star]\!]\} \end{aligned}$$

To see that this defines a monotone function it suffices to consider a constraint

$$lhs' \subseteq rhs' \;\Rightarrow\; lhs \subseteq rhs$$

in $\mathcal{C}_\star[\![e_\star]\!]$ and to observe that lhs' is of the form $\{t\}$; this ensures that $(\widehat{\mathsf{C}}_1,\widehat{\rho}_1) \sqsubseteq (\widehat{\mathsf{C}}_2,\widehat{\rho}_2)$ implies $F_i(\widehat{\mathsf{C}}_1,\widehat{\rho}_1) \sqsubseteq F_i(\widehat{\mathsf{C}}_2,\widehat{\rho}_2)$ (for $i = 1,2$) because if $\{t\} \subseteq (\widehat{\mathsf{C}}_1,\widehat{\rho}_1)[\![rhs']\!]$ then also $\{t\} \subseteq (\widehat{\mathsf{C}}_2,\widehat{\rho}_2)[\![rhs']\!]$. Since $\widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Env}}_\star$ is a complete lattice this means that $F_\star$ has a least fixed point and it turns out also to be the least solution to the set $\mathcal{C}_\star[\![e_\star]\!]$ of constraints:

Proposition 3.22
$lfp(F_\star) = \bigsqcap\{(\widehat{\mathsf{C}},\widehat{\rho}) \mid (\widehat{\mathsf{C}},\widehat{\rho}) \models_c \mathcal{C}_\star[\![e_\star]\!]\}$

Proof It is easy to verify that:

$$F_\star(\widehat{\mathsf{C}},\widehat{\rho}) \sqsubseteq (\widehat{\mathsf{C}},\widehat{\rho}) \text{ iff } (\widehat{\mathsf{C}},\widehat{\rho}) \models_c \mathcal{C}_\star[\![e_\star]\!]$$

Using the formula $lfp(f) = \bigsqcap\{x \mid f(x) \sqsubseteq x\}$ (see Appendix A) the result then follows. ∎

If the size of $e_\star$ is n then an element $(\widehat{\mathsf{C}},\widehat{\rho})$ of $\widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Env}}_\star$ may be viewed as an $O(n)$-tuple of values from $\widehat{\mathbf{Val}}_\star$. Since $\widehat{\mathbf{Val}}_\star$ is a lattice of height $O(n)$ this means that $\widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Val}}_\star$ has height $O(n^2)$ and hence the formula

$$lfp(F_\star) = \bigsqcup_m F_\star^m(\bot)$$

may be used to compute the least fixed point in at most $O(n^2)$ iterations. A naive approach will need to consider all $O(n^2)$ constraints to determine the value of each of the $O(n)$ components of the new iterant; this yields an overall $O(n^5)$ bound on the cost.

Graph formulation. An alternative method for computing the least solution to the set $\mathcal{C}_\star[\![e_\star]\!]$ of constraints is to use a *graph formulation of the constraints.* The graph will have nodes $\mathsf{C}(\ell)$ and $\mathsf{r}(x)$ for $\ell \in \mathbf{Lab}_\star$ and $x \in \mathbf{Var}_\star$. Associated with each node p we have a data field $\mathsf{D}[\mathsf{p}]$ that initially is given by:

$$\mathsf{D}[\mathsf{p}] = \{t \mid (\{t\} \subseteq p) \in \mathcal{C}_\star[\![e_\star]\!]\}$$

The graph will have edges for a subset of the constraints in $\mathcal{C}_\star[\![e_\star]\!]$; each edge will be decorated with the constraint that gives rise to it:

```
INPUT:   C*[e*]

OUTPUT:  (Ĉ, ρ̂)

METHOD:  Step 1: Initialisation
                 W := nil;
                 for q in Nodes do D[q] := ∅;
                 for q in Nodes do E[q] := nil;

         Step 2: Building the graph
                 for cc in C*[e*] do
                     case cc of
                         {t} ⊆ p: add(p,{t});
                         p1 ⊆ p2: E[p1] := cons(cc,E[p1]);
                         {t} ⊆ p ⇒ p1 ⊆ p2:
                                 E[p1] := cons(cc,E[p1]);
                                 E[p] := cons(cc,E[p]);

         Step 3: Iteration
                 while W ≠ nil do
                     q := head(W); W := tail(W);
                     for cc in E[q] do
                         case cc of
                             p1 ⊆ p2: add(p2, D[p1]);
                             {t} ⊆ p ⇒ p1 ⊆ p2:
                                 if t ∈ D[p] then add(p2, D[p1]);

         Step 4: Recording the solution
                 for ℓ in Lab* do Ĉ(ℓ) := D[C(ℓ)];
                 for x in Var* do ρ̂(x) := D[r(x)];

USING:   procedure add(q,d) is
             if ¬ (d ⊆ D[q])
             then  D[q] := D[q] ∪ d;
                   W := cons(q,W);
```

Table 3.7: Algorithm for solving constraints.

- a constraint $p_1 \subseteq p_2$ gives rise to an edge from p_1 to p_2, and
- a constraint $\{t\} \subseteq p \Rightarrow p_1 \subseteq p_2$ gives rise to an edge from p_1 to p_2 and an edge from p to p_2.

Having constructed the graph we now traverse all edges in order to propagate information from one data field to another. We make certain only to traverse

p	$\mathsf{D}[p]$	$\mathsf{E}[p]$
$\mathsf{C}(1)$	$\emptyset$	$[\mathsf{id}_x \subseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(1) \subseteq \mathsf{C}(5)]$
$\mathsf{C}(2)$	id_x	$[\mathsf{id}_y \subseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(3) \subseteq \mathsf{C}(5),\ \mathsf{id}_y \subseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(4) \subseteq \mathsf{r}(\mathsf{y}),$ $\mathsf{id}_x \subseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(1) \subseteq \mathsf{C}(5),\ \mathsf{id}_x \subseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(4) \subseteq \mathsf{r}(\mathsf{x})]$
$\mathsf{C}(3)$	$\emptyset$	$[\mathsf{id}_y \subseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(3) \subseteq \mathsf{C}(5)]$
$\mathsf{C}(4)$	id_y	$[\mathsf{id}_y \subseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(4) \subseteq \mathsf{r}(\mathsf{y}),\ \mathsf{id}_x \subseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(4) \subseteq \mathsf{r}(\mathsf{x})]$
$\mathsf{C}(5)$	$\emptyset$	$[\,]$
$\mathsf{r}(\mathsf{x})$	$\emptyset$	$[\mathsf{r}(\mathsf{x}) \subseteq \mathsf{C}(1)]$
$\mathsf{r}(\mathsf{y})$	$\emptyset$	$[\mathsf{r}(\mathsf{y}) \subseteq \mathsf{C}(3)]$

Figure 3.4: Initialisation of data structures for example program.

an edge from p_1 to p_2 when $\mathsf{D}[p_1]$ is extended with a term not previously there (and this incorporates the situation where $\mathsf{D}[p_1]$ is initially set to a non-empty set). Furthermore, an edge decorated with $\{t\} \subseteq p \Rightarrow p_1 \subseteq p_2$ is only traversed if in fact $t \in \mathsf{D}[\mathsf{p}]$.

To be more specific consider the algorithm of Table 3.7. It takes as *input* a set $\mathcal{C}_\star[\![e_\star]\!]$ of constraints and produces as *output* a solution $(\widehat{\mathsf{C}}, \widehat{\rho}) \in \widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Env}}_\star$. It operates on the following main *data structures*:

- a *worklist* W, i.e. a list of nodes whose outgoing edges should be traversed;
- a *data array* D that for each node gives an element of $\widehat{\mathbf{Val}}_\star$; and
- an *edge array* E that for each node gives a list of constraints from which a list of the successor nodes can be computed.

The set Nodes consists of $\mathsf{C}(\ell)$ for all ℓ in $\mathbf{Lab}_\star$ and $\mathsf{r}(x)$ for all x in $\mathbf{Var}_\star$.

The first step of the algorithm is to initialise the data structures. The second step is to build the graph and to perform the initial assignments to the data fields. This is established using the procedure $\mathsf{add}(q,d)$ that incorporates d into $\mathsf{D}[q]$ and adds q to the worklist if d was not part of $\mathsf{D}[q]$. The third step is to continue propagating contributions along edges as long as the worklist is non-empty. The fourth and final step is to record the solution in a more familiar form.

Example 3.23 Let us consider how the algorithm operates on the expression $((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5$ of Example 3.20. After step 2 the data structure W has been initialised to

$$\mathsf{W} = [\mathsf{C}(4), \mathsf{C}(2)],$$

W	[C(4),C(2)]	[r(x),C(2)]	[C(1),C(2)]	[C(5),C(2)]	[C(2)]	[ ]
p	D[p]	D[p]	D[p]	D[p]	D[p]	D[p]
C(1)	$\emptyset$	$\emptyset$	id_y	id_y	id_y	id_y
C(2)	id_x	id_x	id_x	id_x	id_x	id_x
C(3)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
C(4)	id_y	id_y	id_y	id_y	id_y	id_y
C(5)	$\emptyset$	$\emptyset$	$\emptyset$	id_y	id_y	id_y
r(x)	$\emptyset$	id_y	id_y	id_y	id_y	id_y
r(y)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Figure 3.5: Iteration steps of example program.

and the data structures D and E have been initialised as in Figure 3.4 where we have written id_x for $\{\texttt{fn x => x}^1\}$ and id_y for $\{\texttt{fn y => y}^3\}$. The algorithm will now iterate through the worklist and update the data structures W and D as described by step 3. The various intermediate stages are recorded in Figure 3.5. The algorithm computes the solution in the last column and this agrees with the solution presented in Example 3.5. ∎

The following result shows that the algorithm of Table 3.7 does indeed compute the solution we want:

Proposition 3.24
Given input $\mathcal{C}_\star[\![e_\star]\!]$ the algorithm of Table 3.7 terminates and the result $(\widehat{\mathsf{C}}, \widehat{\rho})$ produced by the algorithm satisfies

$$(\widehat{\mathsf{C}}, \widehat{\rho}) = \bigsqcap \{(\widehat{\mathsf{C}}', \widehat{\rho}') \mid (\widehat{\mathsf{C}}', \widehat{\rho}') \models_c \mathcal{C}_\star[\![e_\star]\!]\}$$

and hence it is the least solution to $\mathcal{C}_\star[\![e_\star]\!]$.

Proof It is immediate that steps 1, 2 and 4 terminate, and this leaves us with step 3. It is immediate that the values of D[q] never decrease and that they can be increased at most a finite number of times. It is also immediate that a node q is added to the worklist only if some value of D[q] actually increased. For each node placed on the worklist only a finite amount of calculation (bounded by the number of outgoing edges) needs to be performed in order to remove the node from the worklist. This guarantees termination.

Next let $(\widehat{\mathsf{C}}', \widehat{\rho}')$ be a solution to $(\widehat{\mathsf{C}}', \widehat{\rho}') \models_c \mathcal{C}_\star[\![e_\star]\!]$. It is possible to show that the following invariant

$$\forall \ell \in \mathbf{Lab}_\star : \mathsf{D}[\mathsf{C}(\ell)] \subseteq \widehat{\mathsf{C}}'(\ell)$$
$$\forall x \in \mathbf{Var}_\star : \mathsf{D}[\mathsf{r}(x)] \subseteq \widehat{\rho}'(x)$$

is maintained at all points after step 1. It follows that $(\widehat{\mathsf{C}}, \widehat{\rho}) \sqsubseteq (\widehat{\mathsf{C}}', \widehat{\rho}')$ upon completion of the algorithm.

We prove that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c \mathcal{C}_\star[\![e_\star]\!]$ by contradiction. So suppose there exists $cc \in \mathcal{C}_\star[\![e_\star]\!]$ such that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c cc$ does *not* hold. If cc is $\{t\} \subseteq p$ then step 2 ensures that $\{t\} \subseteq \mathsf{D}[p]$ and this is maintained throughout the algorithm; hence cc cannot have this form. If cc is $p_1 \subseteq p_2$ it must be the case that the final value of D satisfies $\mathsf{D}[p_1] \neq \emptyset$ since otherwise $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c cc$ would hold; now consider the last time $\mathsf{D}[p_1]$ was modified and note that p_1 was placed on the worklist at that time (by the procedure add); since the final worklist is empty we must have considered the constraint cc (which is in $\mathsf{E}[p_1]$) and updated $\mathsf{D}[p_2]$ accordingly; hence cc cannot have this form. If cc is $\{t\} \subseteq p \Rightarrow p_1 \subseteq p_2$ it must be the case that the final value of D satisfies $\mathsf{D}[p] \neq \emptyset$ as well as $\mathsf{D}[p_1] \neq \emptyset$; now consider the last time one of $\mathsf{D}[p]$ and $\mathsf{D}[p_1]$ was modified and note that p or p_1 was placed on the worklist at that time; since the final worklist is empty we must have considered the constraint cc and updated $\mathsf{D}[p_2]$ accordingly; hence cc cannot have this form. Thus $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c cc$ for all $cc \in \mathcal{C}_\star[\![e_\star]\!]$.

We have now shown that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c \mathcal{C}_\star[\![e_\star]\!]$ and that $(\widehat{\mathsf{C}}, \widehat{\rho}) \sqsubseteq (\widehat{\mathsf{C}}', \widehat{\rho}')$ whenever $(\widehat{\mathsf{C}}', \widehat{\rho}') \models_c \mathcal{C}_\star[\![e_\star]\!]$. It now follows that

$$(\widehat{\mathsf{C}}, \widehat{\rho}) = \bigsqcap \{(\widehat{\mathsf{C}}', \widehat{\rho}') \mid (\widehat{\mathsf{C}}', \widehat{\rho}') \models_c \mathcal{C}_\star[\![e_\star]\!]\}$$

as required. ∎

The proof showing that the algorithm terminates can be refined to show that it takes at most $O(n^3)$ steps if the original expression $e_\star$ has size n. To see this recall that $\mathcal{C}_\star[\![e_\star]\!]$ contains at most $O(n)$ constraints of the form $\{t\} \subseteq p$ or $p_1 \subseteq p_2$, and at most $O(n^2)$ constraints of the form $\{t\} \subseteq p \Rightarrow p_1 \subseteq p_2$. We therefore know that the graph has at most $O(n)$ nodes and $O(n^2)$ edges and that each data field can be enlarged at most $O(n)$ times. Assuming that the operations upon $\mathsf{D}[p]$ take unit time we can perform the following calculations: step 1 takes time $O(n)$, step 2 takes time $O(n^2)$, and step 4 takes time $O(n)$; step 3 traverses each of the $O(n^2)$ edges at most $O(n)$ times and hence takes time $O(n^3)$; it follows that the overall algorithm takes no more than $O(n^3)$ basic steps.

Combining the three phases. From Proposition 3.24 we get that the pair $(\widehat{\mathsf{C}}, \widehat{\rho})$ computed by the algorithm of Table 3.7 is the least solution to $\mathcal{C}[\![e_\star]\!]$, so in particular $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c \mathcal{C}[\![e_\star]\!]$. Proposition 3.21 shows that a solution to the constraints will also be an acceptable analysis result for the syntax directed specification, hence $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star$. Proposition 3.18 shows that a solution that only involves program fragments of $e_\star$ and that is acceptable for the syntax directed specification, also is acceptable for the abstract specification, and therefore $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$. Thus we have the following important corollary:

Corollary 3.25 Assume that $(\widehat{\mathsf{C}}, \widehat{\rho})$ is the solution to the constraints $\mathcal{C}[\![e_\star]\!]$ computed by the algorithm of Table 3.7; then $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$. ∎

It is not the case that any $(\widehat{\mathsf{C}}, \widehat{\rho})$ satisfying $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$ can be obtained using the above approach – see Exercise 3.11 and Mini Project 3.1.

For many applications it is the ability to compute the least $(\widehat{\mathsf{C}}, \widehat{\rho})$ satisfying $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$ that is of primary interest, rather than the ability to check $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$ for a proposed guess $(\widehat{\mathsf{C}}, \widehat{\rho})$. However, there are applications where it is the ability to check $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$ that is of primary concern. As an example consider an *open system* $e_\circ$ that utilises resources of an unspecified library $e_\star$. Then analyses and optimisations of $e_\circ$ must rely on $(\widehat{\mathsf{C}}, \widehat{\rho})$ expressing all properties of interest about the library $e_\star$, i.e. $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$. This ensures that the unspecified library $e_\star$ can be replaced by another library $e'_\star$ as long as $(\widehat{\mathsf{C}}, \widehat{\rho})$ continues to express all properties of interest about the library $e'_\star$, i.e. $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e'_\star$.

3.5 Adding Data Flow Analysis

In Section 3.1 we indicated that our Control Flow Analysis could be extended with Data Flow Analysis components. Basically, this amounts to extending the set $\widehat{\mathbf{Val}}$ to contain abstract values other than just abstractions. We shall first see how this can be done when the data flow component is a powerset and next we shall see how it can be generalised to complete lattices. We shall present the two approaches as abstract specifications only (in the manner of Section 3.1), leaving the syntax directed formulations to the exercises.

3.5.1 Abstract Values as Powersets

Abstract domains. There are several ways to extend the value domain $\widehat{\mathbf{Val}}$ so as to specify both Control Flow Analysis and Data Flow Analysis. A particularly simple approach is to use a set **Data** of *abstract data values* (i.e. abstract properties of booleans and integers) since this allows us to define:

$$\widehat{v} \in \widehat{\mathbf{Val}}_d = \mathcal{P}(\mathbf{Term} \cup \mathbf{Data}) \quad \text{abstract values}$$

For each constant $c \in \mathbf{Const}$ we need an element $d_c \in \mathbf{Data}$ specifying the abstract property of c. Similarly, for each operator $op \in \mathbf{Op}$ we need a total function

$$\widehat{\mathsf{op}} : \widehat{\mathbf{Val}}_d \times \widehat{\mathbf{Val}}_d \to \widehat{\mathbf{Val}}_d$$

telling how op operates on abstract properties. Typically, $\widehat{\mathsf{op}}$ will have a definition of the form

$$\widehat{v}_1 \ \widehat{\mathsf{op}} \ \widehat{v}_2 = \bigcup\{d_{op}(d_1, d_2) \mid d_1 \in \widehat{v}_1 \cap \mathbf{Data}, d_2 \in \widehat{v}_2 \cap \mathbf{Data}\}$$

for some function d_{op} : **Data** $\times$ **Data** $\rightarrow$ $\mathcal{P}$(**Data**) specifying how the operator computes with the abstract properties of integers and booleans.

Example 3.26 For a *Detection of Signs Analysis* we take $\mathbf{Data}_{\mathsf{sign}} = \{\mathsf{tt}, \mathsf{ff}, \mathsf{-}, \mathsf{0}, \mathsf{+}\}$ where tt and ff stand for the two truth values and -, 0, and + for the negative numbers, the number 0, and the positive numbers, respectively. It is then natural to define $d_{\mathtt{true}} = \mathsf{tt}$ and $d_7 = \mathsf{+}$ and similarly for the other constants. Taking $\widehat{+}$ as an example, we can base its definition on the following table

d_{+}	tt	ff	-	0	+
tt	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
ff	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
-	$\emptyset$	$\emptyset$	{-}	{-}	{-, 0, +}
0	$\emptyset$	$\emptyset$	{-}	{0}	{+}
+	$\emptyset$	$\emptyset$	{-, 0, +}	{+}	{+}

and similarly for the other operators. ■

Acceptability relation. The acceptability relation of the combined analysis has the form

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_d e$$

and is presented in Table 3.8. Compared with the analysis of Table 3.1 the clause [*con*] now records that d_c is a possible value of c and the clause [*op*] makes use of the function $\widehat{\mathsf{op}}$ described above. In the case of [*if*] we have made sure only to analyse those branches of the conditional that the analysis of the condition indicates the need for; hence we can be more precise than in the pure Control Flow Analysis – the Data Flow Analysis component of the analysis can influence the outcome of the Control Flow Analysis. In the manner of Exercise 3.3 similar improvements can be made to many of the clauses (see Exercise 3.14) and thereby the specification becomes more flow-sensitive.

Example 3.27 Consider the expression:

```
(let f = (fn x => (if (x^1 > 0^2)^3 then (fn y => y^4)^5
                   else (fn z => 25^6)^7)^8)^9
 in ((f^10 3^11)^12 0^13)^14)^15
```

A pure 0-CFA analysis will not be able to discover that the `else`-branch of the conditional will never be executed so it will conclude that the subterm with label 12 may evaluate to `fn y =>` y^4 as well as `fn z =>` 25^6 as shown in the first column of Figure 3.6. The second column of Figure 3.6 shows that when we combine the analysis with a Detection of Signs Analysis (outlined

$$
\begin{array}{ll}
[con] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d c^\ell \text{ iff } \{d_c\} \subseteq \widehat{\mathsf{C}}(\ell) \\[1ex]
[var] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d x^\ell \text{ iff } \widehat{\rho}(x) \subseteq \widehat{\mathsf{C}}(\ell) \\[1ex]
[fn] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d (\texttt{fn}\ x \texttt{ => } e_0)^\ell \text{ iff } \{\texttt{fn}\ x \texttt{ => } e_0\} \subseteq \widehat{\mathsf{C}}(\ell) \\[1ex]
[fun] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d (\texttt{fun}\ f\ x \texttt{ => } e_0)^\ell \text{ iff } \{\texttt{fun}\ f\ x \texttt{ => } e_0\} \subseteq \widehat{\mathsf{C}}(\ell) \\[1ex]
[app] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d (t_1^{\ell_1}\ t_2^{\ell_2})^\ell \\
& \quad \text{iff} \quad (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d t_1^{\ell_1} \ \wedge\ (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d t_2^{\ell_2} \ \wedge \\
& \qquad (\forall (\texttt{fn}\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\mathsf{C}}(\ell_1) : \\
& \qquad\qquad (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d t_0^{\ell_0} \ \wedge \\
& \qquad\qquad \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\rho}(x) \ \wedge\ \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell)) \ \wedge \\
& \qquad (\forall (\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\mathsf{C}}(\ell_1) : \\
& \qquad\qquad (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d t_0^{\ell_0} \ \wedge \\
& \qquad\qquad \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\rho}(x) \ \wedge\ \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell) \ \wedge \\
& \qquad\qquad \{\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}\} \subseteq \widehat{\rho}(f)) \\[2ex]
[if] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d (\texttt{if}\ t_0^{\ell_0}\ \texttt{then}\ t_1^{\ell_1}\ \texttt{else}\ t_2^{\ell_2})^\ell \\
& \quad \text{iff} \quad (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d t_0^{\ell_0} \ \wedge \\
& \qquad (d_{\texttt{true}} \in \widehat{\mathsf{C}}(\ell_0) \Rightarrow ((\widehat{\mathsf{C}}, \widehat{\rho}) \models_d t_1^{\ell_1} \ \wedge\ \widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\mathsf{C}}(\ell))) \ \wedge \\
& \qquad (d_{\texttt{false}} \in \widehat{\mathsf{C}}(\ell_0) \Rightarrow ((\widehat{\mathsf{C}}, \widehat{\rho}) \models_d t_2^{\ell_2} \ \wedge\ \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\mathsf{C}}(\ell))) \\[2ex]
[let] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d (\texttt{let}\ x = t_1^{\ell_1}\ \texttt{in}\ t_2^{\ell_2})^\ell \\
& \quad \text{iff} \quad (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d t_1^{\ell_1} \ \wedge\ (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d t_2^{\ell_2} \ \wedge \\
& \qquad \widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\rho}(x) \ \wedge\ \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\mathsf{C}}(\ell) \\[2ex]
[op] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d (t_1^{\ell_1}\ op\ t_2^{\ell_2})^\ell \\
& \quad \text{iff} \quad (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d t_1^{\ell_1} \ \wedge\ (\widehat{\mathsf{C}}, \widehat{\rho}) \models_d t_2^{\ell_2} \ \wedge \\
& \qquad \widehat{\mathsf{C}}(\ell_1)\ \widehat{\mathsf{op}}\ \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\mathsf{C}}(\ell)
\end{array}
$$

Table 3.8: Abstract values as powersets.

in Example 3.26) then the analysis can determine that only `fn y => y`^{4} is a possible abstraction at label 12. Note that the Detection of Signs Analysis (correctly) determines that the expression will evaluate to a value with property $\{0\}$. ■

The proof techniques introduced in Section 3.2 should suffice for proving the correctness of the analysis with respect to the operational semantics. A slight extension of the algorithmic techniques presented in Sections 3.3 and 3.4 (and in Mini Project 3.1) suffices for obtaining an implementation of the analysis provided that the set **Data** is finite.

	Section 3.1	Subsection 3.5.1	Subsection 3.5.2	
	$(\widehat{\mathsf{C}}, \widehat{\rho})$	$(\widehat{\mathsf{C}}, \widehat{\rho})$	$(\widehat{\mathsf{C}}, \widehat{\rho})$	$(\widehat{\mathsf{D}}, \widehat{\delta})$
1	$\emptyset$	{+}	$\emptyset$	{+}
2	$\emptyset$	{0}	$\emptyset$	{0}
3	$\emptyset$	{tt}	$\emptyset$	{tt}
4	$\emptyset$	{0}	$\emptyset$	{0}
5	{fn y => y^4}	{fn y => y^4}	{fn y => y^4}	$\emptyset$
6	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
7	{fn z => 25^6}	$\emptyset$	$\emptyset$	$\emptyset$
8	{fn y => y^4, fn z => 25^6}	{fn y => y^4}	{fn y => y^4}	$\emptyset$
9	{fn x => $(\cdots)^8$}	{fn x => $(\cdots)^8$}	{fn x => $(\cdots)^8$}	$\emptyset$
10	{fn x => $(\cdots)^8$}	{fn x => $(\cdots)^8$}	{fn x => $(\cdots)^8$}	$\emptyset$
11	$\emptyset$	{+}	$\emptyset$	{+}
12	{fn y => y^4, fn z => 25^6}	{fn y => y^4}	{fn y => y^4}	$\emptyset$
13	$\emptyset$	{0}	$\emptyset$	{0}
14	$\emptyset$	{0}	$\emptyset$	{0}
15	$\emptyset$	{0}	$\emptyset$	{0}
f	{fn x => $(\cdots)^8$}	{fn x => $(\cdots)^8$}	{fn x => $(\cdots)^8$}	$\emptyset$
x	$\emptyset$	{+}	$\emptyset$	{+}
y	$\emptyset$	{0}	$\emptyset$	{0}
z	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Figure 3.6: Control Flow and Data Flow Analysis for example program.

Finally, we should stress that a solution to the analysis of Table 3.8 does *not* immediately give a solution to the analysis of Table 3.1. More precisely, $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_d e$ does *not* guarantee that $(\widehat{\mathsf{C}}', \widehat{\rho}') \models e$ where $\forall \ell : \widehat{\mathsf{C}}'(\ell) = \widehat{\mathsf{C}}(\ell) \cap \mathbf{Term}$ and $\forall x : \widehat{\rho}'(x) = \widehat{\rho}(x) \cap \mathbf{Term}$. The reason is that the Control Flow Analysis part of Table 3.8 is influenced by the Data Flow Analysis part in the clause [*if*]: if for example the abstract value of the condition does not include $d_{\mathtt{true}}$ then the **then**-branch will not be analysed.

3.5.2 Abstract Values as Complete Lattices

Abstract domains. Clearly $\widehat{\mathbf{Val}}_d = \mathcal{P}(\mathbf{Term} \cup \mathbf{Data})$ is isomorphic to $\mathcal{P}(\mathbf{Term}) \times \mathcal{P}(\mathbf{Data})$. This suggests that the abstract cache $\widehat{\mathsf{C}} : \mathbf{Lab} \to \widehat{\mathbf{Val}}_d$

could be split into a term component and a data component and similarly for the abstract environment $\widehat{\rho} : \mathbf{Var} \to \widehat{\mathbf{Val}}_d$.

Having decoupled $\mathcal{P}(\mathbf{Term})$ and $\mathcal{P}(\mathbf{Data})$ we can now consider replacing $\mathcal{P}(\mathbf{Data})$ by a more general collection of properties. An obvious possibility is to replace $\mathcal{P}(\mathbf{Data})$ by a complete lattice L and perform a development closely related to that of the (forward) Monotone Frameworks of Chapter 2.

So let us define a *monotone structure* to consist of:

- a complete lattice L, and
- a set $\mathcal{F}$ of monotone functions of $L \times L \to L$.

An *instance* of a monotone structure then consists of the structure $(L, \mathcal{F})$ and

- a mapping $\iota_{\cdot}$ from the constants $c \in \mathbf{Const}$ to values in L, and
- a mapping $f_{\cdot}$ from the binary operators $op \in \mathbf{Op}$ to functions of $\mathcal{F}$.

Compared with the instances of the Monotone Frameworks of Section 2.3 we omit the flow component since it will be the responsibility of the Control Flow Analysis to determine this. The component ι has been replaced by the mapping $\iota_{\cdot}$ giving the *extremal value* for all the constants and the component $f_{\cdot}$ mapping labels to transfer functions has been replaced by a mapping of the binary operators to their interpretation.

Example 3.28 A monotone structure corresponding to the development of Subsection 3.5.1 will have L to be $\mathcal{P}(\mathbf{Data})$ and $\mathcal{F}$ to be the monotone functions of $\mathcal{P}(\mathbf{Data}) \times \mathcal{P}(\mathbf{Data}) \to \mathcal{P}(\mathbf{Data})$.

An instance of the monotone structure is then obtained by taking

$$\iota_c = \{d_c\}$$

for all constants c (and with $d_c \in \mathbf{Data}$ as above) and

$$f_{op}(l_1, l_2) = \bigcup \{d_{op}(d_1, d_2) \mid d_1 \in l_1, d_2 \in l_2\}$$

for all binary operators op (and where $d_{op} : \mathbf{Data} \times \mathbf{Data} \to \mathcal{P}(\mathbf{Data})$ is as above). ∎

Example 3.29 A monotone structure for *Constant Propagation Analysis* will have L to be $\mathbf{Z}_\perp^\top \times \mathcal{P}(\{\mathsf{tt}, \mathsf{ff}\})$ and $\mathcal{F}$ to be the monotone functions of $L \times L \to L$.

An instance of the monotone structure is obtained by taking e.g. $\iota_7 = (7, \emptyset)$ and $\iota_{\mathtt{true}} = (\bot, \{\mathtt{tt}\})$. For a binary operator such as + we can take:

$$f_{+}(l_1, l_2) = \begin{cases} (z_1 + z_2, \emptyset) & \text{if } l_1 = (z_1, \cdots), l_2 = (z_2, \cdots), \\ & \quad \text{and } z_1, z_2 \in \mathbf{Z} \\ (\bot, \emptyset) & \text{if } l_1 = (z_1, \cdots), l_2 = (z_2, \cdots), \\ & \quad \text{and } z_1 = \bot \text{ or } z_2 = \bot \\ (\top, \emptyset) & \text{otherwise} \end{cases}$$

■

We can now define the following abstract domains

$$\begin{array}{lllll} \widehat{v} & \in & \widehat{\mathbf{Val}} & = & \mathcal{P}(\mathbf{Term}) & \text{abstract values} \\ \widehat{\rho} & \in & \widehat{\mathbf{Env}} & = & \mathbf{Var} \to \widehat{\mathbf{Val}} & \text{abstract environments} \\ \widehat{\mathsf{C}} & \in & \widehat{\mathbf{Cache}} & = & \mathbf{Lab} \to \widehat{\mathbf{Val}} & \text{abstract caches} \end{array}$$

to take care of the Control Flow Analysis and furthermore

$$\begin{array}{lllll} \widehat{d} & \in & \widehat{\mathbf{Data}} & = & L & \text{abstract data values} \\ \widehat{\delta} & \in & \widehat{\mathbf{DEnv}} & = & \mathbf{Var} \to \widehat{\mathbf{Data}} & \text{abstract data environments} \\ \widehat{\mathsf{D}} & \in & \widehat{\mathbf{DCache}} & = & \mathbf{Lab} \to \widehat{\mathbf{Data}} & \text{abstract data caches} \end{array}$$

to take care of the Data Flow Analysis.

Acceptability relation. The acceptability relation now has the form

$$(\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D e$$

and it is defined by the clauses of Table 3.9. In the clause [*con*] we see that the $\iota_{\cdot}$ component of the instance is used to restrict the value of the $\widehat{\mathsf{D}}$ component of the analysis and in the clause [*op*] we see how the $f_{\cdot}$ component is used. The clause [*if*] has explicit tests for the two branches as in the previous approach thereby allowing the Control Flow Analysis to benefit from results obtained by the Data Flow Analysis component. As in the previous subsection, similar improvements can be made to many of the other clauses so as to produce a more flow-sensitive analysis.

Example 3.30 Returning to the expression of Example 3.27 and the Detection of Signs Analysis we now get the analysis result of the last column of Figure 3.6. So we see that the result is as before. ■

The proof techniques introduced in Section 3.2 should suffice for proving the correctness of the analysis with respect to the operational semantics. A slight extension of the algorithmic techniques presented in Sections 3.3 and 3.4 (and in Mini Project 3.1) suffices for obtaining an implementation of the analysis provided that L satisfies the Ascending Chain Condition (as is the case for Monotone Frameworks).

$$
\begin{array}{ll}
[con] & (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D c^\ell \text{ iff } \iota_c \sqsubseteq \widehat{\mathsf{D}}(\ell) \\[2ex]
[var] & (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D x^\ell \text{ iff } \widehat{\rho}(x) \subseteq \widehat{\mathsf{C}}(\ell) \ \wedge \ \widehat{\delta}(x) \sqsubseteq \widehat{\mathsf{D}}(\ell) \\[2ex]
[fn] & (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D (\texttt{fn}\ x \texttt{ => } e_0)^\ell \text{ iff } \{\texttt{fn}\ x \texttt{ => } e_0\} \subseteq \widehat{\mathsf{C}}(\ell) \\[2ex]
[fun] & (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D (\texttt{fun}\ f\ x \texttt{ => } e_0)^\ell \text{ iff } \{\texttt{fun}\ f\ x \texttt{ => } e_0\} \subseteq \widehat{\mathsf{C}}(\ell) \\[2ex]
[app] & (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D (t_1^{\ell_1}\ t_2^{\ell_2})^\ell \\
& \quad \text{iff} \quad (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D t_1^{\ell_1} \ \wedge \ (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D t_2^{\ell_2} \ \wedge \\
& \qquad (\forall (\texttt{fn}\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\mathsf{C}}(\ell_1) : \\
& \qquad\qquad (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D t_0^{\ell_0} \ \wedge \\
& \qquad\qquad \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\rho}(x) \ \wedge \ \widehat{\mathsf{D}}(\ell_2) \sqsubseteq \widehat{\delta}(x) \ \wedge \\
& \qquad\qquad \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell) \ \wedge \ \widehat{\mathsf{D}}(\ell_0) \sqsubseteq \widehat{\mathsf{D}}(\ell)) \ \wedge \\
& \qquad (\forall (\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\mathsf{C}}(\ell_1) : \\
& \qquad\qquad (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D t_0^{\ell_0} \ \wedge \\
& \qquad\qquad \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\rho}(x) \ \wedge \ \widehat{\mathsf{D}}(\ell_2) \sqsubseteq \widehat{\delta}(x) \ \wedge \\
& \qquad\qquad \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell) \ \wedge \ \widehat{\mathsf{D}}(\ell_0) \sqsubseteq \widehat{\mathsf{D}}(\ell) \ \wedge \\
& \qquad\qquad \{\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}\} \subseteq \widehat{\rho}(f)) \\[2ex]
[if] & (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D (\texttt{if}\ t_0^{\ell_0}\ \texttt{then}\ t_1^{\ell_1}\ \texttt{else}\ t_2^{\ell_2})^\ell \\
& \quad \text{iff} \quad (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D t_0^{\ell_0} \ \wedge \\
& \qquad (\iota_{\texttt{true}} \sqsubseteq \widehat{\mathsf{D}}(\ell_0) \Rightarrow (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D t_1^{\ell_1} \ \wedge \\
& \qquad\qquad\qquad\qquad \widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\mathsf{C}}(\ell) \wedge \\
& \qquad\qquad\qquad\qquad \widehat{\mathsf{D}}(\ell_1) \sqsubseteq \widehat{\mathsf{D}}(\ell)) \ \wedge \\
& \qquad (\iota_{\texttt{false}} \sqsubseteq \widehat{\mathsf{D}}(\ell_0) \Rightarrow (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D t_2^{\ell_2} \ \wedge \\
& \qquad\qquad\qquad\qquad \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\mathsf{C}}(\ell) \wedge \\
& \qquad\qquad\qquad\qquad \widehat{\mathsf{D}}(\ell_2) \sqsubseteq \widehat{\mathsf{D}}(\ell)) \\[2ex]
[let] & (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D (\texttt{let}\ x = t_1^{\ell_1}\ \texttt{in}\ t_2^{\ell_2})^\ell \\
& \quad \text{iff} \quad (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D t_1^{\ell_1} \ \wedge \ (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D t_2^{\ell_2} \ \wedge \\
& \qquad \widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\rho}(x) \wedge \widehat{\mathsf{D}}(\ell_1) \sqsubseteq \widehat{\delta}(x) \ \wedge \\
& \qquad \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\mathsf{C}}(\ell) \wedge \widehat{\mathsf{D}}(\ell_2) \sqsubseteq \widehat{\mathsf{D}}(\ell) \\[2ex]
[op] & (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D (t_1^{\ell_1}\ op\ t_2^{\ell_2})^\ell \\
& \quad \text{iff} \quad (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D t_1^{\ell_1} \ \wedge \ (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models_D t_2^{\ell_2} \ \wedge \\
& \qquad f_{op}(\widehat{\mathsf{D}}(\ell_1), \widehat{\mathsf{D}}(\ell_2)) \sqsubseteq \widehat{\mathsf{D}}(\ell)
\end{array}
$$

Table 3.9: Abstract values as complete lattices.

Staging the specification. Let us briefly consider the following alternative clause for [*if*] where the data flow component *cannot* influence the control flow component because we always make sure that the analysis result is acceptable for both branches:

$$
\begin{array}{ll}
(\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models'_D (\texttt{if } t_0^{\ell_0} \texttt{ then } t_1^{\ell_1} \texttt{ else } t_2^{\ell_2})^{\ell} & \\
\quad \text{iff} & (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models'_D t_0^{\ell_0} \wedge \\
& (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models'_D t_1^{\ell_1} \wedge \widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\mathsf{C}}(\ell) \wedge \widehat{\mathsf{D}}(\ell_1) \sqsubseteq \widehat{\mathsf{D}}(\ell) \wedge \\
& (\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models'_D t_2^{\ell_2} \wedge \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\mathsf{C}}(\ell) \wedge \widehat{\mathsf{D}}(\ell_2) \sqsubseteq \widehat{\mathsf{D}}(\ell)
\end{array}
$$

Unlike what was the case for the analyses of Tables 3.8 and 3.9, a solution to the analysis modified in this way *does* give rise to a solution to the analysis of Table 3.1; to be precise $(\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{\delta}) \models'_D e$ guarantees $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e$.

In terms of implementation this modification means that the constraints for the control flow component ($\widehat{\mathsf{C}}$ and $\widehat{\rho}$) can be solved first, and based on this the constraints for the data flow component ($\widehat{\mathsf{D}}$ and $\widehat{\delta}$) can be solved next. If both sets of constraints are solved for their least solution this will still yield the least solution of the combined set of constraints.

Example 3.31 Let us return to Example 3.30. If we modify the clause for [*if*] as discussed above then the resulting analysis will have $\widehat{\mathsf{C}}$ and $\widehat{\rho}$ as in the column for the pure analysis from Section 3.1 and $\widehat{\mathsf{D}}$ and $\widehat{\delta}$ will associate slightly larger sets with some of the labels and variables:

$$
\begin{array}{rcl}
\widehat{\mathsf{D}}(6) & = & \{+\} \\
\widehat{\mathsf{D}}(14) & = & \{0, +\} \\
\widehat{\mathsf{D}}(15) & = & \{0, +\} \\
\widehat{\delta}(\mathsf{z}) & = & \{0\}
\end{array}
$$

This analysis is less precise than those of Tables 3.8 and 3.9: it will only determine that the expression will evaluate to a value with the property $\{0, +\}$. ■

Imperative constructs and data structures. Mini Project 3.2 shows one way of extending the development to track creation points of data structures. Mini Project 3.4 shows how to deal with imperative constructs in the manner of the imperative language WHILE of Chapter 2.

3.6 Adding Context Information

The Control Flow Analyses presented so far are imprecise in that they cannot distinguish the various instances of function calls from one another. In the

terminology of Section 2.5 the 0-CFA analysis is *context-insensitive* and in the terminology of Control Flow Analysis it is *monovariant.*

Example 3.32 Consider the expression:

$$\begin{array}{l}(\texttt{let f = (fn x => x}^1)^2 \\ \texttt{in } ((\texttt{f}^3\ \texttt{f}^4)^5\ (\texttt{fn y => y}^6)^7)^8)^9\end{array}$$

The least 0-CFA analysis is given by $(\widehat{\mathsf{C}}_{\mathsf{id}}, \widehat{\rho}_{\mathsf{id}})$:

$$\begin{array}{lcl}
\widehat{\mathsf{C}}_{\mathsf{id}}(1) & = & \{\texttt{fn x => x}^1, \texttt{fn y => y}^6\} \\
\widehat{\mathsf{C}}_{\mathsf{id}}(2) & = & \{\texttt{fn x => x}^1\} \\
\widehat{\mathsf{C}}_{\mathsf{id}}(3) & = & \{\texttt{fn x => x}^1\} \\
\widehat{\mathsf{C}}_{\mathsf{id}}(4) & = & \{\texttt{fn x => x}^1\} \\
\widehat{\mathsf{C}}_{\mathsf{id}}(5) & = & \{\texttt{fn x => x}^1, \texttt{fn y => y}^6\} \\
\widehat{\mathsf{C}}_{\mathsf{id}}(6) & = & \{\texttt{fn y => y}^6\} \\
\widehat{\mathsf{C}}_{\mathsf{id}}(7) & = & \{\texttt{fn y => y}^6\} \\
\widehat{\mathsf{C}}_{\mathsf{id}}(8) & = & \{\texttt{fn x => x}^1, \texttt{fn y => y}^6\} \\
\widehat{\mathsf{C}}_{\mathsf{id}}(9) & = & \{\texttt{fn x => x}^1, \texttt{fn y => y}^6\} \\
\widehat{\rho}_{\mathsf{id}}(\texttt{f}) & = & \{\texttt{fn x => x}^1\} \\
\widehat{\rho}_{\mathsf{id}}(\texttt{x}) & = & \{\texttt{fn x => x}^1, \texttt{fn y => y}^6\} \\
\widehat{\rho}_{\mathsf{id}}(\texttt{y}) & = & \{\texttt{fn y => y}^6\}
\end{array}$$

So we see that `x` can be bound to $\texttt{fn x => x}^1$ as well as $\texttt{fn y => y}^6$ and hence the overall expression (label 9) may evaluate to either of these two abstractions. However, it is easy to see that in fact only $\texttt{fn y => y}^6$ is a possible result. ■

To get a more precise analysis it is useful to introduce a mechanism that distinguishes different dynamic instances of variables and labels from one another. This results in a *context-sensitive* analysis and in the terminology of Control Flow Analysis the term *polyvariant* is used. There are several approaches to how this can be done. One simple possibility is to expand the program such that the problem does not arise.

Example 3.33 For the expression of Example 3.32 we could for example consider

```
let f1 = (fn x1 => x1)
in let f2 = (fn x2 => x2)
   in (f1 f2) (fn y => y)
```

and then analyse the expanded expression: the 0-CFA analysis is now able to deduce that `x1` can only be bound to `fn x2 => x2` and that `x2` can only be bound to `fn y => y` so the overall expression will evaluate to `fn y => y` only. ■

A more satisfactory solution to the problem is to extend the analysis with *context information* allowing it to distinguish between the various instances of variables and program points and still analyse the original expression. Examples of such analyses include k-CFA analyses, uniform k-CFA analyses, polynomial k-CFA analyses (mainly of interest for $k > 0$) and the Cartesian Product Algorithm.

3.6.1 Uniform k-CFA Analysis

Abstract domains. A key idea is to introduce context to distinguish between the various dynamic instances of variables and program points. There are many choices concerning how to model contexts and how they can be modified in the course of the analysis. In a *uniform k-CFA* analysis (as well as in a k-CFA analysis) a context δ records the last k dynamic call points; hence in this case contexts will be sequences of labels of length at most k and they will be updated whenever a function application is analysed. This is modelled by taking:

$$\delta \in \Delta = \mathbf{Lab}^{\leq k} \quad \text{context information}$$

Since the contexts will be used to distinguish between the various instances of the variables we will need a *context environment* to determine the context associated with the current instance of a variable:

$$ce \in \mathbf{CEnv} = \mathbf{Var} \to \Delta \quad \text{context environments}$$

The context environment will play a role similar to the environment of the semantics; in particular, this means that we shall extend the *abstract values* to contain a context environment:

$$\widehat{v} \in \widehat{\mathbf{Val}} = \mathcal{P}(\mathbf{Term} \times \mathbf{CEnv}) \quad \text{abstract values}$$

So in addition to recording the abstractions (`fn` x `=>` e and `fun` f x `=>` e) we will also record the context environment at the *definition point* for the free variables of the term. This should be compared with the Structural Operational Semantics of Section 3.2 where the closures contain information about the abstraction as well as the environment determining the values of the free variables at the definition point.

The *abstract environment* $\widehat{\rho}$ will now map a variable and a context to an abstract value:

$$\widehat{\rho} \in \widehat{\mathbf{Env}} = (\mathbf{Var} \times \Delta) \to \widehat{\mathbf{Val}} \quad \text{abstract environments}$$

$$
\begin{array}{ll}
[con] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} c^{\ell} \text{ always} \\[1em]
[var] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} x^{\ell} \text{ iff } \widehat{\rho}(x, ce(x)) \subseteq \widehat{\mathsf{C}}(\ell, \delta) \\[1em]
[fn] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} (\texttt{fn}\ x \texttt{ => } e_0)^{\ell} \text{ iff } \{(\texttt{fn}\ x \texttt{ => } e_0, ce_0)\} \subseteq \widehat{\mathsf{C}}(\ell, \delta) \\
 & \qquad\qquad \text{where } ce_0 = ce \mid FV(\texttt{fn}\ x \texttt{ => } e_0) \\[1em]
[fun] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} (\texttt{fun}\ f\ x \texttt{ => } e_0)^{\ell} \text{ iff } \{(\texttt{fun}\ f\ x \texttt{ => } e_0, ce_0)\} \subseteq \widehat{\mathsf{C}}(\ell, \delta) \\
 & \qquad\qquad \text{where } ce_0 = ce \mid FV(\texttt{fun}\ f\ x \texttt{ => } e_0) \\[1em]
[app] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} (t_1^{\ell_1}\ t_2^{\ell_2})^{\ell} \\
 & \quad \text{iff } (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} t_1^{\ell_1} \ \wedge\ (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} t_2^{\ell_2} \ \wedge \\
 & \quad (\forall (\texttt{fn}\ x \texttt{ => } t_0^{\ell_0}, ce_0) \in \widehat{\mathsf{C}}(\ell_1, \delta) : \\
 & \qquad (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce'_0}_{\delta_0} t_0^{\ell_0} \ \wedge \\
 & \qquad \widehat{\mathsf{C}}(\ell_2, \delta) \subseteq \widehat{\rho}(x, \delta_0) \ \wedge\ \widehat{\mathsf{C}}(\ell_0, \delta_0) \subseteq \widehat{\mathsf{C}}(\ell, \delta) \\
 & \qquad \text{where } \delta_0 = \lceil \delta, \ell \rceil_k \\
 & \qquad \text{and } ce'_0 = ce_0[x \mapsto \delta_0]) \ \wedge \\
 & \quad (\forall (\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}, ce_0) \in \widehat{\mathsf{C}}(\ell_1, \delta) : \\
 & \qquad (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce'_0}_{\delta_0} t_0^{\ell_0} \ \wedge \\
 & \qquad \widehat{\mathsf{C}}(\ell_2, \delta) \subseteq \widehat{\rho}(x, \delta_0) \ \wedge\ \widehat{\mathsf{C}}(\ell_0, \delta_0) \subseteq \widehat{\mathsf{C}}(\ell, \delta) \ \wedge \\
 & \qquad \{(\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}, ce_0)\} \subseteq \widehat{\rho}(f, \delta_0) \\
 & \qquad \text{where } \delta_0 = \lceil \delta, \ell \rceil_k \\
 & \qquad \text{and } ce'_0 = ce_0[f \mapsto \delta_0, x \mapsto \delta_0]) \\[1em]
[if] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} (\texttt{if}\ t_0^{\ell_0}\ \texttt{then}\ t_1^{\ell_1}\ \texttt{else}\ t_2^{\ell_2})^{\ell} \\
 & \quad \text{iff } (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} t_0^{\ell_0} \ \wedge\ (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} t_1^{\ell_1} \ \wedge\ (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} t_2^{\ell_2} \ \wedge \\
 & \quad \widehat{\mathsf{C}}(\ell_1, \delta) \subseteq \widehat{\mathsf{C}}(\ell, \delta) \ \wedge\ \widehat{\mathsf{C}}(\ell_2, \delta) \subseteq \widehat{\mathsf{C}}(\ell, \delta) \\[1em]
[let] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} (\texttt{let}\ x = t_1^{\ell_1}\ \texttt{in}\ t_2^{\ell_2})^{\ell} \\
 & \quad \text{iff } (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} t_1^{\ell_1} \ \wedge\ (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce'}_{\delta} t_2^{\ell_2} \ \wedge \\
 & \quad \widehat{\mathsf{C}}(\ell_1, \delta) \subseteq \widehat{\rho}(x, \delta) \ \wedge\ \widehat{\mathsf{C}}(\ell_2, \delta) \subseteq \widehat{\mathsf{C}}(\ell, \delta) \\
 & \quad \text{where } ce' = ce[x \mapsto \delta] \\[1em]
[op] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} (t_1^{\ell_1}\ op\ t_2^{\ell_2})^{\ell} \text{ iff } (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} t_1^{\ell_1} \ \wedge\ (\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} t_2^{\ell_2}
\end{array}
$$

Table 3.10: Uniform k-CFA analysis.

Typically we will use a context environment to find the context associated with the variable of interest and then use it together with the variable to access the abstract environment. This means that indirectly we get the effect of having local abstract environments in the abstract values although $\widehat{\rho}$ is still a global entity as in the previous sections.

The uniform k-CFA analysis differs from the k-CFA analysis in performing a similar development for the *abstract cache* that now maps a label and a context to an abstract value:

$$\widehat{\mathsf{C}} \in \widehat{\mathbf{Cache}} = (\mathbf{Lab} \times \Delta) \to \widehat{\mathbf{Val}} \quad \text{abstract caches}$$

Given information about the context of interest we can determine the abstract value associated with a label. Again we indirectly get the effect of having a cache for each possible context although it is still a global entity. (In k-CFA one has $\widehat{\mathbf{Cache}} = (\mathbf{Lab} \times \mathbf{CEnv}) \to \widehat{\mathbf{Val}}$.)

Acceptability relation. The acceptability relation for *uniform k-CFA* is presented in Table 3.10. It is defined by formulae of the form

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_{\delta}^{ce} e$$

where ce is the current context environment and δ is the current context. The formula expresses that $(\widehat{\mathsf{C}}, \widehat{\rho})$ is an acceptable analysis of e in the *context* specified by ce and δ. The clauses for the various constructs are very much as those in Table 3.1 and will be explained below.

In the clause [*var*] we use the current context environment ce to determine the context $ce(x)$ of the current instance of the variable x and then the abstract value of the variable is given by $\widehat{\rho}(x, ce(x))$. The current context is δ so we have to ensure that $\widehat{\rho}(x, ce(x)) \subseteq \widehat{\mathsf{C}}(\ell, \delta)$.

In the clause [*fn*] we record the current context environment as part of the abstract value and (as in the Structural Operational Semantics of Table 3.2) we restrict the context environment to the set of variables of interest for the abstraction. The clause [*fun*] is similar.

In the clause [*app*] we analyse the two subexpressions using the same context and context environment as the composite expression. When we find a potential abstract value, say (`fn` x `=>` $t_0^{\ell_0}$, ce_0), that the operator may evaluate to, it will contain a local context environment ce_0 that was created at its definition point. When analysing $t_0^{\ell_0}$ we will have passed through the application point ℓ, so the current context will be updated to include ℓ and this will also be the context associated with the variable x in the updated version of the context environment ce_0 used for the analysis of $t_0^{\ell_0}$. The new context is $\lceil \delta, \ell \rceil_k$ which (as in Section 2.5) denotes the sequence $[\delta, \ell]$ but possibly truncated (by omitting elements on the left) so as have length at most k. In the case where the operator has the form (`fun` f x `=>` $t_0^{\ell_0}$, ce_0) we proceed in a similar way and note that f as well as x will be associated with the new context in the analysis of the body of the function.

The clauses for [*if*], [*let*] and [*op*] are fairly straightforward modifications of those of Table 3.1; however, note that the context of the bound variable of the `let`-construct is the current context (as no application point is passed). We

shall dispense with proving the correctness of the analysis and with showing how it can be implemented.

Example 3.34 We shall now specify a uniform 1-CFA analysis for the expression of Example 3.33:

$$(\texttt{let f = (fn x => x}^1)^2 \texttt{ in } ((\texttt{f}^3\ \texttt{f}^4)^5\ (\texttt{fn y => y}^6)^7)^8)^9$$

The initial context will be Λ, the empty sequence of labels. In the course of the analysis the current context will be modified at the two application points with labels 5 and 8; since we only records call strings of length at most one the only contexts of interest will therefore be Λ, 5 and 8. There are four context environments of interest:

$\mathsf{ce}_0 = [\,]$	the initial (empty) context environment,
$\mathsf{ce}_1 = \mathsf{ce}_0[\texttt{f} \mapsto \Lambda]$	the context environment for the analysis of the body of the `let`-construct,
$\mathsf{ce}_2 = \mathsf{ce}_0[\texttt{x} \mapsto 5]$	the context environment used for the analysis of the body of `f` initiated at the application point 5, and
$\mathsf{ce}_3 = \mathsf{ce}_0[\texttt{x} \mapsto 8]$	the context environment used for the analysis of the body of `f` initiated at the application point 8.

Let us take $\widehat{\mathsf{C}}'_{\mathsf{id}}$ and $\widehat{\rho}'_{\mathsf{id}}$ to be:

$$\begin{array}{ll}
\widehat{\mathsf{C}}'_{\mathsf{id}}(1,5) = \{(\texttt{fn x => x}^1,\mathsf{ce}_0)\} & \widehat{\mathsf{C}}'_{\mathsf{id}}(1,8) = \{(\texttt{fn y => y}^6,\mathsf{ce}_0)\} \\
\widehat{\mathsf{C}}'_{\mathsf{id}}(2,\Lambda) = \{(\texttt{fn x => x}^1,\mathsf{ce}_0)\} & \widehat{\mathsf{C}}'_{\mathsf{id}}(3,\Lambda) = \{(\texttt{fn x => x}^1,\mathsf{ce}_0)\} \\
\widehat{\mathsf{C}}'_{\mathsf{id}}(4,\Lambda) = \{(\texttt{fn x => x}^1,\mathsf{ce}_0)\} & \widehat{\mathsf{C}}'_{\mathsf{id}}(5,\Lambda) = \{(\texttt{fn x => x}^1,\mathsf{ce}_0)\} \\
\widehat{\mathsf{C}}'_{\mathsf{id}}(7,\Lambda) = \{(\texttt{fn y => y}^6,\mathsf{ce}_0)\} & \widehat{\mathsf{C}}'_{\mathsf{id}}(8,\Lambda) = \{(\texttt{fn y => y}^6,\mathsf{ce}_0)\} \\
\widehat{\mathsf{C}}'_{\mathsf{id}}(9,\Lambda) = \{(\texttt{fn y => y}^6,\mathsf{ce}_0)\} & \\
\widehat{\rho}'_{\mathsf{id}}(\texttt{f},\Lambda) = \{(\texttt{fn x => x}^1,\mathsf{ce}_0)\} & \\
\widehat{\rho}'_{\mathsf{id}}(\texttt{x},5) = \{(\texttt{fn x => x}^1,\mathsf{ce}_0)\} & \widehat{\rho}'_{\mathsf{id}}(\texttt{x},8) = \{(\texttt{fn y => y}^6,\mathsf{ce}_0)\}
\end{array}$$

We shall now show that this is an acceptable analysis result for the example expression:

$$(\widehat{\mathsf{C}}'_{\mathsf{id}}, \widehat{\rho}'_{\mathsf{id}}) \models^{\mathsf{ce}_0}_{\Lambda} (\texttt{let f = (fn x => x}^1)^2 \texttt{ in } ((\texttt{f}^3\ \texttt{f}^4)^5\ (\texttt{fn y => y}^6)^7)^8)^9$$

According to clause [*let*], it is sufficient to verify that

$$(\widehat{\mathsf{C}}'_{\mathsf{id}}, \widehat{\rho}'_{\mathsf{id}}) \models^{\mathsf{ce}_0}_{\Lambda} (\texttt{fn x => x}^1)^2$$
$$(\widehat{\mathsf{C}}'_{\mathsf{id}}, \widehat{\rho}'_{\mathsf{id}}) \models^{\mathsf{ce}_1}_{\Lambda} ((\texttt{f}^3\ \texttt{f}^4)^5\ (\texttt{fn y => y}^6)^7)^8$$

because $\widehat{\mathsf{C}}'_{\mathsf{id}}(2,\Lambda) \subseteq \widehat{\rho}'_{\mathsf{id}}(\texttt{f},\Lambda)$ and $\widehat{\mathsf{C}}'_{\mathsf{id}}(8,\Lambda) \subseteq \widehat{\mathsf{C}}'_{\mathsf{id}}(9,\Lambda)$. This is straightforward except for the last clause. Since $\widehat{\mathsf{C}}'_{\mathsf{id}}(5,\Lambda) = \{(\texttt{fn x => x}^1,\mathsf{ce}_0)\}$ it is, according to [*app*], sufficient to verify that

$$(\widehat{\mathsf{C}}'_{\mathsf{id}}, \widehat{\rho}'_{\mathsf{id}}) \models^{\mathsf{ce}_1}_{\Lambda} (\mathtt{f}^3\ \mathtt{f}^4)^5$$
$$(\widehat{\mathsf{C}}'_{\mathsf{id}}, \widehat{\rho}'_{\mathsf{id}}) \models^{\mathsf{ce}_1}_{\Lambda} (\mathtt{fn\ y\ =>\ y}^6)^7$$
$$(\widehat{\mathsf{C}}'_{\mathsf{id}}, \widehat{\rho}'_{\mathsf{id}}) \models^{\mathsf{ce}_3}_{8} \mathtt{x}^1$$

because $\widehat{\mathsf{C}}'_{\mathsf{id}}(7, \Lambda) \subseteq \widehat{\rho}'_{\mathsf{id}}(\mathtt{x}, 8)$ and $\widehat{\mathsf{C}}'_{\mathsf{id}}(1, 8) \subseteq \widehat{\mathsf{C}}'_{\mathsf{id}}(8, \Lambda)$. This is straightforward except for the first clause. Proceeding as above we see that $\widehat{\mathsf{C}}'_{\mathsf{id}}(3, \Lambda) = \{(\mathtt{fn\ x\ =>\ x}^1, \mathsf{ce}_0)\}$ and it is sufficient to verify

$$(\widehat{\mathsf{C}}'_{\mathsf{id}}, \widehat{\rho}'_{\mathsf{id}}) \models^{\mathsf{ce}_1}_{\Lambda} \mathtt{f}^3$$
$$(\widehat{\mathsf{C}}'_{\mathsf{id}}, \widehat{\rho}'_{\mathsf{id}}) \models^{\mathsf{ce}_1}_{\Lambda} \mathtt{f}^4$$
$$(\widehat{\mathsf{C}}'_{\mathsf{id}}, \widehat{\rho}'_{\mathsf{id}}) \models^{\mathsf{ce}_2}_{5} \mathtt{x}^1$$

because $\widehat{\mathsf{C}}'_{\mathsf{id}}(4, \Lambda) \subseteq \widehat{\rho}'_{\mathsf{id}}(\mathtt{x}, 5)$ and $\widehat{\mathsf{C}}'_{\mathsf{id}}(1, 5) \subseteq \widehat{\mathsf{C}}'_{\mathsf{id}}(5, \Lambda)$. This is straightforward. The importance of this example is that it shows that the uniform 1-CFA analysis is strong enough to determine that $\mathtt{fn\ y\ =>\ y}^6$ is the *only* result possible for the overall expression unlike what was the case for the 0-CFA analysis in Example 3.32. We can also see that, since $\widehat{\rho}'_{\mathsf{id}}(\mathtt{y}, \delta) = \emptyset$ for all $\delta \in \{\Lambda, 5, 8\}$ it follows that $\mathtt{fn\ y\ =>\ y}^6$ is never called upon a function. ∎

The resulting analysis will have exponential worst case complexity even for the case where $k = 1$. To see this assume that the expression has size n and that it has p different variables. Then Δ has $O(n)$ elements and hence there will be $O(p \cdot n)$ different pairs (x, δ) and $O(n^2)$ different pairs (ℓ, δ). This means that $(\widehat{\mathsf{C}}, \widehat{\rho})$ can be seen as an $O(n^2)$ tuple of values from $\widehat{\mathbf{Val}}$. Since $\widehat{\mathbf{Val}}$ itself is a powerset of pairs of the form (t, ce) and there are $O(n \cdot n^p)$ such pairs it follows that $\widehat{\mathbf{Val}}$ has height $O(n \cdot n^p)$. Since $p = O(n)$ we have the exponential worst case complexity claimed above.

This should be contrasted with the 0-CFA analysis developed in the previous sections. It corresponds to letting Δ be a singleton. Repeating the above calculations we can see $(\widehat{\mathsf{C}}, \widehat{\rho})$ as an $O(p+n)$ tuple of values from $\widehat{\mathbf{Val}}$, and $\widehat{\mathbf{Val}}$ will be a lattice of height $O(n)$. In total this gives us a polynomial analysis as we already saw in Section 3.4.

The worst case complexity of the uniform k-CFA analysis (as well as the k-CFA analysis) can be improved in different ways. One possibility is to reduce the height of the lattice $\widehat{\mathbf{Val}}$ using the techniques of Chapter 4. Another possibility is to replace all context environments with contexts, i.e. to have $\widehat{\mathbf{Val}} = \mathcal{P}(\mathbf{Term} \times \Delta)$; clearly this will give a lattice of polynomial height. This idea is closely related to the so-called *polynomial k-CFA* analysis where the analogues of context environments are forced to be constant functions, i.e. to map all variables to the same context. In the case of polynomial 1-CFA the analysis is of complexity $O(n^6)$.

Interprocedural analysis revisited. Let us compare the above development with that of Section 2.5 where we considered *interprocedural analysis* for a simple imperative procedure language.

Recall that the abstract domain of interest in Section 2.5 has the form

$$\Delta \to L$$

where Δ is the context information and L is the complete lattice of abstract values of interest. For each label ℓ the analysis will determine two elements $A_\circ(\ell)$ and $A_\bullet(\ell)$ of $\Delta \to L$ describing the situation before and after the elementary block labelled ℓ is executed. So we have

$$A_\circ, A_\bullet : \mathbf{Lab} \to (\Delta \to L)$$

and in the terminology of the present chapter we may regard these functions as abstract caches. There is no analogue of the abstract environment in Section 2.5 – the reason is that the procedure language is so simple that it is not needed: the abstract environment records the context of the free variables and since all free variables in the procedures are global variables there is no need for this component.

We can now reformulate the above development as follows. We can take the abstract domain of interest to be

$$\Delta \to \mathcal{P}(\mathbf{Term} \times \mathbf{CEnv})$$

and reformulate the abstract cache and the abstract environment as having the functionalities:

$$\widehat{\mathrm{C}} : \mathbf{Lab} \to \Delta \to \mathcal{P}(\mathbf{Term} \times \mathbf{CEnv})$$
$$\widehat{\rho} : \mathbf{Var} \to \Delta \to \mathcal{P}(\mathbf{Term} \times \mathbf{CEnv})$$

Thus the abstract caches of the interprocedural analysis and the uniform k-CFA analysis (using call strings) have the same overall functionality and we may conclude that the two analyses are variations over a theme.

3.6.2 The Cartesian Product Algorithm

The *Cartesian Product Algorithm*, abbreviated *CPA*, has been developed for object-oriented languages but the main ideas can be expressed using a variation of our functional language in which functions take m arguments ($m > 0$):

$$t ::= \cdots \mid \texttt{fn}\ x_1, \cdots, x_m \texttt{ => } e_b \mid e_0(e_1, \cdots, e_m)$$

For notational simplicity we shall dispense with recursive functions throughout this subsection. To be faithful to the official description of CPA we shall furthermore impose the well-formedness condition that all function abstractions are closed, i.e. $FV(\texttt{fn}\ x_1, \cdots, x_m \texttt{ => } e_b) = \emptyset$, much as was the case for the procedural language considered in Section 2.5. We leave a more general treatment to Exercise 3.17.

Rephrasing the 0-CFA analysis. The first step is to adapt the specification of the 0-CFA analysis to deal with functions taking m arguments. Even though CPA is a very practical oriented algorithm it will be appropriate to consider the abstract specification in Table 3.1 (rather than the syntax directed specification in Table 3.5). We modify it as follows:

$$(\widehat{\mathsf{C}},\widehat{\rho}) \models (\mathtt{fn}\ x_1,\cdots,x_m \mathtt{=>} e_b)^\ell \text{ iff } \{\mathtt{fn}\ x_1,\cdots,x_m \mathtt{=>} e_b\} \subseteq \widehat{\mathsf{C}}(\ell)$$

$$\begin{array}{l}
(\widehat{\mathsf{C}},\widehat{\rho}) \models (t_0^{\ell_0}(t_1^{\ell_1},\cdots,t_m^{\ell_m}))^\ell \\
\quad \text{iff}\ \ (\widehat{\mathsf{C}},\widehat{\rho}) \models t_0^{\ell_0} \ \wedge\ (\widehat{\mathsf{C}},\widehat{\rho}) \models t_1^{\ell_1} \ \wedge\ \cdots\ \wedge\ (\widehat{\mathsf{C}},\widehat{\rho}) \models t_m^{\ell_m} \ \wedge \\
\qquad \forall(\mathtt{fn}\ x_1,\cdots,x_m \mathtt{=>} t_b^{\ell_b}) \in \widehat{\mathsf{C}}(\ell_0) : \\
\qquad\qquad \widehat{\mathsf{C}}(\ell_1) \times \cdots \times \widehat{\mathsf{C}}(\ell_m) \subseteq \widehat{\rho}(x_1) \times \cdots \times \widehat{\rho}(x_m) \ \wedge \\
\qquad\qquad (\widehat{\mathsf{C}},\widehat{\rho}) \models t_b^{\ell_b} \ \wedge \\
\qquad\qquad \widehat{\mathsf{C}}(\ell_b) \subseteq \widehat{\mathsf{C}}(\ell)
\end{array}$$

The third last conjunct in the clause for function application can be written

$$\widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\rho}(x_1) \wedge \cdots \wedge \widehat{\mathsf{C}}(\ell_m) \subseteq \widehat{\rho}(x_m)$$

in the case where no $\widehat{\mathsf{C}}(\ell_i)$ is empty.

The Cartesian Product Algorithm. We next extend the analysis to take context into account. This will be in the form of the actual arguments supplied to the function:

$$\delta \in \Delta = \mathbf{Term}^m = \mathbf{Term} \times \cdots \times \mathbf{Term} \qquad (m \text{ times})$$

Recalling that $\widehat{\mathbf{Val}} = \mathcal{P}(\mathbf{Term})$ we then redefine the abstract domains as follows:

$$\begin{array}{rcrcl}
\widehat{\rho} & \in & \widehat{\mathbf{Env}} & = & (\mathbf{Var} \times \Delta) \to \widehat{\mathbf{Val}} \\
\widehat{\mathsf{C}} & \in & \widehat{\mathbf{Cache}} & = & (\mathbf{Lab} \times \Delta) \to \widehat{\mathbf{Val}}
\end{array}$$

The key clauses in the CPA analysis then are:

$$(\widehat{\mathsf{C}},\widehat{\rho}) \models^{\delta}_{\mathsf{CPA}} x^\ell \text{ iff } \widehat{\rho}(x,\delta) \subseteq \widehat{\mathsf{C}}(\ell,\delta)$$

$$(\widehat{\mathsf{C}},\widehat{\rho}) \models^{\delta}_{\mathsf{CPA}} (\mathtt{fn}\ x_1,\cdots,x_m \mathtt{=>} e_b)^\ell \text{ iff } \{\mathtt{fn}\ x_1,\cdots,x_m \mathtt{=>} e_b\} \subseteq \widehat{\mathsf{C}}(\ell,\delta)$$

$$\begin{array}{l}
(\widehat{\mathsf{C}},\widehat{\rho}) \models^{\delta}_{\mathsf{CPA}} (t_0^{\ell_0}(t_1^{\ell_1},\cdots,t_m^{\ell_m})^\ell \\
\quad \text{iff}\ \ (\widehat{\mathsf{C}},\widehat{\rho}) \models^{\delta}_{\mathsf{CPA}} t_0^{\ell_0} \ \wedge\ (\widehat{\mathsf{C}},\widehat{\rho}) \models^{\delta}_{\mathsf{CPA}} t_1^{\ell_1} \ \wedge\ \cdots \wedge\ (\widehat{\mathsf{C}},\widehat{\rho}) \models^{\delta}_{\mathsf{CPA}} t_m^{\ell_m} \ \wedge \\
\qquad \forall(\mathtt{fn}\ x_1,\cdots,x_m \mathtt{=>} t_b^{\ell_b}) \in \widehat{\mathsf{C}}(\ell_0,\delta) \\
\qquad\quad \forall\delta_b \in \widehat{\mathsf{C}}(\ell_1,\delta) \times \cdots \times \widehat{\mathsf{C}}(\ell_m,\delta) : \\
\qquad\qquad \{\delta_b\} \subseteq \widehat{\rho}(x_1) \times \cdots \times \widehat{\rho}(x_m) \ \wedge \\
\qquad\qquad (\widehat{\mathsf{C}},\widehat{\rho}) \models^{\delta_b}_{\mathsf{CPA}} t_b^{\ell_b} \ \wedge \\
\qquad\qquad \widehat{\mathsf{C}}(\ell_b,\delta_b) \subseteq \widehat{\mathsf{C}}(\ell,\delta)
\end{array}$$

It is clear from this specification that the body of the function is analysed separately for each possible tuple of arguments and that no merging of data takes place. To be practical we need to implement the analysis using memoisation so that the bodies are only analysed once. This can be done by organising each $(\widehat{\mathsf{C}}, \widehat{\rho}) \models^{\delta_b}_{\mathsf{CPA}} t_b^{\ell_b}$ into so-called *templates* and to maintain them in a global pool; when a template is created it is only added to the pool if it is not already present.

The Cartesian Product Algorithm derives its name from the cartesian product $\widehat{\mathsf{C}}(\ell_1, \delta) \times \cdots \times \widehat{\mathsf{C}}(\ell_m, \delta)$ over which δ_b ranges. The analysis can be implemented in a straightforward *lazy* manner because the product grows monotonically: $\widehat{\mathsf{C}}(\ell_1, \delta) \times \cdots \times \widehat{\mathsf{C}}(\ell_m, \delta)$ will increase each time one of the $\widehat{\mathsf{C}}(\ell_i, \delta)$ increases (assuming that none is empty). A mild generalisation is studied in Exercise 3.17.

Interprocedural analysis revisited. Let us once more compare the development to Section 2.5 but this time to the development based on assumption sets. The development in Section 2.5 is based on

$$A_\circ, A_\bullet : \mathbf{Lab} \to (\Delta \to L)$$

where Δ is the context information and $L = \mathcal{P}(D)$ is the powerset of interest, and the current development can be reformulated as operating on

$$\widehat{\mathsf{C}} : \mathbf{Lab} \to \Delta \to \widehat{\mathbf{Val}}$$
$$\widehat{\rho} : \mathbf{Var} \to \Delta \to \widehat{\mathbf{Val}}$$

where $\widehat{\mathbf{Val}} = \mathcal{P}(\mathbf{Term})$ is the powerset of interest. Clearly D and **Term** can be taken to be equal and the fact that $\Delta = \mathbf{Term}$ then shows that $\Delta = D$ and we may conclude that "small assumption sets" and the Cartesian Product Algorithm are variations over a theme.

Concluding Remarks

Control Flow Analysis for functional languages. Many of the key ideas for Control Flow Analysis have been developed within the context of functional languages. The concept of *k-CFA* analysis seems due to Shivers [156, 157, 158]; other works on 0-CFA-like analyses include [163, 136, 57, 56]. The ideas behind k-CFA and *polynomial k-CFA* analysis were further clarified in [79] that also established the exponential complexity of k-CFA analysis for $k > 0$; it also related a form of *Set Based Analysis* [70] to 0-CFA. The *uniform k-CFA* analyses were introduced in [122] as a simplification of the k-CFA analyses; an obvious variation over this is to record the set of the last k distinct call points (see Exercise 3.15). Yet another variation over the

same theme is *Closure Analysis*; an early and often neglected development may be found in [152].

The formulation of the analyses (as well as the one presented in Table 3.1) would often seem to be more appropriate for a dynamically scoped than for a statically scoped language: The 0-CFA analysis coalesces information about variables having several defining occurrences even if they differ in their scope; clearly the analysis can easily be modified so that it more directly models static scope (see Exercise 3.7) rather than relying on no variable having more than one defining occurrence.

To the extent these developments go beyond 0-CFA they establish additional context (called mementoes, tokens or contours) for representing information concerning the *dynamic* call chain. The most common approach links back to the use of call strings in Section 2.5. The Cartesian Product Algorithm [2] was originally developed for object-oriented programs and amounts to the use of "small assumption sets" in Section 2.5.

Another way to establish context is to represent the *static* call chain. This seems first to be described by [80] as part of their so-called "polymorphic splitting" analysis. A more general set-up was formulated in [122] that also argued for the need to base abstract specifications on coinductive methods – bearing in mind that coinductive and inductive methods may coincide as in the case of syntax directed specifications.

Clearly Control Flow Analysis should be combined with Data Flow Analysis to strengthen the quality of the control information as well as providing the data flow information of interest. Our treatment in Section 3.5 only presents the first steps in this direction: a more ambitious approach is outlined in Mini Project 3.4 which is based in [126]. In fact, some authors would claim that the analysis presented in Section 3.1 should not be regarded as a 0-CFA analysis since it does not include data flow analysis (as in Section 3.5) and does not take evaluation order into account (as in Exercises 3.3 and 3.14); in our view these developments are all variations over a theme.

Most of the papers cited above directly formulate a syntax directed specification, perhaps proving it semantically sound, and perhaps showing how to generate constraints so as to obtain an efficient implementation. The use of abstract specifications first appeared in [80, 122] and has the advantage of being more directly applicable to *open systems* (that allow to interface with the library routines provided by the environment) and also to the ideas of Abstract Interpretation of Chapter 4. In particular, the notion of reachability suggests itself rather naturally [21, 60], it becomes clearer how to integrate ideas from Abstract Interpretation into Control Flow Analysis, and one does not inadvertently restrict oneself to *closed systems* only. (The notion of reachability is considered in Mini Project 3.1 which is based on [60].)

Only few papers [125] discuss the interplay between the choice of specification style for the analysis and the choice of semantics. We have used a small-step Structural Operational Semantics rather than a big-step semantics in order to express the semantic correctness also of looping programs. We have used an environment based semantics in order to ensure that we do not "modify" the bodies of functions before they are called, so that function abstractions can meaningfully be used in the value domains of our analysis [125]. As a consequence we have had to introduce intermediate expressions (closures and bindings) and have had to specify the abstract analysis also for intermediate expressions; for the syntax directed specification and the constraint based analysis this was not necessary given that semantic correctness had already been dealt with. Alternative choices are clearly possible but are likely to sacrifice at least some of the generality offered by the present approach.

Control Flow Analysis for other language paradigms. Another main application of Control Flow Analysis has been for *object-oriented languages*: one simply tracks objects rather than functions [3, 124, 137]. As a reminder of the close links between Data Flow Analysis and Control Flow Analysis we should also mention that some approaches [178, 139] are closer to the presentation of Chapter 2. A common theme among the more advanced studies is the incorporation of context (related to k-CFA) and an abstract store [133] (to deal with imperative aspects like method update). To increase the precision, local versions of the abstract store need to exist at all program points, and abstract reference counts are needed to incorporate a "kill" component (in the manner of Chapter 2). We refer to the above literature for further details of how to formulate such analyses and how to choose a proper balance between precision and cost.

Control Flow Analysis for *concurrent languages* has received relatively little attention [22]. However, variations of the techniques presented in this chapter have been used to analyse functional languages extended with concurrency primitives allowing processes to be created dynamically and to communicate via shared locations or channels [78, 57, 60, 22, 23].

In this book we do not consider *logic programming languages*. However, we should point out that Control Flow Analysis also has applications for logic programming languages and that set based analysis was first developed for this class of languages [72, 73].

Set-Constraint Based Analysis. Control Flow Analysis is just one approach to program analysis where the use of constraints pays off. In this chapter we have have taken the following approach: (i) first we have given an abstract specification of when a proposed solution is acceptable, (ii) then we have developed an algorithm for generating a set of constraints expressing that a proposed solution is acceptable, and (iii) finally we have solved the set of constraints for the least solution. For the solution of set constraints in

step (iii), it is unimportant how the constraints were in fact obtained. For this reason, it is often said that set constraints allow the separation of the specification of an analysis from its implementation and that set constraints are able to deal with forward analyses as well as backward analyses and indeed mixtures of these.

Set constraints [72, 7] have a long history [143, 84]. They allow us to express general inclusions of the form

$$S_1 \subseteq S_2$$

where set expressions, S, set variables, V, and set constructors, C, may be built as follows:

$$\begin{array}{rcl} S & ::= & V \mid \emptyset \mid S_1 \cup S_2 \mid S_1 \cap S_2 \mid C(S_1, \cdots, S_n) \\ & \mid & (S_1 \subseteq S_2) \Rightarrow S_3 \mid (S_1 \neq \emptyset) \Rightarrow S_2 \mid C^{-i}(S) \mid \neg S \mid \cdots \\ V & ::= & X \mid Y \mid \cdots \\ C & ::= & \texttt{true} \mid \texttt{false} \mid 0 \mid \cdots \mid \texttt{cons} \mid \texttt{nil} \mid \cdots \end{array}$$

Set constraints allow the consideration of constructors that are not just nullary and this allows us to record also the shape of data structures, so for example $\texttt{cons}(S_1, S_2) \cup \texttt{nil}$ expresses possibly empty lists whose heads come from S_1 and whose tails come from S_2. The associated projection selects those terms (if any) having the required shape, e.g. $\texttt{cons}^{-1}(S)$ produces the heads that may be present in S. We have seen conditional constraints before and it turns out that projection is so powerful that it can be used to code conditional constraints. Finally, it is sometimes possible to explicitly take the complement of a set but this adds to the complexity of the development. (It means that solutions can no longer be guaranteed using Tarski's Theorem and sometimes a version of Banach's Theorem may be used instead.)

The complexity of solving a system of set constraints depends rather dramatically on the set forming operations allowed and therefore many versions have been considered in the literature. We refer to [6, 135] for an overview of what is known in this area; here we just mention [28] for a general result and [70, 10] for some cubic time fragments.

However, it is worth pointing out that many of these results are worst-case results; benchmark results of Jaganathan and Wright [80] shows e.g. that in practice a 1-CFA analysis may be faster than a 0-CFA analysis despite the fact that the former has exponential worst-case complexity and the latter polynomial worst-case complexity. The reason seems to be that the 0-CFA analysis explores most of its polynomial sized state space whereas the 1-CFA analysis is so precise that it only explores a fraction of its exponentially sized state space.

The basic idea behind many of the solution procedures for set constraints is roughly as follows [9]:

1. Dynamically expand conditional constraints, based on the condition being fulfilled, until no more expansion is possible.
2. Remove all conditional constraints and combine the remaining constraints to obtain the least solution.

This is not quite the algorithm used in Section 3.4 where we were only interested in solving a rather limited class of constraints for 0-CFA analysis. When generating the constraints in Table 3.6 we were able to "guess a universe" $\mathbf{Term}_\star$ that was sufficiently large and this allowed us to generate explicit versions of the conditional constraints; in fact a superset of all those to be considered in step 1 of the above algorithm. Therefore our subsequent constraint solving algorithm in Table 3.7 merely needed to check the already existing constraints and to determine whether or not they could contribute to the solution. In practice, the above "lazy" algorithm is likely to perform much better than the "eager" algorithm of Tables 3.6 and 3.7.

Mini Projects

Mini Project 3.1 Reachability Analysis

The syntax directed analysis of Table 3.5 analyses each subexpression of $e_\star$ "exactly once" rather than "at most once" as really called for. In this mini project we shall study one way to amend this.

The idea is to introduce an *abstract reachability component*

$$\widehat{\mathsf{R}} \in \mathbf{Reach} = \mathbf{Lab} \to \mathcal{P}(\{\mathbf{on}\})$$

and to modify the syntax directed analysis to have a relation of the form

$$(\widehat{\mathsf{C}}, \widehat{\rho}, \widehat{\mathsf{R}}) \models'_s e$$

The idea is that $\mathtt{fn}\ x\ \mathtt{=>}\ t_0^{\ell_0}$ has $\{\mathbf{on}\} \subseteq \widehat{\mathsf{R}}(\ell_0)$ if and only if the function is indeed applied somewhere, and that the "recursive call" $(\widehat{\mathsf{C}}, \widehat{\rho}, \widehat{\mathsf{R}}) \models_s t_0^{\ell_0}$ is performed if and only if $\{\mathbf{on}\} \subseteq \widehat{\mathsf{R}}(\ell_0)$.

1. Modify Table 3.5 to incorporate this idea.
2. Show the following analogue of Proposition 3.18: If $(\widehat{\mathsf{C}}, \widehat{\rho}, \widehat{\mathsf{R}}) \models'_s t_\star^{\ell_\star}$, $\{\mathbf{on}\} \subseteq \widehat{\mathsf{R}}(\ell_\star)$ and $(\widehat{\mathsf{C}}, \widehat{\rho}) \sqsubseteq (\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top)$ then $(\widehat{\mathsf{C}}, \widehat{\rho}) \models t_\star^{\ell_\star}$.
3. Determine whether or not the statements

   if $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$ then $(\widehat{\mathsf{C}}, \widehat{\rho}, \widehat{\mathsf{R}}) \models'_s e_\star$ for some $\widehat{\mathsf{R}}$

   if $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$ and $(\widehat{\mathsf{C}}, \widehat{\rho}) \sqsubseteq (\widehat{\mathsf{C}}_\star^\top, \widehat{\rho}_\star^\top)$ then $(\widehat{\mathsf{C}}, \widehat{\rho}, \widehat{\mathsf{R}}) \models'_s e_\star$ for some $\widehat{\mathsf{R}}$

   hold in general. ■

Mini Project 3.2 Data Structures

The language considered so far only includes simple data like integers and booleans. In this mini project we shall extend the language with more general data structures:

$$e ::= \cdots \mid C(e_1, \cdots, e_n)^\ell \mid (\texttt{case}\ e_0\ \texttt{of}\ C(x_1, \cdots, x_n)\ \texttt{=>}\ e_1\ \texttt{or}\ x\ \texttt{=>}\ e_2)^\ell$$

Here $C \in \mathbf{Constr}$ denotes an n-ary data constructor. A data element is constructed by $C(e_1, \cdots, e_n)$: it has tag C and its components are the values of $e_1, \cdots, e_n$. The case construct will first determine the value v_0 of e_0, if v_0 has the tag C then $x_1, \cdots, x_n$ will be bound to the components of v_0 and e_1 is evaluated. If v_0 does not have tag C then x is bound to v_0 and e_2 is evaluated.

As an example we may have $\mathbf{Constr} = \{\texttt{cons}, \texttt{nil}\}$ so we have the following expression (omitting labels) for reversing a list:

```
let append = fun app xs => fn ys =>
             case xs of cons(z,zs) => cons(z,app zs ys)
                     or xs => ys
in fun rev xs => case xs of cons(y,ys) =>
                            append (rev ys) (cons(y,nil()))
                         or xs => nil()
```

To specify a 0-CFA analysis for this language we shall take

$$\widehat{\mathbf{Val}} = \mathcal{P}(\mathbf{Term} \cup \{C(\ell_1, \cdots, \ell_n) \mid C \in \mathbf{Constr}, \ell_1, \cdots, \ell_n \in \mathbf{Lab}\})$$

As before the terms of interest are $\texttt{fn}\ x\ \texttt{=>}\ e_0$ and $\texttt{fun}\ f\ x\ \texttt{=>}\ e_0$ for recording the abstractions. The new contribution is a number of elements of the form $C(\ell_1, \cdots, \ell_n)$ denoting a data element $C(v_1, \cdots, v_n)$ whose i'th component might have been created by the expression at program point ℓ_i.

1. Develop a syntax directed analogue of the analysis in Table 3.5.

2. Modify the constraint generation algorithm of Table 3.6 to handle the new constructs and make the necessary changes to the constraint solving algorithm of Table 3.7.

For the more ambitious: are there any difficulties in developing an abstract analogue of the analysis in Table 3.1? ■

Mini Project 3.3 A Prototype Implementation

In this mini project we shall implement the pure 0-CFA analysis considered in Section 3.3. As implementation language we shall choose a functional language such as Standard ML or Haskell. We can then define a suitable data type for FUN expressions as follows:

$$
\begin{array}{lcl}
\textsf{type } \textit{var} & = & \textsf{string} \\
\textsf{type } \textit{label} & = & \textsf{int} \\
\textsf{datatype } \textit{const} & = & \textsf{Num of int} \mid \textsf{True} \mid \textsf{False} \\
\textsf{datatype } \textit{exp} & = & \textsf{Label of } \textit{term} * \textit{label} \\
\textsf{and } \textit{term} & = & \textsf{Const of } \textit{const} \mid \textsf{Var of } \textit{var} \\
 & \mid & \textsf{Fn of } \textit{var} * \textit{exp} \mid \textsf{Fun of } \textit{var} * \textit{var} * \textit{exp} \\
 & \mid & \textsf{App of } \textit{exp} * \textit{exp} \mid \textsf{If of } \textit{exp} * \textit{exp} * \textit{exp} \\
 & \mid & \textsf{Let of } \textit{var} * \textit{exp} * \textit{exp} \mid \textsf{Op of string} * \textit{exp} * \textit{exp}
\end{array}
$$

Now proceed as follows:

1. Implement the constraint based control flow analysis of Section 3.4; this includes defining an appropriate data structure *constraints* for (conditional) constraints.

2. Implement the graph based algorithm of Section 3.4 for solving constraints; this involves choosing appropriate data structures for the worklist and the two arrays used by the algorithm.

For the more ambitious: generalise your program to perform some of the more advanced analyses, e.g. by incorporating data flow information or context information. ■

Mini Project 3.4 Imperative Constructs

In Section 3.5 we showed how to incorporate Data Flow Analysis into our Control Flow Analysis; this becomes more challenging when imperative constructs are added to the language:

$$e ::= \cdots \mid (\texttt{new}_\pi\ x\ \texttt{:=}\ e_1\ \texttt{in}\ e_2)^\ell \mid (!\ x)^\ell \mid (x\ \texttt{:=}\ e_0)^\ell \mid (e_1\ ;\ e_2)^\ell$$

Here $\texttt{new}_\pi\ x\ \texttt{:=}\ e_1\ \texttt{in}\ e_2$ creates a new reference variable x to be used in e_2; it is initialised to e_1, its content is obtained by $!\ x$, and it may be updated by $x\ \texttt{:=}\ e_0$. The creation point for the reference variable is indicated by the program point $\pi \in \mathbf{Pnt}$ and the construct $e_1\ ;\ e_2$ merely sequences e_1 and e_2 (and is equivalent to $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ when x does not occur in e_2).

One approach to extending the development of Section 3.5 might be to work with an acceptability relation of the form

$$(\widehat{\mathsf{C}}, \widehat{\rho}, \widehat{\mathsf{S}}_\circ, \widehat{\mathsf{S}}_\bullet) \models_{me} e$$

where

- $\widehat{\mathsf{C}} : \mathbf{Lab} \to \widehat{\mathbf{Val}}$ and $\widehat{\mathsf{C}}(\ell)$ describes the values that the subexpression labelled ℓ may evaluate to,
- $\widehat{\rho} : \mathbf{Var} \to \widehat{\mathbf{Val}}$ and $\widehat{\rho}(x)$ describes the values that x might be bound to,
- $\widehat{\mathsf{S}}_\circ : \mathbf{Lab} \to (\mathbf{Pnt} \to \widehat{\mathbf{Val}})$ and $\widehat{\mathsf{S}}_\circ(\ell)$ describes the states that may be possible *before* the subexpression labelled ℓ is evaluated,
- $\widehat{\mathsf{S}}_\bullet : \mathbf{Lab} \to (\mathbf{Pnt} \to \widehat{\mathbf{Val}})$ and $\widehat{\mathsf{S}}_\bullet(\ell)$ describes the states that may be possible *after* the subexpression labelled ℓ is evaluated, and
- $me : \mathbf{Var} \to \mathbf{Pnt}$ and $me(x)$ indicates the creation point for the reference variable x.

One choice of $\widehat{\mathbf{Val}}$ is $\mathcal{P}(\mathbf{Term}) \times L$ where the first component tracks functions and the second component tracks abstract values. It may be helpful to devise a Structural Operational Semantics for the extended language and to use it as a guide when defining the acceptability relation.

A more advanced treatment would involve context information in the manner of Section 3.6. ■

Exercises

Exercise 3.1 Consider the following expression (omitting labels):

```
let f = fn x => x 1
in  let g = fn y => y+2
    in  let h = fn z => z+3
        in  (f g) + (f h)
```

Add labels to the program and guess an analysis result. Use Table 3.1 to verify that it is indeed an acceptable guess. ■

Exercise 3.2 The specification of the Control Flow Analysis in Table 3.1 uses potentially infinite value spaces and this is not really necessary. To see this choose some expression $e_\star \in \mathbf{Exp}$ that is to be analysed. Let $\mathbf{Var}_\star \subseteq \mathbf{Var}$ be the finite set of variables occurring in $e_\star$, let $\mathbf{Lab}_\star \subseteq \mathbf{Lab}$ be the finite set of labels occurring in $e_\star$, and let $\mathbf{Term}_\star$ be the finite set of subterms of $e_\star$. Next define

$$\begin{array}{rcll} \widehat{v} & \in & \widehat{\mathbf{Val}}_\star & = \mathcal{P}(\mathbf{Term}_\star) \\ \widehat{\rho} & \in & \widehat{\mathbf{Env}}_\star & = \mathbf{Var}_\star \to \widehat{\mathbf{Val}}_\star \\ \widehat{\mathsf{C}} & \in & \widehat{\mathbf{Cache}}_\star & = \mathbf{Lab}_\star \to \widehat{\mathbf{Val}}_\star \end{array}$$

and note that these value spaces are finite. Show that the specification of the analysis in Table 3.1 still makes sense when $(\widehat{\mathsf{C}}, \widehat{\rho})$ is restricted to be in $\widehat{\mathbf{Cache}}_\star \times \widehat{\mathbf{Env}}_\star$. ■

Exercise 3.3 Modify the Control Flow Analysis of Table 3.1 to take account of the left to right evaluation order imposed by a call-by-value semantics: in the clause [*app*] there is no need to analyse the operand if the operator cannot produced any closures. Try to find a program where the modified analysis accepts analysis results $(\widehat{\mathsf{C}}, \widehat{\rho})$ rejected by Table 3.1. ■

Exercise 3.4 So far we have defined "$(\widehat{\mathsf{C}}, \widehat{\rho})$ is an acceptable solution for e" to mean that

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models e \tag{3.8}$$

but an alternative condition is that

$$\exists (\widehat{\mathsf{C}}', \widehat{\rho}') : (\widehat{\mathsf{C}}', \widehat{\rho}') \models e \ \wedge \ (\widehat{\mathsf{C}}', \widehat{\rho}') \sqsubseteq (\widehat{\mathsf{C}}, \widehat{\rho}) \tag{3.9}$$

Show that (3.8) implies (3.9) but not vice versa. Discuss which of (3.8) or (3.9) is the preferable definition. ■

Exercise 3.5 Consider an alternative specification of the analysis in Table 3.1 where the condition

$$(\texttt{fun}\ f\ x \texttt{ => } t_0^{\ell_0}) \in \widehat{\rho}(f)$$

in [*app*] is replaced by

$$\widehat{\mathsf{C}}(\ell_1) \subseteq \widehat{\rho}(f)$$

also in [*app*]. Show that the proof of Theorem 3.10 can be modified accordingly. Discuss the relative precision of the two analyses. ■

Exercise 3.6 Reconsider our decision to use $\widehat{\mathbf{Val}} = \mathcal{P}(\mathbf{Term})$ and consider using $\widehat{\mathbf{Val}} = \mathcal{P}(\mathbf{Exp})$ instead. Show that the specification of the Control Flow Analysis may be modified accordingly but that then Fact 3.11 (and hence the correctness result) would fail. ■

Exercise 3.7 The operational semantics allow us to rename bound variables without changing the semantics; this is in accord with the language being statically scoped (or lexically scoped) rather than dynamically scoped. As an example

$$((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5 =_\alpha ((\texttt{fn x => x}^1)^2\ (\texttt{fn x => x}^3)^4)^5$$

and clearly the two programs have the same semantics.

However, renaming bound variables changes the acceptability of a solution as well as influences the precision of the analysis specified in Table 3.1. Develop an abstract specification of a 0-CFA analysis that is more faithful to the static scoping than that of Table 3.1; it should agree with the specification of Table 3.1 for expressions that do not have multiple defining occurrences. ■

Exercise 3.8 In Section 3.2 we equipped FUN with a call-by-value semantics. An alternative would be to use a *call-by-name* or *lazy* semantics. It can be obtained as a simple modification of the semantics of Tables 3.2 and 3.3 by allowing the environments $\rho \in \mathbf{Env}$ to map variables to intermediate terms (and not just values), by deleting the rules $[app_2]$ and $[let_1]$ and then make some obvious modifications to the axioms $[var]$, $[app_{fn}]$, $[app_{fun}]$ and $[let_2]$; in the case of $[var]$ we will take:

$$\rho \vdash x^\ell \rightarrow it^\ell \quad \text{if } x \in dom(\rho) \text{ and } it = \rho(x)$$

Complete the specification of the semantics and show that the correctness result (Theorem 3.10) still holds for the analysis of Table 3.1.

What does that tell us about the precision of the analysis? ■

Exercise 3.9 Let $\models'_s$ and $\models''_s$ be two relations satisfying the specification of Table 3.5. Show that

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models'_s\ e \text{ iff } (\widehat{\mathsf{C}}, \widehat{\rho}) \models''_s\ e$$

by structural induction on e. ■

Exercise 3.10 Consider Proposition 3.18 and determine whether or not the statement

$$\text{if } (\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star \text{ then } (\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$$

holds in general. ■

Exercise 3.11 Give an example showing that both of the statements

$$\text{if } (\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star \text{ then } (\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star$$
$$\text{if } (\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star \text{ and } (\widehat{\mathsf{C}}, \widehat{\rho}) \sqsubseteq (\widehat{\mathsf{C}}^\top_\star, \widehat{\rho}^\top_\star) \text{ then } (\widehat{\mathsf{C}}, \widehat{\rho}) \models_s e_\star$$

fail in general. ■

Exercise 3.12 Give a direct proof of the correctness of the syntax directed analysis of Table 3.5, i.e. establish an analogue of Theorem 3.10. This involves first extending the syntax directed analysis to the `bind`- and `close`-constructs and next proving that if $\rho \mathcal{R} \widehat{\rho}$, $\rho \vdash ie \rightarrow ie'$ and $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s ie$ then also $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_s ie'$. ■

Exercise 3.13 Consider the system $\mathcal{C}_\star^=[\![e_\star]\!]$ that contains a constraint

$$ls_1 \cup \cdots \cup ls_n = rhs$$

whenever $\mathcal{C}_\star[\![e_\star]\!]$ contains the $n \geq 1$ constraints

$$ls_i \subseteq rhs$$

Show that

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c \mathcal{C}_\star^=[\![e_\star]\!] \text{ implies } (\widehat{\mathsf{C}}, \widehat{\rho}) \models_c \mathcal{C}_\star[\![e_\star]\!]$$

where $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c (ls = rhs)$ is defined in the obvious way. Also show that

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c \mathcal{C}_\star[\![e_\star]\!] \text{ implies } (\widehat{\mathsf{C}}, \widehat{\rho}) \models_c \mathcal{C}_\star^=[\![e_\star]\!]$$

holds in the special case where $(\widehat{\mathsf{C}}, \widehat{\rho})$ is *least* such that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models_c \mathcal{C}_\star[\![e_\star]\!]$. ■

Exercise 3.14 Use the ideas of Exercise 3.3 to develop an improvement of Table 3.8 where expressions are only analysed when absolutely needed. Next develop a syntax directed analysis using the same ideas. Discuss the relationship between the two specifications: are they more closely related than is the case for $\models$ and $\models_s$ of Table 3.1 and 3.5 (see Exercises 3.10 and 3.11)? ■

Exercise 3.15 Modify the abstract specification of the uniform k-CFA analysis so that it does not record the last k function calls but the last k times we called a different function than in the preceding call: if the calling sequence is [1,2,2,1,1] then 2-CFA records [1,1] but the modified analysis records [2,1]. Discuss which of the two analyses (say for $k = 2$) is likely to be most useful in practice. ■

Exercise 3.16 Let us consider a language of first-order recursion equation schemes: the programs have the form

$$\texttt{define}\ D_\star\ \texttt{in}\ e_\star$$

where $D_\star$ is a sequence of function definitions of the form:

$$f(x) = e$$

Here f is a function name, x is the formal parameter and e is the body of the function; the functions defined in D may be mutually recursive and the parameter mechanism is call-by-value. The expressions are given by

$$\begin{array}{rcl} e & ::= & t^\ell \\ t & ::= & c \mid x \mid f\ e \mid \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid e_1\ op\ e_2 \end{array}$$

where $c \in \mathbf{Const}$ and $op \in \mathbf{Op}$ as before; we shall assume that f and x belong to distinct syntactic categories. As an example we may define the Fibonacci function by the following expression (omitting labels):

```
define fib(z) = if z<3 then 0
                else fib (z-1) + fib (z-2)
in      fib x
```

Define a uniform k-CFA analysis for this language. For $k = 0$ and $k = 1$ compare the development with that for the procedure language in Section 2.5. ■

Exercise 3.17 In Subsection 3.6.2 we covered the Cartesian Production Algorithm based on the assumption that all function abstractions are closed. In this exercise we do not make this simplifying assumption and also we wish to deal with recursive functions. Develop an abstract specification of an analysis

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models^{ce}_{\delta} e$$

where $\delta \in \Delta$ is as in Subsection 3.6.2 and

$$ce \in \mathbf{CEnv} = \mathbf{Var} \to \Delta$$

Hint: Modify Table 3.10 to deal with functions taking m arguments and to perform the appropriate determination of new contexts in the clause for function application. ■

Chapter 4

Abstract Interpretation

The purpose of this chapter is to convey some of the essential ideas of Abstract Interpretation. We shall mainly do so in a programming language independent way and thus focus on the design of the property spaces, the functions and computations upon them, and the relationships between them.

We first formulate a notion of correctness for a restricted class of analyses as this will allow us to motivate better some of the key definitions in the development. Then we cover the widening and narrowing techniques that can be used to obtain approximations of the least fixed point and for limiting the number of computation steps needed. Next we consider Galois connections and Galois insertions that allow a costly space of properties to be replaced with one that is less costly. Galois connections can be constructed in a systematic way and can be used to induce one specification of an analysis from another.

4.1 A Mundane Approach to Correctness

To set the scene, imagine some programming language. Its *semantics* identifies some set V of values (like states, closures, double precision reals) and specifies how a program p transforms one value v_1 to another v_2; we may write

$$p \vdash v_1 \rightsquigarrow v_2 \tag{4.1}$$

for this without committing ourselves to the details of the semantics and without necessarily imposing determinacy (that $p \vdash v_1 \rightsquigarrow v_2$ and $p \vdash v_1 \rightsquigarrow v_3$ imply $v_2 = v_3$).

In a similar way, a *program analysis* identifies the set L of properties (like shapes of states, abstract closures, lower and upper bounds for reals) and

specifies how a program p transforms one property l_1 to another l_2; we may write

$$p \vdash l_1 \rhd l_2 \tag{4.2}$$

for this without committing ourselves to the method used for specification of the analysis. However, unlike what is the case for the semantics, it is customary to require $\rhd$ to be deterministic and thereby define a function; this will allow us to write $f_p(l_1) = l_2$ to mean $p \vdash l_1 \rhd l_2$.

In the rest of this section we shall show how to relate the semantics to the analysis. We shall present two approaches based on correctness relations and representation functions, respectively. In both cases we shall define a notion of correctness of the analysis with respect to the semantics and we shall show that the two notions are equivalent.

This is a mundane approach in the sense that it only applies to analyses where properties directly describe sets of values. This is the case for the Constant Propagation Analysis of Section 2.3, the Shape Analysis of Section 2.6 and the Control Flow Analysis of Chapter 3 but it is not the case for the Live Variable Analysis of Section 2.1 where properties are related to relations between values. In the literature, the terms *first-order analyses* versus *second-order analyses* have been used to differentiate between these classes of analyses. It is important to stress that the development of Sections 4.2 to 4.5 apply equally well to both classes.

We begin by showing how the development of Chapters 2 and 3 can be rephrased in the style of (4.1) and (4.2).

Example 4.1 Consider the WHILE language of Chapter 2. Recall that the semantics is a Structural Operational Semantics with transitions of the forms $\langle S, \sigma\rangle \to \langle S', \sigma'\rangle$ and $\langle S, \sigma\rangle \to \sigma'$, where S and S' are statements of **Stmt** and σ and σ' are states of $\mathbf{State} = \mathbf{Var} \to \mathbf{Z}$. With $S_\star$ being the program of interest we shall now write

$$S_\star \vdash \sigma_1 \rightsquigarrow \sigma_2$$

for the reflexive transitive closure of the transition relation, i.e. for:

$$\langle S_\star, \sigma_1\rangle \to^* \sigma_2$$

Note that the set V of values is the set **State**.

We shall now consider the *Constant Propagation Analysis* of Section 2.3. Recall that the analysis of $S_\star$ gives rise to a set of equations $\mathsf{CP}^=$ formulated in terms of an instance of a Monotone Framework: the properties L of interest are given by $\widehat{\mathbf{State}}_{\mathsf{CP}} = (\mathbf{Var}_\star \to \mathbf{Z}^\top)_\perp$, E is $\{\mathit{init}(S_\star)\}$, F is $\mathit{flow}(S_\star)$, and ι is $\lambda x.\top$. Further recall that a solution to the equations is a pair $(\mathsf{CP}_\circ, \mathsf{CP}_\bullet)$

of mappings $\mathsf{CP}_\circ : \mathbf{Lab}_\star \to \widehat{\mathbf{State}}_{\mathsf{CP}}$ and $\mathsf{CP}_\bullet : \mathbf{Lab}_\star \to \widehat{\mathbf{State}}_{\mathsf{CP}}$ satisfying the equations. Given a solution $(\mathsf{CP}_\circ, \mathsf{CP}_\bullet)$ to $\mathsf{CP}^=$ we take

$$S_\star \vdash \widehat{\sigma}_1 \rhd \widehat{\sigma}_2$$

to mean that:

$$\iota = \widehat{\sigma}_1 \ \wedge \ \widehat{\sigma}_2 = \bigsqcup\{\mathsf{CP}_\bullet(\ell) \mid \ell \in \mathit{final}(S_\star)\}$$

Thus for a program $S_\star$ with isolated entries, $\widehat{\sigma}_1$ is the abstract state associated with the entry point of $S_\star$, and $\widehat{\sigma}_2$ is the abstract state associated with the exit points; we use the least upper bound operation ($\sqcup$) on the complete lattice $\widehat{\mathbf{State}}_{\mathsf{CP}}$ to combine the contributions from the (possibly several) exit points of $S_\star$. ■

Example 4.2 Consider the FUN language of Chapter 3. Recall that the semantics is given by a Structural Operational Semantics with transitions of the form $\rho \vdash ie \to ie'$ where ρ is an environment (an element of $\mathbf{Env} = \mathbf{Var} \to_{\mathrm{fin}} \mathbf{Val}$) and ie and ie' are intermediate expressions from $\mathbf{IExp}$. Now let $e_\star$ be the closed expression of interest. We shall write

$$e_\star \vdash v_1 \rightsquigarrow v_2$$

to mean that $e_\star$ when given the argument v_1 will evaluate to the value v_2, i.e. that

$$[\,] \vdash (e_\star \ v_1^{\ell_1})^{\ell_2} \to^* v_2^{\ell_2}$$

where ℓ_1 and ℓ_2 are fresh labels. Note that the set V of values now is the set $\mathbf{Val}$.

We shall next consider the pure *Control Flow Analysis* of Section 3.1. Recall that the result of analysing the expression $e_\star$ is a pair $(\widehat{\mathsf{C}}, \widehat{\rho})$ satisfying $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_\star$ as defined in Table 3.1. Here $\widehat{\mathsf{C}}$ is an element of $\widehat{\mathbf{Cache}} = \mathbf{Lab}_\star \to \widehat{\mathbf{Val}}$ and $\widehat{\rho}$ is an element of $\widehat{\mathbf{Env}} = \mathbf{Var}_\star \to \widehat{\mathbf{Val}}$ where $\widehat{\mathbf{Val}} = \mathcal{P}(\mathbf{Term}_\star)$. For this analysis we shall take the properties L of interest to be pairs $(\widehat{\rho}, \widehat{v})$ of $\widehat{\mathbf{Env}} \times \widehat{\mathbf{Val}}$ and assume that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models (e_\star \ \mathsf{c}^{\ell_1})^{\ell_2}$ for some constant c. Then we define

$$e_\star \vdash (\widehat{\rho}_1, \widehat{v}_1) \rhd (\widehat{\rho}_2, \widehat{v}_2)$$

to mean that when $e_\star$ is given an argument with property $(\widehat{\rho}_1, \widehat{v}_1)$ then the result of the application will have property $(\widehat{\rho}_2, \widehat{v}_2)$:

$$\widehat{\mathsf{C}}(\ell_1) = \widehat{v}_1 \ \wedge \ \widehat{\mathsf{C}}(\ell_2) = \widehat{v}_2 \ \wedge \ \widehat{\rho}_1 = \widehat{\rho}_2 = \widehat{\rho}$$

Note that the "dummy" constant c used as an argument to $e_\star$ is used as a place holder for all potential arguments being described by $\widehat{v}_1$; for this idea to work it is important that the analysis of c puts no constraints on $(\widehat{\mathsf{C}}, \widehat{\rho})$ as in indeed the case for the specification in Table 3.1. ■

4.1.1 Correctness Relations

Every program analysis should be correct with respect to the semantics. For a class of (so-called first-order) program analyses this is established by directly relating properties to values using a *correctness relation*:

$$R : V \times L \to \{\mathit{true}, \mathit{false}\}$$

The intention is that $v\ R\ l$ formalises our claim that the value v is described by the property l.

Correctness formulation. To be useful one has to prove that the correctness relation R is preserved under computation: if the relation holds between the initial value and the initial property then it also holds between the final value and the final property. This may be formulated as the implication

$$v_1\ R\ l_1\ \wedge\ p \vdash v_1 \rightsquigarrow v_2\ \wedge\ p \vdash l_1 \rhd l_2\ \Rightarrow\ v_2\ R\ l_2 \tag{4.3}$$

and is also expressed by the following diagram:

$$\begin{array}{cccc}
p\ \vdash & v_1 & \rightsquigarrow & v_2 \\
 & \vdots & & \vdots \\
 & R & \Rightarrow & R \\
 & \vdots & & \vdots \\
p\ \vdash & l_1 & \rhd & l_2
\end{array}$$

A relation R satisfying a condition like this is often called a *logical relation* and the implication is sometimes written $(p \vdash \cdot \rightsquigarrow \cdot)(R \twoheadrightarrow R)(p \vdash \cdot \rhd \cdot)$.

The theory of Abstract Interpretation comes to life when we augment the set of properties L with a preorder structure and relate this to the correctness relation R. The most common scenario is when $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a complete lattice with partial ordering $\sqsubseteq$ (see Appendix A). We then impose the following relationship between R and L:

$$v\ R\ l_1\ \wedge\ l_1 \sqsubseteq l_2\ \Rightarrow\ v\ R\ l_2 \tag{4.4}$$

$$(\forall l \in L' \subseteq L : v\ R\ l)\ \Rightarrow\ v\ R\ (\bigsqcap L') \tag{4.5}$$

Condition (4.4) says that the smaller the property is with respect to the partial ordering, the better (i.e. more precise) it is. This is an "arbitrary" decision in the sense that we could instead have decided that the larger the property is, the better it is, as is indeed the case in much of the literature on

Data Flow Analysis; luckily the principle of duality from lattice theory (see the Concluding Remarks) tells us that this difference is only a cosmetic one.

Condition (4.5) says that there is always a best property for describing a value. This is important for having to perform only one analysis (using the best property, i.e. the greatest lower bound of the candidates) instead of several analyses (one for each of the candidates). Recall from Appendix A that a subset Y of L is a *Moore family* if and only if $(\bigsqcap Y') \in Y$ for all subsets Y' of Y. We can then see that condition (4.5) is equivalent to the demand that $\{l \mid v\ R\ l\}$ is a Moore family.

Condition (4.5) has two immediate consequences:

$$v\ R\ \top$$

$$v\ R\ l_1\ \wedge\ v\ R\ l_2\ \ \Rightarrow\ \ v\ R\ (l_1 \sqcap l_2)$$

The first formula says that $\top$ describes any value and the second formula says that if we have two descriptions of a value then their greatest lower bound is also a description of the value.

Example 4.3 Returning to the Constant Propagation Analysis of Example 4.1 we can now specify the relation

$$R_{\mathsf{CP}} : \mathbf{State} \times \widehat{\mathbf{State}}_{\mathsf{CP}} \rightarrow \{\mathit{true}, \mathit{false}\}$$

between the values (i.e. the states) and the properties (i.e. the abstract states):

$$\sigma\ R_{\mathsf{CP}}\ \widehat{\sigma}\ \ \text{iff}\ \ \forall x \in FV(S_\star) : (\widehat{\sigma}(x) = \top\ \vee\ \sigma(x) = \widehat{\sigma}(x))$$

Thus $\widehat{\sigma}$ may map some variables to $\top$ but if $\widehat{\sigma}$ maps a variable x to an element in $\mathbf{Z}$ then this must also be the value of $\sigma(x)$.

Let us observe that the conditions (4.4) and (4.5) are fulfilled by the Constant Propagation Analysis. Recall from Section 2.3 that $(\widehat{\mathbf{State}}_{\mathsf{CP}}, \sqsubseteq_{\mathsf{CP}})$ is a complete lattice with the ordering $\sqsubseteq_{\mathsf{CP}}$. It is then straightforward to verify that (4.4) and (4.5) do indeed hold. (Also compare with Exercise 2.7.) ∎

Example 4.4 For the Control Flow Analysis mentioned in Example 4.2 we shall define

$$R_{\mathsf{CFA}} : \mathbf{Val} \times (\widehat{\mathbf{Env}} \times \widehat{\mathbf{Val}}) \rightarrow \{\mathit{true}, \mathit{false}\}$$

to be the relation $\mathcal{V}$ of Section 3.2:

$$v\ R_{\mathsf{CFA}}\ (\widehat{\rho}, \widehat{v})\ \ \text{iff}\ \ v\ \mathcal{V}\ (\widehat{\rho}, \widehat{v})$$

Recall that we have two kinds of values $v \in \mathbf{Val}$, constants c and closures $\texttt{close}\ t\ \texttt{in}\ \rho$, and that $\mathcal{V}$ is given by:

$$v\ \mathcal{V}\ (\widehat{\rho}, \widehat{v}) \text{ iff } \begin{cases} \mathit{true} & \text{if } v = c \\ t \in \widehat{v} \wedge \forall x \in \mathit{dom}(\rho) : \rho(x)\ \mathcal{V}\ (\widehat{\rho}, \widehat{\rho}(x)) & \text{if } v = \texttt{close}\ t\ \texttt{in}\ \rho \end{cases}$$

The correctness condition (4.3) can be reformulated as

$$\begin{array}{l} (v_1\ \mathcal{V}\ (\widehat{\rho}, \widehat{v}_1)\ \wedge\ [\,] \vdash (e_\star\ v_1^{\ell_1})^{\ell_2} \rightarrow^* v_2^{\ell_2}\ \wedge\ (\widehat{\mathsf{C}}, \widehat{\rho}) \models (e_\star\ \mathsf{c}^{\ell_1})^{\ell_2}\ \wedge \\ \widehat{\mathsf{C}}(\ell_1) = \widehat{v}_1\ \wedge\ \widehat{\mathsf{C}}(\ell_2) = \widehat{v}_2)\ \Rightarrow\ v_2\ \mathcal{V}\ (\widehat{\rho}, \widehat{v}_2) \end{array}$$

and it follows from the correctness result established by Theorem 3.10 in Section 3.2 (see Exercise 4.3).

Finally, let us observe that the Control Flow Analysis also satisfies the conditions (4.4) and (4.5). For this we shall equip $\widehat{\mathbf{Env}} \times \widehat{\mathbf{Val}}$ with the partial ordering $\sqsubseteq_{\mathsf{CFA}}$ defined by:

$$(\widehat{\rho}_1, \widehat{v}_1) \sqsubseteq_{\mathsf{CFA}} (\widehat{\rho}_2, \widehat{v}_2) \text{ iff } \widehat{v}_1 \subseteq \widehat{v}_2\ \wedge\ \forall x : \widehat{\rho}_1(x) \subseteq \widehat{\rho}_2(x)$$

This will turn $\widehat{\mathbf{Env}} \times \widehat{\mathbf{Val}}$ into a complete lattice. By induction on $v \in \mathbf{Val}$ one can then easily prove that (4.4) and (4.5) are fulfilled. ∎

4.1.2 Representation Functions

An alternative approach to the use of a correctness relation $R : V \times L \rightarrow \{\mathit{true}, \mathit{false}\}$ between values and properties is to use a *representation function*:

$$\beta : V \rightarrow L$$

The idea is that β maps a value to the *best* property describing it. The correctness criterion for the analysis will then be formulated as follows:

$$\beta(v_1) \sqsubseteq l_1\ \wedge\ p \vdash v_1 \rightsquigarrow v_2\ \wedge\ p \vdash l_1 \rhd l_2\ \Rightarrow\ \beta(v_2) \sqsubseteq l_2 \qquad (4.6)$$

This is also expressed by the following diagram:

$$\begin{array}{ccccc} p\ \vdash & v_1 & \rightsquigarrow & v_2 & \\ \beta & \downarrow & \Rightarrow & \downarrow & \beta \\ & \sqcap\!\mid & & \sqcap\!\mid & \\ p\ \vdash & l_1 & \rhd & l_2 & \end{array}$$

Thus the idea is that if the initial value v_1 is safely described by l_1 then the final value v_2 will be safely described by the result l_2 of the analysis.

Equivalence of correctness formulations. Lemma 4.5 below shows that the formulations (4.3) and (4.6) of the correctness of the analysis are indeed equivalent (when R and β are suitably related). To establish this we shall first show how to define a correctness relation R_β from a given representation function β:

$$v \; R_\beta \; l \quad \text{iff} \quad \beta(v) \sqsubseteq l$$

Next we show how to define a representation function β_R from a correctness relation R:

$$\beta_R(v) \quad = \quad \bigsqcap \{l \mid v \; R \; l\}$$

Lemma 4.5

(i) Given $\beta : V \to L$, then the relation $R_\beta : V \times L \to \{\mathit{true}, \mathit{false}\}$ satisfies conditions (4.4) and (4.5), and furthermore $\beta_{R_\beta} = \beta$.

(ii) Given $R : V \times L \to \{\mathit{true}, \mathit{false}\}$ satisfying conditions (4.4) and (4.5), then β_R is well-defined and $R_{\beta_R} = R$.

Hence the two formulations (4.3) and (4.6) of correctness are equivalent. ■

Proof To prove (i) we first observe that condition (4.4) is immediate since $\sqsubseteq$ is transitive. Condition (4.5) is immediate because when $\beta(v)$ is a lower bound for L' we have $\beta(v) \sqsubseteq \bigsqcap L'$. The calculation $\beta_{R_\beta}(v) = \bigsqcap\{l \mid v \; R_\beta \; l\} = \bigsqcap\{l \mid \beta(v) \sqsubseteq l\} = \beta(v)$ then concludes the proof of (i).

To prove (ii) we observe that from $v \; R \; l$ we get $\beta_R(v) \sqsubseteq l$ and hence $v \; R_{\beta_R} \; l$. Conversely, from $v \; R_{\beta_R} \; l$ we get $\beta_R(v) \sqsubseteq l$; writing $L' = \{l \mid v \; R \; l\}$ it is clear that (4.5) gives $v \; R \; (\bigsqcap L')$ and this amounts to $v \; R \; (\beta_R(v))$; we then get the desired result $v \; R \; l$ by (4.4). ■

Motivated by these results we shall say that the relation R is *generated by* the representation function β whenever $v \; R \; l$ is equivalent to $\beta(v) \sqsubseteq l$. This relationship is illustrated in Figure 4.1: The relation R expresses that v is described by all the properties above $\beta(v)$ and β expresses that among all the properties that describe v, $\beta(v)$ is the best.

Example 4.6 For the Constant Propagation Analysis studied in Examples 4.1 and 4.3 we shall define

$$\beta_{\mathsf{CP}} : \mathbf{State} \to \widehat{\mathbf{State}}_{\mathsf{CP}}$$

as the injection of **State** into $\widehat{\mathbf{State}}_{\mathsf{CP}}$: $\beta_{\mathsf{CP}}(\sigma) = \lambda x.\sigma(x)$. It is straightforward to verify that R_{CP} is *generated by* β_{CP}, i.e.

$$\sigma \; R_{\mathsf{CP}} \; \widehat{\sigma} \;\Leftrightarrow\; \beta_{\mathsf{CP}}(\sigma) \sqsubseteq_{\mathsf{CP}} \widehat{\sigma}$$

using the definition of the ordering $\sqsubseteq_{\mathsf{CP}}$ on $\widehat{\mathbf{State}}_{\mathsf{CP}}$. ■

Figure 4.1: Correctness relation R *generated by* representation function β.

Example 4.7 For the Control Flow Analysis studied in Examples 4.2 and 4.4 we shall define

$$\beta_{\mathsf{CFA}} : \mathbf{Val} \to \widehat{\mathbf{Env}} \times \widehat{\mathbf{Val}}$$

inductively on the structure of values $v \in \mathbf{Val}$:

$$\beta_{\mathsf{CFA}}(v) = \begin{cases} (\lambda x.\emptyset, \emptyset) & \text{if } v = c \\ (\beta^E_{\mathsf{CFA}}(\rho), \{t\}) & \text{if } v = \texttt{close}\ t\ \texttt{in}\ \rho \end{cases}$$

The first clause reflects that we do not collect constants in a pure 0-CFA analysis. In the second clause we only have one closure so the abstract value will be a singleton set and we construct the associated "minimal" abstract environment by extending β_{CFA} to operate on environments. To do that we shall "merge" all the abstract environments occurring in $\bigcup\{\beta_{\mathsf{CFA}}(\rho(x)) \mid x \in \mathbf{Var}_\star\}$; this reflects that the 0-CFA analysis uses one global abstract environment to describe all the possible local environments of the semantics. So we define $\beta^E_{\mathsf{CFA}} : \mathbf{Env} \to \widehat{\mathbf{Env}}$ by:

$$\begin{aligned} \beta^E_{\mathsf{CFA}}(\rho)(x) &= \bigcup\{\widehat{\rho}_y(x) \mid \beta_{\mathsf{CFA}}(\rho(y)) = (\widehat{\rho}_y, \widehat{v}_y) \text{ and } y \in dom(\rho)\} \\ &\quad \cup \begin{cases} \widehat{v}_x & \text{if } x \in dom(\rho) \text{ and } \beta_{\mathsf{CFA}}(\rho(x)) = (\widehat{\rho}_x, \widehat{v}_x) \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

To show that R_{CFA} is *generated by* β_{CFA} we have to show that:

$$v\ R_{\mathsf{CFA}}\ (\widehat{\rho}, \widehat{v}) \Leftrightarrow \beta_{\mathsf{CFA}}(v) \sqsubseteq_{\mathsf{CFA}} (\widehat{\rho}, \widehat{v})$$

This follows by induction on $v \in \mathbf{Val}$ and we leave the details to Exercise 4.4. ∎

4.1.3 A Modest Generalisation

We shall conclude this section by performing a modest generalisation of the development performed so far. A program p specifies how one value v_1 is transformed into another value v_2:

$$p \vdash v_1 \rightsquigarrow v_2$$

Here $v_1 \in V_1$ and $v_2 \in V_2$ and we shall subsequently refrain from imposing the condition that $V_1 = V_2$; thus we shall allow the programs to have different "argument" and "result" types – for example, this will be the case for most functional programs. The analysis of p specifies how a property l_1 is transformed into a property l_2:

$$p \vdash l_1 \rhd l_2$$

Here $l_1 \in L_1$ and $l_2 \in L_2$ and again we shall refrain from imposing the restriction that $L_1 = L_2$. As previously argued it is natural to demand that $p \vdash l_1 \rhd l_2$ specifies a function

$$f_p : L_1 \rightarrow L_2$$

given by $f_p(l_1) = l_2$ iff $p \vdash l_1 \rhd l_2$.

Turning to the correctness conditions we shall now assume that we have two correctness relations, one for V_1 and L_1 and one for V_2 and L_2:

$$R_1 : V_1 \times L_1 \rightarrow \{true, false\} \textit{ generated by } \beta_1 : V_1 \rightarrow L_1$$
$$R_2 : V_2 \times L_2 \rightarrow \{true, false\} \textit{ generated by } \beta_2 : V_2 \rightarrow L_2$$

Correctness of f_p now amounts to

$$v_1 \; R_1 \; l_1 \;\wedge\; p \vdash v_1 \rightsquigarrow v_2 \quad \Rightarrow \quad v_2 \; R_2 \; f_p(l_1)$$

for all $v_1 \in V_1$, $v_2 \in V_2$ and $l_1 \in L_1$. Using the concept of *logical relations* (briefly mentioned above) this can be written as:

$$(p \vdash \cdot \rightsquigarrow \cdot)\,(R_1 \twoheadrightarrow R_2)\, f_p$$

To be precise, $\rightsquigarrow (R_1 \twoheadrightarrow R_2)\, f$ means that:

$$\forall v_1, v_2, l_1 : v_1 \rightsquigarrow v_2 \;\wedge\; v_1 \; R_1 \; l_1 \;\Rightarrow\; v_2 \; R_2 \; f(l_1)$$

Higher-order formulation. We can now ask whether the relation $R_1 \twoheadrightarrow R_2$ defined above is a correctness relation. Lemma 4.8 below shows that this is indeed the case and furthermore that we can find a representation function β such that $R_1 \twoheadrightarrow R_2$ is *generated by* β. The representation function

β can be defined from the representation functions β_1 and β_2 and it will be denoted $\beta_1 \twoheadrightarrow \beta_2$:

$$(\beta_1 \twoheadrightarrow \beta_2)(\rightsquigarrow) = \lambda l_1 . \bigsqcup\{\beta_2(v_2) \mid \beta_1(v_1) \sqsubseteq l_1 \ \wedge \ v_1 \rightsquigarrow v_2\}$$

Lemma 4.8 If R_i is a correctness relation for V_i and L_i that is *generated by* the representation function $\beta_i : V_i \to L_i$ (for $i = 1, 2$) then $R_1 \twoheadrightarrow R_2$ is a correctness relation and it is *generated by* the representation function $\beta_1 \twoheadrightarrow \beta_2$. ∎

Proof We shall prove $\rightsquigarrow (R_1 \twoheadrightarrow R_2) f \Leftrightarrow (\beta_1 \twoheadrightarrow \beta_2)(\rightsquigarrow) \sqsubseteq f$. We calculate:

$$\begin{aligned}
(\beta_1 \twoheadrightarrow \beta_2)(\rightsquigarrow) \sqsubseteq f \quad &\Leftrightarrow \quad \forall l_1 : \bigsqcup\{\beta_2(v_2) \mid \beta_1(v_1) \sqsubseteq l_1 \wedge v_1 \rightsquigarrow v_2\} \sqsubseteq f(l_1) \\
&\Leftrightarrow \quad \forall l_1, v_1, v_2 : (\beta_1(v_1) \sqsubseteq l_1 \wedge v_1 \rightsquigarrow v_2 \Rightarrow \beta_2(v_2) \sqsubseteq f(l_1)) \\
&\Leftrightarrow \quad \forall l_1, v_1, v_2 : (v_1 \ R_1 \ l_1 \wedge v_1 \rightsquigarrow v_2 \Rightarrow v_2 \ R_2 \ f(l_1)) \\
&\Leftrightarrow \quad \rightsquigarrow (R_1 \twoheadrightarrow R_2) f
\end{aligned}$$

Note that it now follows (from Lemma 4.5) that if each R_i satisfies conditions (4.4) and (4.5) then so does $R_1 \twoheadrightarrow R_2$. ∎

Example 4.9 Consider the program `plus` with the semantics given by

$$\texttt{plus} \vdash (z_1, z_2) \rightsquigarrow z_1 + z_2$$

where $z_1, z_2 \in \mathbf{Z}$. A very precise analysis might use the complete lattices $(\mathcal{P}(\mathbf{Z}), \subseteq)$ and $(\mathcal{P}(\mathbf{Z} \times \mathbf{Z}), \subseteq)$ as follows:

$$f_{\text{plus}}(ZZ) = \{z_1 + z_2 \mid (z_1, z_2) \in ZZ\}$$

where $ZZ \subseteq \mathbf{Z} \times \mathbf{Z}$. Consider now the correctness relations R_{Z} and $R_{\mathrm{Z}\times\mathrm{Z}}$ *generated by* the representation functions:

$$\begin{aligned}
\beta_{\mathrm{Z}}(z) &= \{z\} \\
\beta_{\mathrm{Z}\times\mathrm{Z}}(z_1, z_2) &= \{(z_1, z_2)\}
\end{aligned}$$

The correctness of the analysis of `plus` can now be expressed by

$$\forall z_1, z_2, z, ZZ : \texttt{plus} \vdash (z_1, z_2) \rightsquigarrow z \ \wedge \ (z_1, z_2) \ R_{\mathrm{Z}\times\mathrm{Z}} \ ZZ \ \Rightarrow \ z \ R_{\mathrm{Z}} \ f_{\text{plus}}(ZZ)$$

or more succinctly

$$(\texttt{plus} \vdash \cdot \rightsquigarrow \cdot)\,(R_{\mathrm{Z}\times\mathrm{Z}} \twoheadrightarrow R_{\mathrm{Z}})\, f_{\text{plus}}$$

The representation function $\beta_{\mathrm{Z}\times\mathrm{Z}} \twoheadrightarrow \beta_{\mathrm{Z}}$ satisfies

$$(\beta_{\mathrm{Z}\times\mathrm{Z}} \twoheadrightarrow \beta_{\mathrm{Z}})(p \vdash \cdot \rightsquigarrow \cdot) = \lambda ZZ.\{z \mid (z_1, z_2) \in ZZ \ \wedge \ p \vdash (z_1, z_2) \rightsquigarrow z\}$$

so the correctness can also be expressed as $(\beta_{\mathrm{Z}\times\mathrm{Z}} \twoheadrightarrow \beta_{\mathrm{Z}})(\texttt{plus} \vdash \cdot \rightsquigarrow \cdot) \sqsubseteq f_{\text{plus}}$. ∎

Figure 4.2: The complete lattice **Interval** = (**Interval**, $\sqsubseteq$).

The above example illustrates how the abstract concepts can give a more succinct formulation of the correctness of the analysis. In the following we shall see several cases where we move freely between what we may call a "concrete" formulation of a property and an "abstract" formulation of the same property. And we shall see that the latter often will allow us to reuse general results so that we do not have to redevelop parts of the theory for each application considered.

4.2 Approximation of Fixed Points

It should be clear by now that complete lattices play a crucial role in program analysis and in the remainder of this chapter we shall tacitly *assume* that property spaces such as L and M are indeed complete lattices. We refer to Appendix A for the basic notions of complete lattices and monotone functions. The following example introduces an interesting complete lattice that forms the basis of many analyses over the integers.

Example 4.10 We shall now present a complete lattice that may be used for *Array Bound Analysis*, i.e. for determining if an array index is always within the bounds of the array – if this is the case then a number of run-time checks can be eliminated.

The lattice (**Interval**, $\sqsubseteq$) of intervals over **Z** may be described as follows.

The elements are

$$\textbf{Interval} = \{\bot\} \cup \{[z_1, z_2] \mid z_1 \leq z_2, z_1 \in \mathbf{Z} \cup \{-\infty\}, z_2 \in \mathbf{Z} \cup \{\infty\}\}$$

where the ordering $\leq$ on $\mathbf{Z}$ is extended to an ordering on $\mathbf{Z}' = \mathbf{Z} \cup \{-\infty, \infty\}$ by setting $-\infty \leq z$, $z \leq \infty$, and $-\infty \leq \infty$ (for all $z \in \mathbf{Z}$). Intuitively, $\bot$ denotes the empty interval and $[z_1, z_2]$ is the interval from z_1 to z_2 including the end points if they are in $\mathbf{Z}$. We shall use *int* to range over elements of **Interval**.

The partial ordering $\sqsubseteq$ on **Interval** is depicted in Figure 4.2; the idea is that $int_1 \sqsubseteq int_2$ corresponds to $\{z \mid z \text{ is in } int_1\} \subseteq \{z \mid z \text{ is in } int_2\}$ where the meaning of "is in" should be immediate. To give a succinct definition of the partial ordering we define the infimum and supremum operations on intervals as follows:

$$\mathsf{inf}(int) = \begin{cases} \infty & \text{if } int = \bot \\ z_1 & \text{if } int = [z_1, z_2] \end{cases}$$

$$\mathsf{sup}(int) = \begin{cases} -\infty & \text{if } int = \bot \\ z_2 & \text{if } int = [z_1, z_2] \end{cases}$$

This allows us to define:

$$int_1 \sqsubseteq int_2 \quad \text{iff} \quad \mathsf{inf}(int_2) \leq \mathsf{inf}(int_1) \wedge \mathsf{sup}(int_1) \leq \mathsf{sup}(int_2)$$

We claim that (**Interval**, $\sqsubseteq$) is indeed a complete lattice. We shall prove this by showing that each subset of **Interval** has a least upper bound and then refer to Lemma A.2 of Appendix A to get that (**Interval**, $\sqsubseteq$) is a complete lattice. So let Y be a subset of **Interval**. The idea is that each interval *int* of Y should be "contained in" the interval $\bigsqcup Y$ defined by:

$$\bigsqcup Y = \begin{cases} \bot & \text{if } Y \subseteq \{\bot\} \\ [\ \mathsf{inf}'\{\mathsf{inf}(int) \mid int \in Y\}\ ,\ \mathsf{sup}'\{\mathsf{sup}(int) \mid int \in Y\}\ ] & \\ & \text{otherwise} \end{cases}$$

where inf' and sup' are the infimum and supremum operators on $\mathbf{Z}'$ corresponding to the ordering $\leq$ on $\mathbf{Z}'$; they are given by $\mathsf{inf}'(\emptyset) = \infty$, $\mathsf{inf}'(Z) = z'$ if $z' \in Z$ is the least element of Z, and $\mathsf{inf}'(Z) = -\infty$ otherwise; and similarly $\mathsf{sup}'(\emptyset) = -\infty$, $\mathsf{sup}'(Z) = z'$ if $z' \in Z$ is the greatest element of Z, and $\mathsf{sup}'(Z) = \infty$ otherwise. It is now straightforward to show that $\bigsqcup Y$ is indeed the least upper bound of Y. ∎

Given a complete lattice $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ the effect of a program, p, in transforming one property, l_1, into another, l_2, i.e. $p \vdash l_1 \rhd l_2$, is normally given by an equation

$$f(l_1) = l_2$$

for a monotone function $f : L \to L$ dependent on the program p. Note that the demand that f is monotone is very natural for program analysis; it merely says that if l_1' describes at least the values that l_1 does then also $f(l_1')$ describes at least the values that $f(l_1)$ does.

For recursive or iterative program constructs we ideally want to obtain the least fixed point, $lfp(f)$, as the result of a *finite* iterative process. However, the iterative sequence $(f^n(\bot))_n$ need *not* eventually stabilise nor need its least upper bound necessarily equal $lfp(f)$. This might suggest considering the iterative sequence $(f^n(\top))_n$ and, even when it does not eventually stabilise, we can always terminate the iteration at an arbitrary point in time. While this is safe (thanks to condition (4.4) of Section 4.1) it turns out to be grossly imprecise in practice.

Fixed points. We shall begin by recalling some of the properties of fixed points of monotone functions over complete lattices; we refer to Appendix A for the details of this development. So consider a monotone function $f : L \to L$ on a complete lattice $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. A *fixed point* of f is an element $l \in L$ such that $f(l) = l$ and we write

$$Fix(f) = \{l \mid f(l) = l\}$$

for the set of fixed points. The function f is *reductive* at l if and only if $f(l) \sqsubseteq l$ and we write

$$Red(f) = \{l \mid f(l) \sqsubseteq l\}$$

for the set of elements upon which f is reductive; we shall say that f itself is reductive if $Red(f) = L$. Similarly, the function f is *extensive* at l if and only if $f(l) \sqsupseteq l$ and we write

$$Ext(f) = \{l \mid f(l) \sqsupseteq l\}$$

for the set of elements upon which f is extensive; we shall say that f itself is extensive if $Ext(f) = L$.

Since L is a complete lattice it is always the case that the set $Fix(f)$ will have a greatest lower bound in L and we denote it by $lfp(f)$; this is actually the *least fixed point* of f because Tarski's Theorem (Proposition A.10) ensures that:

$$lfp(f) = \bigsqcap Fix(f) = \bigsqcap Red(f) \in Fix(f) \subseteq Red(f)$$

Similarly, the set $Fix(f)$ will have a least upper bound in L and we denote it by $gfp(f)$; this is actually the *greatest fixed point* of f because Tarski's Theorem ensures that:

$$gfp(f) = \bigsqcup Fix(f) = \bigsqcup Ext(f) \in Fix(f) \subseteq Ext(f)$$

In *Denotational Semantics* it is customary to iterate to the least fixed point by taking the least upper bound of the sequence $(f^n(\bot))_n$. However, we have not

Figure 4.3: Fixed points of f.

imposed any continuity requirements on f (e.g. that $f(\bigsqcup_n l_n) = \bigsqcup_n(f(l_n))$ for all ascending chains $(l_n)_n$) and consequently we cannot be sure to actually reach the fixed point. In a similar way one could consider the greatest lower bound of the sequence $(f^n(\top))_n$. One can show that

$$f^n(\bot) \sqsubseteq \bigsqcup_n f^n(\bot) \sqsubseteq lfp(f) \sqsubseteq gfp(f) \sqsubseteq \bigsqcap_n f^n(\top) \sqsubseteq f^n(\top)$$

as is illustrated in Figure 4.3; indeed all inequalities (i.e. $\sqsubseteq$) can be strict (i.e. $\neq$).

4.2.1 Widening Operators

Since we cannot guarantee that the iterative sequence $(f^n(\bot))_n$ eventually stabilises nor that its least upper bound necessarily equals $lfp(f)$, we must consider another way of approximating $lfp(f)$. The idea is now to replace it by a new sequence $(f^n_\nabla)_n$ that is known to eventually stabilise and to do so with a value that is a safe (upper) approximation of the least fixed point. The construction of the new sequence is parameterised on the operator ∇, called a *widening operator*; the precision of the approximated fixed point as well as the cost of computing it depends on the actual choice of widening operator.

Upper bound operators. In preparation for the development, an operator $\check{\sqcup} : L \times L \to L$ on a complete lattice $L = (L, \sqsubseteq)$ is called an *upper bound operator* if

$$l_1 \sqsubseteq (l_1 \check{\sqcup} l_2) \sqsupseteq l_2$$

for all $l_1, l_2 \in L$, i.e. it always returns an element larger than both its arguments. Note that we do *not* require $\check{\sqcup}$ to be monotone, commutative, associative, nor absorptive (i.e. that $l \check{\sqcup} l = l$).

Let $(l_n)_n$ be a sequence of elements of L and let $\phi : L \times L \to L$ be a total function on L. We shall now use ϕ to construct a new sequence $(l_n^\phi)_n$ defined by:

$$l_n^\phi = \begin{cases} l_n & \text{if } n = 0 \\ l_{n-1}^\phi \ \phi \ l_n & \text{if } n > 0 \end{cases}$$

The following result expresses that any sequence can be turned into an ascending chain by an upper bound operator:

Fact 4.11 If $(l_n)_n$ is a sequence and $\check{\sqcup}$ is an upper bound operator then $(l_n^{\check{\sqcup}})_n$ is an ascending chain; furthermore $l_n^{\check{\sqcup}} \sqsupseteq \bigsqcup\{l_0, l_1, \cdots, l_n\}$ for all n. ∎

Proof To prove that $(l_n^{\check{\sqcup}})_n$ is an ascending chain it is sufficient to prove that $l_n^{\check{\sqcup}} \sqsubseteq l_{n+1}^{\check{\sqcup}}$ for all n. If $n = 0$ we calculate $l_0^{\check{\sqcup}} = l_0 \sqsubseteq l_0 \check{\sqcup} l_1 = l_1^{\check{\sqcup}}$. For the inductive step we have $l_n^{\check{\sqcup}} \sqsubseteq l_n^{\check{\sqcup}} \check{\sqcup} l_{n+1} = l_{n+1}^{\check{\sqcup}}$ as required.

To prove that $l_n^{\check{\sqcup}} \sqsupseteq \bigsqcup\{l_0, l_1, \cdots, l_n\}$ we first observe that it holds trivially for $n = 0$. For the inductive step we have $l_{n+1}^{\check{\sqcup}} = l_n^{\check{\sqcup}} \check{\sqcup} l_{n+1} \sqsupseteq \bigsqcup\{l_0, l_1, \cdots, l_n\} \sqcup l_{n+1} = \bigsqcup\{l_0, l_1, \cdots, l_n, l_{n+1}\}$ and the result follows. ∎

Example 4.12 Consider the complete lattice (**Interval**, $\sqsubseteq$) of Figure 4.2 and let *int* be an arbitrary but fixed element of **Interval**. Consider the following operator $\check{\sqcup}^{int}$ defined on **Interval**:

$$int_1 \ \check{\sqcup}^{int} \ int_2 = \begin{cases} int_1 \sqcup int_2 & \text{if } int_1 \sqsubseteq int \vee int_2 \sqsubseteq int_1 \\ [-\infty, \infty] & \text{otherwise} \end{cases}$$

Note that the operation is not symmetric: for $int = [0, 2]$ we e.g. have $[1, 2] \check{\sqcup}^{int} [2, 3] = [1, 3]$ whereas $[2, 3] \check{\sqcup}^{int} [1, 2] = [-\infty, \infty]$.

It is immediate that $\check{\sqcup}^{int}$ is an upper bound operator. Consider now the sequence:

$$[0, 0], [1, 1], [2, 2], [3, 3], [4, 4], [5, 5], \cdots$$

If $int = [0, \infty]$, then the upper bound operator will transform the above sequence into the ascending chain:

$$[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [0, 5], \cdots$$

Figure 4.4: The widening operator ∇ applied to f.

However, if $int = [0, 2]$, then we will get the following ascending chain

$$[0,0], [0,1], [0,2], [0,3], [-\infty, \infty], [-\infty, \infty], \cdots$$

which eventually stabilises. ■

Widening operators. We can now introduce a special class of upper bound operators that will help us to approximate the least fixed points: An operator $\nabla : L \times L \to L$ is a *widening operator* if and only if:

- it is an upper bound operator, and
- for all ascending chains $(l_n)_n$ the ascending chain $(l_n^\nabla)_n$ eventually stabilises.

Note that it follows from Fact 4.11 that $(l_n^\nabla)_n$ is indeed an ascending chain.

The idea is as follows: Given a monotone function $f : L \to L$ on a complete lattice L and given a widenimg operator ∇ on L, we shall calculate the sequence $(f_\nabla^n)_n$ defined by

$$f_\nabla^n = \begin{cases} \bot & \text{if } n = 0 \\ f_\nabla^{n-1} & \text{if } n > 0 \ \wedge \ f(f_\nabla^{n-1}) \sqsubseteq f_\nabla^{n-1} \\ f_\nabla^{n-1} \ \nabla \ f(f_\nabla^{n-1}) & \text{otherwise} \end{cases}$$

As in Fact 4.11 it follows that this is an ascending chain and Proposition 4.13 below will ensure that this sequence eventually stabilises. From Fact 4.14 below we shall see that this means that we will eventually have $f(f_\nabla^m) \sqsubseteq f_\nabla^m$ for some value of m (corresponding to the second clause in the definition of f_∇^n). This means that f is reductive at f_∇^m and from Tarski's Theorem (Proposition A.10) we then know that $f_\nabla^m \sqsupseteq lfp(f)$ must be the case; hence we take

$$lfp_\nabla(f) = f_\nabla^m$$

as the desired safe approximation of $lfp(f)$. This is illustrated in Figure 4.4. We shall now establish the necessary results.

Proposition 4.13
If ∇ is a widening operator then the ascending chain $(f_\nabla^n)_n$ eventually stabilises.

In preparation for the proof of Proposition 4.13 we shall first show the following technical result:

Fact 4.14 If ∇ is a widening operator then:

(i) the sequence $(f_\nabla^n)_n$ is an ascending chain;

(ii) if $f(f_\nabla^m) \sqsubseteq f_\nabla^m$ for some m then the sequence $(f_\nabla^n)_n$ eventually stabilises and furthermore $\forall n > m : f_\nabla^n = f_\nabla^m$ and $\bigsqcup_n f_\nabla^n = f_\nabla^m$;

(iii) if $(f_\nabla^n)_n$ eventually stabilises then there exists an m such that $f(f_\nabla^m) \sqsubseteq f_\nabla^m$; and

(iv) if $(f_\nabla^n)_n$ eventually stabilises then $\bigsqcup_n f_\nabla^n \sqsupseteq lfp(f)$.

These claims also hold if ∇ is just an upper bound operator. ∎

Proof The proof of (i) is analogous to that of Fact 4.11 so we omit the details.

To prove (ii) we assume that $f(f_\nabla^m) \sqsubseteq f_\nabla^m$ for some m. By induction on $n > m$ we prove that $f_\nabla^n = f_\nabla^m$: for $n = m + 1$ it follows from the assumption and for the inductive step we note that $f(f_\nabla^n) \sqsubseteq f_\nabla^n$ will be the case so $f_\nabla^{n+1} = f_\nabla^n$. Thus $(f_\nabla^n)_n$ eventually stabilises and it follows that $\bigsqcup_n f_\nabla^n = f_\nabla^m$.

To prove (iii) we assume that $(f_\nabla^n)_n$ eventually stabilises. This means that there exists m such that $\forall n > m : f_\nabla^n = f_\nabla^m$. By way of contradiction assume that $f(f_\nabla^m) \sqsubseteq f_\nabla^m$ does not hold; then $f_\nabla^m = f_\nabla^{m+1} = f_\nabla^m \nabla f(f_\nabla^m) \sqsupseteq f(f_\nabla^m)$ and we have the desired contradiction.

To prove (iv) we observe that (ii) and (iii) give $\bigsqcup_n f_\nabla^n = f_\nabla^m$ for some m where $f(f_\nabla^m) \sqsubseteq f_\nabla^m$. Hence $f_\nabla^m \in Red(f)$ and by Tarski's Theorem (Proposition A.10) this shows $f_\nabla^m \sqsupseteq lfp(f)$. ∎

We now turn to the proof of Proposition 4.13:

Proof By way of contradiction we shall assume that the ascending chain $(f_\nabla^n)_n$ never stabilises; i.e.:

$$\forall n_0 : \exists n \geq n_0 : f_\nabla^n \neq f_\nabla^{n_0}$$

It follows that $f(f_\nabla^{n-1}) \sqsubseteq f_\nabla^{n-1}$ never holds for any $n > 0$; because if it did, then Fact 4.14 gives that $(f_\nabla^n)_n$ eventually stabilises and our hypothesis would be contradicted. This means that the definition of $(f_\nabla^n)_n$ specialises to:

$$f_\nabla^n = \begin{cases} \bot & \text{if } n = 0 \\ f_\nabla^{n-1} \nabla f(f_\nabla^{n-1}) & \text{otherwise} \end{cases}$$

Now define the sequence $(l_n)_n$ by

$$l_n = \begin{cases} \bot & \text{if } n = 0 \\ f(f_\nabla^{n-1}) & \text{if } n > 0 \end{cases}$$

and note that $(l_n)_n$ is an ascending chain because $(f_\nabla^n)_n$ is an ascending chain (Fact 4.14) and f is monotone. We shall now prove that

$$\forall n : l_n^\nabla = f_\nabla^n$$

by induction on n: for $n = 0$ it is immediate and for $n > 0$ we have $l_n^\nabla = l_{n-1}^\nabla \nabla l_n = f_\nabla^{n-1} \nabla f(f_\nabla^{n-1}) = f_\nabla^n$. Since $(l_n)_n$ is an ascending chain and ∇ is a widening operator it follows that the sequence $(l_n^\nabla)_n$ eventually stabilises, i.e. $(f_\nabla^n)_n$ eventually stabilises. This provides the desired contradiction and proves the result. ∎

Example 4.15 Consider the complete lattice (**Interval**, $\sqsubseteq$) of Figure 4.2. Let K be a *finite* set of integers, e.g. the set of integers explicitly mentioned in a given program. We shall now define a widening operator ∇_K based on K. The idea is that $[z_1, z_2] \nabla_K [z_3, z_4]$ is something like

$$[\ \mathsf{LB}(z_1, z_3)\ ,\ \mathsf{UB}(z_2, z_4)\ ]$$

where $\mathsf{LB}(z_1, z_3) \in \{z_1\} \cup K \cup \{-\infty\}$ is the best possible lower bound and $\mathsf{UB}(z_2, z_4) \in \{z_2\} \cup K \cup \{\infty\}$ is the best possible upper bound. In this way a change in any of the bounds of the interval $[z_1, z_2]$ can only take place in a finite number of steps (corresponding to the elements of K).

For the precise definition we let $z_i \in \mathbf{Z}' = \mathbf{Z} \cup \{-\infty, \infty\}$ and write:

$$\mathsf{LB}_K(z_1, z_3) = \begin{cases} z_1 & \text{if } z_1 \leq z_3 \\ k & \text{if } z_3 < z_1 \ \wedge\ k = \mathsf{max}\{k \in K \mid k \leq z_3\} \\ -\infty & \text{if } z_3 < z_1 \ \wedge\ \forall k \in K : z_3 < k \end{cases}$$

$$\mathsf{UB}_K(z_2, z_4) = \begin{cases} z_2 & \text{if } z_4 \leq z_2 \\ k & \text{if } z_2 < z_4 \ \wedge\ k = \mathsf{min}\{k \in K \mid z_4 \leq k\} \\ \infty & \text{if } z_2 < z_4 \ \wedge\ \forall k \in K : k < z_4 \end{cases}$$

We can now define $\nabla = \nabla_K$ by:

$$int_1 \ \nabla \ int_2 = \begin{cases} \bot & \text{if } int_1 = int_2 = \bot \\ [\ \mathsf{LB}_K(\mathsf{inf}(int_1), \mathsf{inf}(int_2)), \mathsf{UB}_K(\mathsf{sup}(int_1), \mathsf{sup}(int_2))\ ] & \\ & \text{otherwise} \end{cases}$$

As an example consider the ascending chain $(int_n)_n$:

$$[0,1], [0,2], [0,3], [0,4], [0,5], [0,6], [0,7], \cdots$$

and assume that $K = \{3,5\}$. Then $(int_n^\nabla)_n$ will be the chain

$$[0,1], [0,3], [0,3], [0,5], [0,5], [0,\infty], [0,\infty], \cdots$$

It is straightforward to show that ∇ is indeed an upper bound operator. To show that it is a widening operator we consider an ascending chain $(int_n)_n$ and must show that the ascending chain $(int_n^\nabla)_n$ eventually stabilises. By way of contradiction suppose that $(int_n^\nabla)_n$ does not eventually stabilise. Then at least one of the following properties will hold:

$$(\forall n : \mathsf{inf}(int_n^\nabla) > -\infty) \quad \wedge \quad \mathsf{inf}(\bigsqcup_n int_n^\nabla) = -\infty$$
$$(\forall n : \mathsf{sup}(int_n^\nabla) < \infty) \quad \wedge \quad \mathsf{sup}(\bigsqcup_n int_n^\nabla) = \infty$$

Without loss of generality we can assume that the second property holds. Hence there must exist an infinite sequence $n_1 < n_2 < \cdots$ such that

$$\forall i : \infty > \mathsf{sup}(int_{n_i+1}^\nabla) > \mathsf{sup}(int_{n_i}^\nabla)$$

and by finiteness of K there must be some j such that

$$\forall i \geq j : \infty > \mathsf{sup}(int_{n_i+1}^\nabla) > \mathsf{sup}(int_{n_i}^\nabla) > \mathsf{max}(K)$$

where for the purpose of this definition we set $\mathsf{max}(\emptyset) = -\infty$. Since we also have

$$int_{n_j+1}^\nabla = int_{n_j}^\nabla \ \nabla \ int_{n_j+1}$$

it must be the case that $\mathsf{sup}(int_{n_j+1}) > \mathsf{sup}(int_{n_j}^\nabla)$ as otherwise $\mathsf{sup}(int_{n_j+1}^\nabla) = \mathsf{sup}(int_{n_j}^\nabla)$. But then the construction yields

$$\mathsf{sup}(int_{n_j+1}^\nabla) = \infty$$

and we have the desired contradiction. This shows that ∇ is a widening operator on the complete lattice of intervals. ∎

4.2.2 Narrowing Operators

Using the technique of widening we managed to arrive at an upper approximation f_∇^m of the least fixed point of f. However, we have $f(f_\nabla^m) \sqsubseteq f_\nabla^m$ so f is reductive at f_∇^m and this immediately suggests a way of improving the approximation by considering the iterative sequence $(f^n(f_\nabla^m))_n$. Since $f_\nabla^m \in Red(f)$ this will be a descending chain with $f^n(f_\nabla^m) \in Red(f)$ and hence $f^n(f_\nabla^m) \sqsupseteq lfp(f)$ for all n. Once again we have no reason to believe that this descending chain eventually stabilises although it is of course safe to stop at an arbitrary point. This scenario is not quite the one we saw above but inspired by the notion of widening we can define the notion of narrowing that encapsulates a termination criterion. This development can safely be omitted on a first reading.

An operator $\Delta : L \times L \to L$ is a *narrowing operator* if:

- $l_2 \sqsubseteq l_1 \;\Rightarrow\; l_2 \sqsubseteq (l_1 \;\Delta\; l_2) \sqsubseteq l_1$ for all $l_1, l_2 \in L$, and
- for all descending chains $(l_n)_n$ the sequence $(l_n^\Delta)_n$ eventually stabilises.

Note that we do *not* require Δ to be monotone, commutative, associative or absorptive. One can show that $(l_n^\Delta)_n$ is a descending chain when $(l_n)_n$ is a descending chain (Exercise 4.10).

The idea is as follows: For f_∇^m satisfying $f(f_\nabla^m) \sqsubseteq f_\nabla^m$, i.e. $lfp_\nabla(f) = f_\nabla^m$, we now construct the sequence $([f]_\Delta^n)_n$ by

$$[f]_\Delta^n = \begin{cases} f_\nabla^m & \text{if } n = 0 \\ [f]_\Delta^{n-1} \;\Delta\; f([f]_\Delta^{n-1}) & \text{if } n > 0 \end{cases}$$

Lemma 4.16 below guarantees that this is a descending chain where all elements satisfy $lfp(f) \sqsubseteq [f]_\Delta^n$. Proposition 4.17 below tells us that this chain eventually stabilises so $[f]_\Delta^{m'} = [f]_\Delta^{m'+1}$ for some value m'. We shall therefore take

$$lfp_\nabla^\Delta(f) = [f]_\Delta^{m'}$$

as the desired approximation of $lfp(f)$. The complete development is illustrated in Figures 4.4 and 4.5. We shall now establish the required results.

Lemma 4.16 If Δ is a narrowing operator and $f(f_\nabla^m) \sqsubseteq f_\nabla^m$ then $([f]_\Delta^n)_n$ is a descending chain in $Red(f)$ and

$$[f]_\Delta^n \sqsupseteq f^n(f_\nabla^m) \sqsupseteq lfp(f)$$

for all n. ∎

Proof By induction on n we prove:

$$f^{n+1}(f_\nabla^m) \;\sqsubseteq\; f([f]_\Delta^n) \;\sqsubseteq\; [f]_\Delta^{n+1} \;\sqsubseteq\; [f]_\Delta^n \tag{4.7}$$

Figure 4.5: The narrowing operator Δ applied to f.

For the basis ($n = 0$) it is immediate that

$$f^{n+1}(f^m_\nabla) \sqsubseteq f([f]^n_\Delta) \sqsubseteq [f]^n_\Delta$$

using that $f(f^m_\nabla) \sqsubseteq f^m_\nabla$. By construction of $[f]^{n+1}_\Delta$ we then get

$$f([f]^n_\Delta) \sqsubseteq [f]^{n+1}_\Delta \sqsubseteq [f]^n_\Delta$$

and together these two results establish the basis of the induction proof.

For the inductive step we may apply f to the induction hypothesis (4.7) and get

$$f^{n+2}(f^m_\nabla) \sqsubseteq f^2([f]^n_\Delta) \sqsubseteq f([f]^{n+1}_\Delta) \sqsubseteq f([f]^n_\Delta)$$

since f is assumed to be monotone. Using the induction hypothesis we also have $f([f]^n_\Delta) \sqsubseteq [f]^{n+1}_\Delta$ so we obtain:

$$f^{n+2}(f^m_\nabla) \sqsubseteq f([f]^{n+1}_\Delta) \sqsubseteq [f]^{n+1}_\Delta$$

By construction of $[f]^{n+2}_\Delta$ we get

$$f([f]^{n+1}_\Delta) \sqsubseteq [f]^{n+2}_\Delta \sqsubseteq [f]^{n+1}_\Delta$$

and together these two results complete the proof of (4.7).

From (4.7) it now follows that $([f]^n_\Delta)_n$ is a descending chain in $Red(f)$. It also follows that $f^n(f^m_\nabla) \sqsubseteq [f]^n_\Delta$ holds for $n > 0$ and it clearly also holds for $n = 0$.

From the assumption $f(f_\nabla^m) \sqsubseteq f_\nabla^m$ it is immediate that $f(f^n(f_\nabla^m)) \sqsubseteq f^n(f_\nabla^m)$ for $n \geq 0$ and hence $f^n(f_\nabla^m) \in Red(f)$. But then $f^n(f_\nabla^m) \sqsupseteq lfp(f)$. This completes the proof. ∎

Proposition 4.17
If Δ is a narrowing operator and $f(f_\nabla^m) \sqsubseteq f_\nabla^m$ then the descending chain $([f]_\Delta^n)_n$ eventually stabilises.

Proof Define the sequence $(l_n)_n$ by

$$l_n = \begin{cases} f_\nabla^m & \text{if } n = 0 \\ f([f]_\Delta^{n-1}) & \text{if } n > 0 \end{cases}$$

and note that this defines a descending chain because $([f]_\Delta^n)_n$ is a descending chain and because $f([f]_\Delta^0) \sqsubseteq f_\nabla^m$. Therefore the sequence $(l_n^\Delta)_n$ eventually stabilises. We now prove by induction on n that:

$$l_n^\Delta = [f]_\Delta^n$$

The basis ($n = 0$) is immediate. For the inductive step we calculate $l_{n+1}^\Delta = l_n^\Delta \ \Delta \ l_{n+1} = [f]_\Delta^n \ \Delta f([f]_\Delta^n) = [f]_\Delta^{n+1}$. It thus follows that $([f]_\Delta^n)_n$ eventually stabilises. ∎

It is important to stress that narrowing operators are *not* the dual concept of widening operators. In particular, the sequence $(f_\nabla^n)_n$ may step outside $Ext(f)$ in order to end in $Red(f)$, whereas the sequence $([f]_\Delta^n)_n$ stays in $Red(f)$ all the time.

Example 4.18 Consider the complete lattice (**Interval**, $\sqsubseteq$) of Figure 4.2. Basically there are two kinds of infinite descending chains in **Interval**: those with elements of the form $[-\infty, z]$ and those with elements of the form $[z, \infty]$ where $z \in \mathbf{Z}$. Consider an infinite sequence of the latter form; it will have elements

$$[z_1, \infty], [z_2, \infty], [z_3, \infty], \cdots$$

where $z_1 < z_2 < z_3 < \cdots$. The idea is now to define a narrowing operator Δ_N that will force the sequence to stabilise when $z_i \geq N$ for some fixed non-negative integer N. Similarly, for a descending chain with elements of the form $[-\infty, z_i]$ the narrowing operator will force it to stabilise when $z_i \leq -N$. Formally, we shall define $\Delta = \Delta_N$ by

$$int_1 \ \Delta \ int_2 \quad = \quad \begin{cases} \bot & \text{if } int_1 = \bot \ \vee \ int_2 = \bot \\ [z_1, z_2] & \text{otherwise} \end{cases}$$

where

$$z_1 = \begin{cases} \mathsf{inf}(int_1) & \text{if } N < \mathsf{inf}(int_2) \wedge \mathsf{sup}(int_2) = \infty \\ \mathsf{inf}(int_2) & \text{otherwise} \end{cases}$$

$$z_2 = \begin{cases} \mathsf{sup}(int_1) & \text{if } \mathsf{inf}(int_2) = -\infty \wedge \mathsf{sup}(int_2) < -N \\ \mathsf{sup}(int_2) & \text{otherwise} \end{cases}$$

So consider e.g. the infinite descending chain $([n, \infty])_n$

$$[0, \infty], [1, \infty], [2, \infty], [3, \infty], [4, \infty], [5, \infty], \cdots$$

and assume that $N = 3$. Then the operator will give the sequence $([n, \infty]^\Delta)_n$:

$$[0, \infty], [1, \infty], [2, \infty], [3, \infty], [3, \infty], [3, \infty], \cdots$$

Let us show that Δ is indeed a narrowing operator. It is immediate to verify that

$$int_2 \sqsubseteq int_1 \text{ implies } int_2 \sqsubseteq int_1 \ \Delta \ int_2 \sqsubseteq int_1$$

by cases on whether $int_2 = \bot$ or $int_2 \neq \bot$. We shall then show that if $(int_n)_n$ is a descending chain then the sequence $(int_n^\Delta)_n$ eventually stabilises. So assume that $(int_n)_n$ is a descending chain. One can then show that $(int_n^\Delta)_n$ is a descending chain (Exercise 4.10). Next suppose by way of contradiction that $(int_n^\Delta)_n$ never eventually stabilises. It follows that there exists $n_1 \geq 0$ such that:

$$int_n^\Delta \neq [-\infty, \infty] \text{ for all } n \geq n_1$$

It furthermore follows for all $n \geq n_1$ that int_n^Δ must have:

$$\mathsf{sup}(int_n^\Delta) = \infty \quad \text{or} \quad \mathsf{inf}(int_n^\Delta) = -\infty$$

Without loss of generality let us assume that all $n \geq n_1$ have $\mathsf{sup}(int_n^\Delta) = \infty$ and $\mathsf{inf}(int_n^\Delta) \in \mathbf{Z}$. Hence there exists $n_2 \geq n_1$ such that:

$$\mathsf{inf}(int_n^\Delta) > N \text{ for all } n \geq n_2$$

But then $int_n^\Delta = int_{n_2}^\Delta$ for all $n \geq n_2$ and we have the desired contradiction. This shows that Δ is a narrowing operator. ∎

4.3 Galois Connections

Sometimes calculations on a complete lattice L may be too costly or even uncomputable and this may motivate replacing L by a simpler lattice M. An example is when L is the powerset of integers and M is a lattice of intervals. So rather than performing the analysis $p \vdash l_1 \rhd l_2$ in L, the idea will be to find a description of the elements of L in M and to perform the analysis

$p \vdash m_1 \rhd m_2$ in M. To express the relationship between L and M it is customary to use an *abstraction function*

$$\alpha : L \to M$$

giving a representation of the elements of L as elements of M and a *concretisation function*

$$\gamma : M \to L$$

that expresses the meaning of elements of M in terms of elements of L. We shall write

$$(L, \alpha, \gamma, M)$$

or

$$L \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} M$$

for this setup and would expect that α and γ should be somehow related. We shall study this relationship in the present section and then return to the connection between $p \vdash l_1 \rhd l_2$ and $p \vdash m_1 \rhd m_2$ in Section 4.5; in Section 4.4 we shall study the systematic construction of such relationships.

We define (L, α, γ, M) to be a *Galois connection* between the complete lattices $(L, \sqsubseteq)$ and $(M, \sqsubseteq)$ if and only if

$$\alpha : L \to M \text{ and } \gamma : M \to L \text{ are monotone functions}$$

that satisfy:

$$\gamma \circ \alpha \sqsupseteq \lambda l.l \tag{4.8}$$

$$\alpha \circ \gamma \sqsubseteq \lambda m.m \tag{4.9}$$

Conditions (4.8) and (4.9) express that we do not lose safety by going back and forth between the two lattices although we may lose precision. In the case of (4.8) this ensures that if we start with an element $l \in L$ we can first find a description $\alpha(l)$ of it in M and next determine which element $\gamma(\alpha(l))$ of L that describes $\alpha(l)$; this need not be l but it will be a safe approximation to l, i.e. $l \sqsubseteq \gamma(\alpha(l))$. This is illustrated in Figure 4.6.

Example 4.19 Let $\mathcal{P}(\mathbf{Z}) = (\mathcal{P}(\mathbf{Z}), \subseteq)$ be the complete lattice of sets of integers and let **Interval** = (**Interval**, $\sqsubseteq$) be the complete lattice of Figure 4.2. We shall now define a Galois connection

$$(\mathcal{P}(\mathbf{Z}), \alpha_{\mathrm{ZI}}, \gamma_{\mathrm{ZI}}, \mathbf{Interval})$$

between $\mathcal{P}(\mathbf{Z})$ and **Interval**.

Figure 4.6: The Galois connection (L, α, γ, M).

The concretisation function $\gamma_{\mathrm{ZI}} : \mathbf{Interval} \to \mathcal{P}(\mathbf{Z})$ is defined by

$$\gamma_{\mathrm{ZI}}(int) = \{z \in \mathbf{Z} \mid \mathsf{inf}(int) \leq z \leq \mathsf{sup}(int)\}$$

where inf and sup are as in Example 4.10. Thus γ_{ZI} will extract the set of elements described by the interval, e.g. $\gamma_{\mathrm{ZI}}([0,3]) = \{0,1,2,3\}$ and $\gamma_{\mathrm{ZI}}([0,\infty]) = \{z \in \mathbf{Z} \mid z \geq 0\}$.

The abstraction function $\alpha_{\mathrm{ZI}} : \mathcal{P}(\mathbf{Z}) \to \mathbf{Interval}$ is defined by

$$\alpha_{\mathrm{ZI}}(Z) = \begin{cases} \bot & \text{if } Z = \emptyset \\ [\mathsf{inf}'(Z), \mathsf{sup}'(Z)] & \text{otherwise} \end{cases}$$

where inf′ and sup′ are as in Example 4.10. Thus α_{ZI} will determine the smallest interval that includes all the elements of the set, e.g. $\alpha_{\mathrm{ZI}}(\{0,1,3\}) = [0,3]$ and $\alpha_{\mathrm{ZI}}(\{2 * z \mid z > 0\}) = [2,\infty]$.

Let us verify that $(\mathcal{P}(\mathbf{Z}), \alpha_{\mathrm{ZI}}, \gamma_{\mathrm{ZI}}, \mathbf{Interval})$ is indeed a Galois connection. It is easy to see that α_{ZI} and γ_{ZI} are monotone functions. We shall next prove that (4.8) holds, i.e. that $\gamma_{\mathrm{ZI}} \circ \alpha_{\mathrm{ZI}} \sqsupseteq \lambda Z.Z$. If $Z \neq \emptyset$ we have:

$$\begin{aligned} \gamma_{\mathrm{ZI}}(\alpha_{\mathrm{ZI}}(Z)) &= \gamma_{\mathrm{ZI}}([\mathsf{inf}'(Z), \mathsf{sup}'(Z)]) \\ &= \{z \in \mathbf{Z} \mid \mathsf{inf}'(Z) \leq z \leq \mathsf{sup}'(Z)\} \\ &\supseteq Z \end{aligned}$$

For $Z = \emptyset$ we trivially have $\gamma_{\mathrm{ZI}}(\alpha_{\mathrm{ZI}}(\emptyset)) = \gamma_{\mathrm{ZI}}(\bot) = \emptyset$ so we have proved (4.8). Intuitively, this condition expresses that if we start with a subset of $\mathbf{Z}$, find the smallest interval containing it, and next determine the corresponding subset of $\mathbf{Z}$, then we will get a (possibly) larger subset of $\mathbf{Z}$ than the one we started with.

Finally, we shall prove (4.9), i.e. that $\alpha_{\mathrm{ZI}} \circ \gamma_{\mathrm{ZI}} \sqsubseteq \lambda int.int$. Consider first $int = [z_1, z_2]$ where we have:

$$\begin{aligned} \alpha_{\mathrm{ZI}}(\gamma_{\mathrm{ZI}}([z_1, z_2])) &= \alpha_{\mathrm{ZI}}(\{z \in \mathbf{Z} \mid z_1 \leq z \leq z_2\}) \\ &= [z_1, z_2] \end{aligned}$$

For $int = \bot$ we trivially have $\alpha_{\mathrm{ZI}}(\gamma_{\mathrm{ZI}}(\bot)) = \alpha_{\mathrm{ZI}}(\emptyset) = \bot$ so we have proved (4.9). Intuitively, the condition expresses that if we start with an interval, determine the corresponding subset of $\mathbf{Z}$, and next find the smallest interval containing this set, then the resulting interval will be included in the interval we started with; actually we showed that the two intervals are equal. ∎

Adjunctions. There is an alternative formulation of the Galois connection (L, α, γ, M) that is frequently easier to work with. We define (L, α, γ, M) to be an *adjunction* between complete lattices $L = (L, \sqsubseteq)$ and $M = (M, \sqsubseteq)$ if and only if

$$\alpha : L \to M \text{ and } \gamma : M \to L \text{ are total functions}$$

that satisfy

$$\alpha(l) \sqsubseteq m \quad \Leftrightarrow \quad l \sqsubseteq \gamma(m) \tag{4.10}$$

for all $l \in L$ and $m \in M$.

Condition (4.10) expresses that α and γ "respect" the orderings of the two lattices: If an element $l \in L$ is safely described by the element $m \in M$, i.e. $\alpha(l) \sqsubseteq m$, then it is also the case that the element described by m is safe with respect to l, i.e. $l \sqsubseteq \gamma(m)$.

Proposition 4.20
(L, α, γ, M) is an adjunction if and only if (L, α, γ, M) is a Galois connection.

Proof First assume that (L, α, γ, M) is a Galois connection and let us show that it is an adjunction. So assume first that $\alpha(l) \sqsubseteq m$; since γ is monotone we get $\gamma(\alpha(l)) \sqsubseteq \gamma(m)$; using that $\gamma \circ \alpha \sqsupseteq \lambda l.l$ we then get $l \sqsubseteq \gamma(\alpha(l)) \sqsubseteq \gamma(m)$ as required. The proof showing that $l \sqsubseteq \gamma(m)$ implies $\alpha(l) \sqsubseteq m$ is analogous.

Next assume that (L, α, γ, M) is an adjunction and let us prove that it is a Galois connection. First we prove that $\gamma \circ \alpha \sqsupseteq \lambda l.l$: for $l \in L$ we trivially have $\alpha(l) \sqsubseteq \alpha(l)$ and using that $\alpha(l) \sqsubseteq m \Rightarrow l \sqsubseteq \gamma(m)$ we get $l \sqsubseteq \gamma(\alpha(l))$ as required. The proof showing that $\alpha \circ \gamma \sqsubseteq \lambda m.m$ is analogous. To complete the proof we also have to show that α and γ are monotone. To see that α is monotone suppose that $l_1 \sqsubseteq l_2$; we have already proved that $\gamma \circ \alpha \sqsupseteq \lambda l.l$ so we have $l_1 \sqsubseteq l_2 \sqsubseteq \gamma(\alpha(l_2))$; using that $l \sqsubseteq \gamma(m) \Rightarrow \alpha(l) \sqsubseteq m$ we then get $\alpha(l_1) \sqsubseteq \alpha(l_2)$. The proof showing that γ is monotone is analogous. ∎

Galois connections defined by extraction functions. We shall now see that representation functions (introduced in Section 4.1) can be used to define Galois connections. So consider once again the representation function $\beta : V \to L$ mapping the values of V to the properties of the complete lattice L. It gives rise to a Galois connection

$$(\mathcal{P}(V), \alpha, \gamma, L)$$

between $\mathcal{P}(V)$ and L where the abstraction and concretisation functions are defined by

$$\begin{array}{rcl} \alpha(V') & = & \bigsqcup\{\beta(v) \mid v \in V'\} \\ \gamma(l) & = & \{v \in V \mid \beta(v) \sqsubseteq l\} \end{array}$$

for $V' \subseteq V$ and $l \in L$. Let us pause for a minute to see that this indeed defines an adjunction:

$$\begin{array}{rcl} \alpha(V') \sqsubseteq l & \Leftrightarrow & \bigsqcup\{\beta(v) \mid v \in V'\} \sqsubseteq l \\ & \Leftrightarrow & \forall v \in V' : \beta(v) \sqsubseteq l \\ & \Leftrightarrow & V' \subseteq \gamma(l) \end{array}$$

It follows from Proposition 4.20 that we also have a Galois connection. It is also immediate that $\alpha(\{v\}) = \beta(v)$ as illustrated by the diagram:

$$\begin{tikzcd} \mathcal{P}(V) \arrow[rr, shift right, "\alpha"'] & & L \arrow[ll, shift right, "\gamma"'] \\ & V \arrow[ul, "\{\cdot\}"] \arrow[ur, "\beta"'] & \end{tikzcd}$$

A special case of the above construction that is frequently useful is when $L = (\mathcal{P}(D), \subseteq)$ for some set D and we have an *extraction function*

$$\eta : V \to D$$

mapping the values of V to their descriptions in D. We will then define the representation function $\beta_\eta : V \to \mathcal{P}(D)$ by $\beta_\eta(v) = \{\eta(v)\}$ and the Galois connection between $\mathcal{P}(V)$ and $\mathcal{P}(D)$ will now be written

$$(\mathcal{P}(V), \alpha_\eta, \gamma_\eta, \mathcal{P}(D))$$

where

$$\begin{array}{rcl} \alpha_\eta(V') & = & \bigcup\{\beta_\eta(v) \mid v \in V'\} \quad = \quad \{\eta(v) \mid v \in V'\} \\ \gamma_\eta(D') & = & \{v \in V \mid \beta_\eta(v) \subseteq D'\} \quad = \quad \{v \mid \eta(v) \in D'\} \end{array}$$

for $V' \subseteq V$ and $D' \subseteq D$. The relationship between η, β_η, α_η and γ_η is illustrated by the diagram:

Figure 4.7: The complete lattice $\mathcal{P}(\mathbf{Sign}) = (\mathcal{P}(\mathbf{Sign}), \subseteq)$.

$$\begin{tikzcd}
\mathcal{P}(V) \arrow[r, shift right, "\alpha_\eta"'] & \mathcal{P}(D) \arrow[l, shift right, "\gamma_\eta"'] \\
V \arrow[u, "\{\cdot\}"] \arrow[r, "\eta"] \arrow[ur, "\beta_\eta"'] & D \arrow[u, "\{\cdot\}"']
\end{tikzcd}$$

Example 4.21 Let us consider the two complete lattices $(\mathcal{P}(\mathbf{Z}), \subseteq)$ and $(\mathcal{P}(\mathbf{Sign}), \subseteq)$ where $\mathbf{Sign} = \{-, 0, +\}$; see Figure 4.7. The extraction function

$$\mathsf{sign} : \mathbf{Z} \rightarrow \mathbf{Sign}$$

simply defines the signs of the integers and is specified by:

$$\mathsf{sign}(z) = \begin{cases} - & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ + & \text{if } z > 0 \end{cases}$$

The above construction then gives us a Galois connection

$$(\mathcal{P}(\mathbf{Z}), \alpha_{\mathsf{sign}}, \gamma_{\mathsf{sign}}, \mathcal{P}(\mathbf{Sign}))$$

with

$$\begin{aligned} \alpha_{\mathsf{sign}}(Z) &= \{\mathsf{sign}(z) \mid z \in Z\} \\ \gamma_{\mathsf{sign}}(S) &= \{z \in \mathbf{Z} \mid \mathsf{sign}(z) \in S\} \end{aligned}$$

where $Z \subseteq \mathbf{Z}$ and $S \subseteq \mathbf{Sign}$. ∎

4.3.1 Properties of Galois Connections

We shall present three interesting results. The first result says that a Galois connection is fully determined by either one of the abstraction and concretisation functions:

Lemma 4.22 If (L, α, γ, M) is a Galois connection then:

(i) α uniquely determines γ by $\gamma(m) = \bigsqcup\{l \mid \alpha(l) \sqsubseteq m\}$ and γ uniquely determines α by $\alpha(l) = \bigsqcap\{m \mid l \sqsubseteq \gamma(m)\}$.

(ii) α is completely additive and γ is completely multiplicative.

In particular $\alpha(\bot) = \bot$ and $\gamma(\top) = \top$. ∎

Proof To show (i) we shall first show that γ is determined by α. Since (L, α, γ, M) is an adjunction by Proposition 4.20 we have $\gamma(m) = \bigsqcup\{l \mid l \sqsubseteq \gamma(m)\} = \bigsqcup\{l \mid \alpha(l) \sqsubseteq m\}$. This shows that α uniquely determines γ: if both (L, α, γ_1, M) and (L, α, γ_2, M) are Galois connection then $\gamma_1(m) = \bigsqcup\{l \mid \alpha(l) \sqsubseteq m\} = \gamma_2(m)$ for all m and hence $\gamma_1 = \gamma_2$.

Similarly, we have $\alpha(l) = \bigsqcap\{m \mid \alpha(l) \sqsubseteq m\} = \bigsqcap\{m \mid l \sqsubseteq \gamma(m)\}$ and this shows that γ uniquely determines α.

To show (ii) consider $L' \subseteq L$; using Proposition 4.20 we then have

$$\begin{aligned}
\alpha(\bigsqcup L') \sqsubseteq m \quad &\Leftrightarrow \quad \bigsqcup L' \sqsubseteq \gamma(m) \\
&\Leftrightarrow \quad \forall l \in L' : l \sqsubseteq \gamma(m) \\
&\Leftrightarrow \quad \forall l \in L' : \alpha(l) \sqsubseteq m \\
&\Leftrightarrow \quad \bigsqcup\{\alpha(l) \mid l \in L'\} \sqsubseteq m
\end{aligned}$$

and it follows that $\alpha(\bigsqcup L') = \bigsqcup\{\alpha(l) \mid l \in L'\}$.

The proof that $\gamma(\bigsqcap M') = \bigsqcap\{\gamma(m) \mid m \in M'\}$ is analogous. ∎

Motivated by Lemma 4.22 we shall say that if (L, α, γ, M) is a Galois connection then α is the *lower adjoint* (or left adjoint) of γ and that γ is the *upper adjoint* (or right adjoint) of α.

The next result shows that it suffices to specify either a completely additive abstraction function or a completely multiplicative concretisation function in order to obtain a Galois connection:

Lemma 4.23 If $\alpha : L \to M$ is completely additive then there exists $\gamma : M \to L$ such that (L, α, γ, M) is a Galois connection. Similarly, if $\gamma : M \to L$ is completely multiplicative then there exists $\alpha : L \to M$ such that (L, α, γ, M) is a Galois connection. ∎

Proof Consider the claim for α and define γ by:

$$\gamma(m) = \bigsqcup\{l' \mid \alpha(l') \sqsubseteq m\}$$

Then we have $\alpha(l) \sqsubseteq m \Rightarrow l \in \{l' \mid \alpha(l') \sqsubseteq m\} \Rightarrow l \sqsubseteq \gamma(m)$ where the last implication follows from the definition of γ. For the other direction we first observe that $l \sqsubseteq \gamma(m) \Rightarrow \alpha(l) \sqsubseteq \alpha(\gamma(m))$ because α is completely additive and hence monotone. Now

$$\begin{aligned}\alpha(\gamma(m)) &= \alpha(\bigsqcup\{l' \mid \alpha(l') \sqsubseteq m\}) \\ &= \bigsqcup\{\alpha(l') \mid \alpha(l') \sqsubseteq m\} \\ &\sqsubseteq m\end{aligned}$$

so $l \sqsubseteq \gamma(m) \Rightarrow \alpha(l) \sqsubseteq m$. It follows that (L, α, γ, M) is an adjunction and hence a Galois connection by Proposition 4.20.

The proof of the claim for γ is analogous. ■

The final result shows that we neither lose nor gain precision by iterating abstraction and concretisation:

Fact 4.24 If (L, α, γ, M) is a Galois connection then $\alpha \circ \gamma \circ \alpha = \alpha$ and $\gamma \circ \alpha \circ \gamma = \gamma$. ■

Proof We have $\lambda l.l \sqsubseteq \gamma \circ \alpha$ and since α is monotone we get $\alpha \sqsubseteq \alpha \circ (\gamma \circ \alpha)$. Similarly $(\alpha \circ \gamma) \circ \alpha \sqsubseteq \alpha$ follows from $\alpha \circ \gamma \sqsubseteq \lambda m.m$. Thus $\alpha \circ \gamma \circ \alpha = \alpha$.

The proof of $\gamma \circ \alpha \circ \gamma = \gamma$ is analogous. ■

Example 4.25 As a somewhat more complex example consider the complete lattices **Interval** = (**Interval**, $\sqsubseteq$) and $\mathcal{P}$(**Sign**) = ($\mathcal{P}$(**Sign**), $\subseteq$) of Figures 4.2 and 4.7, respectively.

Let us define a concretisation function $\gamma_{\mathrm{IS}} : \mathcal{P}(\mathbf{Sign}) \to \mathbf{Interval}$ by:

$$\begin{aligned}\gamma_{\mathrm{IS}}(\{-,0,+\}) &= [-\infty,\infty] & \qquad \gamma_{\mathrm{IS}}(\{-,0\}) &= [-\infty,0] \\ \gamma_{\mathrm{IS}}(\{-,+\}) &= [-\infty,\infty] & \gamma_{\mathrm{IS}}(\{0,+\}) &= [0,\infty] \\ \gamma_{\mathrm{IS}}(\{-\}) &= [-\infty,-1] & \gamma_{\mathrm{IS}}(\{0\}) &= [0,0] \\ \gamma_{\mathrm{IS}}(\{+\}) &= [1,\infty] & \gamma_{\mathrm{IS}}(\emptyset) &= \bot\end{aligned}$$

To determine whether or not there exists an abstraction function

$$\alpha_{\mathrm{IS}} : \mathbf{Interval} \to \mathcal{P}(\mathbf{Sign})$$

such that (**Interval**, $\alpha_{\mathrm{IS}}, \gamma_{\mathrm{IS}}, \mathcal{P}$(**Sign**)) is a Galois connection, we shall simply determine whether or not γ_{IS} is completely multiplicative: If it is, then Lemma 4.23 guarantees that there does indeed exist a Galois connection and Lemma 4.22 tells us how to construct the abstraction function. If γ_{IS} is

not completely multiplicative then Lemma 4.22 guarantees that there cannot exist a Galois connection. In order to determine whether or not γ_{IS} is completely multiplicative we shall use Lemma A.4 of Appendix A: It is immediate to verify that $\mathcal{P}(\mathbf{Sign})$ is finite and that γ_{IS} satisfies conditions (i) and (ii) of Lemma A.4. To verify condition (iii) we need to compare $\gamma_{\mathrm{IS}}(S_1 \cap S_2)$ to $\gamma_{\mathrm{IS}}(S_1) \sqcap \gamma_{\mathrm{IS}}(S_2)$ for all pairs $(S_1, S_2) \in \mathcal{P}(\mathbf{Sign}) \times \mathcal{P}(\mathbf{Sign})$ of incomparable sets of signs, i.e. all pairs in the following list:

$$\begin{array}{lll} (\{-,0\},\{-,+\}), & (\{-,0\},\{0,+\}), & (\{-,0\},\{+\}), \\ (\{-,+\},\{0,+\}), & (\{-,+\},\{0\}), & (\{0,+\},\{-\}), \\ (\{-\},\{0\}), & (\{-\},\{+\}), & (\{0\},\{+\}) \end{array}$$

In checking the pair $(\{-,0\},\{-,+\})$ we calculate

$$\begin{array}{rclcl} \gamma_{\mathrm{IS}}(\{-,0\} \cap \{-,+\}) & = & \gamma_{\mathrm{IS}}(\{-\}) & = & [-\infty,-1] \\ \gamma_{\mathrm{IS}}(\{-,0\}) \sqcap \gamma_{\mathrm{IS}}(\{-,+\}) & = & [-\infty,0] \sqcap [-\infty,\infty] & = & [-\infty,0] \end{array}$$

and we deduce that γ_{IS} is *not* completely multiplicative. Hence according to Lemma 4.22 we cannot find any $\alpha_{\mathrm{IS}} : \mathbf{Interval} \to \mathcal{P}(\mathbf{Sign})$ such that $(\mathbf{Interval}, \alpha_{\mathrm{IS}}, \gamma_{\mathrm{IS}}, \mathcal{P}(\mathbf{Sign}))$ is a Galois connection. ∎

The mundane approach to correctness. We shall now return to further motivating that the connection between L and M should be a Galois connection. Recall from Section 4.1 that the semantic correctness of an analysis may be expressed by a *correctness relation* R between the values of V and the properties of L or it may be expressed by a representation function β mapping the values of V to their description in L. When we replace L with some other complete lattice M we would obviously like the correctness results still to hold. We shall now see that if there is a Galois connection (L, α, γ, M) between L and M then we can construct a correctness relation between V and M and a representation function from V to M.

Let us first focus on the correctness relation. So let $R : V \times L \to \{true, false\}$ be a correctness relation that satisfies the conditions (4.4) and (4.5) of Section 4.1. Further let (L, α, γ, M) be a Galois connection between the complete lattices L and M. It is then natural to define $S : V \times M \to \{true, false\}$ by

$$v \; S \; m \;\; \text{iff} \;\; v \; R \; (\gamma(m))$$

and we shall now argue that this is a correctness relation between V and M fulfilling conditions corresponding to (4.4) and (4.5). We have

$$\begin{array}{rcl} (v \; S \; m_1) \;\wedge\; m_1 \sqsubseteq m_2 & \Rightarrow & v \; R \; (\gamma(m_1)) \;\wedge\; \gamma(m_1) \sqsubseteq \gamma(m_2) \\ & \Rightarrow & v \; R \; (\gamma(m_2)) \\ & \Rightarrow & v \; S \; m_2 \end{array}$$

using that γ is monotone and that R satisfies condition (4.4); this shows that S also satisfies condition (4.4). Also for all $M' \subseteq M$ we have

$$\begin{aligned}(\forall m \in M' : v\ S\ m) &\Rightarrow (\forall m \in M' : v\ R\ (\gamma(m))) \\ &\Rightarrow v\ R\ (\bigsqcap\{\gamma(m) \mid m \in M'\}) \\ &\Rightarrow v\ R\ (\gamma(\bigsqcap M')) \\ &\Rightarrow v\ S\ (\bigsqcap M')\end{aligned}$$

using that γ is completely multiplicative (Lemma 4.22) and that R satisfies condition (4.5); this shows that S also satisfies condition (4.5). Hence S defines a correctness relation between V and M.

Continuing the above line of reasoning assume now that R is *generated by* the *representation function* $\beta : V \to L$, i.e. $v\ R\ l \Leftrightarrow \beta(v) \sqsubseteq l$. Since (L, α, γ, M) is a Galois connection and hence an adjunction (Proposition 4.20) we may calculate

$$\begin{aligned}v\ S\ m &\Leftrightarrow v\ R\ (\gamma(m)) \\ &\Leftrightarrow \beta(v) \sqsubseteq \gamma(m) \\ &\Leftrightarrow (\alpha \circ \beta)(v) \sqsubseteq m\end{aligned}$$

showing that S is *generated by* $\alpha \circ \beta : V \to M$. This shows how the Galois connection facilitates the definition of correctness relations and representation functions and concludes our motivation for why it is natural to require that the connection between L and M is specified by a Galois connection.

4.3.2 Galois Insertions

For a Galois connection (L, α, γ, M) there may be several elements of M that describe the same element of L, i.e. γ need not be injective, and this means that M may contain elements that are not relevant for the approximation of L.

The concept of Galois insertion is intended to rectify this: (L, α, γ, M) is a *Galois insertion* between the complete lattices $L = (L, \sqsubseteq)$ and $M = (M, \sqsubseteq)$ if and only if

$$\alpha : L \to M \text{ and } \gamma : M \to L \text{ are monotone functions}$$

that satisfy:

$$\begin{aligned}\gamma \circ \alpha &\sqsupseteq \lambda l.l \\ \alpha \circ \gamma &= \lambda m.m\end{aligned}$$

Thus we now require that we do not lose precision by first doing a concretisation and then an abstraction. As a consequence M cannot contain elements

Figure 4.8: The Galois insertion (L, α, γ, M).

that do not describe elements of L, i.e. M does not contain superfluous elements.

Example 4.26 The calculations of Example 4.19 show that

$$(\mathcal{P}(\mathbf{Z}), \alpha_{\mathrm{ZI}}, \gamma_{\mathrm{ZI}}, \mathbf{Interval})$$

is indeed a Galois insertion: we start with an interval, use γ_{ZI} to determine the set of integers it describes and next use α_{ZI} to determine the smallest interval containing this set and we get exactly the same interval as we started with. ∎

The concept of a Galois insertion is illustrated in Figure 4.8 and is further clarified by:

Lemma 4.27 For a Galois connection (L, α, γ, M) the following claims are equivalent:

(i) (L, α, γ, M) is a Galois insertion;

(ii) α is surjective: $\forall m \in M : \exists l \in L : \alpha(l) = m$;

(iii) γ is injective: $\forall m_1, m_2 \in M : \gamma(m_1) = \gamma(m_2) \Rightarrow m_1 = m_2$; and

(iv) γ is an order-similarity: $\forall m_1, m_2 \in M :$
$$\gamma(m_1) \sqsubseteq \gamma(m_2) \Leftrightarrow m_1 \sqsubseteq m_2.$$ ∎

Proof First we prove that (i) ⇒ (iii). For this we calculate

$$\gamma(m_1) = \gamma(m_2) \;\Rightarrow\; \alpha(\gamma(m_1)) = \alpha(\gamma(m_2)) \;\Rightarrow\; m_1 = m_2$$

where the last step follows from (i).

Next we prove that (iii) $\Rightarrow$ (ii). For $m \in M$ we have $\gamma(\alpha(\gamma(m)) = \gamma(m)$ by Fact 4.24 and hence $\alpha(\gamma(m)) = m$ by (iii).

Now we prove that (ii) $\Rightarrow$ (iv). That $m_1 \sqsubseteq m_2 \Rightarrow \gamma(m_1) \sqsubseteq \gamma(m_2)$ is immediate since γ is monotone. Next suppose that $\gamma(m_1) \sqsubseteq \gamma(m_2)$; by monotonicity of α this gives $\alpha(\gamma(m_1)) \sqsubseteq \alpha(\gamma(m_2))$; using (ii) we can write $m_1 = \alpha(l_1)$ and $m_2 = \alpha(l_2)$ for some $l_1, l_2 \in L$; using Fact 4.24 this then gives $m_1 \sqsubseteq m_2$ as desired.

Finally we prove that (iv) $\Rightarrow$ (i). For this we calculate

$$\alpha(\gamma(m_1)) \sqsubseteq m_2 \;\Leftrightarrow\; \gamma(m_1) \sqsubseteq \gamma(m_2) \;\Leftrightarrow\; m_1 \sqsubseteq m_2$$

where the first step is using Proposition 4.20 and the last step is using (iv). This shows that $\alpha(\gamma(m_1))$ and m_1 have the same upper bounds and hence are equal. ∎

Lemma 4.27 has an interesting consequence for a Galois connection given by an extraction function $\eta : V \to D$: it is a Galois insertion if and only if η is surjective.

Example 4.28 Consider the complete lattices $(\mathcal{P}(\mathbf{Z}), \subseteq)$ and $(\mathcal{P}(\mathbf{Sign} \times \mathbf{Parity}), \subseteq)$ where $\mathbf{Sign} = \{-, 0, +\}$ as before and $\mathbf{Parity} = \{\mathsf{odd}, \mathsf{even}\}$. Define the extraction function $\mathsf{signparity} : \mathbf{Z} \to \mathbf{Sign} \times \mathbf{Parity}$ by:

$$\mathsf{signparity}(z) = \begin{cases} (\mathsf{sign}(z), \mathsf{odd}) & \text{if } z \text{ is odd} \\ (\mathsf{sign}(z), \mathsf{even}) & \text{if } z \text{ is even} \end{cases}$$

This gives rise to a Galois connection $(\mathcal{P}(\mathbf{Z}), \alpha_{\mathsf{signparity}}, \gamma_{\mathsf{signparity}}, \mathcal{P}(\mathbf{Sign} \times \mathbf{Parity}))$. The property $(0, \mathsf{odd})$ describes no integers so clearly $\mathsf{signparity}$ is not surjective and we have an example of a Galois connection that is not a Galois insertion. ∎

Construction of Galois insertions. Given a Galois connection (L, α, γ, M) it is always possible to obtain a Galois insertion by enforcing that the concretisation function γ is injective. Basically, this amounts to removing elements from the complete lattice M using a *reduction operator*

$$\varsigma : M \to M$$

defined from the Galois connection. We have the following result:

Proposition 4.29
Let (L, α, γ, M) be a Galois connection and define the reduction operator $\varsigma : M \to M$ by

$$\varsigma(m) = \bigsqcap \{m' \mid \gamma(m) = \gamma(m')\}$$

Then $\varsigma[M] = (\{\varsigma(m) \mid m \in M\}, \sqsubseteq_M)$ is a complete lattice and $(L, \alpha, \gamma, \varsigma[M])$ is a Galois insertion.

Figure 4.9: The reduction operator $\varsigma : M \to M$.

This is illustrated in Figure 4.9: The idea is that two elements of M are identified by ς if they map to the same value by γ; in particular, m and $\alpha(\gamma(m))$ will be identified.

In preparation for the proof we shall first show the following two results:

Fact 4.30 Let (L, α, γ, M) be a Galois connection. Then

$$\alpha[L] = (\{\alpha(l) \mid l \in L\}, \sqsubseteq_M)$$

is a complete lattice. ∎

Proof Clearly $\alpha[L] \subseteq M$ is partially ordered by the ordering $\sqsubseteq_M$ of $M = (M, \sqsubseteq_M)$. We now want to show for $M' \subseteq \alpha[L] \subseteq M$ that

$$\bigsqcup_{\alpha[L]} M' = \bigsqcup_M M'$$

We first show that $\bigsqcup_M M' \in \alpha[L]$ which means that there exists $l \in L$ such that $\alpha(l) = \bigsqcup_M M'$. For this take $l = \bigsqcup_L \gamma[M']$ which clearly exists in L. Since $\alpha \circ \gamma \circ \alpha = \alpha$ (Fact 4.24) and α is completely additive (Lemma 4.22) we then have

$$\begin{aligned} \alpha(l) &= \alpha(\textstyle\bigsqcup_L \gamma[M']) \\ &= \textstyle\bigsqcup_M \alpha[\gamma[M']] \\ &= \textstyle\bigsqcup_M M' \end{aligned}$$

thus showing $\bigsqcup_M M' \in \alpha[L]$. Since $\bigsqcup_M M'$ is the least upper bound of M' in M it follows that it also is in $\alpha[L]$. By Lemma A.2 we then have the result. ∎

Fact 4.31 If (L, α, γ, M) is a Galois connection and

$$\varsigma(m) = \bigsqcap\{m' \mid \gamma(m') = \gamma(m)\}$$

then

$$\varsigma(m) = \alpha(\gamma(m))$$

and hence $\alpha[L] = \varsigma[M]$. ∎

Proof We have $\varsigma(m) \sqsubseteq \alpha(\gamma(m))$ because $\gamma(m) = \gamma(\alpha(\gamma(m)))$ using Fact 4.24. That $\alpha(\gamma(m)) \sqsubseteq \varsigma(m)$ is equivalent to $\gamma(m) \sqsubseteq \gamma(\varsigma(m))$ using Proposition 4.20 and $\gamma(\varsigma(m)) = \bigsqcap\{\gamma(m') \mid \gamma(m') = \gamma(m)\} = \gamma(m)$ follows from Lemma 4.22.

Next consider $\alpha(l)$ for $l \in L$; by Fact 4.24 we have $\alpha(l) = \alpha(\gamma(\alpha(l)))$ and hence $\alpha(l) = \varsigma(\alpha(l))$; this shows $\alpha[L] \subseteq \varsigma[M]$. Finally consider $\varsigma(m)$ for $m \in M$; then $\varsigma(m) = \alpha(\gamma(m))$; this shows $\varsigma[M] \subseteq \alpha[L]$. ∎

We now turn to the proof of Proposition 4.29:

Proof Facts 4.30 and 4.31 give that $\varsigma[M] = \alpha[L]$ is a complete lattice. Since (L, α, γ, M) is an adjunction (Proposition 4.20) it follows that also $(L, \alpha, \gamma, \alpha[L])$ is. Since α is surjective on $\alpha[L]$ it follows from Lemma 4.27 that we have a Galois insertion. ∎

Reduction operators defined by extraction functions. We shall now specialise the construction of Proposition 4.29 to the setting where a Galois connection $(\mathcal{P}(V), \alpha_\eta, \gamma_\eta, \mathcal{P}(D))$ is given by an *extraction function* $\eta : V \to D$. Then the reduction operator ς_η is given by

$$\varsigma_\eta(D') = D' \cap \eta[V]$$

where $\eta[V] = \{\eta(v) \mid v \in V\}$ denotes the image of η and furthermore $\varsigma_\eta[\mathcal{P}(D)]$ is isomorphic (see Appendix A) to $\mathcal{P}(\eta[V])$ (see Exercise 4.14). The resulting Galois insertion will therefore be isomorphic to $(\mathcal{P}(V), \alpha_\eta, \gamma_\eta, \mathcal{P}(\eta[V]))$: Formally, the Galois connection $(L_1, \alpha_1, \gamma_1, M_1)$ is *isomorphic* to the Galois connection $(L_2, \alpha_2, \gamma_2, M_2)$ when there are isomorphisms $\theta_L : L_1 \to L_2$ and $\theta_M : M_1 \to M_2$ (see Appendix A) such that $\alpha_2 = \theta_M \circ \alpha_1 \circ \theta_L^{-1}$ and $\gamma_2 = \theta_L \circ \gamma_1 \circ \theta_M^{-1}$.

Example 4.32 Returning to Example 4.28 we can use the above technique to construct a Galois insertion. Now

$$\mathsf{signparity}[\mathbf{Z}] = \{(\mathsf{-}, \mathsf{odd}), (\mathsf{-}, \mathsf{even}), (\mathsf{0}, \mathsf{even}), (\mathsf{+}, \mathsf{odd}), (\mathsf{+}, \mathsf{even})\}$$

showing that $(\mathcal{P}(\mathbf{Z}), \alpha_{\mathsf{signparity}}, \gamma_{\mathsf{signparity}}, \mathcal{P}(\mathsf{signparity}[\mathbf{Z}]))$ is the resulting Galois insertion. ∎

4.4 Systematic Design of Galois Connections

Sequential composition. When developing a program analysis it is often useful to do so in stages: The starting point will typically be a complete lattice $(L_0, \sqsubseteq)$ fairly closely related to the semantics; an example is

$(\mathcal{P}(V), \subseteq)$. We may then decide to use a more approximate set of properties and introduce the complete lattice $(L_1, \sqsubseteq)$ related to L_0 by a Galois connection $(L_0, \alpha_1, \gamma_1, L_1)$. This step can then be repeated any number of times: We replace one complete lattice L_i of properties with a more approximate complete lattice $(L_{i+1}, \sqsubseteq)$ related to L_i by a Galois connection $(L_i, \alpha_{i+1}, \gamma_{i+1}, L_{i+1})$. This process will stop when we have an analysis with the required computational properties. So the situation can be depicted as follows:

$$L_0 \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} L_1 \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} L_2 \underset{\alpha_3}{\overset{\gamma_3}{\leftrightarrows}} \cdots \underset{\alpha_k}{\overset{\gamma_k}{\leftrightarrows}} L_k$$

The above sequence of approximations of the analysis could as well have been done in one step, i.e. the "functional composition" of two Galois connections is also a Galois connection. Formally, if $(L_0, \alpha_1, \gamma_1, L_1)$ and $(L_1, \alpha_2, \gamma_2, L_2)$ are Galois connections, then

$$(L_0, \alpha_2 \circ \alpha_1, \gamma_1 \circ \gamma_2, L_2)$$

is also a Galois connection. To verify this we simply observe that $\alpha_2(\alpha_1(l_0)) \sqsubseteq l_2 \Leftrightarrow \alpha_1(l_0) \sqsubseteq \gamma_2(l_2) \Leftrightarrow l_0 \sqsubseteq \gamma_1(\gamma_2(l_2))$ and, using Proposition 4.20, this shows the result. A similar result holds for Galois insertions because the functional composition of surjective functions yields a surjective function.

Each of the Galois connections $(L_i, \alpha_{i+1}, \gamma_{i+1}, L_{i+1})$ may have been obtained by combining other Galois connections and we shall shortly introduce a number of techniques for doing so. We shall illustrate these techniques in a sequence of examples that in the end will give us a finite complete lattice that has turned out to be very useful in practice for performing an *Array Bound Analysis.*

Example 4.33 One of the components in the Array Bound Analysis is concerned with approximating the difference in magnitude between two numbers (typically the bound and the index). We shall proceed in two stages: First we shall approximate pairs (z_1, z_2) of integers by their difference in magnitude $|z_1| - |z_2|$ and next we shall further approximate this difference using a finite lattice. The two Galois connections will be defined by extraction functions and they will then be combined by taking their functional composition.

The first stage is specified by the Galois connection

$$(\mathcal{P}(\mathbf{Z} \times \mathbf{Z}), \alpha_{\mathsf{diff}}, \gamma_{\mathsf{diff}}, \mathcal{P}(\mathbf{Z}))$$

where $\mathsf{diff} : \mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}$ is the extraction function calculating the difference in magnitude:

$$\mathsf{diff}(z_1, z_2) = |z_1| - |z_2|$$

The abstraction and concretisation functions α_{diff} and γ_{diff} will then be

$$\begin{array}{rcl} \alpha_{\mathsf{diff}}(ZZ) & = & \{|z_1|-|z_2| \mid (z_1,z_2) \in ZZ\} \\ \gamma_{\mathsf{diff}}(Z) & = & \{(z_1,z_2) \mid |z_1|-|z_2| \in Z\} \end{array}$$

for $ZZ \subseteq \mathbf{Z} \times \mathbf{Z}$ and $Z \subseteq \mathbf{Z}$.

The second stage is specified by the Galois connection

$$(\mathcal{P}(\mathbf{Z}), \alpha_{\mathsf{range}}, \gamma_{\mathsf{range}}, \mathcal{P}(\mathbf{Range}))$$

where $\mathbf{Range} = \{\mathsf{<-1}, \mathsf{-1}, \mathsf{0}, \mathsf{+1}, \mathsf{>+1}\}$. The extraction function $\mathsf{range} : \mathbf{Z} \to \mathbf{Range}$ clarifies the meaning of the elements of $\mathbf{Range}$:

$$\mathsf{range}(z) = \begin{cases} \mathsf{<-1} & \text{if } z < -1 \\ \mathsf{-1} & \text{if } z = -1 \\ \mathsf{0} & \text{if } z = 0 \\ \mathsf{+1} & \text{if } z = 1 \\ \mathsf{>+1} & \text{if } z > 1 \end{cases}$$

The abstraction and concretisation functions α_{range} and γ_{range} will then be

$$\begin{array}{rcl} \alpha_{\mathsf{range}}(Z) & = & \{\mathsf{range}(z) \mid z \in Z\} \\ \gamma_{\mathsf{range}}(R) & = & \{z \mid \mathsf{range}(z) \in R\} \end{array}$$

for $Z \subseteq \mathbf{Z}$ and $R \subseteq \mathbf{Range}$.

We then have that the functional composition

$$(\mathcal{P}(\mathbf{Z} \times \mathbf{Z}), \alpha_{\mathrm{R}}, \gamma_{\mathrm{R}}, \mathcal{P}(\mathbf{Range}))$$

where $\alpha_{\mathrm{R}} = \alpha_{\mathsf{range}} \circ \alpha_{\mathsf{diff}}$ and $\gamma_{\mathrm{R}} = \gamma_{\mathsf{diff}} \circ \gamma_{\mathsf{range}}$, is a Galois connection. We obtain the following formulae for the abstraction and concretisation functions:

$$\begin{array}{rcl} \alpha_{\mathrm{R}}(ZZ) & = & \{\mathsf{range}(|z_1|-|z_2|) \mid (z_1,z_2) \in ZZ\} \\ \gamma_{\mathrm{R}}(R) & = & \{(z_1,z_2) \mid \mathsf{range}(|z_1|-|z_2|) \in R\} \end{array}$$

Using Exercise 4.15 we see that this is the Galois connection specified by the extraction function $\mathsf{range} \circ \mathsf{diff} : \mathbf{Z} \times \mathbf{Z} \to \mathbf{Range}$. ∎

A catalogue of combination techniques. We have seen that the "functional (or sequential) composition" of Galois connections gives rise to a new Galois connection. It is also useful to be able to do "parallel combinations" and this will be the topic for the remainder of this section. We may have analyses for the *individual* components of a composite structure and may want to combine them into an analysis for the whole structure; we shall see two techniques for that: the independent attribute method and the relational method. We shall also show how Galois connections can be used

to approximate the total function spaces and the monotone function spaces. Alternatively, we may have several analyses of the *same* structure and may want to combine them into one analysis and here the direct product and the direct tensor product allow us to do that. Using the notion of Galois insertions this leads to a study of the reduced product and the reduced tensor product.

The benefit of having such a catalogue of techniques is that a relative small set of "basic" analyses, whose correctness have been established, can be used to construct rather sophisticated analyses and, from an implementation point of view, opens up the possibility of reusing existing implementations. Finally, it is worth stressing that the complete lattices used in program analysis obviously can be constructed without using these techniques but that it often provides additional insights to view them as combinations of simpler Galois connections.

4.4.1 Component-wise Combinations

The first techniques we shall consider are applicable when we have several analyses of *individual* components of a structure and we want to combine them into a single analysis.

Independent attribute method. Let $(L_1, \alpha_1, \gamma_1, M_1)$ and $(L_2, \alpha_2, \gamma_2, M_2)$ be Galois connections. The *independent attribute method* will then give rise to a Galois connection

$$(L_1 \times L_2, \alpha, \gamma, M_1 \times M_2)$$

where:

$$\begin{aligned} \alpha(l_1, l_2) &= (\alpha_1(l_1), \alpha_2(l_2)) \\ \gamma(m_1, m_2) &= (\gamma_1(m_1), \gamma_2(m_2)) \end{aligned}$$

To see that this indeed does define a Galois connection we simply calculate

$$\begin{aligned} \alpha(l_1, l_2) \sqsubseteq (m_1, m_2) &\Leftrightarrow (\alpha_1(l_1), \alpha_2(l_2)) \sqsubseteq (m_1, m_2) \\ &\Leftrightarrow \alpha_1(l_1) \sqsubseteq m_1 \ \wedge \ \alpha_2(l_2) \sqsubseteq m_2 \\ &\Leftrightarrow l_1 \sqsubseteq \gamma_1(m_1) \ \wedge \ l_2 \sqsubseteq \gamma_2(m_2) \\ &\Leftrightarrow (l_1, l_2) \sqsubseteq (\gamma_1(m_1), \gamma_2(m_2)) \\ &\Leftrightarrow (l_1, l_2) \sqsubseteq \gamma(m_1, m_2) \end{aligned}$$

and use Proposition 4.20. A similar result holds for Galois insertions (Exercise 4.17).

Example 4.34 The Array Bound Analysis will contain a component that performs a Detection of Signs Analysis on pairs of integers. As a starting point, we take the Galois connection

$$(\mathcal{P}(\mathbf{Z}), \alpha_{\mathsf{sign}}, \gamma_{\mathsf{sign}}, \mathcal{P}(\mathbf{Sign}))$$

specified by the extraction function sign in Example 4.21. It can be used to analyse both components of a pair of integers so using the independent attribute method we will get a Galois connection

$$(\mathcal{P}(\mathbf{Z}) \times \mathcal{P}(\mathbf{Z}), \alpha_{\mathrm{SS}}, \gamma_{\mathrm{SS}}, \mathcal{P}(\mathbf{Sign}) \times \mathcal{P}(\mathbf{Sign}))$$

where α_{SS} and γ_{SS} are given by

$$\begin{array}{rcl} \alpha_{\mathrm{SS}}(Z_1, Z_2) & = & (\{\mathsf{sign}(z) \mid z \in Z_1\}, \{\mathsf{sign}(z) \mid z \in Z_2\}) \\ \gamma_{\mathrm{SS}}(S_1, S_2) & = & (\{z \mid \mathsf{sign}(z) \in S_1\}, \{z \mid \mathsf{sign}(z) \in S_2\}) \end{array}$$

where $Z_i \subseteq \mathbf{Z}$ and $S_i \subseteq \mathbf{Sign}$.

This Galois connection cannot be described using an extraction function because neither $\mathcal{P}(\mathbf{Z}) \times \mathcal{P}(\mathbf{Z})$ nor $\mathcal{P}(\mathbf{Sign}) \times \mathcal{P}(\mathbf{Sign})$ is a powerset. However, they are both isomorphic to powersets:

$$\begin{array}{rcl} \mathcal{P}(\mathbf{Z}) \times \mathcal{P}(\mathbf{Z}) & \cong & \mathcal{P}(\{1,2\} \times \mathbf{Z}) \\ \mathcal{P}(\mathbf{Sign}) \times \mathcal{P}(\mathbf{Sign}) & \cong & \mathcal{P}(\{1,2\} \times \mathbf{Sign}) \end{array}$$

By defining the extraction function $\mathsf{twosigns} : \{1,2\} \times \mathbf{Z} \to \{1,2\} \times \mathbf{Sign}$ using the formula $\mathsf{twosigns}(i, z) = (i, \mathsf{sign}(z))$ we obtain a Galois connection

$$(\mathcal{P}(\{1,2\} \times \mathbf{Z}), \alpha_{\mathsf{twosigns}}, \gamma_{\mathsf{twosigns}}, \mathcal{P}(\{1,2\} \times \mathbf{Sign}))$$

that is isomorphic to $(\mathcal{P}(\mathbf{Z}) \times \mathcal{P}(\mathbf{Z}), \alpha_{\mathrm{SS}}, \gamma_{\mathrm{SS}}, \mathcal{P}(\mathbf{Sign}) \times \mathcal{P}(\mathbf{Sign}))$.

In general the independent attribute method often leads to imprecision. An expression like `(x,-x)` in the source language may have a value in $\{(z, -z) \mid z \in \mathbf{Z}\}$ but in the present setting where we use $\mathcal{P}(\mathbf{Z}) \times \mathcal{P}(\mathbf{Z})$ to represent sets of pairs of integers we cannot do better than representing $\{(z, -z) \mid z \in \mathbf{Z}\}$ by $(\mathbf{Z}, \mathbf{Z})$ and hence the best property describing it in the analysis of Example 4.34 will be $\alpha_{\mathrm{SS}}(\mathbf{Z}, \mathbf{Z}) = (\{-, 0, +\}, \{-, 0, +\})$. Thus we lose all information about the relative signs of the two components. ∎

Relational method. In the independent attribute method there is absolutely no interplay between the two pairs of abstraction and concretisation functions. It is possible to do better by allowing the two components of the analysis to interact with one another so as to get more precise descriptions.

Let $(\mathcal{P}(V_1), \alpha_1, \gamma_1, \mathcal{P}(D_1))$ and $(\mathcal{P}(V_2), \alpha_2, \gamma_2, \mathcal{P}(D_2))$ be Galois connections. The *relational method* will give rise to the Galois connection

$$(\mathcal{P}(V_1 \times V_2), \alpha, \gamma, \mathcal{P}(D_1 \times D_2))$$

where

$$\begin{aligned}\alpha(VV) &= \bigcup\{\alpha_1(\{v_1\}) \times \alpha_2(\{v_2\}) \mid (v_1, v_2) \in VV\} \\ \gamma(DD) &= \{(v_1, v_2) \mid \alpha_1(\{v_1\}) \times \alpha_2(\{v_2\}) \subseteq DD\}\end{aligned}$$

where $VV \subseteq V_1 \times V_2$ and $DD \subseteq D_1 \times D_2$. Let us check that this does indeed define a Galois connection. From the definition it is immediate that α is completely additive and hence that there exists a Galois connection (Lemma 4.23). It remains to show that γ (as defined above) is the upper adjoint of α. For this we can use Lemma 4.22 and calculate

$$\begin{aligned}\gamma(DD) &= \{(v_1, v_2) \mid \alpha_1(\{v_1\}) \times \alpha_2(\{v_2\}) \subseteq DD\} \\ &= \{(v_1, v_2) \mid \alpha(\{(v_1, v_2)\}) \subseteq DD\} \\ &= \bigcup\{VV \mid \alpha(VV) \subseteq DD\}\end{aligned}$$

where one way of proving the equality is to use that α is completely additive. This shows the required result.

It is instructive to see how the relational method is simplified if the Galois connections $(\mathcal{P}(V_i), \alpha_i, \gamma_i, \mathcal{P}(D_i))$ are given by extraction functions $\eta_i : V_i \to D_i$, i.e. if $\alpha_i(V_i') = \{\eta_i(v_i) \mid v_i \in V_i'\}$ and $\gamma_i(D_i') = \{v_i \mid \eta_i(v_i) \in D_i'\}$. We then have

$$\begin{aligned}\alpha(VV) &= \{(\eta_1(v_1), \eta_2(v_2)) \mid (v_1, v_2) \in VV\} \\ \gamma(DD) &= \{(v_1, v_2) \mid (\eta_1(v_1), \eta_2(v_2)) \in DD\}\end{aligned}$$

which also can be obtained directly from the extraction function $\eta : V_1 \times V_2 \to D_1 \times D_2$ defined by $\eta(v_1, v_2) = (\eta_1(v_1), \eta_2(v_2))$.

Example 4.35 Let us return to Example 4.34 and show how the relational method can be used to construct a more precise analysis. We will now get a Galois connection

$$(\mathcal{P}(\mathbf{Z} \times \mathbf{Z}), \alpha_{SS'}, \gamma_{SS'}, \mathcal{P}(\mathbf{Sign} \times \mathbf{Sign}))$$

where $\alpha_{SS'}$ and $\gamma_{SS'}$ are given by

$$\begin{aligned}\alpha_{SS'}(ZZ) &= \{(\mathsf{sign}(z_1), \mathsf{sign}(z_2)) \mid (z_1, z_2) \in ZZ\} \\ \gamma_{SS'}(SS) &= \{(z_1, z_2) \mid (\mathsf{sign}(z_1), \mathsf{sign}(z_2)) \in SS\}\end{aligned}$$

where $ZZ \subseteq \mathbf{Z} \times \mathbf{Z}$ and $SS \subseteq \mathbf{Sign} \times \mathbf{Sign}$. This corresponds to using an extraction function $\mathsf{twosigns}' : \mathbf{Z} \times \mathbf{Z} \to \mathbf{Sign} \times \mathbf{Sign}$ given by $\mathsf{twosigns}'(z_1, z_2) = (\mathsf{sign}(z_1), \mathsf{sign}(z_2))$.

Once again consider the expression `(x,-x)` in the source language that has a value in $\{(z, -z) \mid z \in \mathbf{Z}\}$. In the present setting $\{(z, -z) \mid z \in \mathbf{Z}\}$ is an

element of $\mathcal{P}(\mathbf{Z} \times \mathbf{Z})$ and it is described by the set $\alpha_{SS'}(\{(z, -z) \mid z \in \mathbf{Z}\}) = \{(-,+),(0,0),(+,-)\}$ of $\mathcal{P}(\mathbf{Sign} \times \mathbf{Sign})$. Hence the information about the relative signs of the two components is preserved. This will be the Galois connection that we will use in our further development of the Array Bound Analysis. ∎

The above treatment of the relational method can be extended in a very general way. Let us write $\mathcal{P}(V_1) \otimes \mathcal{P}(V_2)$ for $\mathcal{P}(V_1 \times V_2)$ and similarly $\mathcal{P}(D_1) \otimes \mathcal{P}(D_2)$ for $\mathcal{P}(D_1 \times D_2)$. It is possible to perform a more general development using a notion of *tensor product* for which $L_1 \otimes L_2$ exists even when the complete lattices L_1 and L_2 are not powersets. We refer to the Concluding Remarks for information about this.

Total function space. In Appendix A it is established that if L is a complete lattice then so is the *total function space* $S \to L$ for S being a set. We have a similar result for Galois connections:

Let (L, α, γ, M) be a Galois connection and let S be a set. Then we obtain a Galois connection

$$(S \to L, \alpha', \gamma', S \to M)$$

by taking

$$\begin{aligned} \alpha'(f) &= \alpha \circ f \\ \gamma'(g) &= \gamma \circ g \end{aligned}$$

To see this we first observe that α' and γ' are monotone functions because α and γ are; furthermore

$$\begin{aligned} \gamma'(\alpha'(f)) &= \gamma \circ \alpha \circ f \sqsupseteq f \\ \alpha'(\gamma'(g)) &= \alpha \circ \gamma \circ g \sqsubseteq g \end{aligned}$$

follow since (L, α, γ, M) is a Galois connection. A similar result holds for Galois insertions. The construction is illustrated by the following commuting diagrams:

$$\begin{tikzcd} L \arrow[rr, "\alpha"] & & M \\ & S \arrow[ul, "f"] \arrow[ur, "\alpha'(f)"'] & \end{tikzcd} \qquad \begin{tikzcd} L & & M \arrow[ll, "\gamma"'] \\ & S \arrow[ul, "\gamma'(g)"] \arrow[ur, "g"'] & \end{tikzcd}$$

Example 4.36 Assume that we have some analysis mapping the program variables to properties from the complete lattice L, i.e. operating on the abstract states $\mathbf{Var} \to L$. Given a Galois connection (L, α, γ, M) the above construction will show us how the abstract states of $\mathbf{Var} \to L$ are approximated by the abstract states of $\mathbf{Var} \to M$. ∎

Monotone function space. In Appendix A it is established that the *monotone function space* between two complete lattices is a complete lattice. We have a similar result for Galois connections:

Let $(L_1, \alpha_1, \gamma_1, M_1)$ and $(L_2, \alpha_2, \gamma_2, M_2)$ be Galois connections. Then we obtain the Galois connection

$$(L_1 \to L_2, \alpha, \gamma, M_1 \to M_2)$$

by taking

$$\begin{aligned} \alpha(f) &= \alpha_2 \circ f \circ \gamma_1 \\ \gamma(g) &= \gamma_2 \circ g \circ \alpha_1 \end{aligned}$$

To check this we first observe that the functions α and γ are monotone because α_2 and γ_2 are; next we calculate

$$\begin{aligned} \gamma(\alpha(f)) &= (\gamma_2 \circ \alpha_2) \circ f \circ (\gamma_1 \circ \alpha_1) \sqsupseteq f \\ \alpha(\gamma(g)) &= (\alpha_2 \circ \gamma_2) \circ g \circ (\alpha_1 \circ \gamma_1) \sqsubseteq g \end{aligned}$$

using the monotonicity of $f : L_1 \to L_2$ and $g : M_1 \to M_2$ together with (4.8) and (4.9). A similar result holds for Galois insertions (Exercise 4.17).

This construction is illustrated by the following commuting diagrams:

$$\begin{tikzcd} L_1 \arrow[r, "f"] & L_2 \arrow[d, "\alpha_2"] \\ M_1 \arrow[u, "\gamma_1"] \arrow[r, "\alpha(f)"'] & M_2 \end{tikzcd} \qquad \begin{tikzcd} L_1 \arrow[r, "\gamma(g)"] \arrow[d, "\alpha_1"'] & L_2 \\ M_1 \arrow[r, "g"'] & M_2 \arrow[u, "\gamma_2"'] \end{tikzcd}$$

4.4.2 Other Combinations

So far our constructions have shown how to combine Galois connections dealing with individual components of the data into Galois connections dealing with composite data. We shall now show how two analyses dealing with the

same data can be combined into one analysis; this amounts to performing two analyses in parallel. We shall consider two variants of this analysis, one "corresponding" to the independent attribute method and one "corresponding" to the relational method.

Direct product. Let $(L, \alpha_1, \gamma_1, M_1)$ and $(L, \alpha_2, \gamma_2, M_2)$ be Galois connections. The *direct product* of the two Galois connections will be the Galois connection

$$(L, \alpha, \gamma, M_1 \times M_2)$$

where α and γ are given by:

$$\begin{array}{rcl} \alpha(l) & = & (\alpha_1(l), \alpha_2(l)) \\ \gamma(m_1, m_2) & = & \gamma_1(m_1) \sqcap \gamma_2(m_2) \end{array}$$

To see that this indeed defines a Galois connection we calculate

$$\begin{array}{rcl} \alpha(l) \sqsubseteq (m_1, m_2) & \Leftrightarrow & \alpha_1(l) \sqsubseteq m_1 \wedge \alpha_2(l) \sqsubseteq m_2 \\ & \Leftrightarrow & l \sqsubseteq \gamma_1(m_1) \wedge l \sqsubseteq \gamma_2(m_2) \\ & \Leftrightarrow & l \sqsubseteq \gamma(m_1, m_2) \end{array}$$

and then use Proposition 4.20 to get the result.

Example 4.37 Let us consider how this construction can be used to combine the detection of signs analysis for pairs of integers given in Example 4.35 with the analysis of difference in magnitude given in Example 4.33. We get the Galois connection

$$(\mathcal{P}(\mathbf{Z} \times \mathbf{Z}), \alpha_{\mathrm{SSR}}, \gamma_{\mathrm{SSR}}, \mathcal{P}(\mathbf{Sign} \times \mathbf{Sign}) \times \mathcal{P}(\mathbf{Range}))$$

where α_{SSR} and γ_{SSR} are given by:

$$\begin{array}{rcl} \alpha_{\mathrm{SSR}}(ZZ) & = & (\{(\mathsf{sign}(z_1), \mathsf{sign}(z_2)) \mid (z_1, z_2) \in ZZ\}, \\ & & \{\mathsf{range}(|z_1| - |z_2|) \mid (z_1, z_2) \in ZZ\}) \\ \gamma_{\mathrm{SSR}}(SS, R) & = & \{(z_1, z_2) \mid (\mathsf{sign}(z_1), \mathsf{sign}(z_2)) \in SS\} \\ & & \cap \{(z_1, z_2) \mid \mathsf{range}(|z_1| - |z_2|) \in R\} \end{array}$$

Note that the expression `(x, 3*x)` in the source language has a value in $\{(z, 3 * z) \mid z \in \mathbf{Z}\}$ which is described by $\alpha_{\mathrm{SSR}}(\{(z, 3 * z) \mid z \in \mathbf{Z}\}) = (\{(-,-), (0,0), (+,+)\}, \{0, <-1\})$. Thus we do not exploit the fact that if the pair is described by $(0,0)$ then the difference in magnitude will indeed be described by 0 whereas if the pair is described by $(-,-)$ or $(+,+)$ then the difference in magnitude will indeed be described by <-1. ∎

Direct tensor product. In the direct product there is no interplay between the two abstraction functions and as we saw above this gives rise to the same loss of precision as in the independent attribute method. It is possible to do better by letting the two components interact with one another. Again we shall only consider the simple case of powersets so let $(\mathcal{P}(V), \alpha_i, \gamma_i, \mathcal{P}(D_i))$ be Galois connections. Then the *direct tensor product* is the Galois connection

$$(\mathcal{P}(V), \alpha, \gamma, \mathcal{P}(D_1 \times D_2))$$

where α and γ are defined by:

$$\begin{array}{rcl} \alpha(V') & = & \bigcup\{\alpha_1(\{v\}) \times \alpha_2(\{v\}) \mid v \in V'\} \\ \gamma(DD) & = & \{v \mid \alpha_1(\{v\}) \times \alpha_2(\{v\}) \subseteq DD\} \end{array}$$

where $V' \subseteq V$ and $DD \subseteq D_1 \times D_2$. To verify that this defines a Galois connection we calculate

$$\begin{array}{rcl} \alpha(V') \subseteq DD & \Leftrightarrow & \forall v \in V' : \alpha_1(\{v\}) \times \alpha_2(\{v\}) \subseteq DD \\ & \Leftrightarrow & \forall v \in V' : v \in \gamma(DD) \\ & \Leftrightarrow & V' \subseteq \gamma(DD) \end{array}$$

and then use Proposition 4.20.

The construction can be simplified if the two Galois connections $(\mathcal{P}(V), \alpha_i, \gamma_i, \mathcal{P}(D_i))$ are given by extraction functions $\eta_i : V \to D_i$, i.e. if $\alpha_i(V') = \{\eta_i(v) \mid v \in V'\}$ and $\gamma_i(D_i') = \{v \mid \eta_i(v) \in D_i'\}$. Then we have

$$\begin{array}{rcl} \alpha(V') & = & \{(\eta_1(v), \eta_2(v)) \mid v \in V'\} \\ \gamma(DD) & = & \{v \mid (\eta_1(v), \eta_2(v)) \in DD\} \end{array}$$

which also can be obtained directly from the extraction function $\eta : V \to D_1 \times D_2$ defined by $\eta(v) = (\eta_1(v), \eta_2(v))$.

Example 4.38 Let us return to Example 4.37 and show how the direct tensor product gives a more precise analysis. We will now get a Galois connection

$$(\mathcal{P}(\mathbf{Z} \times \mathbf{Z}), \alpha_{\mathrm{SSR}'}, \gamma_{\mathrm{SSR}'}, \mathcal{P}(\mathbf{Sign} \times \mathbf{Sign} \times \mathbf{Range}))$$

where

$$\begin{array}{rcl} \alpha_{\mathrm{SSR}'}(ZZ) & = & \{(\mathsf{sign}(z_1), \mathsf{sign}(z_2), \mathsf{range}(|z_1| - |z_2|)) \mid (z_1, z_2) \in ZZ\} \\ \gamma_{\mathrm{SSR}'}(SSR) & = & \{(z_1, z_2) \mid (\mathsf{sign}(z_1), \mathsf{sign}(z_2), \mathsf{range}(|z_1| - |z_2|)) \in SSR\} \end{array}$$

for $ZZ \subseteq \mathbf{Z} \times \mathbf{Z}$ and $SSR \subseteq \mathbf{Sign} \times \mathbf{Sign} \times \mathbf{Range}$.

It is worth pointing out that this Galois connection is also obtainable from the extraction function

$$\mathsf{twosignsrange} : \mathbf{Z} \times \mathbf{Z} \to \mathbf{Sign} \times \mathbf{Sign} \times \mathbf{Range}$$

defined by $\mathsf{twosignsrange}(z_1, z_2) = (\mathsf{sign}(z_1), \mathsf{sign}(z_2), \mathsf{range}(|z_1| - |z_2|))$.

Returning to the precision of the analysis we will now have $\alpha_{\mathrm{SSR}'}(\{(z, 3 * z) \mid z \in \mathbf{Z}\}) = \{(\mathsf{-}, \mathsf{-}, \mathsf{<-1}), (\mathsf{0}, \mathsf{0}, \mathsf{0}), (\mathsf{+}, \mathsf{+}, \mathsf{<-1})\}$ and hence have a more precise description than in Example 4.37.

However, it is worth noticing that the above Galois connection is not a Galois insertion. To see this consider the two elements $\emptyset$ and $\{(\mathsf{0}, \mathsf{0}, \mathsf{<-1})\}$ of $\mathcal{P}(\mathbf{Sign} \times \mathbf{Sign} \times \mathbf{Range})$ and observe that

$$\gamma_{\mathrm{SSR}'}(\emptyset) = \emptyset = \gamma_{\mathrm{SSR}'}(\{(\mathsf{0}, \mathsf{0}, \mathsf{<-1})\})$$

Thus $\gamma_{\mathrm{SSR}'}$ is not injective and hence Lemma 4.27 shows that we do not have a Galois insertion. ∎

Reduced product and reduced tensor product. The construction of Proposition 4.29 gives us a general method for turning a Galois connection into a Galois insertion. This technique can now be combined with the other techniques for combining Galois connections and this is of particular interest for the direct product and the direct tensor product.

Let $(L, \alpha_1, \gamma_1, M_1)$ and $(L, \alpha_2, \gamma_2, M_2)$ be Galois connections. Then the *reduced product* is the Galois insertion

$$(L, \alpha, \gamma, \varsigma[M_1 \times M_2])$$

where

$$\begin{aligned}
\alpha(l) &= (\alpha_1(l), \alpha_2(l)) \\
\gamma(m_1, m_2) &= \gamma_1(m_1) \sqcap \gamma_2(m_2) \\
\varsigma(m_1, m_2) &= \bigsqcap \{(m_1', m_2') \mid \gamma_1(m_1) \sqcap \gamma_2(m_2) = \gamma_1(m_1') \sqcap \gamma_2(m_2')\}
\end{aligned}$$

To see that this is indeed a Galois insertion recall that we already know that the direct product $(L, \alpha, \gamma, M_1 \times M_2)$ is a Galois connection and that Proposition 4.29 then shows that $(L, \alpha, \gamma, \varsigma[M_1 \times M_2])$ is a Galois insertion.

Next let $(\mathcal{P}(V), \alpha_i, \gamma_i, \mathcal{P}(D_i))$ be Galois connections for $i = 1, 2$. Then the *reduced tensor product* is the Galois insertion

$$(\mathcal{P}(V), \alpha, \gamma, \varsigma[\mathcal{P}(D_1 \times D_2)])$$

where

$$\begin{aligned}
\alpha(V') &= \bigcup \{\alpha_1(\{v\}) \times \alpha_2(\{v\}) \mid v \in V'\} \\
\gamma(DD) &= \{v \mid \alpha_1(\{v\}) \times \alpha_2(\{v\}) \subseteq DD\} \\
\varsigma(DD) &= \bigcap \{DD' \mid \gamma(DD) = \gamma(DD')\}
\end{aligned}$$

Again it follows from Proposition 4.29 that this is indeed a Galois insertion.

Example 4.39 Let us return to Example 4.38 where we noted that the complete lattice $\mathcal{P}(\mathbf{Sign} \times \mathbf{Sign} \times \mathbf{Range})$ contains more than one element that describes the empty set of $\mathcal{P}(\mathbf{Z} \times \mathbf{Z})$. The superfluous elements will be removed by the construction of Proposition 4.29. The function $\varsigma_{\mathrm{SSR}'}$ will amount to

$$\varsigma_{\mathrm{SSR}'}(SSR) = \bigcap\{SSR' \mid \gamma_{\mathrm{SSR}'}(SSR) = \gamma_{\mathrm{SSR}'}(SSR')\}$$

where $SSR, SSR' \subseteq \mathbf{Sign} \times \mathbf{Sign} \times \mathbf{Range}$. In particular, $\varsigma_{\mathrm{SSR}'}$ will map the singleton sets constructed from the 16 elements

$$\begin{array}{llll}
(-,0,<-1), & (-,0,-1), & (-,0,0), & \\
(0,-,0), & (0,-,+1), & (0,-,>+1), & \\
(0,0,<-1), & (0,0,-1), & (0,0,+1), & (0,0,>+1), \\
(0,+,0), & (0,+,+1), & (0,+,>+1), & \\
(+,0,<-1), & (+,0,-1), & (+,0,0) &
\end{array}$$

to the empty set. The remaining 29 elements of $\mathbf{Sign} \times \mathbf{Sign} \times \mathbf{Range}$ are

$$\begin{array}{lllll}
(-,-,<-1), & (-,-,-1), & (-,-,0), & (-,-,+1), & (-,-,>+1), \\
(-,0,+1), & (-,0,>+1), & & & \\
(-,+,<-1), & (-,+,-1), & (-,+,0), & (-,+,+1), & (-,+,>+1), \\
(0,-,<-1), & (0,-,-1), & (0,0,0), & (0,+,<-1), & (0,+,-1), \\
(+,-,<-1), & (+,-,-1), & (+,-,0), & (+,-,+1), & (+,-,>+1), \\
(+,0,+1), & (+,0,>+1), & & & \\
(+,+,<-1), & (+,+,-1), & (+,+,0), & (+,+,+1), & (+,+,>+1)
\end{array}$$

and they describe disjoint subsets of $\mathbf{Z} \times \mathbf{Z}$. Let us call the above set of 29 elements for **AB** (for Array Bound); then $\varsigma_{\mathrm{SSR}'}[\mathcal{P}(\mathbf{Sign} \times \mathbf{Sign} \times \mathbf{Range})]$ is isomorphic to $\mathcal{P}(\mathbf{AB})$.

To conclude the development of the complete lattice and the associated Galois connection for the Array Bound Analysis we shall simply construct the reduced tensor product of the Galois connections of Examples 4.35 and 4.33. This will yield a *Galois insertion* isomorphic to

$$(\mathcal{P}(\mathbf{Z} \times \mathbf{Z}), \alpha_{\mathrm{SSR}'}, \gamma_{\mathrm{SSR}'}, \mathcal{P}(\mathbf{AB}))$$

Note that from an implementation point of view the last step of the construction has paid off: if we had stopped with the direct tensor product in Example 4.38 then the properties would need 45 bits for their representation whereas now 29 bits suffice.

Summary. The Array Bound Analysis has been designed from three simple Galois connections specified by extraction functions:

(i) an analysis approximating integers by their sign (Example 4.21),

(ii) an analysis approximating pairs of integers by their difference in magnitude (Example 4.33), and

(iii) an analysis approximating integers by their closeness to 0, 1 and -1 (Example 4.33).

We have illustrated different ways of combining these analyses:

(iv) the relational product of analysis (i) with itself,

(v) the functional composition of analysis (ii) and (iii), and

(vi) the reduced tensor product of analysis (iv) and (v).

It is worth noting that because the resulting complete lattice $\mathcal{P}(\mathbf{AB})$ is a powerset then it is indeed possible to obtain the very same Galois insertion using an extraction function $\mathsf{twosignsrange}' : \mathbf{Z} \times \mathbf{Z} \to \mathbf{AB}$. ∎

4.5 Induced Operations

We shall now show that Galois connections are indeed useful for transforming computations into more approximate computations that have better time, space, or termination behaviour. We can do so in two different ways. In both cases we assume that we have an analysis using the complete lattice L and that we have a Galois connection (L, α, γ, M).

One possibility is to replace the analysis using L with an analysis using M. In Subsection 4.5.1 we shall show that if the analysis using M is an upper approximation to the analysis *induced* from L then the correctness properties are preserved. We shall illustrate this approach in Subsection 4.5.2 for the Monotone Frameworks considered in Chapter 2.

An alternative is only to use the complete lattice M for approximating the fixed point computations in L. So rather than performing all computations on the more approximate lattice M the idea is only to use M to ensure convergence of fixed point computations and not needlessly reduce the precision of all other operations. We shall illustrate this in Subsection 4.5.3.

4.5.1 Inducing along the Abstraction Function

Now suppose that we have Galois connections $(L_i, \alpha_i, \gamma_i, M_i)$ such that each M_i is a more approximate version of L_i (for $i = 1, 2$). One way to make use

of this is to replace an existing analysis $f_p : L_1 \to L_2$ with a new and more approximate analysis $g_p : M_1 \to M_2$. We already saw in Section 4.4 that

$$\alpha_2 \circ f_p \circ \gamma_1 \text{ is a candidate for } g_p$$

(just as $\gamma_2 \circ g_p \circ \alpha_1$ would be a candidate for f_p). The analysis $\alpha_2 \circ f_p \circ \gamma_1$ is said to be *induced* by f_p and the two Galois connections. This is illustrated by the diagram:

$$\begin{tikzcd}
L_1 \arrow[r, "f_p"] & L_2 \arrow[d, "\alpha_2"] \\
M_1 \arrow[u, "\gamma_1"] \arrow[r, "\alpha_2 \circ f_p \circ \gamma_1"'] & M_2
\end{tikzcd}$$

Example 4.40 Let us return to Example 4.9 where we studied the simple program `plus` and specified the very precise analysis

$$f_{\text{plus}}(ZZ) = \{z_1 + z_2 \mid (z_1, z_2) \in ZZ\}$$

using the complete lattices $(\mathcal{P}(\mathbf{Z}), \subseteq)$ and $(\mathcal{P}(\mathbf{Z} \times \mathbf{Z}), \subseteq)$. In Example 4.21 we introduced the Galois connection

$$(\mathcal{P}(\mathbf{Z}), \alpha_{\mathsf{sign}}, \gamma_{\mathsf{sign}}, \mathcal{P}(\mathbf{Sign}))$$

for approximating sets of integers by sets of signs. In Example 4.35 we used the relational method to get the Galois connection

$$(\mathcal{P}(\mathbf{Z} \times \mathbf{Z}), \alpha_{\text{SS}'}, \gamma_{\text{SS}'}, \mathcal{P}(\mathbf{Sign} \times \mathbf{Sign}))$$

operating on pairs of integers. We now want to induce a more approximate analysis for the `plus` program

$$g_{\text{plus}} : \mathcal{P}(\mathbf{Sign} \times \mathbf{Sign}) \to \mathcal{P}(\mathbf{Sign})$$

from the existing analysis f_{plus}. To do so we take

$$g_{\text{plus}} = \alpha_{\mathsf{sign}} \circ f_{\text{plus}} \circ \gamma_{\text{SS}'}$$

and simply calculate (for $SS \subseteq \mathbf{Sign} \times \mathbf{Sign}$)

$$\begin{aligned}
g_{\text{plus}}(SS) &= \alpha_{\mathsf{sign}}(f_{\text{plus}}(\gamma_{\text{SS}'}(SS))) \\
&= \alpha_{\mathsf{sign}}(f_{\text{plus}}(\{(z_1, z_2) \in \mathbf{Z} \times \mathbf{Z} \mid (\mathsf{sign}(z_1), \mathsf{sign}(z_2)) \in SS\})) \\
&= \alpha_{\mathsf{sign}}(\{z_1 + z_2 \mid z_1, z_2 \in \mathbf{Z}, (\mathsf{sign}(z_1), \mathsf{sign}(z_2)) \in SS\}) \\
&= \{\mathsf{sign}(z_1 + z_2) \mid z_1, z_2 \in \mathbf{Z}, (\mathsf{sign}(z_1), \mathsf{sign}(z_2)) \in SS\} \\
&= \bigcup\{s_1 \oplus s_2 \mid (s_1, s_2) \in SS\}
\end{aligned}$$

where $\oplus : \mathbf{Sign} \times \mathbf{Sign} \to \mathcal{P}(\mathbf{Sign})$ is the "addition" operator on signs (so e.g. $+ \oplus + = \{+\}$ and $+ \oplus - = \{-, 0, +\}$). ∎

The mundane approach to correctness. We shall now follow the approach of Section 4.1 and show that correctness of f_p carries over to g_p. For this assume that:

$$R_i : V_i \times L_i \to \{true, false\} \text{ is } \textit{generated by } \beta_i : V_i \to L_i$$

The correctness of the analysis $f_p : L_1 \to L_2$ is then expressed by

$$(p \vdash \cdot \rightsquigarrow \cdot)\,(R_1 \twoheadrightarrow R_2)\, f_p$$

where $R_1 \twoheadrightarrow R_2$ is *generated by* $\beta_1 \twoheadrightarrow \beta_2$ (Lemma 4.8). As argued in Section 4.3 we get a correctness relation S_i for V_i and M_i by letting

$$S_i : V_i \times M_i \to \{true, false\} \text{ be } \textit{generated by } \alpha_i \circ \beta_i : V_i \to M_i$$

which is equivalent to saying that $v_i\ S_i\ m_i \Leftrightarrow v_i\ R_i\ (\gamma_i(m_i))$. The correctness relation for the analysis using M_1 and M_2 will be $S_1 \twoheadrightarrow S_2$ which will be *generated by* $(\alpha_1 \circ \beta_1) \twoheadrightarrow (\alpha_2 \circ \beta_2)$ (Lemma 4.8). We now have the following useful result:

Lemma 4.41 If $(L_i, \alpha_i, \gamma_i, M_i)$ are Galois connections, and $\beta_i : V_i \to L_i$ are representation functions then

$$((\alpha_1 \circ \beta_1) \twoheadrightarrow (\alpha_2 \circ \beta_2))(\rightsquigarrow) = \alpha_2 \circ ((\beta_1 \twoheadrightarrow \beta_2)(\rightsquigarrow)) \circ \gamma_1$$

holds for all $\rightsquigarrow$. ∎

Proof To see this we simply calculate

$$\begin{aligned}
((\alpha_1 \circ \beta_1) \twoheadrightarrow (\alpha_2 \circ \beta_2))(\rightsquigarrow)(m_1) &= \bigsqcup\{\alpha_2(\beta_2(v_2)) \mid \alpha_1(\beta_1(v_1)) \sqsubseteq m_1 \wedge v_1 \rightsquigarrow v_2\} \\
&= \alpha_2(\bigsqcup\{\beta_2(v_2) \mid \beta_1(v_1) \sqsubseteq \gamma_1(m_1) \wedge v_1 \rightsquigarrow v_2\}) \\
&= \alpha_2((\beta_1 \twoheadrightarrow \beta_2)(\rightsquigarrow)(\gamma_1(m_1)))
\end{aligned}$$

and the result follows. ∎

We shall now show that Lemma 4.41 yields:

$$(p \vdash \cdot \rightsquigarrow \cdot)\,(R_1 \twoheadrightarrow R_2)\, f_p \ \wedge\ \alpha_2 \circ f_p \circ \gamma_1 \sqsubseteq g_p \ \Rightarrow\ (p \vdash \cdot \rightsquigarrow \cdot)\,(S_1 \twoheadrightarrow S_2)\, g_p$$

This just means that if f_p is correct and if g_p is an upper approximation to the induced analysis then also g_p is correct. So suppose that

$$(p \vdash \cdot \rightsquigarrow \cdot)\,(R_1 \twoheadrightarrow R_2)\, f_p$$

and that $\alpha_2 \circ f_p \circ \gamma_1 \sqsubseteq g_p$. Since $(L_i, \alpha_i, \gamma_i, M_i)$ are Galois connections and f_p and g_p are monotone we get $f_p \sqsubseteq \gamma_2 \circ g_p \circ \alpha_1$ as illustrated in the diagrams:

$$\begin{tikzcd} L_1 \arrow[r, "f_p"] & L_2 \arrow[d, "\alpha_2"] \arrow[dl, phantom, "\sqsubseteq"] \\ M_1 \arrow[u, "\gamma_1"] \arrow[r, "g_p"'] & M_2 \end{tikzcd} \qquad \begin{tikzcd} L_1 \arrow[r, "f_p"] \arrow[d, "\alpha_1"'] & L_2 \arrow[dl, phantom, "\sqsubseteq"] \\ M_1 \arrow[r, "g_p"'] & M_2 \arrow[u, "\gamma_2"'] \end{tikzcd}$$

It follows that $(\beta_1 \twoheadrightarrow \beta_2)(p \vdash \cdot \rightsquigarrow \cdot) \sqsubseteq \gamma_2 \circ g_p \circ \alpha_1$ and hence

$$\alpha_2 \circ (\beta_1 \twoheadrightarrow \beta_2)(p \vdash \cdot \rightsquigarrow \cdot) \circ \gamma_1 \sqsubseteq g_p$$

By Lemma 4.41 this is the desired result.

We shall say that a function $f_p : L_1 \to L_2$ is *optimal* for the program p if and only if correctness of a function $f' : L_1 \to L_2$ amounts to $f_p \sqsubseteq f'$. An equivalent formulation is that f_p is optimal if and only if

$$(\beta_1 \twoheadrightarrow \beta_2)(p \vdash \cdot \rightsquigarrow \cdot) = f_p$$

Lemma 4.41 may then be read as saying that if $f_p : L_1 \to L_2$ is optimal then so is $\alpha_2 \circ f_p \circ \gamma_1 : M_1 \to M_2$.

Fixed points in the induced analysis. Let us next consider the situation where the analysis $f_p : L_1 \to L_2$ requires the computation of the *least fixed point* of a monotone function $F : (L_1 \to L_2) \to (L_1 \to L_2)$ so that $f_p = \mathit{lfp}(F)$. The Galois connections $(L_i, \alpha_i, \gamma_i, M_i)$ give rise to a Galois connection $(L_1 \to L_2, \alpha, \gamma, M_1 \to M_2)$ between the monotone function spaces as shown in Section 4.4. We can now apply our technique of inducing and let $G : (M_1 \to M_2) \to (M_1 \to M_2)$ be an upper approximation to $\alpha \circ F \circ \gamma$. It will be natural to take $g_p : M_1 \to M_2$ to be $g_p = \mathit{lfp}(G)$. That correctness of f_p carries over to g_p follows from the following general result:

Lemma 4.42 Assume that (L, α, γ, M) is a Galois connection and let $f : L \to L$ and $g : M \to M$ be monotone functions satisfying that g is an upper approximation to the function induced by f, i.e.:

$$\alpha \circ f \circ \gamma \sqsubseteq g$$

Then for all $m \in M$:

$$g(m) \sqsubseteq m \Rightarrow f(\gamma(m)) \sqsubseteq \gamma(m)$$

and furthermore $\mathit{lfp}(f) \sqsubseteq \gamma(\mathit{lfp}(g))$ and $\alpha(\mathit{lfp}(f)) \sqsubseteq \mathit{lfp}(g)$. ■

Proof First assume $g(m) \sqsubseteq m$. The assumption $\alpha \circ f \circ \gamma \sqsubseteq g$ gives $\alpha(f(\gamma(m))) \sqsubseteq g(m)$ and hence $\alpha(f(\gamma(m))) \sqsubseteq m$. Using that a Galois connection is an adjunction (Proposition 4.20) we get $f(\gamma(m)) \sqsubseteq \gamma(m)$ as required.

To prove the second result we observe that $\{\gamma(m) \mid g(m) \sqsubseteq m\} \subseteq \{l \mid f(l) \sqsubseteq l\}$ follows from the previous result. Hence we get (using Lemma 4.22):

$$\gamma(\bigsqcap\{m \mid g(m) \sqsubseteq m\}) = \bigsqcap\{\gamma(m) \mid g(m) \sqsubseteq m\} \sqsupseteq \bigsqcap\{l \mid f(l) \sqsubseteq l\}$$

Using Tarski's Theorem (Proposition A.10) twice we have $\mathit{lfp}(g) = \bigsqcap \mathit{Red}(g) = \bigsqcap\{m \mid g(m) \sqsubseteq m\}$ and $\mathit{lfp}(f) = \bigsqcap \mathit{Red}(f) = \bigsqcap\{l \mid f(l) \sqsubseteq l\}$ so it follows that $\gamma(\mathit{lfp}(g)) \sqsupseteq \mathit{lfp}(f)$ as required. Then $\alpha(\mathit{lfp}(f)) \sqsubseteq \mathit{lfp}(g)$ follows because a Galois connection is an adjunction. ∎

4.5.2 Application to Data Flow Analysis

Generalised Monotone Frameworks. To illustrate how these techniques can be applied we shall now consider a generalisation of the Monotone Frameworks of Section 2.3. So let a *generalised Monotone Framework* consist of:

- a complete lattice $L = (L, \sqsubseteq)$.

Here we do not demand that L satisfies the Ascending Chain Condition and we do not specify the space $\mathcal{F}$ of transfer functions as we shall be taking $\mathcal{F}$ to be the entire space of monotone functions from L to L (which clearly contains the identity function and is closed under composition of functions).

An *instance* A of a generalised Monotone Framework then consists of:

- the complete lattice, L, of the framework;
- a finite flow, $F \subseteq \mathbf{Lab} \times \mathbf{Lab}$;
- a finite set of extremal labels, $E \subseteq \mathbf{Lab}$;
- an extremal value, $\iota \in L$; and
- a mapping, $f_\cdot$, from the labels $\mathbf{Lab}$ of F and E to monotone transfer functions from L to L.

As in Chapter 2 this gives rise to a set $\mathsf{A}^{\sqsupseteq}$ of constraints

$$A_\circ(\ell) \sqsupseteq \bigsqcup\{A_\bullet(\ell') \mid (\ell', \ell) \in F\} \sqcup \iota_E^\ell \qquad \text{where } \iota_E^\ell = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases}$$

$$A_\bullet(\ell) \sqsupseteq f_\ell(A_\circ(\ell))$$

where ℓ ranges over the labels $\mathbf{Lab}$ of F and E. We write $(A_\circ, A_\bullet) \models \mathsf{A}^{\sqsupseteq}$ whenever $A_\circ, A_\bullet : \mathbf{Lab} \to L$ is a solution to the constraints $\mathsf{A}^{\sqsupseteq}$. It is useful to consider the associated monotone function

$$\vec{f} : (\mathbf{Lab} \to L) \times (\mathbf{Lab} \to L) \to (\mathbf{Lab} \to L) \times (\mathbf{Lab} \to L)$$

defined by:

$$\vec{f}(A_\circ, A_\bullet) = (\ \lambda\ell.\bigsqcup\{A_\bullet(\ell') \mid (\ell',\ell) \in F\} \sqcup \iota_E^\ell\ ,\ \lambda\ell.f_\ell(A_\circ(\ell))\ )$$

We then have the following important result (in the manner of Section 4.4):

$$(A_\circ, A_\bullet) \sqsupseteq \vec{f}(A_\circ, A_\bullet) \text{ is equivalent to } (A_\circ, A_\bullet) \models \mathsf{A}^{\sqsupseteq}$$

Galois connections and Monotone Frameworks. Let now (L, α, γ, M) be a Galois connection and consider an instance B of the generalised Monotone Framework M that satisfies

- the mapping $g_.$ from the labels **Lab** of F and E to monotone transfer functions of $M \to M$ satisfies $g_\ell \sqsupseteq \alpha \circ f_\ell \circ \gamma$ for all ℓ; and
- the extremal value $\jmath$ satisfies $\jmath \sqsupseteq \alpha(\iota)$

and otherwise B is as A, i.e. has the same F and E.

As above we get a set of constraints $\mathsf{B}^{\sqsupseteq}$ and we write $(B_\circ, B_\bullet) \models \mathsf{B}^{\sqsupseteq}$ whenever $B_\circ, B_\bullet : \mathbf{Lab} \to M$ is a solution to the constraints. The alternative formulation is $(B_\circ, B_\bullet) \sqsupseteq \vec{g}(B_\circ, B_\bullet)$ where $\vec{g} : (\mathbf{Lab} \to M) \times (\mathbf{Lab} \to M) \to (\mathbf{Lab} \to M) \times (\mathbf{Lab} \to M)$ is the monotone function associated with the constraints.

We shall now see that whenever we have a solution to the constraints obtained from B then we also have a solution to the constraints obtained from A. This can be expressed by:

$$(B_\circ, B_\bullet) \models \mathsf{B}^{\sqsupseteq} \text{ implies } (\gamma \circ B_\circ, \gamma \circ B_\bullet) \models \mathsf{A}^{\sqsupseteq}$$

We can give a direct ("concrete") proof of this result but it is instructive to see how it follows from the general ("abstract") results established earlier. The idea is to "lift" the Galois connection (L, α, γ, M) to a Galois connection

$$((\mathbf{Lab} \to L) \times (\mathbf{Lab} \to L), \alpha', \gamma', (\mathbf{Lab} \to M) \times (\mathbf{Lab} \to M))$$

using the techniques for total function spaces and the independent attribute method presented in Section 4.4. The assumptions $g_\ell \sqsupseteq \alpha \circ f_\ell \circ \gamma$ (for all ℓ) and $\jmath \sqsupseteq \alpha(\iota)$ can then be used to establish

$$\vec{g} \sqsupseteq \alpha' \circ \vec{f} \circ \gamma'$$

as the following calculations show

$$\begin{aligned}(\alpha' \circ \vec{f} \circ \gamma')(B_\circ, B_\bullet) &= (\lambda\ell.\bigsqcup\{\alpha(\gamma(B_\bullet(\ell'))) \mid (\ell',\ell) \in F\} \sqcup \alpha(\iota_E^\ell), \\ &\qquad \lambda\ell.\alpha(f_\ell(\gamma(B_\circ(\ell))))) \\ &\sqsubseteq \vec{g}(B_\circ, B_\bullet)\end{aligned}$$

where we have used that $\alpha(\bot) = \bot$. We can now use Lemma 4.42 to obtain

$$\vec{g}(B_\circ, B_\bullet) \sqsubseteq (B_\circ, B_\bullet) \text{ implies } \vec{f}(\gamma'(B_\circ, B_\bullet)) \sqsubseteq \gamma'(B_\circ, B_\bullet)$$

and it follows that if $(B_\circ, B_\bullet) \models \mathsf{B}^{\sqsupseteq}$ then $(\gamma \circ B_\circ, \gamma \circ B_\bullet) \models \mathsf{A}^{\sqsupseteq}$ as stated above.

The mundane approach to correctness. The above result shows that any solution to the constraints obtained for B also is a solution to the constraints obtained from A. We shall now show that semantic correctness of A implies semantic correctness of B.

Let us reconsider the approach taken to semantic correctness in Section 4.1; here $F = \mathit{flow}(S_\star)$ and $E = \{\mathit{init}(S_\star)\}$. For the analysis A this calls for using a *representation function*

$$\beta : \mathbf{State} \to L$$

and the correctness of all solutions to $\mathsf{A}^{\sqsupseteq}$ then amounts to the claim:

$$\begin{array}{l}\text{Assume that } (A_\circ, A_\bullet) \models \mathsf{A}^{\sqsupseteq} \text{ and } \langle S_\star, \sigma_1\rangle \to^* \sigma_2; \\ \text{then } \beta(\sigma_1) \sqsubseteq \iota \text{ implies } \beta(\sigma_2) \sqsubseteq \bigsqcup\{A_\bullet(\ell) \mid \ell \in \mathit{final}(S_\star)\}.\end{array} \tag{4.11}$$

For the analysis B it follows from Section 4.3 that it is natural to use the representation function

$$\alpha \circ \beta : \mathbf{State} \to M$$

and the correctness of all solutions to $\mathsf{B}^{\sqsupseteq}$ then amounts to the claim:

$$\begin{array}{l}\text{Assume that } (B_\circ, B_\bullet) \models \mathsf{B}^{\sqsupseteq} \text{ and } \langle S_\star, \sigma_1\rangle \to^* \sigma_2; \\ \text{then } (\alpha \circ \beta)(\sigma_1) \sqsubseteq \jmath \text{ implies } (\alpha \circ \beta)(\sigma_2) \sqsubseteq \bigsqcup\{B_\bullet(\ell) \mid \ell \in \mathit{final}(S_\star)\}.\end{array} \tag{4.12}$$

We know that B is an upper approximation of the analysis induced from A and shall now prove that (4.11) implies (4.12). To do so we shall need to strengthen the relationship between the extremal values of A and B by *assuming* that $\jmath$ satisfies

$$\gamma(\jmath) = \iota$$

from which $\jmath \sqsupseteq \alpha(\iota)$ readily follows. For the proof that (4.11) implies (4.12) suppose that:

$$(B_\circ, B_\bullet) \models \mathsf{B}^{\sqsupseteq},\ \langle S_\star, \sigma_1\rangle \to^* \sigma_2 \text{ and } (\alpha \circ \beta)(\sigma_1) \sqsubseteq \jmath$$

It follows that:

$$(\gamma \circ B_\circ, \gamma \circ B_\bullet) \models \mathsf{A}^{\sqsupseteq},\ \langle S_\star, \sigma_1\rangle \to^* \sigma_2 \text{ and } \beta(\sigma_1) \sqsubseteq \gamma(\jmath) \sqsubseteq \iota$$

From (4.11) we get $\beta(\sigma_2) \sqsubseteq \bigsqcup\{\gamma \circ B_\bullet(\ell) \mid \ell \in \mathit{final}(S_\star)\}$ and hence $\beta(\sigma_2) \sqsubseteq \gamma(\bigsqcup\{B_\bullet(\ell) \mid \ell \in \mathit{final}(S_\star)\})$ showing the desired $(\alpha \circ \beta)(\sigma_2) \sqsubseteq \bigsqcup\{B_\bullet(\ell) \mid \ell \in \mathit{final}(S_\star)\}$.

A Worked Example

As a concrete example we shall now consider an analysis SS for the WHILE language that approximates how *sets* of states are transformed into *sets* of states. First we prove that it is correct. Then we show that the Constant Propagation Analysis of Section 2.3 is an upper approximation of an analysis induced from SS. This will then establish the correctness of the *Constant Propagation Analysis.*

Sets of states analysis. The analysis SS approximating the sets of states will be a generalised Monotone Framework with:

- the complete lattice $(\mathcal{P}(\mathbf{State}), \subseteq)$.

Given a label consistent statement $S_\star$ in **Stmt** we can now specify the instance as follows:

- the flow F is $\mathit{flow}(S_\star)$;
- the set E of extremal labels is $\{\mathit{init}(S_\star)\}$;
- the extremal value ι is **State**; and
- the transfer functions are given by $f_\cdot^{\mathsf{SS}}$:

$$\begin{array}{lcll} f_\ell^{\mathsf{SS}}(\Sigma) & = & \{\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \mid \sigma \in \Sigma\} & \text{if } [x := a]^\ell \text{ is in } S_\star \\ f_\ell^{\mathsf{SS}}(\Sigma) & = & \Sigma & \text{if } [\texttt{skip}]^\ell \text{ is in } S_\star \\ f_\ell^{\mathsf{SS}}(\Sigma) & = & \Sigma & \text{if } [b]^\ell \text{ is in } S_\star \end{array}$$

where $\Sigma \subseteq \mathbf{State}$.

Correctness. The following result shows that this analysis is correct in the sense explained above:

Lemma 4.43 Assume that $(SS_\circ, SS_\bullet) \models \mathsf{SS}^{\supseteq}$ and $\langle S_\star, \sigma_1\rangle \to^* \sigma_2$; then $\sigma_1 \in \mathbf{State}$ implies $\sigma_2 \in \bigcup\{SS_\bullet(\ell) \mid \ell \in \mathit{final}(S_\star)\}$. ∎

Proof From Section 2.2 we have:

$$\langle S, \sigma\rangle \to \langle S', \sigma'\rangle \text{ implies } \mathit{final}(S) \supseteq \mathit{final}(S') \ \wedge \ \mathit{flow}(S) \supseteq \mathit{flow}(S')$$

and as in Chapter 2 it is immediate that

$$\mathit{flow}(S) \supseteq \mathit{flow}(S') \wedge (SS_\circ, SS_\bullet) \models \mathsf{SS}^{\supseteq}(S) \text{ implies } (SS_\circ, SS_\bullet) \models \mathsf{SS}^{\supseteq}(S')$$

It then suffices to show

$$\begin{array}{l} (SS_\circ, SS_\bullet) \models \mathsf{SS}^{\supseteq}(S) \wedge \langle S, \sigma\rangle \to \sigma' \ \wedge \ \sigma \in SS_\circ(\mathit{init}(S)) \\ \text{implies } \sigma' \in \cup\{SS_\bullet(\ell) \mid \ell \in \mathit{final}(S)\} \\ (SS_\circ, SS_\bullet) \models \mathsf{SS}^{\supseteq}(S) \wedge \langle S, \sigma\rangle \to \langle S', \sigma'\rangle \ \wedge \ \sigma \in SS_\circ(\mathit{init}(S)) \\ \text{implies } \sigma' \in SS_\circ(\mathit{init}(S')) \end{array}$$

since then an induction on the length of the derivation sequence $\langle S_\star, \sigma_1\rangle \rightarrow^* \sigma_2$ will give the result. The proof proceeds by induction on the inference in the semantics. We only consider a few of the interesting cases.

The case $\langle [x := a]^\ell, \sigma\rangle \rightarrow \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$. Then $\mathsf{SS}^{\supseteq}(S)$ will contain the equation

$$SS_\bullet(\ell) \supseteq \{\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \mid \sigma \in SS_\circ(\ell)\}$$

and since $\mathit{init}([x := a]^\ell) = \ell$ and $\mathit{final}([x := a]^\ell) = \{\ell\}$ we see that that the required relationship holds: if $\sigma \in SS_\circ(\ell)$ then $\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \in SS_\bullet(\ell)$.

The case $\langle S_1; S_2, \sigma\rangle \rightarrow \langle S_1'; S_2, \sigma'\rangle$ because $\langle S_1, \sigma\rangle \rightarrow \langle S_1', \sigma'\rangle$. From the assumption $\sigma \in SS_\circ(\mathit{init}(S_1; S_2))$ we get $\sigma \in SS_\circ(\mathit{init}(S_1))$ and the induction hypothesis gives $\sigma' \in SS_\circ(\mathit{init}(S_1'))$. But then $\sigma' \in SS_\circ(\mathit{init}(S_1'; S_2))$ as required.

The case $\langle S_1; S_2, \sigma\rangle \rightarrow \langle S_2, \sigma'\rangle$ because $\langle S_1, \sigma\rangle \rightarrow \sigma'$. From the assumption $\sigma \in SS_\circ(\mathit{init}(S_1; S_2))$ we get $\sigma \in SS_\circ(\mathit{init}(S_1))$ and the induction hypothesis gives $\sigma' \in \bigcup\{SS_\bullet(\ell) \mid \ell \in \mathit{final}(S_1)\}$. We have

$$\{(\ell, \mathit{init}(S_2)) \mid \ell \in \mathit{final}(S_1)\} \subseteq \mathit{flow}(S_1; S_2)$$

and since we have the constraints

$$SS_\circ(\ell) \supseteq \bigcup\{SS_\bullet(\ell') \mid (\ell', \ell) \in \mathit{flow}(S_1; S_2)\}$$

for all ℓ we get

$$SS_\circ(\mathit{init}(S_2)) \supseteq \bigcup\{SS_\bullet(\ell) \mid \ell \in \mathit{final}(S_1)\}$$

and hence $\sigma' \in SS_\circ(\mathit{init}(S_2))$ as required.

The remaining cases follow the same pattern and are omitted. ∎

Remark. The SS analysis is unlikely to be optimal and hence is unlikely to equal the collecting semantics (see Section 1.5 or Exercise 4.5). This may be demonstrated by exhibiting an example where $\bigcup\{SS_\bullet(\ell) \mid \ell \in \mathit{final}(S_\star)\}$ is strictly larger than $\{\sigma' \mid \langle S_\star, \sigma\rangle \rightarrow^* \sigma' \wedge \sigma \in \mathbf{State}\}$ and it is fairly easy to do so. To obtain a specification of the collecting semantics we should let transfer functions be associated with edges rather than nodes as this would allow us to record the outcome of tests (see Exercise 2.11). ∎

Constant Propagation Analysis. The analysis of Section 2.3 is specified by a generalised Monotone Framework consisting of

- the complete lattice $\widehat{\mathbf{State}}_{\mathsf{CP}} = ((\mathbf{Var} \rightarrow \mathbf{Z}^\top)_\perp, \sqsubseteq)$.

The instance for the statement $S_\star$ is determined by

- the flow F is $\mathit{flow}(S_\star)$;
- the set E of extremal labels is $\{\mathit{init}(S_\star)\}$;
- the extremal value ι is $\lambda x.\top$; and
- the transfer functions are given by the mapping $f_\cdot^{\mathsf{CP}}$ defined in Table 2.7.

Galois connection. The relationship between the two analyses is established by defining the representation function

$$\beta_{\mathsf{CP}} : \mathbf{State} \to \widehat{\mathbf{State}}_{\mathsf{CP}}$$

by $\beta_{\mathsf{CP}}(\sigma) = \sigma$ (as in Example 4.6). As in Section 4.3 this gives rise to a Galois connection $(\mathcal{P}(\mathbf{State}), \alpha_{\mathsf{CP}}, \gamma_{\mathsf{CP}}, \widehat{\mathbf{State}}_{\mathsf{CP}})$ where $\alpha_{\mathsf{CP}}(\Sigma) = \bigsqcup\{\beta_{\mathsf{CP}}(\sigma) \mid \sigma \in \Sigma\}$ and $\gamma_{\mathsf{CP}}(\widehat{\sigma}) = \{\sigma \mid \beta_{\mathsf{CP}}(\sigma) \sqsubseteq \widehat{\sigma}\}$. One can now show that for all labels ℓ

$$f_\ell^{\mathsf{CP}} \sqsupseteq \alpha_{\mathsf{CP}} \circ f_\ell^{\mathsf{SS}} \circ \gamma_{\mathsf{CP}}$$

as well as $\gamma_{\mathsf{CP}}(\lambda x.\top) = \mathbf{State}$. Let us only consider the case where $[x := a]^\ell$ occurs in S_* and calculate

$$\begin{aligned}
\alpha_{\mathsf{CP}}(f_\ell^{\mathsf{SS}}(\gamma_{\mathsf{CP}}(\widehat{\sigma}))) &= \alpha_{\mathsf{CP}}(f_\ell^{\mathsf{SS}}(\{\sigma \mid \sigma \sqsubseteq \widehat{\sigma}\})) \\
&= \alpha_{\mathsf{CP}}(\{\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \mid \sigma \sqsubseteq \widehat{\sigma}\}) \\
&= \bigsqcup\{\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \mid \sigma \sqsubseteq \widehat{\sigma}\} \\
&\sqsubseteq \widehat{\sigma}[x \mapsto \bigsqcup\{\mathcal{A}[\![a]\!]\sigma \mid \sigma \sqsubseteq \widehat{\sigma}\}] \\
&\sqsubseteq f_\ell^{\mathsf{CP}}(\widehat{\sigma})
\end{aligned}$$

and where the last step follows from $\bigsqcup\{\mathcal{A}[\![a]\!]\sigma \mid \sigma \sqsubseteq \widehat{\sigma}\} \sqsubseteq \mathcal{A}_{\mathsf{CP}}[\![a]\!]\widehat{\sigma}$ which can be proved by a straightforward structural induction on a.

Thus we conclude that CP is an upper approximation to the analysis induced from SS by the Galois connection and hence it is correct.

4.5.3 Inducing along the Concretisation Function

Widening operator induced by Galois connection. Suppose that we have a Galois connection (L, α, γ, M) between the complete lattices L and M, and also a monotone function $f : L \to L$. Often the motivation for approximating f arises because a fixed point of f is desired, and the ascending chain $(f^n(\bot))_n$ does not eventually stabilise (or may do so in too many iterations). Instead of using $\alpha \circ f \circ \gamma : M \to M$ to remedy this situation it is often possible to consider a *widening operator* $\nabla_M : M \times M \to M$ and use it to define $\nabla_L : L \times L \to L$ by the formula:

$$l_1 \ \nabla_L \ l_2 = \gamma(\alpha(l_1) \ \nabla_M \ \alpha(l_2))$$

If ∇_L turns out to be a widening operator we then know how to approximate the least fixed point of $f : L \to L$ while calculating over L. This has the advantage that the coarser structure of M is only used to ensure convergence and does not needlessly reduce the precision of all other operations. The following result gives sufficient criteria for this approach to work:

Proposition 4.44
Let (L, α, γ, M) be a Galois connection and let $\nabla_M : M \times M \to M$ be an upper bound operator. Then the formula

$$l_1 \ \nabla_L \ l_2 = \gamma(\alpha(l_1) \ \nabla_M \ \alpha(l_2))$$

defines an upper bound operator $\nabla_L : L \times L \to L$. It defines a widening operator if one of the following two conditions are fulfilled:

(i) M satisfies the Ascending Chain Condition, or

(ii) (L, α, γ, M) is a Galois insertion and $\nabla_M : M \times M \to M$ is a widening operator.

Proof First we prove that ∇_L is an upper bound operator. Since ∇_M is an upper bound operator we have $\alpha(l_i) \sqsubseteq \alpha(l_1)\nabla_M\alpha(l_2)$. Using that a Galois connection is an adjunction we get $l_i \sqsubseteq \gamma(\alpha(l_1)\nabla_M\alpha(l_2))$, i.e. $l_i \sqsubseteq l_1\nabla_L l_2$.

Assume now that condition (i) is fulfilled and consider an ascending chain $(l_n)_n$ in L. We know that also $(l_n^{\nabla_L})_n$ is an ascending chain and that $l_n^{\nabla_L} \in \gamma[M]$ for $n > 0$. Hence $(\alpha(l_n^{\nabla_L}))_n$ is an ascending chain and since M satisfies the Ascending Chain Condition there exists $n_0 \geq 1$ such that $\alpha(l_n^{\nabla_L}) = \alpha(l_{n_0}^{\nabla_L})$ for all $n \geq n_0$. So $\gamma(\alpha(l_n^{\nabla_L})) = \gamma(\alpha(l_{n_0}^{\nabla_L}))$ for all $n \geq n_0$ and using that $\gamma \circ \alpha \circ \gamma = \gamma$ (Fact 4.24) we get $l_n^{\nabla_L} = l_{n_0}^{\nabla_L}$ for all $n \geq n_0$. This completes the proof.

Assume next that condition (ii) is fulfilled and consider again an ascending chain $(l_n)_n$ in L. Since α is monotone it follows that $(\alpha(l_n)_n)$ is an ascending chain in M. Now, ∇_M is a widening operator on M so there exists n_0 such that $(\alpha(l_n))^{\nabla_M} = (\alpha(l_{n_0}))^{\nabla_M}$ for $n \geq n_0$. We shall now prove that

$$(\alpha(l_n))^{\nabla_M} \quad = \quad \alpha(l_n^{\nabla_L}) \tag{4.13}$$

for all $n \geq 0$. The case $n = 0$ is immediate since $(\alpha(l_0))^{\nabla_M} = \alpha(l_0) = \alpha(l_0^{\nabla_L})$. For the induction step we assume that $(\alpha(l_n))^{\nabla_M} = \alpha(l_n^{\nabla_L})$. Then

$$\begin{aligned}(\alpha(l_{n+1}))^{\nabla_M} &= (\alpha(l_n))^{\nabla_M} \ \nabla_M \ \alpha(l_{n+1}) \\ &= \alpha(l_n^{\nabla_L}) \ \nabla_M \ \alpha(l_{n+1})\end{aligned}$$

and

$$\begin{aligned}\alpha(l_{n+1}^{\nabla_L}) &= \alpha(l_n^{\nabla_L} \ \nabla_L \ l_{n+1}) \\ &= \alpha(\gamma(\alpha(l_n^{\nabla_L}) \ \nabla_M \ \alpha(l_{n+1}))) \\ &= \alpha(l_n^{\nabla_L}) \ \nabla_M \ \alpha(l_{n+1})\end{aligned}$$

since (L, α, γ, M) is a Galois insertion.

Using (4.13) we thus have that there exists n_0 such that $\alpha(l_n^{\nabla_L}) = \alpha(l_{n_0}^{\nabla_L})$ for all $n \geq n_0$. But then $\gamma(\alpha(l_n^{\nabla_L})) = \gamma(\alpha(l_{n_0}^{\nabla_L}))$ and hence $l_n^{\nabla_L} = l_{n_0}^{\nabla_L}$ for all $n \geq n_0$ as was shown above. This completes the proof. ∎

Precision of induced widening operator. The following result compares the precision of using the widening operator ∇_L with the precision of using the widening operator ∇_M.

Lemma 4.45 If (L, α, γ, M) is a Galois insertion such that $\gamma(\bot_M) = \bot_L$, and if $\nabla_M : M \times M \to M$ is a widening operator, then the widening operator $\nabla_L : L \times L \to L$ defined by $l_1 \ \nabla_L \ l_2 = \gamma(\alpha(l_1) \ \nabla_M \ \alpha(l_2))$ satisfies

$$lfp_{\nabla_L}(f) = \gamma(lfp_{\nabla_M}(\alpha \circ f \circ \gamma))$$

for all monotone functions $f : L \to L$. ∎

Proof By Proposition 4.44 we already know that ∇_L is a widening operator. Hence there exists $n_f \geq 0$ such that $lfp_{\nabla_L}(f) = f^{n_f}_{\nabla_L} = f^n_{\nabla_L}$ for all $n \geq n_f$. Next write $g = \alpha \circ f \circ \gamma$ and recall that ∇_M is a widening operator. Hence there exists $n_g \geq 0$ such that $lfp_{\nabla_M}(g) = g^{n_g}_{\nabla_M} = g^n_{\nabla_M}$ for all $n \geq n_g$. To obtain the desired result it therefore suffices to prove

$$f^n_{\nabla_L} \quad = \quad \gamma(g^n_{\nabla_M}) \tag{4.14}$$

by induction on n. The base case ($n = 0$) is immediate since $f^0_{\nabla_L} = \bot_L$ and $g^0_{\nabla_M} = \bot_M$ and we assumed that $\bot_L = \gamma(\bot_M)$.

To prepare for the induction step we prove that (4.14) implies that:

$$f(f^n_{\nabla_L}) \sqsubseteq f^n_{\nabla_L} \quad \Leftrightarrow \quad g(g^n_{\nabla_M}) \sqsubseteq g^n_{\nabla_M} \tag{4.15}$$

For "$\Rightarrow$" we calculate (using (4.14) and that (L, α, γ, M) is a Galois connection):

$$\begin{array}{rcl} f(f^n_{\nabla_L}) \sqsubseteq f^n_{\nabla_L} & \Rightarrow & \alpha(f(f^n_{\nabla_L})) \sqsubseteq \alpha(f^n_{\nabla_L}) \\ & \Rightarrow & \alpha(f(\gamma(g^n_{\nabla_M}))) \sqsubseteq \alpha(\gamma(g^n_{\nabla_M})) \\ & \Rightarrow & g(g^n_{\nabla_M}) \sqsubseteq \alpha(\gamma(g^n_{\nabla_M})) \\ & \Rightarrow & g(g^n_{\nabla_L}) \sqsubseteq g^n_{\nabla_M} \end{array}$$

For "$\Leftarrow$" we calculate (using (4.14) and that (L, α, γ, M) is a Galois connection):

$$\begin{array}{rcl} g(g^n_{\nabla_M}) \sqsubseteq g^n_{\nabla_M} & \Rightarrow & \gamma(g(g^n_{\nabla_M})) \sqsubseteq \gamma(g^n_{\nabla_M}) \\ & \Rightarrow & \gamma(\alpha(f(\gamma(g^n_{\nabla_M})))) \sqsubseteq \gamma(g^n_{\nabla_M}) \\ & \Rightarrow & \gamma(\alpha(f(f^n_{\nabla_L}))) \sqsubseteq f^n_{\nabla_L} \\ & \Rightarrow & f(f^n_{\nabla_L}) \sqsubseteq f^n_{\nabla_L} \end{array}$$

Returning to the induction step ($n > 0$) of the proof of (4.14) we calculate:

$$\begin{array}{rcl} f^n_{\nabla_L} & = & \left\{ \begin{array}{ll} f^{n-1}_{\nabla_L} & \text{if } f(f^{n-1}_{\nabla_L}) \sqsubseteq f^{n-1}_{\nabla_L} \\ f^{n-1}_{\nabla_L} \ \nabla_L \ f(f^{n-1}_{\nabla_L}) & \text{otherwise} \end{array} \right. \\ & = & \left\{ \begin{array}{ll} f^{n-1}_{\nabla_L} & \text{if } g(g^{n-1}_{\nabla_M}) \sqsubseteq g^{n-1}_{\nabla_M} \\ f^{n-1}_{\nabla_L} \ \nabla_L \ f(f^{n-1}_{\nabla_L}) & \text{otherwise} \end{array} \right. \\ & = & \left\{ \begin{array}{ll} \gamma(g^{n-1}_{\nabla_M}) & \text{if } g(g^{n-1}_{\nabla_M}) \sqsubseteq g^{n-1}_{\nabla_M} \\ \gamma(\alpha(\gamma(g^{n-1}_{\nabla_M})) \ \nabla_M \ \alpha(f(\gamma(g^{n-1}_{\nabla_M})))) & \text{otherwise} \end{array} \right. \end{array}$$

$$
\begin{aligned}
&= \gamma\left(\begin{cases} g^{n-1}_{\nabla_M} & \text{if } g(g^{n-1}_{\nabla_M}) \sqsubseteq g^{n-1}_{\nabla_M} \\ \alpha(\gamma(g^{n-1}_{\nabla_M}))\ \nabla_M\ g(g^{n-1}_{\nabla_M}) & \text{otherwise} \end{cases}\right) \\
&= \gamma\left(\begin{cases} g^{n-1}_{\nabla_M} & \text{if } g(g^{n-1}_{\nabla_M}) \sqsubseteq g^{n-1}_{\nabla_M} \\ g^{n-1}_{\nabla_M}\ \nabla_M\ g(g^{n-1}_{\nabla_M}) & \text{otherwise} \end{cases}\right) \\
&= \gamma(g^{n}_{\nabla_M})
\end{aligned}
$$

In this calculation we have used that (4.14) and (4.15) hold for $n-1$, the definition of ∇_L, and that (L, α, γ, M) is a Galois insertion. ∎

This result then provides the formal justification for the motivating remarks in the beginning of the subsection. To be specific let M be of finite height, let (L, α, γ, M) be a Galois insertion satisfying $\gamma(\bot_M) = \bot_L$, and let ∇_M be the least upper bound operator $\sqcup_M$. Then the above lemma shows that

$$lfp_{\nabla_L}(f) = \gamma(lfp(\alpha \circ f \circ \gamma))$$

which means that $lfp_{\nabla_L}(f)$ *equals* the result we would have obtained if we decided to work with $\alpha \circ f \circ \gamma : M \to M$ instead of the given $f : L \to L$; furthermore the number of iterations needed turn out to be the same. Since the greater precision of L over M is available for all other operations, this suggests that the use of widening operators is often preferable to the approach of Subsection 4.5.1.

Concluding Remarks

In this chapter we have only been able to touch upon a few of the central concepts of Abstract Interpretation; this has mainly been based on [37, 39, 35, 40]. Mini Project 4.1 and the series of examples leading up to Example 4.39 are based on [179]. Mini Project 4.3 is inspired by [126, 68]. Much more can be said both about the development of the theory and about its applications. In this section we briefly discuss some of the more important concepts that have been omitted so far.

Upper closure operators. An *upper closure operator* $\rho : L \to L$ is a monotone function that is extensive (i.e. satisfies $\rho \sqsupseteq \lambda l.l$) and idempotent (i.e. $\rho \circ \rho = \rho$). Such operators arise naturally in Abstract Interpretation [39] because whenever (L, α, γ, M) is a Galois connection the function $\gamma \circ \alpha : L \to L$ is easily seen to be an upper closure operator. Furthermore, if $\rho : L \to L$ is an upper closure operator then the image $\rho[L] = \{\rho(l) \mid l \in L\}$ of L under ρ equals the set $Fix(\rho) = \{l \in L \mid l = \rho(l)\}$ of fixed points of ρ and is a complete lattice under the partial ordering of L; in fact $(L, \rho, \lambda l.l, \rho[L])$ is a Galois insertion.

It follows that upper closure operators may be used to represent Galois connections by simply demanding that the more approximate space M is actually

a subset of L and that no essential features are lost by doing this. This then opens up the possibility for directly comparing the precision of various Galois connections over the same complete lattice L by simply relating the closure operators. The relation $\rho_1 \sqsubseteq \rho_2$ is naturally defined by $\forall l \in L : \rho_1(l) \sqsubseteq \rho_2(l)$ but turns out to be equivalent to the condition that $\rho_2(L) \subseteq \rho_1(L)$ and represents the fact that ρ_2 is more approximate than ρ_1.

Having defined an ordering on the set of upper closure operators on L one can next show that the set is a complete lattice: the least element is given by the upper closure operator $\lambda l.l$ and the greatest element by $\lambda l.\top$. The binary greatest lower bound operator $\sqcap$ is of special interest: it gives rise to the construction of the reduced product [39] (see Section 4.3.)

A number of additional constructs can be explained by means of upper closure operators: we just mention reduced cardinal power [39] and disjunctive completion [39]. By "inverting" some of these constructions it may then be possible to find the "optimal bases" with respect to a given combination: for reduced product the notion of pseudo-complementation has been used to find the minimal factors [33, 34], and for disjunctive completion one can identify the basic irreducible properties that are inherently necessary for the analysis [63, 64].

Stronger properties on the complete lattices. Being a complete lattice is a rather weak notion compared to being a powerset. By considering more structure on the complete lattices, say distributivity, and identifying the elements that correspond to singletons, e.g. atoms or join irreducible elements, it is frequently possible to lift some of the stronger results that hold for powersets to a larger class of complete lattices.

Since powersets are isomorphic to bit vectors this gives a way of finding more general conditions on analyses for when they are as efficient as the *Bit Vector Frameworks*. This is of particular interest in the case of fixed points, where one can use properties of distributive lattices and distributive analyses to give rather low bounds on the number of iterations needed for the ascending chain $(f^n(\bot))_n$ to stabilise [129, 118].

Another line of work concerns the development of the *tensor product* for complete lattices that are not also powersets [111, 113, 115]. Several notions of tensor product have been studied in lattice theory but the development of tensor products suitable for program analysis was first done in [111].

Concrete analyses. In this chapter we have concentrated on introducing some of the key notions in the theory of Abstract Interpretation and only occasionally have we hinted at concrete applications.

One of the main applications of Abstract Interpretation has been in the area of logic programming. To implement a program efficiently it is important to have precise information about the substitutions that may reach the various

program points; a central question to be asked for a substitution is whether or not it is ground. A number of analyses have been designed for this and most of these build upon the framework of Abstract Interpretation. This includes the design of iteration strategies based on widening, and the decomposition of base domains using the techniques mentioned above under upper closure operators.

Another main application of Abstract Interpretation has been to approximate subsets of n-dimensional vector spaces over integers or rationals. For the purpose of this discussion we shall limit ourselves to at most two dimensions (the line and the plane). In the case of one dimension there are two main techniques. One we already illustrated: the lattice of intervals, and it may be generalised to consider (possibly finite) unions of intervals. The other technique records sets of numbers modulo some base value, e.g. $\{x \mid x \bmod k_1 = k_2\}$. Clearly these two analyses can be combined. In the case of two (or more) dimensions it is straightforward to perform an independent attribute analysis where the techniques above are applied component-wise for each dimension.

A substantial effort has been devoted to developing more interesting relational analyses for two (or more) dimensions where the choice of axes is of less importance for the ability to approximate subsets of vector space. An early method was the affine subspaces of Karr [94] where sets of the form $\{(x, y) \mid k_1 x + k_2 y = k_3\}$ can be described. The generalisation from equality to inequality, and allowing to take intersections of such subsets, was considered by Cousot and Halbwachs [41] and resulted in a study of convex polygons. Generalisations and combinations of these ideas have been developed by Granger [66, 67] and by Masdupuy [105].

An interesting line of work pioneered by Deutsch [44, 45] is to change the problem of describing regular sets of words over a finite alphabet to the problem of describing sets of integer vectors. This is by no means trivial but once it has been achieved it opens up the possibility for using all of the above techniques to represent also regular sets of words. This is very important for the analysis of higher-order and concurrent programs, as shown by Deutsch and Colby [31, 32], since it can describe the shape of activation records and communication patterns in much greater precision than other comparative techniques.

We should also mention techniques for building the abstract space of properties "dynamically" [24] and for using widening and narrowing to improve the performance of chaotic iteration [25].

Duality. The dual $\sqsubseteq^d$ of a partial ordering $\sqsubseteq$ is obtained by defining $l_1 \sqsubseteq^d l_2$ if and only if $l_2 \sqsubseteq l_1$; thus we could write $\sqsubseteq^d$ as $\sqsupseteq$. Any concept defined in terms of partial orderings can be dualised by replacing all partial orderings by their dual. In this way the dual least element is the greatest

element, and the dual least upper bound is the greatest lower bound etc. The principle of lattice duality of Lattice Theory says that if any statement about partially ordered sets is true then so is the dual statement. (Interestingly the concept of monotonicity is its own dual.) However, we should like to point out that the dual of a complete lattice may of course differ from the complete lattice itself; pictorially we represent this by drawing the complete lattice "up-side down".

The principle of lattice duality is important for program analysis because it gives an easy way of relating the literature on Abstract Interpretation to the "classical" literature on Data Flow Analysis: simply dualise the complete lattices. So in Abstract Interpretation the greatest element is trivially safe and conveys no information whereas in "classical" Data Flow Analysis it is the least element that has this role. Similarly, in Abstract Interpretation we are interested in least fixed points whereas in "classical" Data Flow Analysis we are interested in greatest fixed points.

Staying within the basic approach of Abstract Interpretation, that going up in the complete lattice means losing information, it is still possible to dualise much of the development: in particular we can define the notion of dual Galois connections. To see why this may be worthwhile consider the following scenario. In program analysis we aim at establishing an element $l_\ell \in L$ for describing the set of values that may reach a given program point ℓ. In program transformation it is frequently the case that a certain transformation Ξ is valid only if the set of values that reach a certain point have certain properties; we may formulate this as the condition $l_\ell \sqsubseteq l_\Xi$. Now if we want to be more approximate we approximate l_ℓ to l'_ℓ and l_Ξ to l'_Ξ and formulate the approximate condition $l'_\ell \sqsubseteq l'_\Xi$. To ensure that $l'_\ell \sqsubseteq l'_\Xi$ implies $l_\ell \sqsubseteq l_\Xi$ we demand that $l_\ell \sqsubseteq l'_\ell$ and that $l'_\Xi \sqsubseteq l_\Xi$. Thus properties of program points are approximated by going *up* in the complete lattice, for which Galois connections are useful, whereas enabling conditions for program transformations are approximated by going *down* in the complete lattice, and for this the concept of dual Galois connections is useful.

A final word of advice concerns the interplay between Abstract Interpretation and *Denotational Semantics*. In Denotational Semantics the least element conveys absolutely no information, and we learn more when things get larger according to the partial order; had there been a greatest element it would have denoted conflicting information. This is quite opposite to the situation in Abstract Interpretation where the greatest element conveys absolutely no information and we learn more when things get smaller according to the partial order; the least element often denotes non-reachability. Hence it would be dangerous to simply apply too many of the intuitions from Denotational Semantics when performing Abstract Interpretation not least because both formalisms ask for least fixed points and therefore are *not duals* of one another.

Mini Projects

Mini Project 4.1 A Galois Connection for Lists

In a series of examples leading up to Example 4.39 we constructed a Galois insertion for recording the relationship between pairs of integers; it was given by

$$(\mathcal{P}(\mathbf{Z} \times \mathbf{Z}), \alpha_{\mathsf{SSR}'}, \gamma_{\mathsf{SSR}'}, \mathcal{P}(\mathbf{AB}))$$

where $\mathbf{AB} \subseteq \mathbf{Sign} \times \mathbf{Sign} \times \mathbf{Range}$ contained only 29 elements (out of the 45 possibilities).

In this mini project we are going to construct a Galois insertion for recording the relationship between pairs of lists. Let V be the domain of lists of finite length over some simple data type. We write $x = [x_1, \cdots, x_n]$ for a list with n elements whose first element is x_1; when $n = 0$ we write $x = [\ ]$. Next let $x = [x_1, \cdots, x_n]$ and $y = [y_1, \cdots, y_m]$ be two lists. They have the same head if and only if $n > 0$, $m > 0$ and $x_1 = y_1$. The list x is a suffix of y if and only if there exists $k \geq 0$ such that $n + k = m$ and $x_i = y_{i+k}$ for $i \in \{1, \cdots, n\}$. Finally we write $\mathit{length}(x) = n$ and $\mathit{length}(y) = m$.

The Galois insertion should have the form

$$(\mathcal{P}(V \times V), \alpha, \gamma, \mathcal{P}'(\mathbf{LR}))$$

where $\mathbf{LR} \subseteq \mathcal{P}(\{\mathsf{H}, \mathsf{S}\}) \times \mathbf{Range}$. Here H means that the lists have the same head, S means that one list is a suffix of the other, and the range components describe $\mathit{length}(x) - \mathit{length}(y)$ where x is the first list and y the second list.

Complete the details of the specification. ∎

Mini Project 4.2 Correctness of the Shape Analysis

We shall now return to the *Shape Analysis* of Section 2.6 and show how it gives rise to a Galois connection. Recall that the semantics uses configurations with a state $\sigma \in \mathbf{State}$ and a heap component $\mathcal{H} \in \mathbf{Heap}$ and that the analysis works on shape graphs consisting of an abstract state S, an abstract heap H and a sharing component is.

We shall begin by defining a function *vars* that given a location and a state will determine the associated abstract location:

$$\mathit{vars}(\xi)(\sigma) = n_X \text{ where } X = \{x \mid \sigma(x) = \xi\}$$

Proceed as follows:

1. Define a representation function

$$\beta_{\mathsf{SA}} : \mathbf{State} \times \mathbf{Heap} \to \mathcal{P}(\mathbf{SG})$$

that to each state and heap associates a singleton set consisting of a compatible shape graph (as defined in Section 2.6) and construct the associated Galois connection

$$(\mathcal{P}(\mathbf{State} \times \mathbf{Heap}), \alpha_{\mathsf{SA}}, \gamma_{\mathsf{SA}}, \mathcal{P}(\mathbf{SG}))$$

Is it a Galois insertion?

To establish the correctness of the analysis we shall follow the approach of Subsection 4.5.2:

2. Specify an analysis SH approximating the sets of pairs of states and heaps as a generalised Monotone Framework over the complete lattice $(\mathcal{P}(\mathbf{State} \times \mathbf{Heap}), \subseteq)$; write f_ℓ^{SH} for the associated transfer functions. Prove the correctness of the analysis SH (i.e. prove an analogue of Lemma 4.43).

3. Show that $f_\ell^{\mathsf{SA}} \sqsupseteq \alpha_{\mathsf{SA}} \circ f_\ell^{\mathsf{SH}} \circ \gamma_{\mathsf{SA}}$ for all transfer functions and conclude that the Shape Analysis is correct. Determine whether or not $f_\ell^{\mathsf{SA}} = \alpha_{\mathsf{SA}} \circ f_\ell^{\mathsf{SH}} \circ \gamma_{\mathsf{SA}}$ holds for all transfer functions. (See Exercise 2.23.) ■

Mini Project 4.3 Application to Control Flow Analysis

In this mini project we shall perform an analogue of the development of Subsection 4.5.2 for the Control and Data Flow Analysis of Section 3.5.

1. Specify a "sets of values" analysis

$$(\widehat{\mathsf{C}}_{\mathsf{SV}}, \widehat{\rho}_{\mathsf{SV}}) \models_{\mathsf{SV}} e$$

in the manner of Subsection 3.5.1 (by taking $\mathbf{Data} = \mathbf{Val}$ where $\mathbf{Val}$ is as in Section 3.2). Formulate and prove a semantic correctness result in the manner of Example 4.40 and Theorem 3.10.

2. Let a *monotone structure* $(L, \mathcal{F})$ be given as in Subsection 3.5.2 and consider a Galois connection $(\mathcal{P}(\mathbf{Val}), \alpha, \gamma, L)$. Motivated by the judgements of the acceptability relation $(\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{d}) \models_D e$ construct a Galois connection:

$$(\{(\widehat{\mathsf{C}}_{\mathsf{SV}}, \widehat{\rho}_{\mathsf{SV}}) \mid \cdots\}, \alpha', \gamma', \{(\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{d}) \mid \cdots\})$$

Formulate and prove a result intended to establish

$$(\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{d}) \models_D e \quad \Rightarrow \quad \gamma'((\widehat{\mathsf{C}}, \widehat{\mathsf{D}}, \widehat{\rho}, \widehat{d})) \models_{\mathsf{SV}} e$$

and argue that this shows the semantic correctness of the Control and Data Flow Analysis. ■

Figure 4.10: The complete lattice (**Sign**′, $\sqsubseteq$).

Exercises

Exercise 4.1 For the complete lattice (**Sign**′, $\sqsubseteq$) of Figure 4.10 define a correctness relation $R_{ZS'} : \mathbf{Z} \times \mathbf{Sign}' \rightarrow \{true, false\}$. Verify that it indeed has the properties (4.4) and (4.5). Next define a representation function $\beta_{ZS'} : \mathbf{Z} \rightarrow \mathbf{Sign}'$ and show that the $R_{ZS'}$ constructed above is indeed generated by $\beta_{ZS'}$. ■

Exercise 4.2 Show that if (4.4) and (4.5) hold for R and L then we also have:

$$v \; R \; l_1 \;\wedge\; v \; R \; l_2 \quad \Rightarrow \quad v \; R \; (l_1 \sqcup l_2)$$

and more generally:

$$(\forall l \in L' \neq \emptyset : v \; R \; l) \quad \Rightarrow \quad v \; R \; (\bigsqcup L')$$

Give an example showing that $v \; R \; \bot$ fails even though (4.4) and (4.5) are fulfilled. ■

Exercise 4.3 Show that the Control Flow Analysis of Chapter 3 is indeed correct in the sense of condition (4.3) as claimed in Example 4.4. To do so first show that

$$(\widehat{\mathsf{C}}, \widehat{\rho}) \models v^\ell \text{ iff } v \; \mathcal{V} \; (\widehat{\rho}, \widehat{\mathsf{C}}(\ell))$$

whenever v is a value (i.e. a closure or a constant). Next show that

$$[\,] \vdash (e_\star \; v_1^{\ell_1})^\ell \rightarrow^* v^\ell \wedge (\widehat{\mathsf{C}}, \widehat{\rho}) \models (e_\star \; v_1^{\ell_1})^\ell \;\Rightarrow\; (\widehat{\mathsf{C}}, \widehat{\rho}) \models v^\ell$$

is a corollary of Theorem 3.10. Finally assume the premise of (4.3) and use the above results to obtain the desired result. ■

Exercise 4.4 Show that the relation R_{CFA} defined in Example 4.4 is generated by the representation function β_{CFA} also defined in Example 4.7. To do so prove that

$$v \; R_{\mathsf{CFA}} \; (\widehat{\rho}, \widehat{v}) \text{ iff } \beta_{\mathsf{CFA}}(v) \sqsubseteq_{\mathsf{CFA}} (\widehat{\rho}, \widehat{v})$$

by induction on the size of v; only the case where v is `close` t `in` ρ is non-trivial. ■

Exercise 4.5 Define $L_i = (\mathcal{P}(V_i), \subseteq)$ (for $i = 1, 2$) and define $f_p : L_1 \to L_2$ by

$$f_p(l_1) = \{v_2 \in V_2 \mid \exists v_1 \in l_1 : p \vdash v_1 \rightsquigarrow v_2\}$$

Show that f_p is monotone. Next show that $(p \vdash \cdot \rightsquigarrow \cdot)(R_1 \twoheadrightarrow R_2)\, f_p$ where $v_i \; R_i \; l_i$ is defined by $v_i \in l_i$. Also, show that for $f' : L_1 \to L_2$ we have $(p \vdash \cdot \rightsquigarrow \cdot)(R_1 \twoheadrightarrow R_2)\, f'$ if and only if $f_p \sqsubseteq f'$. A semantics that associates a program p with f_p as defined here is sometimes called a *collecting semantics*. Finally, note that R_i is generated by β_i defined by $\beta_i(v_i) = \{v_i\}$; show that $f_p = (\beta_1 \twoheadrightarrow \beta_2)(p \vdash \cdot \rightsquigarrow \cdot)$. ■

Exercise 4.6 Show that all of

- $\sqcup$
- $\lambda(l_1, l_2).\top$
- $\lambda(l_1, l_2).\begin{cases} l_1 & \text{if } l_2 \sqsubseteq l_1 \\ \top & \text{otherwise} \end{cases}$
- $\lambda(l_1, l_2).\begin{cases} l_2 & \text{if } l_1 = \bot \\ l_1 & \text{if } l_2 \sqsubseteq l_1 \;\wedge\; l_1 \neq \bot \\ \top & \text{otherwise} \end{cases}$
- $\lambda(l_1, l_2).\begin{cases} l_1 \sqcup l_2 & \text{if } l_1 \sqsubseteq l' \vee l_2 \sqsubseteq l_1 \\ \top & \text{otherwise} \end{cases}$

are upper bound operators (where l' is some element of L). Determine which of them that are also widening operators. Try to find sufficient conditions on l' such that the operator involving l' is a widening operator. ■

Exercise 4.7 Show that if L satisfies the Ascending Chain Condition then an operator on L is a widening operator if and only if it is an upper bound operator. Conclude that if L satisfies the Ascending Chain Condition then the least upper bound operator $\sqcup : L \times L \to L$ is a widening operator. ■

Exercise 4.8 Consider changing the definition of f^0_∇ from $f^0_\nabla = \bot$ to $f^0_\nabla = l_0$ for some $l_0 \in L$. Possible assumptions on l_0 are:

- $l_0 = f(\bot)$;
- $l_0 = f^{27}(\bot)$;
- $l_0 \in Ext(f)$;
- l_0 arbitrary.

Which of these suffice for proving Fact 4.14 and Proposition 4.13? ■

Exercise 4.9 Let ∇_K be as in Example 4.15 and define

$$int_1 \nabla int_2 = \begin{cases} int_1 \sqcup int_2 & \text{if } int_1 \sqsubseteq int' \vee int_2 \sqsubseteq int_1 \\ int_1 \nabla_K int_2 & \text{otherwise} \end{cases}$$

where int' is an interval satisfying $\mathsf{inf}(int') > -\infty$ and $\mathsf{sup}(int') < \infty$. Show that

$$\forall int_1, int_2 : int_1 \nabla int_2 \sqsubseteq int_1 \nabla_K int_2$$

and that the inequality may be strict. Show that ∇ is an upper bound operator. Determine whether or not ∇ is a widening operator. ■

Exercise 4.10 Let $(l_n)_n$ be a descending chain and let $\Delta : L \times L \to L$ be a total function that satisfies $l'_2 \sqsubseteq l'_1 \Rightarrow l'_2 \sqsubseteq (l'_1 \Delta l'_2) \sqsubseteq l'_1$ for all $l'_1, l'_2 \in L$. Show that the sequence $(l_n^\Delta)_n$ is a descending chain and that $l_n^\Delta \sqsupseteq l_n$ for all n. ■

Exercise 4.11 Consider the following alternative strategy to narrowing for improving the approximation $f_\nabla^m \in Red(f)$ to the fixed point $lfp(f)$ of the function $f : L \to L$. A *descending chain truncator* is a function Υ that maps descending chains $(l_n)_n$ to a non-negative number such that

> if $(l_n)_n$ and $(l'_n)_n$ are descending chains and $\forall n \leq \Upsilon((l_n)_n) : l_n = l'_n$ then $\Upsilon((l_n)_n) = \Upsilon((l'_n)_n)$.

This ensures that Υ is finitely calculable. The truncated descending chain then is

$$(f_\nabla^m, \cdots, f^n(f_\nabla^m), \cdots, f^{m'}(f_\nabla^m))$$

where $m' = \Upsilon((f^n(f_\nabla^m))_n)$ and the desired approximation to $lfp(f)$ is

$$lfp_\nabla^\Upsilon(f) = f^{m'}(f_\nabla^m)$$

Prove that this development can be used as an alternative to narrowing and try to determine the relationship between the two concepts. ■

Exercise 4.12 Show that if L satisfies the Descending Chain Condition then the binary greatest lower bound operator $\sqcap : L \times L \to L$ is a narrowing operator. ■

Exercise 4.13 Consider the complete lattice **Interval** of Figure 4.2 and the complete lattice **Sign′** of Figure 4.10 and define $\gamma_{IS'}$ by

$$\begin{array}{rclrcl} \gamma_{IS'}(\top) & = & [-\infty, \infty] & \gamma_{IS'}(-0) & = & [-\infty, 0] \\ \gamma_{IS'}(0+) & = & [0, \infty] & \gamma_{IS'}(-) & = & [-\infty, -1] \\ \gamma_{IS'}(0) & = & [0, 0] & \gamma_{IS'}(+) & = & [1, \infty] \\ \gamma_{IS'}(\bot) & = & \bot & & & \end{array}$$

Show that there exists a Galois connection between **Interval** and **Sign′** with $\gamma_{IS'}$ as the upper adjoint. ■

Exercise 4.14 Let $(\mathcal{P}(V), \alpha_\eta, \gamma_\eta, \mathcal{P}(D))$ be a Galois connection that is given by an extraction function $\eta : V \to D$. Show that α_η is surjective if and only if η is. Conclude that $(\mathcal{P}(V), \alpha_\eta, \gamma_\eta, \mathcal{P}(D))$ is a Galois insertion if and only if η is surjective. Next show that $\varsigma_\eta[\mathcal{P}(D)] = \alpha_\eta[\mathcal{P}(V)]$ is isomorphic (see Appendix A) to $\mathcal{P}(\eta[V])$ where $\eta[V]$ is the image of η and finally verify that $\varsigma_\eta(V') = V' \cap \eta[V]$. ■

Exercise 4.15 Assume that the Galois connections $(\mathcal{P}(D_i), \alpha_{\eta_{i+1}}, \gamma_{\eta_{i+1}}, \mathcal{P}(D_{i+1}))$ are given by the extraction functions $\eta_{i+1} : D_i \to D_{i+1}$. Show that the composition of the Galois connections, $(\mathcal{P}(D_0), \alpha, \gamma, \mathcal{P}(D_2))$, will have $\alpha = \alpha_{\eta_2} \circ \alpha_{\eta_1} = \alpha_{\eta_2 \circ \eta_1}$ and $\gamma = \gamma_{\eta_1} \circ \gamma_{\eta_2} = \gamma_{\eta_2 \circ \eta_1}$; i.e. the Galois connection is given by the extraction function $\eta_2 \circ \eta_1$. ■

Exercise 4.16 Let $(\mathcal{P}(V_1), \alpha_1, \gamma_1, \mathcal{P}(D_1))$ and $(\mathcal{P}(V_2), \alpha_2, \gamma_2, \mathcal{P}(D_2))$ be Galois connections given by the extraction functions $\eta_1 : V_1 \to D_1$ and $\eta_2 : V_2 \to D_2$. Furthermore assume that V_1 and V_2 are disjoint and similarly for D_1 and D_2. Define an extraction function

$$\eta : V_1 \cup V_2 \to D_1 \cup D_2$$

by

$$\eta(v) = \begin{cases} \eta_1(v) & \text{if } v \in V_1 \\ \eta_2(v) & \text{if } v \in V_2 \end{cases}$$

Show that this defines a Galois connection

$$(\mathcal{P}(V_1 \cup V_2), \alpha_\eta, \gamma_\eta, \mathcal{P}(D_1 \cup D_2))$$

and reformulate it as an isomorphic Galois connection

$$(\mathcal{P}(V_1) \times \mathcal{P}(V_2), \alpha_\eta, \gamma_\eta, \mathcal{P}(D_1) \times \mathcal{P}(D_2))$$

in the manner of the independent attribute method. How important is it that V_1 and V_2 are disjoint and that D_1 and D_2 are disjoint? ■

Exercise 4.17 Let $(L_1, \alpha_1, \gamma_1, M_1)$ and $(L_2, \alpha_2, \gamma_2, M_2)$ be Galois insertions. First define

$$\begin{array}{rcl} \alpha(l_1, l_2) & = & (\alpha_1(l_1), \alpha_2(l_2)) \\ \gamma(m_1, m_2) & = & (\gamma_1(m_1), \gamma_2(m_2)) \end{array}$$

and show that $(L_1 \times L_2, \alpha, \gamma, M_1 \times M_2)$ is a Galois insertion. Then define

$$\begin{array}{rcl} \alpha(f) & = & \alpha_2 \circ f \circ \gamma_1 \\ \gamma(g) & = & \gamma_2 \circ g \circ \alpha_1 \end{array}$$

and show that $(L_1 \to L_2, \alpha, \gamma, M_1 \to M_2)$ is a Galois insertion. ■

Exercise 4.18 Let $(\mathcal{P}(V_1), \alpha_1, \gamma_1, \mathcal{P}(D_1))$ and $(\mathcal{P}(V_2), \alpha_2, \gamma_2, \mathcal{P}(D_2))$ be Galois insertions. Define

$$\begin{array}{rcl} \alpha(VV) & = & \bigcup\{\alpha_1(\{v_1\}) \times \alpha_2(\{v_2\}) \mid (v_1, v_2) \in VV\} \\ \gamma(DD) & = & \{(v_1, v_2) \mid \alpha_1(\{v_1\}) \times \alpha_2(\{v_2\}) \subseteq DD\} \end{array}$$

and determine whether or not $(\mathcal{P}(V_1 \times V_2), \alpha, \gamma, \mathcal{P}(D_1 \times D_2))$ is a Galois insertion. ■

Exercise 4.19 Let $(L, \alpha_1, \gamma_1, M_1)$ and $(L, \alpha_2, \gamma_2, M_2)$ be Galois insertions. Define

$$\begin{array}{rcl} \alpha(l) & = & (\alpha_1(l), \alpha_2(l)) \\ \gamma(m_1, m_2) & = & \gamma_1(m_1) \sqcap \gamma_2(m_2) \end{array}$$

and determine whether or not $(L, \alpha, \gamma, M_1 \times M_2)$ is a Galois insertion. ■

Exercise 4.20 Let $(L_1, \alpha_1, \gamma_1, M_1)$ be a Galois connection and define

$$\begin{array}{rcl} \alpha(f) & = & \alpha_1 \circ f \circ \gamma_1 \\ \gamma(g) & = & \gamma_1 \circ g \circ \alpha_1 \end{array}$$

(in the manner of Section 4.4). Do any of the following equalities

$$\begin{array}{rclrcl} \alpha(\lambda l.l) & = & \lambda m.m & \gamma(\lambda m.m) & = & \lambda l.l \\ \alpha(f_1 \circ f_2) & = & \alpha(f_1) \circ \alpha(f_2) & \gamma(g_1 \circ g_2) & = & \gamma(g_1) \circ \gamma(g_2) \end{array}$$

necessarily hold? Which of the equalities hold when $(L_1, \alpha_1, \gamma_1, M_1)$ is known to be a Galois insertion? ■

Exercise 4.21 Consider the Galois insertion

$$(\mathcal{P}(\mathbf{Z} \times \mathbf{Z}), \alpha_{\mathrm{SSR}'}, \gamma_{\mathrm{SSR}'}, \mathcal{P}(\mathbf{AB}))$$

developed in Example 4.39. Determine for each of the sets

$$\begin{array}{l}
\{(x,y) \mid x = y\} \\
\{(x,y) \mid x = -y\} \\
\{(x,y) \mid x = y+1\} \\
\{(x,y) \mid x = y+3\} \\
\{(x,y) \mid x \geq y\} \\
\{(x,y) \mid x \geq y+1\} \\
\{(x,y) \mid x^2 + y^2 \leq 100\}
\end{array}$$

the best description in $\mathcal{P}(\mathbf{AB})$. ■

Exercise 4.22 Let $(L_i, \alpha_i, \gamma_i, M_i)$ be Galois connections for $i = 1, 2, 3$. Use the approach of Section 4.4 to define

$$\begin{array}{rcl}
\alpha(f) & = & \cdots \\
\gamma(g) & = & \cdots
\end{array}$$

such that $((L_1 \times L_2) \to L_3, \alpha, \gamma, (M_1 \times M_2) \to M_3)$ is a Galois connection. Next let all of $(L_i, \alpha_i, \gamma_i, M_i)$ be the Galois connection

$$(\mathcal{P}(\mathbf{Z}), \alpha_{\mathrm{ZI}}, \gamma_{\mathrm{ZI}}, \mathbf{Interval})$$

of Example 4.19 that relates the set of integers to the intervals. Let `plus` : $\mathcal{P}(\mathbf{Z}) \times \mathcal{P}(\mathbf{Z}) \to \mathcal{P}(\mathbf{Z})$ be the "pointwise" application of addition defined by:

$$\lambda(Z_1, Z_2).\{z_1 + z_2 \mid z_1 \in Z_1 \wedge z_2 \in Z_2\}$$

Next define

$$\alpha(\texttt{plus}) = \lambda(int_1, int_2).\cdots$$

and supply the details of the definition. ■

Exercise 4.23 Let L be the complete lattice of sets of integers, let M be the complete lattice of intervals of Example 4.10, let (L, α, γ, M) be the Galois insertion of Example 4.19, and let $\nabla_M : M \times M \to M$ be the widening operator of Example 4.15. Observe that the formula

$$l_1 \nabla_L l_2 = \gamma(\alpha(l_1) \nabla_M \alpha(l_2))$$

defines a widening operator $\nabla_L : L \times L \to L$ and develop a formula for it (in the manner of the formula for ∇_M of Example 4.15). ■

Exercise 4.24 Suppose that (L, α, γ, M) is a Galois connection or a Galois insertion and that possibly M satisfies the Descending Chain Condition. Let $\Delta_M : M \times M \to M$ be a narrowing operator and try to determine if the formula

$$l_1 \Delta_L l_2 = \gamma(\alpha(l_1) \Delta_M \alpha(l_2))$$

defines a narrowing operator $\Delta_L : L \times L \to L$. ■

Chapter 5

Type and Effect Systems

So far our techniques have applied equally well to typed and untyped programming languages. This flexibility does not apply to the development to be performed in this chapter: here we demand that our programming language is typed because we will use the syntax for types in order to express the program analysis properties of interest (as was already illustrated in Section 1.6).

We shall first present an Annotated Type System for *Control Flow Analysis* in Section 5.1, demonstrate its semantic soundness and other theoretical properties in Section 5.2, and then in Section 5.3 show how to obtain an algorithm for computing the annotated types (and prove that it is sound and complete). In Sections 5.4 and 5.5 we give examples of other analyses specified by Type and Effect Systems. In Section 5.4 we study Type and Effect Systems with rules for subtyping, polymorphism and polymorphic recursion and illustrate their use in an analysis for tracking *Side Effects*, an *Exception Analysis* and an analysis for *Region Inference*. Finally, in Section 5.5 we show that the annotations can be given more structure and we illustrate this for a *Communication Analysis*.

5.1 Control Flow Analysis

Syntax of the FUN language. To illustrate the approach we shall make use of the functional language FUN also considered in Chapter 3; that the approach also applies to the imperative language of Chapter 2 was briefly sketched in Section 1.6. However, in this chapter we shall use a slightly different labelling scheme from the one in Chapter 3; the syntactic category of interest is

$$e \in \mathbf{Exp} \quad \text{expressions}$$

and it is defined by:

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \texttt{fn}_\pi\ x \texttt{ => } e_0 \mid \texttt{fun}_\pi\ f\ x \texttt{ => } e_0 \mid e_1\ e_2 \\
 & \mid & \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \mid \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \mid e_1\ op\ e_2
\end{array}
$$

The *program points*, $\pi \in \mathbf{Pnt}$, are used to name the function abstractions in the program; this could also be done using the notion of labelled terms from Chapter 3 but for our present purposes we do not need the full generality of this machinery – the reason is that now we will use the types to record information that was previously associated with labels. Hence our syntax just makes use of expressions and dispenses with terms.

As in the previous chapters we shall assume that a countable set of variables is given and that constants (including the truth values), binary operators (including the natural arithmetic, boolean and relational operators) and program points are left unspecified:

$$
\begin{array}{rcll}
c & \in & \mathbf{Const} & \text{constants} \\
op & \in & \mathbf{Op} & \text{binary operators} \\
f, x & \in & \mathbf{Var} & \text{variables} \\
\pi & \in & \mathbf{Pnt} & \text{program points}
\end{array}
$$

Example 5.1 The functional program `(fn x => x) (fn y => y)` considered in Chapters 1 and 3 is now written as

$$(\texttt{fn}_X\ \texttt{x => x})\ (\texttt{fn}_Y\ \texttt{y => y})$$

just as we did in Example 1.5. ∎

Example 5.2 The expression loop of Example 3.2 is now written:

```
let g = (funF f x => f (fnY y => y))
in g (fnZ z => z)
```

Recall that this is a looping program: g is first applied to the identity function $\texttt{fn}_Z$ `z => z` but it ignores its argument and calls itself recursively with the function $\texttt{fn}_Y$ `y => y`. ∎

5.1.1 The Underlying Type System

The analyses will be specified as extensions of the ordinary type system in order to record the program properties of interest. For this reason the ordinary type system is sometimes called the *underlying type system* and we shall start by specifying it.

$$[con] \quad \Gamma \vdash_{\mathsf{UL}} c : \tau_c$$

$$[var] \quad \Gamma \vdash_{\mathsf{UL}} x : \tau \qquad \text{if } \Gamma(x) = \tau$$

$$[fn] \quad \frac{\Gamma[x \mapsto \tau_x] \vdash_{\mathsf{UL}} e_0 : \tau_0}{\Gamma \vdash_{\mathsf{UL}} \mathtt{fn}_\pi\ x\ \mathtt{=>}\ e_0 : \tau_x \to \tau_0}$$

$$[fun] \quad \frac{\Gamma[f \mapsto \tau_x \to \tau_0][x \mapsto \tau_x] \vdash_{\mathsf{UL}} e_0 : \tau_0}{\Gamma \vdash_{\mathsf{UL}} \mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e_0 : \tau_x \to \tau_0}$$

$$[app] \quad \frac{\Gamma \vdash_{\mathsf{UL}} e_1 : \tau_2 \to \tau_0 \quad \Gamma \vdash_{\mathsf{UL}} e_2 : \tau_2}{\Gamma \vdash_{\mathsf{UL}} e_1\ e_2 : \tau_0}$$

$$[if] \quad \frac{\Gamma \vdash_{\mathsf{UL}} e_0 : \mathtt{bool} \quad \Gamma \vdash_{\mathsf{UL}} e_1 : \tau \quad \Gamma \vdash_{\mathsf{UL}} e_2 : \tau}{\Gamma \vdash_{\mathsf{UL}} \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau}$$

$$[let] \quad \frac{\Gamma \vdash_{\mathsf{UL}} e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash_{\mathsf{UL}} e_2 : \tau_2}{\Gamma \vdash_{\mathsf{UL}} \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 : \tau_2}$$

$$[op] \quad \frac{\Gamma \vdash_{\mathsf{UL}} e_1 : \tau^1_{op} \quad \Gamma \vdash_{\mathsf{UL}} e_2 : \tau^2_{op}}{\Gamma \vdash_{\mathsf{UL}} e_1\ op\ e_2 : \tau_{op}}$$

Table 5.1: The Underlying Type System.

Types. Let us first introduce the notions of *types* and *type environments*:

$$\begin{array}{lll} \tau & \in & \textbf{Type} \quad \text{types} \\ \Gamma & \in & \textbf{TEnv} \quad \text{type environments} \end{array}$$

We shall assume that the types are given by

$$\tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \to \tau_2$$

where `int` and `bool` are the only two kinds of base types and as usual we use arrows for function types. Each constant $c \in$ **Const** has a type that we shall denote τ_c so e.g. `true` has type $\tau_{\mathtt{true}} = \mathtt{bool}$ and 7 has type $\tau_7 = \mathtt{int}$. Each binary operator *op* will expect two arguments of type τ^1_{op} and τ^2_{op}, respectively, and give a result of type τ_{op} – an example is the relational operation $\leq$ that expects two arguments of type `int` and gives a result of type `bool`. For the sake of simplicity we shall *assume* that all the constants have base types and that all binary operators expect arguments of base types and return values of base types.

The type environments are given by:

$$\Gamma ::= [\ ] \mid \Gamma[x \mapsto \tau]$$

Formally, Γ is a list but nevertheless we shall feel free to regard it as a finite mapping: we write $dom(\Gamma)$ for $\{x \mid \Gamma \text{ contains } [x \mapsto \cdots]\}$; we write $\Gamma(x) = \tau$ if $x \in dom(\Gamma)$ and the *rightmost* occurrence of $[x \mapsto \cdots]$ in Γ is $[x \mapsto \tau]$, and we write $\Gamma \mid X$ for the type environment obtained from Γ by removing all occurrences of $[x \mapsto \cdots]$ with $x \notin X$. For the sake of readability we shall write $[x \mapsto \tau]$ for $[\,][x \mapsto \tau]$.

Typing judgements. The general form of a typing is given by

$$\Gamma \vdash_{\mathsf{UL}} e : \tau$$

that says that the expression e has type τ assuming that any free variable has type given by Γ. The axioms and rules for the judgements are listed in Table 5.1 and are explained below.

The axioms [*con*] and [*var*] are straightforward: the first uses the predefined type for the constant and the second consults the type environment. In the rules [*fn*] and [*fun*] we *guess* a type for the bound variables and determine the types of the bodies under the additional assumptions; the rule [*fun*] has the implicit requirement that the guess of the type for f matches that of the resulting function. As a consequence of these two rules the type system is *nondeterministic* in the sense that $\Gamma \vdash_{\mathsf{UL}} e : \tau_1$ and $\Gamma \vdash_{\mathsf{UL}} e : \tau_2$ does not necessarily imply that $\tau_1 = \tau_2$.

The rule [*app*] requires that the operator and the operand of the application can be typed and implicitly it requires that the type of the operator is a function type where the type before the arrow equals that of the operand – in this way we express that the types of the formal and actual parameter must be equal.

The rules [*if*], [*let*] and [*op*] are straightforward. In particular, the `let`-construct `let` x `=` e_1 `in` e_2 admits exactly the same typings as the application ($\mathtt{fn}_\pi$ x `=>` e_2) e_1 and regardless of the choice of π. In Sections 5.4 and 5.5 we shall consider a polymorphic `let`-construct where `let` x `=` e_1 `in` e_2 may admit more typings than ($\mathtt{fn}_\pi$ x `=>` e_2) e_1.

Example 5.3 Let us show that the expression loop

```
let g = (funF f x => f (fnY y => y))
in g (fnZ z => z)
```

of Example 5.2 has type $\tau \to \tau$ for each type τ. We shall first consider the expression $\mathtt{fun}_F$ `f x => f (`$\mathtt{fn}_Y$ `y => y)` where we write Γ_{fx} for the type environment $[\mathtt{f} \mapsto (\tau \to \tau) \to (\tau \to \tau)][\mathtt{x} \mapsto \tau \to \tau]$. Then we get

$$\Gamma_{fx} \vdash_{\mathsf{UL}} \mathtt{f} : (\tau \to \tau) \to (\tau \to \tau)$$

$$\Gamma_{fx} \vdash_{\mathsf{UL}} \mathtt{fn}_Y\ \mathtt{y} \Rightarrow \mathtt{y} : \tau \to \tau$$

using the axiom [*var*] and the rule [*fn*]. The rule [*app*] then gives

$$\Gamma_{\mathsf{fx}} \vdash_{\mathsf{UL}} \mathtt{f\ (fn_Y\ y\ =>\ y)} : \tau \to \tau$$

and we have a judgement matching the premise of the rule [*fun*]. Hence we get

$$[\,] \vdash_{\mathsf{UL}} \mathtt{fun_F\ f\ x\ =>\ f\ (fn_Y\ y\ =>\ y)} : (\tau \to \tau) \to (\tau \to \tau)$$

Taking Γ_{g} to be the type environment $[\mathsf{g} \mapsto (\tau \to \tau) \to (\tau \to \tau)]$ we get

$$\Gamma_{\mathsf{g}} \vdash_{\mathsf{UL}} \mathtt{g\ (fn_Z\ z\ =>\ z)} : \tau \to \tau$$

using the axiom [*var*] and the rules [*fn*] and [*app*]. The rule [*let*] can now be used to show that the expression loop has type $\tau \to \tau$. ■

5.1.2 The Analysis

Annotated types. That a function has type $\tau_1 \to \tau_2$ means that given an argument of type τ_1 it will return a value of type τ_2 in case it terminates. To get a *Control Flow Analysis* we shall annotate the type with information, $\varphi \in \mathbf{Ann}$, about which function it might be. The annotations are given by

$$\varphi \quad \in \quad \mathbf{Ann} \quad \text{annotations}$$

where:

$$\varphi ::= \{\pi\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset$$

So φ will be a *set* of function names – describing the set of function definitions that can result in a function of a given type; as will be discussed below, we shall feel free to write $\{\pi_1, \ldots, \pi_n\}$ for $\{\pi_1\} \cup \cdots \cup \{\pi_n\}$. We now take

$$\begin{array}{lcll} \widehat{\tau} & \in & \widehat{\mathbf{Type}} & \text{annotated types} \\ \widehat{\Gamma} & \in & \widehat{\mathbf{TEnv}} & \text{annotated type environments} \end{array}$$

and define:

$$\begin{array}{lcl} \widehat{\tau} & ::= & \mathtt{int} \mid \mathtt{bool} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \\ \widehat{\Gamma} & ::= & [\,] \mid \widehat{\Gamma}[x \mapsto \widehat{\tau}] \end{array}$$

We shall write $\lfloor \widehat{\tau} \rfloor$ for the *underlying type* corresponding to the annotated type $\widehat{\tau}$; it is defined as follows:

$$\begin{array}{rcl} \lfloor \mathtt{int} \rfloor & = & \mathtt{int} \\ \lfloor \mathtt{bool} \rfloor & = & \mathtt{bool} \\ \lfloor \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \rfloor & = & \lfloor \widehat{\tau}_1 \rfloor \to \lfloor \widehat{\tau}_2 \rfloor \end{array}$$

As an example we have $\lfloor \mathtt{int} \xrightarrow{\{X\}} \mathtt{int} \rfloor = (\mathtt{int} \to \mathtt{int})$. Furthermore, we extend the notation to operate on type environments so $\lfloor \widehat{\Gamma} \rfloor(x) = \lfloor \widehat{\Gamma}(x) \rfloor$ for all x.

$$[con] \quad \widehat{\Gamma} \vdash_{\mathsf{CFA}} c : \tau_c$$

$$[var] \quad \widehat{\Gamma} \vdash_{\mathsf{CFA}} x : \widehat{\tau} \qquad \text{if } \widehat{\Gamma}(x) = \widehat{\tau}$$

$$[fn] \quad \frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}_0}{\widehat{\Gamma} \vdash_{\mathsf{CFA}} \mathtt{fn}_\pi\ x \mathtt{=>} e_0 : \widehat{\tau}_x \xrightarrow{\{\pi\}\cup\varphi} \widehat{\tau}_0}$$

$$[fun] \quad \frac{\widehat{\Gamma}[f \mapsto \widehat{\tau}_x \xrightarrow{\{\pi\}\cup\varphi} \widehat{\tau}_0][x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}_0}{\widehat{\Gamma} \vdash_{\mathsf{CFA}} \mathtt{fun}_\pi\ f\ x \mathtt{=>} e_0 : \widehat{\tau}_x \xrightarrow{\{\pi\}\cup\varphi} \widehat{\tau}_0}$$

$$[app] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CFA}} e_1 : \widehat{\tau}_2 \xrightarrow{\varphi} \widehat{\tau}_0 \quad \widehat{\Gamma} \vdash_{\mathsf{CFA}} e_2 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathsf{CFA}} e_1\ e_2 : \widehat{\tau}_0}$$

$$[if] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CFA}} e_0 : \mathtt{bool} \quad \widehat{\Gamma} \vdash_{\mathsf{CFA}} e_1 : \widehat{\tau} \quad \widehat{\Gamma} \vdash_{\mathsf{CFA}} e_2 : \widehat{\tau}}{\widehat{\Gamma} \vdash_{\mathsf{CFA}} \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \widehat{\tau}}$$

$$[let] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CFA}} e_1 : \widehat{\tau}_1 \quad \widehat{\Gamma}[x \mapsto \widehat{\tau}_1] \vdash_{\mathsf{CFA}} e_2 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathsf{CFA}} \mathtt{let}\ x \mathtt{=} e_1\ \mathtt{in}\ e_2 : \widehat{\tau}_2}$$

$$[op] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CFA}} e_1 : \tau^1_{op} \quad \widehat{\Gamma} \vdash_{\mathsf{CFA}} e_2 : \tau^2_{op}}{\widehat{\Gamma} \vdash_{\mathsf{CFA}} e_1\ op\ e_2 : \tau_{op}}$$

Table 5.2: Control Flow Analysis.

Judgements. The judgements of the *Control Flow Analysis* have the form

$$\widehat{\Gamma} \vdash_{\mathsf{CFA}} e : \widehat{\tau}$$

and are defined by the axioms and rules of Table 5.2. The clauses [*fn*] and [*fun*] annotate the arrow of the resulting function type with the information that the abstraction named π should be included in the set of possible functions; the use of $\{\pi\} \cup \varphi$ indicates that we may want to include other names as well and we shall say that the Type and Effect System allows *subeffecting* (see the Concluding Remarks). In Example 5.5 below we shall give an example where subeffecting is indeed needed to analyse the expression. The remaining clauses of Table 5.2 are straightforward modifications of the similar clauses of Table 5.1.

Example 5.4 Let us return to the expression

$$(\mathtt{fn}_X\ \mathtt{x} \mathtt{=>} \mathtt{x})\ (\mathtt{fn}_Y\ \mathtt{y} \mathtt{=>} \mathtt{y})$$

of Example 5.1. Writing $\widehat{\tau}_{\mathsf{Y}}$ for $\texttt{int} \xrightarrow{\{\mathsf{Y}\}} \texttt{int}$ we have the following inference tree:

$$\dfrac{\dfrac{[\texttt{x} \mapsto \widehat{\tau}_{\mathsf{Y}}] \vdash_{\mathsf{CFA}} \texttt{x} : \widehat{\tau}_{\mathsf{Y}}}{[\,] \vdash_{\mathsf{CFA}} \texttt{fn}_{\mathsf{X}}\ \texttt{x => x} : \widehat{\tau}_{\mathsf{Y}} \xrightarrow{\{\mathsf{X}\}} \widehat{\tau}_{\mathsf{Y}}} \qquad \dfrac{[\texttt{y} \mapsto \texttt{int}] \vdash_{\mathsf{CFA}} \texttt{y} : \texttt{int}}{[\,] \vdash_{\mathsf{CFA}} \texttt{fn}_{\mathsf{Y}}\ \texttt{y => y} : \widehat{\tau}_{\mathsf{Y}}}}{[\,] \vdash_{\mathsf{CFA}} (\texttt{fn}_{\mathsf{X}}\ \texttt{x => x})\ (\texttt{fn}_{\mathsf{Y}}\ \texttt{y => y}) : \widehat{\tau}_{\mathsf{Y}}}$$

Note that the whole inference tree is needed to get full information about the control flow properties of the expression. If we label all the subexpressions

$$((\texttt{fn}_{\mathsf{X}}\ \texttt{x => x}^1)^2\ (\texttt{fn}_{\mathsf{Y}}\ \texttt{y => y}^3)^4)^5$$

as in Chapter 3 then we can list the types of the subexpressions as follows

ℓ	1	2	3	4	5
$\widehat{\mathsf{C}}(\ell)$	$\widehat{\tau}_{\mathsf{Y}}$	$\widehat{\tau}_{\mathsf{Y}} \xrightarrow{\{\mathsf{X}\}} \widehat{\tau}_{\mathsf{Y}}$	int	$\widehat{\tau}_{\mathsf{Y}}$	$\widehat{\tau}_{\mathsf{Y}}$

and we are close to the information supplied by $\widehat{\mathsf{C}}$ in Chapter 3. The information corresponding to $\widehat{\rho}$ can be obtained by "merging" information from the various type environments of the inference tree (see Exercise 5.4). ■

Example 5.5 Consider once again the expression loop

```
let g = (funF f x => f (fnY y => y))
in g (fnZ z => z)
```

and let us write $\widehat{\Gamma}_{\mathsf{fx}}$ for $[\texttt{f} \mapsto (\widehat{\tau} \xrightarrow{\{\mathsf{Y},\mathsf{Z}\}} \widehat{\tau}) \xrightarrow{\{\mathsf{F}\}} (\widehat{\tau} \xrightarrow{\emptyset} \widehat{\tau})][\texttt{x} \mapsto \widehat{\tau} \xrightarrow{\{\mathsf{Y},\mathsf{Z}\}} \widehat{\tau}]$. Using the clause [*fn*] we have

$$\widehat{\Gamma}_{\mathsf{fx}} \vdash_{\mathsf{CFA}} \texttt{fn}_{\mathsf{Y}}\ \texttt{y => y} : \widehat{\tau} \xrightarrow{\{\mathsf{Y},\mathsf{Z}\}} \widehat{\tau}$$

where we exploit that subeffecting allows us to enlarge the annotation from the minimal $\{\mathsf{Y}\}$ to $\{\mathsf{Y},\mathsf{Z}\}$. Using this we can construct an inference tree for:

$$[\,] \vdash_{\mathsf{CFA}} \texttt{fun}_{\mathsf{F}}\ \texttt{f x => f}\ (\texttt{fn}_{\mathsf{Y}}\ \texttt{y => y}) : (\widehat{\tau} \xrightarrow{\{\mathsf{Y},\mathsf{Z}\}} \widehat{\tau}) \xrightarrow{\{\mathsf{F}\}} (\widehat{\tau} \xrightarrow{\emptyset} \widehat{\tau})$$

Next let $\widehat{\Gamma}_{\mathsf{g}}$ be $[\texttt{g} \mapsto (\widehat{\tau} \xrightarrow{\{\mathsf{Y},\mathsf{Z}\}} \widehat{\tau}) \xrightarrow{\{\mathsf{F}\}} (\widehat{\tau} \xrightarrow{\emptyset} \widehat{\tau})]$. We now exploit that the annotation $\{\mathsf{Z}\}$ can be enlarged to $\{\mathsf{Z},\mathsf{Y}\}$ in the clause [*fn*] so that:

$$\widehat{\Gamma}_{\mathsf{g}} \vdash_{\mathsf{CFA}} \texttt{fn}_{\mathsf{Z}}\ \texttt{z => z} : \widehat{\tau} \xrightarrow{\{\mathsf{Z},\mathsf{Y}\}} \widehat{\tau}$$

Since $\{\mathsf{Z},\mathsf{Y}\} = \{\mathsf{Y},\mathsf{Z}\}$ we can use the clause [*app*] and get

$$\widehat{\Gamma}_{\mathsf{g}} \vdash_{\mathsf{CFA}} \texttt{g}\ (\texttt{fn}_{\mathsf{Z}}\ \texttt{z => z}) : \widehat{\tau} \xrightarrow{\emptyset} \widehat{\tau}$$

[*unit*]	$\varphi = \varphi \cup \emptyset$	[*idem*]	$\varphi = \varphi \cup \varphi$
[*com*]	$\varphi_1 \cup \varphi_2 = \varphi_2 \cup \varphi_1$	[*ass*]	$\varphi_1 \cup (\varphi_2 \cup \varphi_3) = (\varphi_1 \cup \varphi_2) \cup \varphi_3$
[*ref*]	$\varphi = \varphi$		
[*trans*]	$\dfrac{\varphi_1 = \varphi_2 \quad \varphi_2 = \varphi_3}{\varphi_1 = \varphi_3}$	[*cong*]	$\dfrac{\varphi_1 = \varphi_1' \quad \varphi_2 = \varphi_2'}{\varphi_1 \cup \varphi_2 = \varphi_1' \cup \varphi_2'}$

Table 5.3: Equality of Annotations.

and eventually $[\,] \vdash_{\mathsf{CFA}} \mathsf{loop} : \widehat{\tau} \xrightarrow{\emptyset} \widehat{\tau}$. This can be interpreted as saying that the expression loop does not terminate: the type is $\widehat{\tau} \xrightarrow{\emptyset} \widehat{\tau}$ but the annotation $\emptyset$ indicates that there will be no function abstractions with the given type.

Actually we can show

$$[\,] \vdash_{\mathsf{CFA}} \mathtt{fun}_{\mathsf{F}}\ \mathtt{f\ x} \Rightarrow \mathtt{f\ (fn}_{\mathsf{Y}}\ \mathtt{y} \Rightarrow \mathtt{y)} : (\widehat{\tau} \xrightarrow{\{\mathsf{Y},\mathsf{Z}\}} \widehat{\tau}) \xrightarrow{\{\mathsf{F}\}} (\widehat{\tau} \xrightarrow{\varphi} \widehat{\tau})$$

for every annotation φ and hence we have $[\,] \vdash_{\mathsf{CFA}} \mathsf{loop} : \widehat{\tau} \xrightarrow{\varphi} \widehat{\tau}$ for every annotation φ; clearly the judgement with $\varphi = \emptyset$ is the most informative. ∎

Equivalence of annotations. There are a few subtleties involved in using this simple-minded system for Control Flow Analysis. One is the implicit decision to regard a type like $\widehat{\tau} \xrightarrow{\{\mathsf{Y},\mathsf{Z}\}} \widehat{\tau}$ as being equal to $\widehat{\tau} \xrightarrow{\{\mathsf{Z},\mathsf{Y}\}} \widehat{\tau}$.

Concerning this subtlety, we already explained that we feel free to write $\{\pi_1, \cdots, \pi_n\}$ for $\{\pi_1\} \cup \cdots \cup \{\pi_n\}$. To be utterly formal we should really say that we write $\{\pi_1, \ldots, \pi_n\}$ for $((\emptyset \cup \{\pi_1\}) \cup \cdots) \cup \{\pi_n\}$.

Next we allow to replace $\tau_1 \xrightarrow{\varphi_1} \tau_2$ by $\tau_1 \xrightarrow{\varphi_2} \tau_2$ whenever φ_1 and φ_2 are "equal as sets". To be utterly formal this can be axiomatised by the axioms and rules of Table 5.3: the axioms [*unit*], [*idem*], [*com*] and [*ass*] express that set union has a unit and is idempotent, commutative and associative, and the axioms and rules [*trans*], [*ref*] and [*cong*] ensure that equality is an equivalence relation as well as a congruence.

Finally, we allow to replace $\widehat{\tau}_1$ by $\widehat{\tau}_2$ if they have the same underlying types and all annotations on corresponding function arrows are "equal as sets". To be utterly formal we could axiomatise this by:

$$\widehat{\tau} = \widehat{\tau} \qquad\qquad \frac{\widehat{\tau}_1 = \widehat{\tau}_1' \quad \widehat{\tau}_2 = \widehat{\tau}_2' \quad \varphi = \varphi'}{(\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2) = (\widehat{\tau}_1' \xrightarrow{\varphi'} \widehat{\tau}_2')}$$

It is customary to be informal about these fine technical points. But to avoid confusion one should at the very least point out that annotations are

considered equal modulo the existence of a unit, commutativity, associativity, and idempotence; the abbreviation *UCAI* is often used for this.

Conservative extension. Another subtlety is the ability to give the abstraction $\mathtt{fn}_{\mathsf{Y}}$ `y => y` the type $\widehat{\tau} \xrightarrow{\{\mathsf{Y},\mathsf{Z}\}} \widehat{\tau}$. Suppose for a moment that the two rules for function abstraction did *not* have a $\{\pi\} \cup \varphi$ annotation on the function arrows but only a $\{\pi\}$. Then $\mathtt{fn}_{\mathsf{Y}}$ `y => y` would have type $\widehat{\tau} \xrightarrow{\{\mathsf{Y}\}} \widehat{\tau}$ but not $\widehat{\tau} \xrightarrow{\{\mathsf{Y},\mathsf{Z}\}} \widehat{\tau}$; consequently the program from Example 5.5 would have no type in the Annotated Type System for Control Flow Analysis! This is a very undesirable property: we ought to be able to analyse all programs.

To ensure that our system does not have the above deficiency we shall formally prove that the Control Flow Analysis of Table 5.2 is a *conservative extension* of the underlying type system of Table 5.1. This is expressed by:

Fact 5.6

(i) If $\widehat{\Gamma} \vdash_{\mathsf{CFA}} e : \widehat{\tau}$ then $\lfloor \widehat{\Gamma} \rfloor \vdash_{\mathsf{UL}} e : \lfloor \widehat{\tau} \rfloor$.

(ii) If $\Gamma \vdash_{\mathsf{UL}} e : \tau$ then there exists $\widehat{\Gamma}$ and $\widehat{\tau}$ such that $\widehat{\Gamma} \vdash_{\mathsf{CFA}} e : \widehat{\tau}$, $\lfloor \widehat{\Gamma} \rfloor = \Gamma$ and $\lfloor \widehat{\tau} \rfloor = \tau$. ∎

Proof The proof of (i) is straightforward. For the proof of (ii) one annotates all arrows with the set of all program points in e. ∎

This result paves the way for extending $\lfloor \cdot \rfloor$ to operate on entire typings: applied to the typing $\widehat{\Gamma} \vdash_{\mathsf{CFA}} e : \widehat{\tau}$ it produces a typing of the form $\lfloor \widehat{\Gamma} \rfloor \vdash_{\mathsf{UL}} e : \lfloor \widehat{\tau} \rfloor$. As an example, the typing of Example 5.5 is transformed into the typing of Example 5.3 (assuming that $\lfloor \widehat{\tau} \rfloor = \tau$).

In Section 5.4 we shall study explicit inference rules for *subeffecting*: this is a related technique for ensuring that the analysis is a conservative extension of the underlying type system.

5.2 Theoretical Properties

Having specified the analysis we shall now ensure that it is semantically correct. Furthermore, the fact that the analysis is a conservative extension of the underlying type system motivates the following result: whenever we have a typing in the underlying type system then the set of typings of the Control Flow Analysis constitutes a Moore family. So as in Section 3.2 (and Exercise 2.7) every acceptable expression can be analysed and it has a best analysis.

As in Sections 2.2 and 3.2, the material of this section may be skimmed through on a first reading; however, we re-iterate that it is frequently when

$$[con] \quad \vdash c \longrightarrow c$$

$$[fn] \quad \vdash (\mathtt{fn}_\pi\ x \mathtt{=>} e_0) \longrightarrow (\mathtt{fn}_\pi\ x \mathtt{=>} e_0)$$

$$[fun] \quad \vdash (\mathtt{fun}_\pi\ f\ x \mathtt{=>} e_0) \longrightarrow \mathtt{fn}_\pi\ x \mathtt{=>} (e_0[f \mapsto \mathtt{fun}_\pi\ f\ x \mathtt{=>} e_0])$$

$$[app] \quad \frac{\vdash e_1 \longrightarrow (\mathtt{fn}_\pi\ x \mathtt{=>} e_0) \quad \vdash e_2 \longrightarrow v_2 \quad \vdash e_0[x \mapsto v_2] \longrightarrow v_0}{\vdash e_1\ e_2 \longrightarrow v_0}$$

$$[if_1] \quad \frac{\vdash e_0 \longrightarrow \mathtt{true} \quad \vdash e_1 \longrightarrow v_1}{\vdash \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \longrightarrow v_1}$$

$$[if_2] \quad \frac{\vdash e_0 \longrightarrow \mathtt{false} \quad \vdash e_2 \longrightarrow v_2}{\vdash \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \longrightarrow v_2}$$

$$[let] \quad \frac{\vdash e_1 \longrightarrow v_1 \quad \vdash e_2[x \mapsto v_1] \longrightarrow v_2}{\vdash \mathtt{let}\ x \mathtt{=} e_1\ \mathtt{in}\ e_2 \longrightarrow v_2}$$

$$[op] \quad \frac{\vdash e_1 \longrightarrow v_1 \quad \vdash e_2 \longrightarrow v_2}{\vdash e_1\ op\ e_2 \longrightarrow v} \quad \text{if } v_1\ \mathbf{op}\ v_2 = v$$

Table 5.4: Natural Semantics for FUN.

conducting the correctness proof that the final and subtle errors in the analysis are found and corrected!

5.2.1 Natural Semantics

To prove the semantic correctness of the analysis we need to define the semantics. Many kinds of semantics would be appropriate but among the operational semantics we shall now prefer a *Natural Semantics* (i.e. big-step operational semantics) without environments because this makes semantic correctness somewhat easier to establish. This is related to the discussion in the Concluding Remarks of Chapter 3 about the difference between using a big-step operational semantics without environments and a small-step operational semantics with environments; thus our correctness statement will be somewhat weak in case of looping programs.

Transitions. The Natural Semantics will have transitions of the form

$$\vdash e \longrightarrow v$$

meaning that the expression e evaluates to the value v. We shall *assume* that $e \in \mathbf{Exp}$ is a *closed* expression, i.e. $FV(e) = \emptyset$, meaning that e does not have

any free variables. The *values*

$$v \in \mathbf{Val} \quad \text{values}$$

will be a subset of the expressions given by the syntax

$$v ::= c \mid \texttt{fn}_\pi\ x\ \texttt{=>}\ e_0 \qquad \text{provided that } FV(\texttt{fn}_\pi\ x\ \texttt{=>}\ e_0) = \emptyset$$

where we demand that values are closed expressions. Compared to the Structural Operational Semantics of Section 3.2, it is not necessary to introduce environments since the bound variables will be syntactically replaced by their value as soon as they become free. Hence there is neither need for a `close`-construct nor for a `bind`-construct.

As usual we shall write $e_1[x \mapsto e_2]$ for substituting e_2 for all free occurrences of x in e_1. It will be the case that e_2 is closed whenever we use this notation and therefore there is no risk of variable capture and hence no need to rename bound variables. Throughout we shall *assume* that $\texttt{fun}_\pi\ f\ x\ \texttt{=>}\ e_0$ uses distinct variables for f and x. The semantics is given by Table 5.4 and is explained below.

The axioms [*con*] and [*fn*] express that the constant and the function abstraction, respectively, evaluate to themselves. For recursive function abstractions we shall unfold the recursion one level as expressed by the axiom [*fun*]; note that the function abstraction being created inherits the program point of the recursive function abstraction. The rule [*app*] for application expresses that first we evaluate the operator, then the operand, and next we substitute the actual parameter for the formal parameter in the body of the abstraction and evaluate the body. We have only one rule for application because the axiom [*fun*] ensures that all function abstractions will be of the form $\texttt{fn}_\pi\ x\ \texttt{=>}\ e_0$. The rules for the conditional, the `let`-construct and the binary operators should be straightforward.

Example 5.7 Consider the expression $(\texttt{fn}_X\ \texttt{x => x})\ (\texttt{fn}_Y\ \texttt{y => y})$ of Example 5.1. Using the axiom [*fn*] we have

$$\vdash \texttt{fn}_X\ \texttt{x => x} \longrightarrow \texttt{fn}_X\ \texttt{x => x}$$
$$\vdash \texttt{fn}_Y\ \texttt{y => y} \longrightarrow \texttt{fn}_Y\ \texttt{y => y}$$
$$\vdash \texttt{x}[\texttt{x} \mapsto \texttt{fn}_Y\ \texttt{y => y}] \longrightarrow \texttt{fn}_Y\ \texttt{y => y}$$

and we can apply the rule [*app*] to get:

$$\vdash (\texttt{fn}_X\ \texttt{x => x})\ (\texttt{fn}_Y\ \texttt{y => y}) \longrightarrow \texttt{fn}_Y\ \texttt{y => y}$$

In Example 3.7 we showed how the Structural Operational Semantics deals with this expression. ∎

Example 5.8 Next consider the expression loop

```
let g = (funF f x => f (fnY y => y))
in g (fnZ z => z)
```

and let us see how this looping program is modelled in the Natural Semantics. First we observe that the axiom [*fun*] gives

$$\vdash \texttt{fun}_{\mathsf{F}}\ \texttt{f x => f (fn}_{\mathsf{Y}}\ \texttt{y => y)} \longrightarrow$$
$$\texttt{fn}_{\mathsf{F}}\ \texttt{x => ((fun}_{\mathsf{F}}\ \texttt{f x => f (fn}_{\mathsf{Y}}\ \texttt{y => y)) (fn}_{\mathsf{Y}}\ \texttt{y => y))}$$

so we have replaced the recursive call of `f` with the recursive function definition itself. Turning to the body of the `let`-construct we have to replace the occurrence of `g` with the abstraction:

$$\texttt{fn}_{\mathsf{F}}\ \texttt{x => ((fun}_{\mathsf{F}}\ \texttt{f x => f (fn}_{\mathsf{Y}}\ \texttt{y => y)) (fn}_{\mathsf{Y}}\ \texttt{y => y))}$$

The operator will now evaluate to this value and the operand $\texttt{fn}_{\mathsf{Z}}$ `z => z` will evaluate to itself. So the next step is to determine a value v such that we have an inference tree for

$$\vdash \texttt{(fun}_{\mathsf{F}}\ \texttt{f x => f (fn}_{\mathsf{Y}}\ \texttt{y => y)) (fn}_{\mathsf{Y}}\ \texttt{y => y)} \longrightarrow v \tag{5.1}$$

and after that we are in a position to use the rule for application. The evaluation of the operator in (5.1) proceeds as before and so does the evaluation of the operand and once more we are left with the obligation to construct an inference tree for the judgement (5.1). Thus we have encountered a *circularity* and we see that the *looping* of the program is modelled in the semantics by the *absence* of an inference tree. In Example 3.8 we showed how the Structural Operational Semantics deals with the expression loop. ∎

It is immediate to verify that if e is a closed expression and $\vdash e \longrightarrow v$ then all $\vdash e' \longrightarrow v'$ occurring in the corresponding inference tree (in particular $\vdash e \longrightarrow v$ itself) will have both e' and v' to be closed.

5.2.2 Semantic Correctness

To be able to express the semantic correctness of the analysis we need to assume that the types of the binary operators *op* and their semantics are suitably related. Recall that for the underlying type system we assume that *op* takes two arguments with the base types τ^1_{op} and τ^2_{op} and gives a result of type τ_{op} and since base types do not have any annotations we shall now *assume* that:

> If $[\,] \vdash_{\mathsf{CFA}} v_1 : \tau^1_{op}$ and $[\,] \vdash_{\mathsf{CFA}} v_2 : \tau^2_{op}$ then $[\,] \vdash_{\mathsf{CFA}} v : \tau_{op}$
> where $v = v_1\ \mathbf{op}\ v_2$.

This ensures that when given arguments of appropriate types the operator will return a value of the expected type.

The *semantic correctness* of the analysis now expresses that the annotated type foreseen by the analysis will also be the annotated type of the value obtained by evaluating the expression; this is expressed by the following subject reduction result:

Theorem 5.9
If $[\,] \vdash_{\mathsf{CFA}} e : \widehat{\tau}$, and $\vdash e \longrightarrow v$ then $[\,] \vdash_{\mathsf{CFA}} v : \widehat{\tau}$.

It follows that if $[\,] \vdash e : \widehat{\tau}_1 \xrightarrow{\varphi_0} \widehat{\tau}_2$ and $\vdash e \longrightarrow \mathbf{fn}_\pi\ x \texttt{=>} e_0$ then $\pi \in \varphi_0$; hence the analysis correctly tracks the closures that can result from a given expression. Also note that if $[\,] \vdash e : \widehat{\tau}_1 \xrightarrow{\emptyset} \widehat{\tau}_2$ then e cannot terminate.

In preparation for the proof of the theorem we need a few technical results of the sort that are standard for type systems. The first result expresses that the type environment may be extended with information that does not influence the analysis result:

Fact 5.10 If $\widehat{\Gamma}_1 \vdash_{\mathsf{CFA}} e : \widehat{\tau}$ and $\widehat{\Gamma}_1(x) = \widehat{\Gamma}_2(x)$ for all $x \in FV(e)$ then $\widehat{\Gamma}_2 \vdash_{\mathsf{CFA}} e : \widehat{\tau}$. ■

Proof The proof is by induction on the inference tree for $\widehat{\Gamma}_1 \vdash_{\mathsf{CFA}} e : \widehat{\tau}$ using that it is the rightmost occurrence of a variable in a type environment that determines its type. ■

The next result shows that we can safely substitute an expression of the correct annotated type for a variable:

Lemma 5.11 Assume $[\,] \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}_0$ and $\widehat{\Gamma}[x \mapsto \widehat{\tau}_0] \vdash_{\mathsf{CFA}} e : \widehat{\tau}$. Then $\widehat{\Gamma} \vdash_{\mathsf{CFA}} e[x \mapsto e_0] : \widehat{\tau}$. ■

Proof The proof is by structural induction on e. Most cases are straightforward so we shall only present the cases of variables and function abstractions.

The case y. We assume that

$$\widehat{\Gamma}[x \mapsto \widehat{\tau}_0] \vdash_{\mathsf{CFA}} y : \widehat{\tau}$$

so from the axiom [*var*] of Table 5.2 we have $(\widehat{\Gamma}[x \mapsto \widehat{\tau}_0])(y) = \widehat{\tau}$. If $x = y$ then $y[x \mapsto e_0] = e_0$ and $\widehat{\tau} = \widehat{\tau}_0$, and from $[\,] \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}_0$ and Fact 5.10 we get the required $\widehat{\Gamma} \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}$. If $x \neq y$ then $y[x \mapsto e_0] = y$ and it is easy to see that $\widehat{\Gamma} \vdash_{\mathsf{CFA}} y : \widehat{\tau}$.

The case $\mathbf{fn}_\pi\ y \texttt{=>} e$. We assume that

$$\widehat{\Gamma}[x \mapsto \widehat{\tau}_0] \vdash_{\mathsf{CFA}} \mathbf{fn}_\pi\ y \texttt{=>} e : \widehat{\tau}$$

so, according to rule [*fn*] of Table 5.2, it must be the case that $\widehat{\tau} = \widehat{\tau}_y \xrightarrow{\{\pi\}\cup\varphi} \widehat{\tau}'$ and:

$$\widehat{\Gamma}[x \mapsto \widehat{\tau}_0][y \mapsto \widehat{\tau}_y] \vdash_{\mathsf{CFA}} e : \widehat{\tau}'$$

If $x = y$ then it follows from Fact 5.10 that $\widehat{\Gamma}[y \mapsto \widehat{\tau}_y] \vdash_{\mathsf{CFA}} e : \widehat{\tau}'$ and since $(\mathbf{fn}_\pi\ y \texttt{ => } e)[x \mapsto e_0] = \mathbf{fn}_\pi\ y \texttt{ => } e$ the result follows. So assume that $x \neq y$; then $(\mathbf{fn}_\pi\ y \texttt{ => } e)[x \mapsto e_0] = \mathbf{fn}_\pi\ y \texttt{ => } (e[x \mapsto e_0])$. It follows from Fact 5.10 that $\widehat{\Gamma}[y \mapsto \widehat{\tau}_y][x \mapsto \widehat{\tau}_0] \vdash_{\mathsf{CFA}} e : \widehat{\tau}'$ and then we get $\widehat{\Gamma}[y \mapsto \widehat{\tau}_y] \vdash_{\mathsf{CFA}} e[x \mapsto e_0] : \widehat{\tau}'$ from the induction hypothesis. Now the result follows using the rule [*fn*]. ∎

We now turn to the proof of Theorem 5.9:

Proof The proof proceeds by induction on the inference tree for $\vdash e \longrightarrow v$.

The cases [*con*] and [*fn*] are immediate.

The case [*fun*]. We assume that

$$\vdash \mathbf{fun}_\pi\ f\ x \texttt{ => } e_0 \longrightarrow \mathbf{fn}_\pi\ x \texttt{ => } e_0[f \mapsto \mathbf{fun}_\pi\ f\ x \texttt{ => } e_0]$$

where f and x are distinct variables. Also we have

$$[\,] \vdash_{\mathsf{CFA}} \mathbf{fun}_\pi\ f\ x \texttt{ => } e_0 : \widehat{\tau}_x \xrightarrow{\{\pi\}\cup\varphi_0} \widehat{\tau}_0$$

and according to Table 5.2 this is because $[f \mapsto \widehat{\tau}_x \xrightarrow{\{\pi\}\cup\varphi_0} \widehat{\tau}_0][x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}_0$. Since $[f \mapsto \widehat{\tau}_x \xrightarrow{\{\pi\}\cup\varphi_0} \widehat{\tau}_0][x \mapsto \widehat{\tau}_x]$ equals $[x \mapsto \widehat{\tau}_x][f \mapsto \widehat{\tau}_x \xrightarrow{\{\pi\}\cup\varphi} \widehat{\tau}_0]$ (because we assumed that f and x are distinct) it follows from Lemma 5.11 that

$$[x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{CFA}} e_0[f \mapsto \mathbf{fun}_\pi\ f\ x \texttt{ => } e_0] : \widehat{\tau}_0$$

and hence $[\,] \vdash_{\mathsf{CFA}} \mathbf{fn}_\pi\ x \texttt{ => } e_0[f \mapsto \mathbf{fun}_\pi\ f\ x \texttt{ => } e_0] : \widehat{\tau}_x \xrightarrow{\{\pi\}\cup\varphi_0} \widehat{\tau}_0$ which is the desired result.

The case [*app*]. We assume that

$$\vdash e_1\ e_2 \longrightarrow v_0$$

because $\vdash e_1 \longrightarrow \mathbf{fn}_\pi\ x \texttt{ => } e_0$, $\vdash e_2 \longrightarrow v_2$ and $\vdash e_0[x \mapsto v_2] \longrightarrow v_0$. Also we have

$$[\,] \vdash_{\mathsf{CFA}} e_1\ e_2 : \widehat{\tau}_0$$

and according to Table 5.2 this is because $[\,] \vdash_{\mathsf{CFA}} e_1 : \widehat{\tau}_2 \xrightarrow{\varphi} \widehat{\tau}_0$ and $[\,] \vdash_{\mathsf{CFA}} e_2 : \widehat{\tau}_2$. The induction hypothesis applied to the inference tree for e_1 gives:

$$[\,] \vdash_{\mathsf{CFA}} \mathbf{fn}_\pi\ x \texttt{ => } e_0 : \widehat{\tau}_2 \xrightarrow{\varphi} \widehat{\tau}_0$$

According to Table 5.2 this can only be the case if $\pi \in \varphi$ and $[x \mapsto \widehat{\tau}_2] \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}_0$. The induction hypothesis applied to the inference tree for e_2 gives

$$[\,] \vdash_{\mathsf{CFA}} v_2 : \widehat{\tau}_2$$

and by Lemma 5.11 we now get $[\,] \vdash_{\mathsf{CFA}} e_0[x \mapsto v_2] : \widehat{\tau}_0$. The induction hypothesis applied to the inference tree for $e_0[x \mapsto v_2]$ now gives

$$[\,] \vdash_{\mathsf{CFA}} v_0 : \widehat{\tau}_0$$

and this is the desired result.

The case [if_1]. We assume that

$$\vdash \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow v_1$$

because $\vdash e_0 \longrightarrow \texttt{true}$ and $\vdash e_1 \longrightarrow v_1$. Also we have

$$[\,] \vdash_{\mathsf{CFA}} \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \widehat{\tau}$$

and from Table 5.2 we see that this is because $[\,] \vdash_{\mathsf{CFA}} e_0 : \texttt{bool}$, $[\,] \vdash_{\mathsf{CFA}} e_1 : \widehat{\tau}$ and $[\,] \vdash_{\mathsf{CFA}} e_2 : \widehat{\tau}$. The induction hypothesis gives

$$[\,] \vdash_{\mathsf{CFA}} v_1 : \widehat{\tau}$$

and this is the desired result.

The case [if_2] is analogous.

The case [*let*]. We assume that

$$\vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow v_2$$

because $\vdash e_1 \longrightarrow v_1$ and $\vdash e_2[x \mapsto v_1] \longrightarrow v_2$. Also we have

$$[\,] \vdash_{\mathsf{CFA}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \widehat{\tau}_2$$

and this is because $[\,] \vdash_{\mathsf{CFA}} e_1 : \widehat{\tau}_1$ and $[x \mapsto \widehat{\tau}_1] \vdash_{\mathsf{CFA}} e_2 : \widehat{\tau}_2$. The induction hypothesis now gives:

$$[\,] \vdash_{\mathsf{CFA}} v_1 : \widehat{\tau}_1$$

From Lemma 5.11 we then get $[\,] \vdash_{\mathsf{CFA}} e_2[x \mapsto v_1] : \widehat{\tau}_2$ and the induction hypothesis gives

$$[\,] \vdash_{\mathsf{CFA}} v_2 : \widehat{\tau}_2$$

as required.

The case [*op*]. We assume that

$$\vdash e_1\ op\ e_2 \longrightarrow v$$

because $\vdash e_1 \longrightarrow v_1$, $\vdash e_2 \longrightarrow v_2$ and $v_1\ \mathbf{op}\ v_2 = v$. Also we assume

$$[\,] \vdash_{\mathsf{CFA}} e_1\ op\ e_2 : \tau_{op}$$

and this can only be because $[\,] \vdash_{\mathsf{CFA}} e_1 : \tau^1_{op}$ and $[\,] \vdash_{\mathsf{CFA}} e_2 : \tau^2_{op}$. The induction hypothesis gives $[\,] \vdash_{\mathsf{CFA}} v_1 : \tau^1_{op}$ and $[\,] \vdash_{\mathsf{CFA}} v_2 : \tau^2_{op}$ and the desired result that $[\,] \vdash_{\mathsf{CFA}} v : \tau_{op}$ then follows from the stated assumptions about the relationship between the semantics and the types of the binary operators. ∎

5.2.3 Existence of Solutions

In Chapter 3 we showed that the set of solutions to the Control Flow Analysis constituted a Moore family: the greatest lower bound of a set of solutions is also a solution. (Also see Exercise 2.7.) From that result it then follows that all programs can be analysed and that there is a best analysis.

Complete lattice of annotations. A similar development is possible here except that special care has to be taken concerning the nature of annotations. We would like to be able to regard the set **Ann** of annotations as a complete lattice, and this necessitates a partial ordering

$$\varphi_1 \sqsubseteq \varphi_2 \quad \text{or} \quad \varphi_1 \subseteq \varphi_2$$

that intuitively means that the set φ_1 of program points is included in the set φ_2 of program points. One way to formalise this is to say

$$\exists \varphi' : \varphi_1 \cup \varphi' = \varphi_2$$

where we rely on the axiomatisation of "equal as sets" presented in Table 5.3. Another way is by means of an explicit system of rules and axioms for inferring judgements of the form $\varphi_1 \subseteq \varphi_2$; we shall dispense with these details (but see Exercise 5.3).

One way to ensure that $(\mathbf{Ann}, \sqsubseteq)$ is a complete lattice is to demand that all annotations are subsets of a given finite set; for this one might replace **Pnt** by the finite set $\mathbf{Pnt}_\star$ of program points occurring in the expression of interest. Another possibility will be to change the syntax of annotations to allow expressing an arbitrary subset of **Pnt**. Either approach works, since all that suffices for the subsequent development is to *assume* that:

$$(\mathbf{Ann}, \sqsubseteq) \text{ is a complete lattice isomorphic to } (\mathcal{P}(\mathbf{Pnt}), \subseteq).$$

The partial ordering on **Ann** will sometimes be written as $\sqsubseteq$ and sometimes as $\subseteq$.

Complete lattice of annotated types. We can now extend the partial ordering on annotations to operate on annotated types that have the same underlying type. For this let $\tau \in \mathbf{Type}$ be a type and write

$$\widehat{\mathbf{Type}}[\tau]$$

for the set of annotated types $\widehat{\tau}$ with underlying type τ, i.e. such that $\lfloor \widehat{\tau} \rfloor = \tau$. Next define $\widehat{\tau}_1 \sqsubseteq \widehat{\tau}_2$ for $\widehat{\tau}_1, \widehat{\tau}_2 \in \widehat{\mathbf{Type}}[\tau]$ to mean that whenever $\widehat{\tau}_i$ has the annotation φ_i in a given position then $\varphi_1 \subseteq \varphi_2$. Formally this is achieved by:

$$\widehat{\tau} \sqsubseteq \widehat{\tau} \qquad\qquad \frac{\widehat{\tau}_1 \sqsubseteq \widehat{\tau}_1' \quad \varphi \subseteq \varphi' \quad \widehat{\tau}_2 \sqsubseteq \widehat{\tau}_2'}{\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \sqsubseteq \widehat{\tau}_1' \xrightarrow{\varphi'} \widehat{\tau}_2'}$$

As an example, $(\texttt{int} \xrightarrow{\varphi_1} \texttt{int}) \xrightarrow{\varphi_2} \texttt{int} \sqsubseteq (\texttt{int} \xrightarrow{\varphi_3} \texttt{int}) \xrightarrow{\varphi_4} \texttt{int}$ will be the case if and only if $\varphi_1 \subseteq \varphi_3$ and $\varphi_2 \subseteq \varphi_4$. (Note that this ordering is *not* contravariant unlike what will be the case for the subtyping to be introduced in Section 5.4.) Clearly the least element in $\widehat{\mathbf{Type}}[\tau]$ will have $\emptyset$ on all function arrows occurring in τ and the greatest element will have **Pnt** on all function arrows.

It is straightforward to prove that each $(\widehat{\mathbf{Type}}[\tau], \sqsubseteq)$ is a complete lattice. In a similar way the partial ordering can be extended to operate also on type environments having the same underlying structure. Finally, suppose that $\Gamma \vdash_{\mathsf{UL}} e : \tau$ and write

$$\mathsf{JUDG}_{\mathsf{CFA}}[\Gamma \vdash_{\mathsf{UL}} e : \tau]$$

for the set of typings $\widehat{\Gamma} \vdash_{\mathsf{CFA}} e : \widehat{\tau}$ such that $\lfloor \cdot \rfloor$ maps $\widehat{\Gamma} \vdash_{\mathsf{CFA}} e : \widehat{\tau}$ to $\Gamma \vdash_{\mathsf{UL}} e : \tau$ (and hence $\lfloor \widehat{\Gamma} \rfloor = \Gamma$ and $\lfloor \widehat{\tau} \rfloor = \tau$). This facilitates extending the partial ordering $\sqsubseteq$ to operate on the set $\mathsf{JUDG}_{\mathsf{CFA}}[\Gamma \vdash_{\mathsf{UL}} e : \tau]$ of typings.

Moore family result. We are now ready to state the result about Moore families:

Proposition 5.12
$\mathsf{JUDG}_{\mathsf{CFA}}[\Gamma \vdash_{\mathsf{UL}} e : \tau]$ is a Moore family whenever $\Gamma \vdash_{\mathsf{UL}} e : \tau$.

Proof We assume that $\Gamma \vdash_{\mathsf{UL}} e : \tau$ and prove that $\mathsf{JUDG}_{\mathsf{CFA}}[\Gamma \vdash_{\mathsf{UL}} e : \tau]$ is a Moore family. For this we proceed by induction on the shape of the inference tree for $\Gamma \vdash_{\mathsf{UL}} e : \tau$. We shall only give the proof for the cases [*var*], [*fn*] and [*app*]; the other cases follow the same overall pattern. In all cases let

$$Y = \{(\widehat{\Gamma}^i \vdash_{\mathsf{CFA}} e : \widehat{\tau}^i) \mid i \in I\}$$

be a subset of $\mathsf{JUDG}_{\mathsf{CFA}}[\Gamma \vdash_{\mathsf{UL}} e : \tau]$. Using that $(\widehat{\mathbf{Type}}[\tau'], \sqsubseteq)$ is a complete lattice for all choices of $\tau' \in \mathbf{Type}$ we get that $\bigsqcap Y$ exists and that it is defined in a pointwise manner. (Note that if $I = \emptyset$ then $\bigsqcap Y$ is obtained from $\Gamma \vdash_{\mathsf{UL}} e : \tau$ by placing $\bigsqcap \emptyset = \mathbf{Pnt}$ as annotation on all function arrows.) It remains to show that $\bigsqcap Y \in \mathsf{JUDG}_{\mathsf{CFA}}[\Gamma \vdash_{\mathsf{UL}} e : \tau]$.

The case [*var*]. The result follows from $(\bigsqcap_i \widehat{\Gamma}^i)(x) = \bigsqcap_i (\widehat{\Gamma}^i(x))$.

The case [*fn*]. We have $\Gamma \vdash_{\mathsf{UL}} \mathbf{fn}_\pi\ x \Rightarrow e_0 : \tau_x \to \tau_0$ because $\Gamma[x \mapsto \tau_x] \vdash_{\mathsf{UL}} e_0 : \tau_0$. For $i \in I$ we have $\widehat{\Gamma}^i \vdash_{\mathsf{CFA}} \mathbf{fn}_\pi\ x \Rightarrow e_0 : \widehat{\tau}^i_x \xrightarrow{\{\pi\} \cup \varphi^i} \widehat{\tau}^i_0$ because of

$$\widehat{\Gamma}^i[x \mapsto \widehat{\tau}^i_x] \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}^i_0$$

and clearly $\widehat{\Gamma}^i[x \mapsto \widehat{\tau}^i_x] \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}^i_0$ is an element of $\mathsf{JUDG}_{\mathsf{CFA}}[\Gamma[x \mapsto \tau_x] \vdash_{\mathsf{UL}} e_0 : \tau_0]$. By the induction hypothesis we get $(\bigsqcap_i \widehat{\Gamma}^i)[x \mapsto \bigsqcap_i \widehat{\tau}^i_x] \vdash_{\mathsf{CFA}} e_0 : \bigsqcap_i \widehat{\tau}^i_0$ and hence

$$\bigsqcap_i \widehat{\Gamma}^i \vdash_{\mathsf{CFA}} \mathbf{fn}_\pi\ x \Rightarrow e_0 : \bigsqcap_i \widehat{\tau}^i_x \xrightarrow{\{\pi\} \cup \varphi} \bigsqcap_i \widehat{\tau}^i_0$$

where $\varphi = \bigcap_i \varphi^i$.

The case [*app*]. We now have $\Gamma \vdash_{\mathsf{UL}} e_1\ e_2 : \tau_0$ because $\Gamma \vdash_{\mathsf{UL}} e_1 : \tau_2 \to \tau_0$ and $\Gamma \vdash_{\mathsf{UL}} e_2 : \tau_2$. For all $i \in I$ we have $\widehat{\Gamma}^i \vdash_{\mathsf{CFA}} e_1\ e_2 : \widehat{\tau}^i_0$ because

$$\widehat{\Gamma}^i \vdash_{\mathsf{CFA}} e_1 : \widehat{\tau}^i_2 \xrightarrow{\varphi^i} \widehat{\tau}^i_0 \quad \text{and} \quad \widehat{\Gamma}^i \vdash_{\mathsf{CFA}} e_2 : \widehat{\tau}^i_2$$

and clearly $\widehat{\Gamma}^i \vdash_{\mathsf{CFA}} e_1 : \widehat{\tau}_2^i \xrightarrow{\varphi^i} \widehat{\tau}_0^i$ is an element of $\mathsf{JUDG}_{\mathsf{CFA}}[\Gamma \vdash_{\mathsf{UL}} e_1 : \tau_2 \to \tau_0]$ and $\widehat{\Gamma}^i \vdash_{\mathsf{CFA}} e_2 : \widehat{\tau}_2^i$ is an element of $\mathsf{JUDG}_{\mathsf{CFA}}[\Gamma \vdash_{\mathsf{UL}} e_2 : \tau_2]$. By the induction hypothesis we get $\bigsqcap_i \widehat{\Gamma}^i \vdash e_1 : \bigsqcap_i \widehat{\tau}_2^i \xrightarrow{\varphi} \bigsqcap_i \widehat{\tau}_0^i$ and $\bigsqcap_i \widehat{\Gamma}^i \vdash e_2 : \bigsqcap_i \widehat{\tau}_2^i$ where $\varphi = \bigcap_i \varphi^i$ and hence

$$\bigsqcap_i \widehat{\Gamma}^i \vdash e_1\ e_2 : \bigsqcap_i \widehat{\tau}_0^i$$

which is the desired result. ∎

Example 5.13 Consider the expression e:

```
f (fnX x => x+1) + f (fnY y => y+2) + (fnZ z => z+3)(4)
```

In the underlying type system we have:

$$[f \mapsto (\mathtt{int} \to \mathtt{int}) \to \mathtt{int}] \vdash \mathtt{e} : \mathtt{int}$$

In the Control Flow Analysis we have

$$[\mathtt{f} \mapsto (\mathtt{int} \xrightarrow{\varphi_1} \mathtt{int}) \xrightarrow{\varphi_2} \mathtt{int}] \vdash \mathtt{e} : \mathtt{int}$$

whenever $\{\mathsf{X}, \mathsf{Y}\} \subseteq \varphi_1$. The least solution therefore has $\varphi_1 = \{\mathsf{X}, \mathsf{Y}\}$ and $\varphi_2 = \emptyset$. This clearly tells us that `f` is only applied to `(fnX x => x+1)` and `(fnY y => y+2)` and not to `(fnZ z => z+3)`. A larger solution like $\varphi_1 = \{\mathsf{X}, \mathsf{Y}, \mathsf{Z}\}$ and $\varphi_2 = \{\mathsf{V}\}$ would not convey this information. Hence it seems sensible to ask for the least solution with respect to $\sqsubseteq$ and the existence of this solution is guaranteed by Proposition 5.12. ∎

5.3 Inference Algorithms

The main difference between an analysis expressed in the form of an inference system (as in Table 5.2) and in the form of an algorithm is that the user of the inference system is expected to have sufficient foresight to be able to *guess* the right types and annotations whereas the implementation of the algorithm will make use of a mechanism for making *tentative* guesses that are later *refined*. Let us first consider the simple case corresponding to the underlying type system of Table 5.1.

5.3.1 An Algorithm for the Underlying Type System

Augmented types. The algorithm corresponding to the type system of Table 5.1 will work on *augmented types* that allow the use of type variables to reflect that the details of a type are not fully determined yet:

τ	$\in$	**AType**	augmented types
α	$\in$	**TVar**	type variables

We shall take:

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2 \mid \alpha \\ \alpha & ::= & \texttt{'a} \mid \texttt{'b} \mid \texttt{'c} \mid \cdots \end{array}$$

Substitutions. A *substitution* is a finite and partial mapping from type variables to augmented types, we write

$$\theta : \mathbf{TVar} \to_{\text{fin}} \mathbf{AType}$$

and note that the domain $dom(\theta) = \{\alpha \mid \theta \text{ is defined on } \alpha\}$ is finite. We shall allow to view a substitution as a total function from type variables to augmented types, setting $\theta\ \alpha = \alpha$ whenever $\alpha \notin dom(\theta)$. We shall say that a substitution θ is *defined* on α if and only if $\alpha \in dom(\theta)$.

The substitution θ is called a *ground substitution* if and only if it maps all type variables in its domain to ordinary types, i.e. if $\forall \alpha \in dom(\theta) : \theta\ \alpha \in \mathbf{Type}$. The substitution θ is said to *cover* Γ, respectively τ, if and only if it is defined on all the type variables in Γ, respectively τ. Substitutions can be applied to augmented types in a pointwise manner:

$$\begin{array}{rcl} \theta\ \texttt{int} & = & \texttt{int} \\ \theta\ \texttt{bool} & = & \texttt{bool} \\ \theta(\tau_1 \to \tau_2) & = & (\theta\ \tau_1) \to (\theta\ \tau_2) \\ \theta\ \alpha & = & \tau \ \text{ if } \theta\ \alpha = \tau \end{array}$$

We shall write $\theta_1 \circ \theta_2$ for the composition of θ_1 and θ_2, i.e. $(\theta_1 \circ \theta_2)\tau = \theta_1(\theta_2\ \tau)$ for all augmented types τ.

The idea. The *type reconstruction algorithm*, called $\mathcal{W}_{\mathsf{UL}}$, is given two arguments: an augmented type environment Γ (mapping program variables to augmented types) and an expression e. If it succeeds in finding a typing for the expression then it will return its augmented type τ and a substitution θ telling how the type environment has to be refined in order to obtain a typing. As an example we will have

$$\mathcal{W}_{\mathsf{UL}}([x \mapsto \texttt{'a}],\ \texttt{1 + (x 2)}) = (\texttt{int}, [\texttt{'a} \mapsto \texttt{int} \to \texttt{int}])$$

because during the inspection of the expression `1 + (x 2)` it becomes clear that `x` must be a function from integers to integers if the expression is to be correctly typed. So the idea is that if

$$\mathcal{W}_{\mathsf{UL}}(\Gamma, e) = (\tau, \theta) \quad \text{then} \quad \theta_G(\theta\ \Gamma) \vdash_{\mathsf{UL}} e : \theta_G\ \tau$$

for every ground substitution θ_G that covers $\theta\ \Gamma$ and τ, i.e. whenever we replace all the type variables with ordinary types in a consistent way. When

this property holds we shall say that the algorithm is *syntactically sound.* In order for the algorithm to be *syntactically complete* it is also required that all typings obtainable in the inference system can be reconstructed from the results of the algorithm. We shall discuss these properties at length in Subsection 5.3.3.

The algorithm. The algorithm $\mathcal{W}_{\mathsf{UL}}$ is specified in Table 5.5 and is explained below. The algorithm asks for *fresh type variables* at several places. By this is meant type variables that do not occur in the argument to $\mathcal{W}_{\mathsf{UL}}$ and that have not been generated as fresh variables elsewhere; this could be formalised by supplying $\mathcal{W}_{\mathsf{UL}}$ with yet another parameter for tracking the type variables that remain fresh but it is customary not to do so. There is a small amount of nondeterminism in $\mathcal{W}_{\mathsf{UL}}$ in that there may be many choices for the fresh variables; this could be made precise by assuming they are all numbered and by always supplying the candidate with the smallest number but again it is customary not to do so.

In the clause for constants we simply note that the type of c is τ_c and there is no need to adjust our assumptions Γ so we return the identity substitution *id*: we demand that $id\ \alpha = \alpha$ for all α and could take *id* to be the empty mapping. The clause for variables is similar except that now we consult the type environment Γ to determine the augmented type of x.

For function abstraction we assume that the formal parameter has type α_x for α_x being a fresh type variable – so far we have no constraints on the type of the formal parameter. Then we call $\mathcal{W}_{\mathsf{UL}}$ recursively on the function body to determine its type under the assumption that x has type α_x. The resulting type τ_0 and substitution θ_0 are then used to construct the overall type; in particular, θ_0 is used to replace the type variable α_x with a more refined type since the analysis of the function body may have provided additional information about the type of the formal parameter.

The clause for recursive function definition is somewhat more complicated. It starts out in a way similar to the previous clause and requires fresh type variables α_x and α_0 so that we can supply augmented types $\alpha_x \to \alpha_0$ and α_x for the occurrences of f and x, respectively, in the analysis of the function body. However, this will result in two possible types for the function body: one is the type variable α_0 (modified by the substitution θ_0 obtained by analysing the function body) and the other is the type τ_0 obtained from the analysis of the function body. These two types have to be equal according to the rule [*fn*] of Table 5.1 and to ensure this we shall use a *unification procedure* $\mathcal{U}_{\mathsf{UL}}$; its definition will be discussed in detail below but the idea is that given two augmented types τ_1 and τ_2, $\mathcal{U}_{\mathsf{UL}}(\tau_1, \tau_2)$ will return a substitution θ that makes them equal i.e. such that $\theta\ \tau_1 = \theta\ \tau_2$. In the clause for recursive function definition we get that $\theta_1(\tau_0) = \theta_1(\theta_0\ \alpha_0)$ so the overall type will be $\theta_1(\theta_0\ \alpha_x) \to \theta_1\ \tau_0$. Also we record that the assumptions of Γ have to be modified by θ_0 as well as θ_1.

$$\mathcal{W}_{\mathsf{UL}}(\Gamma, c) = (\tau_c,\ \ id)$$

$$\mathcal{W}_{\mathsf{UL}}(\Gamma, x) = (\Gamma(x),\ \ id)$$

$$\begin{array}{lll}\mathcal{W}_{\mathsf{UL}}(\Gamma, \texttt{fn}_\pi\ x\ \texttt{=>}\ e_0) = & \text{let} & \alpha_x \text{ be fresh} \\ & & (\tau_0, \theta_0) = \mathcal{W}_{\mathsf{UL}}(\Gamma[x \mapsto \alpha_x], e_0) \\ & \text{in} & ((\theta_0\ \alpha_x) \to \tau_0,\ \ \theta_0)\end{array}$$

$$\begin{array}{lll}\mathcal{W}_{\mathsf{UL}}(\Gamma, \texttt{fun}_\pi\ f\ x\ \texttt{=>}\ e_0) = & & \\ & \text{let} & \alpha_x, \alpha_0 \text{ be fresh} \\ & & (\tau_0, \theta_0) = \mathcal{W}_{\mathsf{UL}}(\Gamma[f \mapsto \alpha_x \to \alpha_0][x \mapsto \alpha_x], e_0) \\ & & \theta_1 = \mathcal{U}_{\mathsf{UL}}(\tau_0, \theta_0\ \alpha_0) \\ & \text{in} & (\theta_1(\theta_0\ \alpha_x) \to \theta_1\ \tau_0,\ \ \theta_1 \circ \theta_0)\end{array}$$

$$\begin{array}{lll}\mathcal{W}_{\mathsf{UL}}(\Gamma, e_1\ e_2) = & \text{let} & (\tau_1, \theta_1) = \mathcal{W}_{\mathsf{UL}}(\Gamma, e_1) \\ & & (\tau_2, \theta_2) = \mathcal{W}_{\mathsf{UL}}(\theta_1\ \Gamma, e_2) \\ & & \alpha \text{ be fresh} \\ & & \theta_3 = \mathcal{U}_{\mathsf{UL}}(\theta_2\ \tau_1, \tau_2 \to \alpha) \\ & \text{in} & (\theta_3\ \alpha,\ \ \theta_3 \circ \theta_2 \circ \theta_1)\end{array}$$

$$\begin{array}{lll}\mathcal{W}_{\mathsf{UL}}(\Gamma, \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2) = & \text{let} & (\tau_0, \theta_0) = \mathcal{W}_{\mathsf{UL}}(\Gamma, e_0) \\ & & (\tau_1, \theta_1) = \mathcal{W}_{\mathsf{UL}}(\theta_0\ \Gamma, e_1) \\ & & (\tau_2, \theta_2) = \mathcal{W}_{\mathsf{UL}}(\theta_1(\theta_0\ \Gamma), e_2) \\ & & \theta_3 = \mathcal{U}_{\mathsf{UL}}(\theta_2(\theta_1\ \tau_0), \texttt{bool}) \\ & & \theta_4 = \mathcal{U}_{\mathsf{UL}}(\theta_3\ \tau_2, \theta_3(\theta_2\ \tau_1)) \\ & \text{in} & (\theta_4(\theta_3\ \tau_2),\ \ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1)\end{array}$$

$$\begin{array}{lll}\mathcal{W}_{\mathsf{UL}}(\Gamma, \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) = & \text{let} & (\tau_1, \theta_1) = \mathcal{W}_{\mathsf{UL}}(\Gamma, e_1) \\ & & (\tau_2, \theta_2) = \mathcal{W}_{\mathsf{UL}}((\theta_1\ \Gamma)[x \mapsto \tau_1], e_2) \\ & \text{in} & (\tau_2,\ \ \theta_2 \circ \theta_1)\end{array}$$

$$\begin{array}{lll}\mathcal{W}_{\mathsf{UL}}(\Gamma, e_1\ op\ e_2) = & \text{let} & (\tau_1, \theta_1) = \mathcal{W}_{\mathsf{UL}}(\Gamma, e_1) \\ & & (\tau_2, \theta_2) = \mathcal{W}_{\mathsf{UL}}(\theta_1\ \Gamma, e_2) \\ & & \theta_3 = \mathcal{U}_{\mathsf{UL}}(\theta_2\ \tau_1, \tau_{op}^1) \\ & & \theta_4 = \mathcal{U}_{\mathsf{UL}}(\theta_3\ \tau_2, \tau_{op}^2) \\ & \text{in} & (\tau_{op},\ \ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1)\end{array}$$

Table 5.5: Algorithm $\mathcal{W}_{\mathsf{UL}}$ for the underlying type system.

Also the clause for function application relies on the unification procedure. In this clause we call $\mathcal{W}_{\mathsf{UL}}$ recursively on the operator and the operand and we use unification to ensure that the type of the operator τ_1 (modified by θ_2) is a function type with an argument type that equals the type τ_2 of the operand: $\theta_2\ \tau_1$ has to have the form $\tau_2 \to \alpha$. Again we have to record that the assumptions of Γ have to be modified by all three substitutions that have

been constructed.

By now the clause for conditional is straightforward and similarly the clauses for the `let`-construct and the binary operator. (Note that the `let`-construct is *not* polymorphic and therefore no special action is needed to determine the type of the bound variable; we shall return to the problem of polymorphism later.)

Example 5.14 Consider the expression $(\mathtt{fn}_X\ \mathtt{x} \Rightarrow \mathtt{x})\ (\mathtt{fn}_Y\ \mathtt{y} \Rightarrow \mathtt{y})$ of Example 5.1. The call

$$\mathcal{W}_{\mathsf{UL}}([\,], (\mathtt{fn}_X\ \mathtt{x} \Rightarrow \mathtt{x})\ (\mathtt{fn}_Y\ \mathtt{y} \Rightarrow \mathtt{y}))$$

gives rise to the call

$$\mathcal{W}_{\mathsf{UL}}([\,], \mathtt{fn}_X\ \mathtt{x} \Rightarrow \mathtt{x})$$

which will create the fresh type variable $'\mathsf{a}$ and return the pair $('\mathsf{a} \to '\mathsf{a}, id)$. We now have the call

$$\mathcal{W}_{\mathsf{UL}}([\,], \mathtt{fn}_Y\ \mathtt{y} \Rightarrow \mathtt{y})$$

that creates the fresh type variable $'\mathsf{b}$ and returns $('\mathsf{b} \to '\mathsf{b}, id)$. Thus we get the following call of $\mathcal{U}_{\mathsf{UL}}$

$$\mathcal{U}_{\mathsf{UL}}('\mathsf{a} \to '\mathsf{a}, ('\mathsf{b} \to '\mathsf{b}) \to '\mathsf{c})$$

where $'\mathsf{c}$ is a fresh type variable. As we shall see in Example 5.15 below this gives rise to the substitution $['\mathsf{a} \mapsto '\mathsf{b} \to '\mathsf{b}]['\mathsf{c} \mapsto '\mathsf{b} \to '\mathsf{b}]$ and the initial call of $\mathcal{W}_{\mathsf{UL}}$ will return $'\mathsf{b} \to '\mathsf{b}$ and $['\mathsf{a} \mapsto '\mathsf{b} \to '\mathsf{b}]['\mathsf{c} \mapsto '\mathsf{b} \to '\mathsf{b}]$. ■

Analogy. The placement of substitutions in Table 5.5 may seem ad hoc at first sight. As an aid to the intuition we shall therefore offer the following analogy. Consider a society where a number of laws have been passed. One such law might stipulate that when the owner of a personal computer sells it, the owner is obliged to comply with the law and in particular provide the buyer with the original disks for all software that remains on the computer. Let us focus our attention on a particular owner preparing to sell a computer and who is determining an acceptable selling price. Then parliament passes a law stating that whenever original software disks are passed on from one private person to another, the previous owner has to pay a fee of ten percent of the original buying price to a government agency combating software piracy. From this day on the owner of the computer needs to reconsider all the considerations made in order to determine the acceptable selling price. In short: whenever a new law is passed all existing considerations need to be reconsidered in order to remain valid. — Coming back to Table 5.5, the *typings* determined by $\mathcal{W}_{\mathsf{UL}}$ correspond to the considerations of the owner, and the *substitutions* produced by $\mathcal{U}_{\mathsf{UL}}$ to the new laws being passed by parliament: whenever a new substitution ("law") is constructed all existing

$$
\begin{array}{rcl}
\mathcal{U}_{\mathsf{UL}}(\texttt{int},\texttt{int}) & = & id \\
\mathcal{U}_{\mathsf{UL}}(\texttt{bool},\texttt{bool}) & = & id \\
\mathcal{U}_{\mathsf{UL}}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') & = & \text{let } \theta_1 = \mathcal{U}_{\mathsf{UL}}(\tau_1, \tau_1') \\
 & & \phantom{\text{let }} \theta_2 = \mathcal{U}_{\mathsf{UL}}(\theta_1\ \tau_2, \theta_1\ \tau_2') \\
 & & \text{in } \theta_2 \circ \theta_1 \\
\mathcal{U}_{\mathsf{UL}}(\tau, \alpha) & = & \begin{cases} [\alpha \mapsto \tau] & \text{if } \alpha \text{ does not occur in } \tau \\ & \text{or if } \alpha \text{ equals } \tau \\ \text{fail} & \text{otherwise} \end{cases} \\
\mathcal{U}_{\mathsf{UL}}(\alpha, \tau) & = & \begin{cases} [\alpha \mapsto \tau] & \text{if } \alpha \text{ does not occur in } \tau \\ & \text{or if } \alpha \text{ equals } \tau \\ \text{fail} & \text{otherwise} \end{cases} \\
\mathcal{U}_{\mathsf{UL}}(\tau_1, \tau_2) & = & \text{fail} \quad \text{in all other cases}
\end{array}
$$

Table 5.6: Unification of underlying types.

typings ("considerations") must be suitably modified so as to remain valid; this is exactly what is achieved by the carefully chosen use of substitutions in Table 5.5. □

Unification. The algorithm $\mathcal{U}_{\mathsf{UL}}$ for unifying two augmented types is shown in Table 5.6. It takes two augmented types τ_1 and τ_2 as arguments and if it succeeds it returns a substitution θ such that $\theta\ \tau_1 = \theta\ \tau_2$.

In the clause for unifying the two function types $\tau_1 \to \tau_2$ and $\tau_1' \to \tau_2'$ the desired result is obtained in two stages: θ_1 ensures that $\theta_1\ \tau_1 = \theta_1\ \tau_1'$ and hence $(\theta_2 \circ \theta_1)\ \tau_1 = (\theta_2 \circ \theta_1)\ \tau_1'$ and θ_2 ensures that $\theta_2(\theta_1\ \tau_2) = \theta_2(\theta_1\ \tau_2')$; it follows that $\theta_2 \circ \theta_1$ succeeds in unifying the two function types. Note that the algorithm only fails at *top-level* if:

- two types with different *top-level* constructors (by which we mean `int`, `bool`, or $\to$) are to be unified, or
- a type variable is to be unified with a function type containing that type variable.

Example 5.15 In Example 5.14 we had the following call of the unification procedure:

$$\mathcal{U}_{\mathsf{UL}}(\texttt{'a} \to \texttt{'a}, (\texttt{'b} \to \texttt{'b}) \to \texttt{'c})$$

It will first give rise to the call $\mathcal{U}_{\mathsf{UL}}(\texttt{'a}, \texttt{'b} \to \texttt{'b})$ which returns the substitution $[\texttt{'a} \mapsto \texttt{'b} \to \texttt{'b}]$. Then we will have the call $\mathcal{U}_{\mathsf{UL}}(\texttt{'b} \to \texttt{'b}, \texttt{'c})$ and the substitution

$[\text{'c} \mapsto \text{'b} \to \text{'b}]$ is returned. Thus the overall result will be the substitution $[\text{'a} \mapsto \text{'b} \to \text{'b}][\text{'c} \mapsto \text{'b} \to \text{'b}]$ as already used in Example 5.14. ■

5.3.2 An Algorithm for Control Flow Analysis

In applying the above ideas to the inference system for Control Flow Analysis presented in Table 5.2 we are going to face a difficulty. In the underlying type system two types are equal if and only if their syntactic representations are the same; we say that types constitute a *free algebra*. For annotated types, two annotated types *may* be equal even when their syntactic representations are different: $\texttt{int} \xrightarrow{\{\pi_1\}\cup\{\pi_2\}} \texttt{int}$ equals $\texttt{int} \xrightarrow{\{\pi_2\}\cup\{\pi_1\}} \texttt{int}$ because annotations and types are considered equal modulo *UCAI* as discussed in Subsection 5.1.2 – we say that annotated types constitute a *non-free algebra*.

The difficulty we encounter when transferring the development of the previous subsection to the Control Flow Analysis is that the algorithm $\mathcal{W}_{\mathsf{UL}}$ relies on the procedure $\mathcal{U}_{\mathsf{UL}}$ for unifying two types, and that this algorithm only applies to types in a free algebra. One way out of this difficulty would be to use instead an algorithm for unifying modulo UCAI; such algorithms exist but their properties are not so nice as those of $\mathcal{U}_{\mathsf{UL}}$. Another way out of the difficulty, and the approach we shall take, is to arrange it such that a variant of $\mathcal{U}_{\mathsf{UL}}$ can still be used by introducing additional mechanisms for dealing with the annotations: one is the notion of *simple* types and annotations and the other is the use of *constraints*.

Simple types and annotations. The first step will be to restrict the form of the annotated types so that only annotation variables are allowed on the function arrows; later we shall combine this with a set of constraints restricting the values of the annotation variables.

A *simple type* is an augmented annotated type where the only annotations allowed on function arrows are *annotation variables* and where *type variables* are allowed in types, and a *simple annotation* is an annotation where also annotation variables are allowed:

$$\begin{array}{lcll} \widehat{\tau} & \in & \textbf{SType} & \text{simple types} \\ \alpha & \in & \textbf{TVar} & \text{type variables} \\ \beta & \in & \textbf{AVar} & \text{annotation variables} \\ \varphi & \in & \textbf{SAnn} & \text{simple annotations} \end{array}$$

Formally, the syntactic categories are given by:

$$\begin{array}{lcl} \widehat{\tau} & ::= & \texttt{int} \mid \texttt{bool} \mid \widehat{\tau}_1 \xrightarrow{\beta} \widehat{\tau}_2 \mid \alpha \\ \alpha & ::= & \text{'a} \mid \text{'b} \mid \text{'c} \mid \cdots \\ \beta & ::= & \text{'1} \mid \text{'2} \mid \text{'3} \mid \cdots \end{array}$$

$$
\begin{array}{rcl}
\mathcal{U}_{\mathsf{CFA}}(\texttt{int},\texttt{int}) & = & id \\
\mathcal{U}_{\mathsf{CFA}}(\texttt{bool},\texttt{bool}) & = & id \\
\mathcal{U}_{\mathsf{CFA}}(\widehat{\tau}_1 \xrightarrow{\beta} \widehat{\tau}_2, \widehat{\tau}_1' \xrightarrow{\beta'} \widehat{\tau}_2') & = & \text{let } \theta_0 = [\beta' \mapsto \beta] \\
 & & \quad\;\; \theta_1 = \mathcal{U}_{\mathsf{CFA}}(\theta_0\ \widehat{\tau}_1, \theta_0\ \widehat{\tau}_1') \\
 & & \quad\;\; \theta_2 = \mathcal{U}_{\mathsf{CFA}}(\theta_1\ (\theta_0\ \widehat{\tau}_2), \theta_1\ (\theta_0\ \widehat{\tau}_2')) \\
 & & \text{in } \theta_2 \circ \theta_1 \circ \theta_0 \\
\mathcal{U}_{\mathsf{CFA}}(\widehat{\tau}, \alpha) & = & \begin{cases} [\alpha \mapsto \widehat{\tau}] & \text{if } \alpha \text{ does not occur in } \widehat{\tau} \\ & \text{or if } \alpha \text{ equals } \widehat{\tau} \\ \text{fail} & \text{otherwise} \end{cases} \\
\mathcal{U}_{\mathsf{CFA}}(\alpha, \widehat{\tau}) & = & \begin{cases} [\alpha \mapsto \widehat{\tau}] & \text{if } \alpha \text{ does not occur in } \widehat{\tau} \\ & \text{or if } \alpha \text{ equals } \widehat{\tau} \\ \text{fail} & \text{otherwise} \end{cases} \\
\mathcal{U}_{\mathsf{CFA}}(\widehat{\tau}_1, \widehat{\tau}_2) & = & \text{fail} \quad \text{in all other cases}
\end{array}
$$

Table 5.7: Unification of simple types.

$$\varphi \quad ::= \quad \{\pi\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset \mid \beta$$

A *simple type environment* $\widehat{\Gamma}$ then is a mapping from variables to simple types.

Unification of simple types. Simple types constitute a free algebra and so we can apply the same technique as in Subsection 5.3.1 to define a function $\mathcal{U}_{\mathsf{CFA}}$ of the following form:

$$\mathcal{U}_{\mathsf{CFA}}(\widehat{\tau}_1, \widehat{\tau}_2) = \theta$$

Here $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are simple types and θ will be a simple substitution: A *simple substitution* is a substitution that maps type variables to simple types, and that maps annotation variables to annotation *variables* only. Thus a simple substitution applied to a simple type still gives a simple type. As for $\mathcal{U}_{\mathsf{UL}}$ the intention is that θ unifies $\widehat{\tau}_1$ and $\widehat{\tau}_2$, i.e. $\theta\ \widehat{\tau}_1 = \theta\ \widehat{\tau}_2$. In case this is not possible the outcome of $\mathcal{U}_{\mathsf{CFA}}(\widehat{\tau}_1, \widehat{\tau}_2)$ will be a failure. The unification function $\mathcal{U}_{\mathsf{CFA}}$ is defined in Table 5.7 and is explained below.

As before *id* is the identity substitution and $\theta' \circ \theta''$ is the composition of two substitutions. In the clause for unifying the two function types $\widehat{\tau}_1 \xrightarrow{\beta} \widehat{\tau}_2$ and $\widehat{\tau}_1' \xrightarrow{\beta'} \widehat{\tau}_2$ the desired result is obtained by first ensuring that the two annotation variables are equal and then proceeding as in the unification algorithm for the underlying types. Note that $\mathcal{U}_{\mathsf{CFA}}(\widehat{\tau}_1, \widehat{\tau}_2)$ will fail if and only if $\mathcal{U}_{\mathsf{UL}}(\lfloor\widehat{\tau}_1\rfloor, \lfloor\widehat{\tau}_2\rfloor)$ fails.

Example 5.16 Consider the following call of the unification procedure:

$$\mathcal{U}_{\mathsf{CFA}}('\mathsf{a} \xrightarrow{'1} '\mathsf{a}, ('\mathsf{b} \xrightarrow{'2} '\mathsf{b}) \xrightarrow{'3} '\mathsf{c})$$

We construct the substitution $['3 \mapsto '1]$ and then we perform the call

$$\mathcal{U}_{\mathsf{CFA}}('\mathsf{a}, '\mathsf{b} \xrightarrow{'2} '\mathsf{b})$$

which returns the substitution $['\mathsf{a} \mapsto '\mathsf{b} \xrightarrow{'2} '\mathsf{b}]$. Then we make the call

$$\mathcal{U}_{\mathsf{CFA}}('\mathsf{b} \xrightarrow{'2} '\mathsf{b}, '\mathsf{c})$$

and the substitution $['\mathsf{c} \mapsto '\mathsf{b} \xrightarrow{'2} '\mathsf{b}]$ is returned. Thus the overall result will be $['3 \mapsto '1]['\mathsf{a} \mapsto '\mathsf{b} \xrightarrow{'2} '\mathsf{b}]['\mathsf{c} \mapsto '\mathsf{b} \xrightarrow{'2} '\mathsf{b}]$. ∎

The following fact, whose proof we leave to Exercise 5.8, expresses that the algorithm is correct. The first part of the result says that the algorithm is *syntactically sound*: if it succeeds then it produces the desired result. The second part says that the algorithm is *syntactically complete*: if there is some way of unifying the two simple types then the algorithm will succeed in doing so.

Fact 5.17 Let $\widehat{\tau}_1$ and $\widehat{\tau}_2$ be two simple types.

- If $\mathcal{U}_{\mathsf{CFA}}(\widehat{\tau}_1, \widehat{\tau}_2) = \theta$ then θ is a simple substitution such that $\theta\,\widehat{\tau}_1 = \theta\,\widehat{\tau}_2$.
- If there exists a substitution θ'' such that $\theta''\widehat{\tau}_1 = \theta''\widehat{\tau}_2$ then there exists substitutions θ and θ' such that $\mathcal{U}_{\mathsf{CFA}}(\widehat{\tau}_1, \widehat{\tau}_2) = \theta$ and $\theta'' = \theta' \circ \theta$. ∎

Constraints. Annotated types can contain arbitrary annotations and simple types can only contain annotation variables. To make up for this deficiency we shall introduce constraints on the annotation variables. A *constraint* is an inclusion of the form

$$\beta \supseteq \varphi$$

where β is an annotation variable and φ is a simple annotation. A constraint set C is a finite set of such constraints.

We shall write $\theta\ C$ for the set of inclusions obtained by applying the substitution θ to all the individual constraints of C: if $\beta \supseteq \varphi$ is in C then $\theta\ \beta \supseteq \theta\ \varphi$ is in $\theta\ C$. If C is a constraint set and θ is a simple substitution then also $\theta\ C$ is a constraint set.

A *type substitution* is a substitution that is defined on type variables only and that maps them to types in $\widehat{\mathbf{Type}}$ (i.e. to annotated types without type

and annotation variables); we shall say that it *covers* $\widehat{\Gamma}$, respectively $\widehat{\tau}$, if it is defined on all type variables in $\widehat{\Gamma}$, respectively $\widehat{\tau}$. Similarly, an *annotation substitution* is a substitution that is defined on annotation variables only and that maps them to annotations in **Ann** (i.e. to annotations without annotation variables); it covers $\widehat{\Gamma}$, respectively $\widehat{\tau}$ or C, if it is defined on all annotation variables in $\widehat{\Gamma}$, respectively $\widehat{\tau}$ or C. An *annotation* substitution θ_A *solves* a constraint set C, written

$$\theta_A \models C$$

if and only if it covers C and for each $\beta \supseteq \varphi$ in C it is the case that $\theta_A\ \beta$ is a superset of, or is equal to, $\theta_A\ \varphi$.

A *ground substitution* is an annotation substitution on annotation variables and a type substitution on type variables. A substitution θ is a *ground validation* of $(\widehat{\Gamma}, \widehat{\tau}, C)$ if and only if it is a ground substitution that covers $\widehat{\Gamma}, \widehat{\tau}$ and C and such that $\theta \models C$ (or more precisely, $\theta_A \models C$ where θ_A is the restriction of θ to the annotation variables).

The algorithm. We are now ready to define an analogue of the type reconstruction algorithm $\mathcal{W}_{\mathsf{UL}}$. It has the form

$$\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e) = (\widehat{\tau}, \theta, C)$$

where we demand that $\widehat{\Gamma}$ is a simple type environment (i.e. that it maps variables to simple types), and it will be the case that $\widehat{\tau}$ is a simple type, θ is a simple substitution, and C is a constraint set of a very special form: it only contains constraints of the form $\beta \supseteq \{\pi\}$. Since $\mathcal{W}_{\mathsf{CFA}}$ will make use of $\mathcal{U}_{\mathsf{CFA}}$ there is also the possibility that $\mathcal{W}_{\mathsf{CFA}}$ fails. The algorithm $\mathcal{W}_{\mathsf{CFA}}$ is defined by the clauses of Table 5.8 and is explained below.

In the clauses for constants and variables we proceed as before and do not impose any constraints. In the case of function abstraction we shall additionally use a fresh annotation variable β_0; it will be the top-level annotation of the overall type and we shall add a constraint $\beta_0 \supseteq \{\pi\}$ requiring that it includes the label π of the function definition – this corresponds to the annotation $\{\pi\} \cup \varphi$ used in the rule [*fn*] of Table 5.2.

The clause for recursive function definition is modified in a similar way. Here the annotation variable β will be used to annotate the function type of f and the call of $\mathcal{W}_{\mathsf{CFA}}$ on the body of the function may cause it to be modified to $\theta_0\ \beta_0$. Next the call of the unification procedure may cause it to be further modified to $\theta_1\ (\theta_0\ \beta_0)$. Since both θ_0 and θ_1 are simple substitutions we know that $\theta_1\ (\theta_0\ \beta_0)$ is an annotation variable so the resulting type will still be simple. The recursive call of $\mathcal{W}_{\mathsf{CFA}}$ gives rise to a constraint set C_0 that has to be modified using the substitution θ_1 and as in the clause for ordinary function abstraction we have to add a new constraint expressing that the

$$\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, c) = (\tau_c,\ id,\ \emptyset)$$

$$\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, x) = (\widehat{\Gamma}(x),\ id,\ \emptyset)$$

$$\begin{array}{lll}
\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, \texttt{fn}_\pi\ x\ \texttt{=>}\ e_0) = & \text{let} & \alpha_x \text{ be fresh} \\
& & (\widehat{\tau}_0, \theta_0, C_0) = \mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}[x \mapsto \alpha_x], e_0) \\
& & \beta_0 \text{ be fresh} \\
& \text{in} & ((\theta_0\ \alpha_x) \xrightarrow{\beta_0} \widehat{\tau}_0,\ \ \theta_0,\ \ C_0 \cup \{\beta_0 \supseteq \{\pi\}\})
\end{array}$$

$$\begin{array}{ll}
\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, \texttt{fun}_\pi\ f\ x\ \texttt{=>}\ e_0) = & \\
\quad \text{let} & \alpha_x, \alpha_0, \beta_0 \text{ be fresh} \\
& (\widehat{\tau}_0, \theta_0, C_0) = \mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}[f \mapsto \alpha_x \xrightarrow{\beta_0} \alpha_0][x \mapsto \alpha_x], e_0) \\
& \theta_1 = \mathcal{U}_{\mathsf{CFA}}(\widehat{\tau}_0, \theta_0\ \alpha_0) \\
\quad \text{in} & (\theta_1(\theta_0\ \alpha_x) \xrightarrow{\theta_1(\theta_0\ \beta_0)} \theta_1\ \widehat{\tau}_0,\ \ \theta_1 \circ \theta_0, \\
& \qquad (\theta_1\ C_0) \cup \{\theta_1(\theta_0\ \beta_0) \supseteq \{\pi\}\})
\end{array}$$

$$\begin{array}{lll}
\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e_1\ e_2) = & \text{let} & (\widehat{\tau}_1, \theta_1, C_1) = \mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e_1) \\
& & (\widehat{\tau}_2, \theta_2, C_2) = \mathcal{W}_{\mathsf{CFA}}(\theta_1\ \widehat{\Gamma}, e_2) \\
& & \alpha, \beta \text{ be fresh} \\
& & \theta_3 = \mathcal{U}_{\mathsf{CFA}}(\theta_2\ \widehat{\tau}_1, \widehat{\tau}_2 \xrightarrow{\beta} \alpha) \\
& \text{in} & (\theta_3\ \alpha,\ \ \theta_3 \circ \theta_2 \circ \theta_1,\ \ \theta_3\ (\theta_2\ C_1) \cup \theta_3\ C_2)
\end{array}$$

$$\begin{array}{ll}
\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2) = & \\
\quad \text{let} & (\widehat{\tau}_0, \theta_0, C_0) = \mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e_0) \\
& (\widehat{\tau}_1, \theta_1, C_1) = \mathcal{W}_{\mathsf{CFA}}(\theta_0\ \widehat{\Gamma}, e_1) \\
& (\widehat{\tau}_2, \theta_2, C_2) = \mathcal{W}_{\mathsf{CFA}}(\theta_1\ (\theta_0\ \widehat{\Gamma}), e_2) \\
& \theta_3 = \mathcal{U}_{\mathsf{CFA}}(\theta_2\ (\theta_1\ \widehat{\tau}_0), \texttt{bool}) \\
& \theta_4 = \mathcal{U}_{\mathsf{CFA}}(\theta_3\ \widehat{\tau}_2, \theta_3\ (\theta_2\ \tau_1)) \\
\quad \text{in} & (\theta_4\ (\theta_3\ \widehat{\tau}_2),\ \ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1 \circ \theta_0, \\
& \qquad \theta_4\ (\theta_3\ (\theta_2\ (\theta_1\ C_0))) \cup \theta_4\ (\theta_3\ (\theta_2\ C_1)) \cup \theta_4\ (\theta_3\ C_2))
\end{array}$$

$$\begin{array}{ll}
\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) = & \\
\quad \text{let} & (\widehat{\tau}_1, \theta_1, C_1) = \mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e_1) \\
& (\widehat{\tau}_2, \theta_2, C_2) = \mathcal{W}_{\mathsf{CFA}}((\theta_1\ \widehat{\Gamma})[x \mapsto \widehat{\tau}_1], e_2) \\
\quad \text{in} & (\widehat{\tau}_2,\ \ \theta_2 \circ \theta_1,\ \ (\theta_2\ C_1) \cup C_2)
\end{array}$$

$$\begin{array}{lll}
\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e_1\ op\ e_2) = & \text{let} & (\widehat{\tau}_1, \theta_1, C_1) = \mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e_1) \\
& & (\widehat{\tau}_2, \theta_2, C_2) = \mathcal{W}_{\mathsf{CFA}}(\theta_1\ \widehat{\Gamma}, e_2) \\
& & \theta_3 = \mathcal{U}_{\mathsf{CFA}}(\theta_2\ \widehat{\tau}_1, \tau^1_{op}) \\
& & \theta_4 = \mathcal{U}_{\mathsf{CFA}}(\theta_3\ \widehat{\tau}_2, \tau^2_{op}) \\
& \text{in} & (\tau_{op},\ \ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1, \\
& & \qquad \theta_4\ (\theta_3\ (\theta_2\ C_1)) \cup \theta_4\ (\theta_3\ C_2))
\end{array}$$

Table 5.8: Algorithm $\mathcal{W}_{\mathsf{CFA}}$ for Control Flow Analysis.

annotation variable θ_1 (θ_0 β_0) has to contain the label π of the function definition.

The clauses for the remaining constructs are fairly straightforward modifications of the similar clauses for $\mathcal{W}_{\mathsf{UL}}$. Note that the substitutions are applied to the constraint sets as well as the types and type environments.

Example 5.18 Returning to the expression ($\mathtt{fn}_{\mathsf{X}}$ `x => x`) ($\mathtt{fn}_{\mathsf{Y}}$ `y => y`) of Example 5.1 we shall now consider the call:

$$\mathcal{W}_{\mathsf{CFA}}([\,], (\mathtt{fn}_{\mathsf{X}}\ \mathtt{x} \Rightarrow \mathtt{x})\ (\mathtt{fn}_{\mathsf{Y}}\ \mathtt{y} \Rightarrow \mathtt{y}))$$

It gives rise to the call $\mathcal{W}_{\mathsf{CFA}}([\,], \mathtt{fn}_{\mathsf{X}}\ \mathtt{x} \Rightarrow \mathtt{x})$ which creates a fresh type variable $'\mathsf{a}$ and a fresh annotation variable $'1$ and returns $('\mathsf{a} \xrightarrow{'1} '\mathsf{a}, id, \{'1 \supseteq \{\mathsf{X}\}\})$. Then we have the call $\mathcal{W}_{\mathsf{CFA}}([\,], \mathtt{fn}_{\mathsf{Y}}\ \mathtt{y} \Rightarrow \mathtt{y})$ that creates a fresh type variable $'\mathsf{b}$ and a fresh annotation variable $'2$ and returns $('\mathsf{b} \xrightarrow{'2} '\mathsf{b}, id, \{'2 \supseteq \{\mathsf{Y}\}\})$. Thus we get the following call of $\mathcal{U}_{\mathsf{CFA}}$

$$\mathcal{U}_{\mathsf{CFA}}('\mathsf{a} \xrightarrow{'1} '\mathsf{a}, ('\mathsf{b} \xrightarrow{'2} '\mathsf{b}) \xrightarrow{'3} '\mathsf{c})$$

where $'\mathsf{c}$ is a fresh type variable and $'3$ is a fresh annotation variable. This gives rise to $['3 \mapsto '1]['\mathsf{a} \mapsto '\mathsf{b} \xrightarrow{'2} '\mathsf{b}]['\mathsf{c} \mapsto '\mathsf{b} \xrightarrow{'2} '\mathsf{b}]$ as shown in Example 5.16 and the initial call of $\mathcal{W}_{\mathsf{CFA}}$ will return the type $'\mathsf{b} \xrightarrow{'2} '\mathsf{b}$, the substitution $['3 \mapsto '1]['\mathsf{a} \mapsto '\mathsf{b} \xrightarrow{'2} '\mathsf{b}]['\mathsf{c} \mapsto '\mathsf{b} \xrightarrow{'2} '\mathsf{b}]$ and the set $\{'1 \supseteq \{\mathsf{X}\}, '2 \supseteq \{\mathsf{Y}\}\}$ of constraints. This corresponds to the typing obtained in Example 5.4. ∎

Example 5.19 Consider the program loop

```
let g = (funF f x => f (fnY y => y))
in g (fnZ z => z)
```

of Example 5.2 and the call $\mathcal{W}_{\mathsf{CFA}}([\,], \mathsf{loop})$. This will first give rise to a call

$$\mathcal{W}_{\mathsf{CFA}}([\,], \mathtt{fun}_{\mathsf{F}}\ \mathtt{f}\ \mathtt{x} \Rightarrow \mathtt{f}\ (\mathtt{fn}_{\mathsf{Y}}\ \mathtt{y} \Rightarrow \mathtt{y}))$$

that returns the type $('\mathsf{a} \xrightarrow{'2} '\mathsf{a}) \xrightarrow{'1} '\mathsf{b}$ and the set $\{'1 \supseteq \{\mathsf{F}\}, '2 \supseteq \{\mathsf{Y}\}\}$ of constraints. Then we have a call of $\mathcal{W}_{\mathsf{CFA}}$ on the body of the `let`-construct:

$$\mathcal{W}_{\mathsf{CFA}}([\mathtt{g} \mapsto ('\mathsf{a} \xrightarrow{'2} '\mathsf{a}) \xrightarrow{'1} '\mathsf{b}], \mathtt{g}\ (\mathtt{fn}_{\mathsf{Z}}\ \mathtt{z} \Rightarrow \mathtt{z}))$$

The call of $\mathcal{W}_{\mathsf{CFA}}$ on $\mathtt{fn}_{\mathsf{Z}}$ `z => z` will give the type $'\mathsf{c} \xrightarrow{'3} '\mathsf{c}$ and the constraint set $\{'3 \supseteq \{\mathsf{Z}\}\}$. The unification procedure will then be called with the types $('\mathsf{a} \xrightarrow{'2} '\mathsf{a}) \xrightarrow{'1} '\mathsf{b}$ and $('\mathsf{c} \xrightarrow{'3} '\mathsf{c}) \xrightarrow{'4} '\mathsf{d}$ and the resulting substitution is $['1 \mapsto '4, '2 \mapsto '3, '\mathsf{a} \mapsto '\mathsf{c}, '\mathsf{b} \mapsto '\mathsf{d}]$. Thus the application will have type $'\mathsf{d}$ and the constraint set will be $\{'3 \supseteq \{\mathsf{Z}\}\}$. The initial call of $\mathcal{W}_{\mathsf{CFA}}$ on loop returns the type $'\mathsf{d}$ and the constraint set $\{'4 \supseteq \{\mathsf{F}\}, '3 \supseteq \{\mathsf{Y}\}, '3 \supseteq \{\mathsf{Z}\}\}$. This corresponds to the typing obtained in Example 5.5. ∎

As illustrated in the above examples, when the *overall* call of $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e)$ gives $(\widehat{\tau}, \theta, C)$ we are only interested in the effect of θ on the type and annotation variables occurring in $\widehat{\Gamma}$.

5.3.3 Syntactic Soundness and Completeness

We are now ready to prove the correctness of the algorithm; this will take the form of a syntactic soundness result and a syntactic completeness result. Syntactic soundness is a rather straightforward result to prove. This normally also holds for Type and Effect Systems that are more complex than the Control Flow Analysis considered here, although the actual details of the Type and Effect Systems may of course require special treatment going beyond the techniques covered here.

By contrast, syntactic completeness is a somewhat harder result to prove. For a complex Type and Effect System it frequently involves establishing a result about *proof normalisation*: that the rules of the inference system that are not syntax directed need only be used at certain places (see Exercise 5.13). However, in the case of Control Flow Analysis the proof is going to be uncharacteristically simple because both the algorithm $\mathcal{W}_{\mathsf{CFA}}$ and the inference system for Control Flow Analysis are defined in syntax directed ways. Thus the present development does not indicate the breadth of techniques needed for Type and Effect Systems in general and the the Concluding Remarks will contain references to the more general situation.

Syntactic soundness. The soundness result expresses that any information obtained from the algorithm is indeed correct with respect the the inference system:

Theorem 5.20
If $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e) = (\widehat{\tau}, \theta, C)$ and θ_G is a ground validation of $\theta\ \widehat{\Gamma}$, $\widehat{\tau}$ and C then $\theta_G(\theta\ \widehat{\Gamma}) \vdash_{\mathsf{CFA}} e : \theta_G\ \widehat{\tau}$.

This theorem may be reformulated as follows: if $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e) = (\widehat{\tau}, \theta, C)$ and θ_T is a type substitution that covers $\theta\ \widehat{\Gamma}$ and $\widehat{\tau}$, and if θ_A is an annotation substition that covers $\theta\ \widehat{\Gamma}$, $\widehat{\tau}$ and C and that satisfies $\theta_A \models C$, then $\theta_A(\theta_T(\theta\ \widehat{\Gamma})) \vdash_{\mathsf{CFA}} e : \theta_A(\theta_T\ \widehat{\tau})$. To see this first note that $\theta_A \circ \theta_T = \theta_T \circ \theta_A$ is a ground substitution; next note that given a ground substitution θ_G we may obtain an annotation substitution θ_A by restricting θ_G to annotation variables and similarly we may obtain a type substitution θ_T by restricting θ_G to type variables and clearly $\theta_G = \theta_A \circ \theta_T = \theta_T \circ \theta_A$.

Proof The proof proceeds by structural induction on e (because $\mathcal{W}_{\mathsf{CFA}}$ is defined by structural induction on e).

The case c. We have $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, c) = (\tau_c, id, \emptyset)$. Next let θ_G be a ground validation of $\widehat{\Gamma}$; clearly it also covers τ_c (because τ_c is a base type) and it also satisfies $\theta_G \models \emptyset$. From the axiom [*con*] of Table 5.2 it is immediate that

$$\theta_G(\Gamma) \vdash_{\mathsf{CFA}} c : \tau_c$$

and since $\theta_G\ \tau_c = \tau_c$ this is the desired result.

The case x. We have $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, x) = (\widehat{\Gamma}(x), id, \emptyset)$. Next let θ_G be a ground validation of $\widehat{\Gamma}$; clearly it also covers $\widehat{\Gamma}(x)$ and it satisfies $\theta_G \models \emptyset$. From the axiom [*var*] of Table 5.2 it is immediate that

$$\theta_G\ \widehat{\Gamma} \vdash_{\mathsf{CFA}} x : \theta_G(\widehat{\Gamma}(x))$$

and this is the desired result.

The case $\mathbf{fn}_\pi\ x \texttt{=>} e_0$. We shall use the notation established in the clause for $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, \mathbf{fn}_\pi\ x \texttt{=>} e_0)$. So let θ_G be a ground validation of $\theta_0\ \widehat{\Gamma}$, $\theta_0\ \alpha_x \xrightarrow{\beta_0} \widehat{\tau}_0$, and $C_0 \cup \{\beta_0 \supseteq \{\pi\}\}$. Then θ_G is a ground validation of $\theta_0(\widehat{\Gamma}[x \mapsto \alpha_x])$, $\widehat{\tau}_0$, and C_0. Hence by the induction hypothesis we get:

$$\theta_G(\theta_0\ \widehat{\Gamma})[x \mapsto \theta_G(\theta_0\ \alpha_x)] \vdash_{\mathsf{CFA}} e_0 : \theta_G\ \widehat{\tau}_0$$

Since $\theta_G \models C_0 \cup \{\beta_0 \supseteq \{\pi\}\}$ we have $\theta_G\ \beta_0 \supseteq \{\pi\}$ so we can apply the rule [*fn*] of Table 5.2 and get

$$\theta_G(\theta_0\ \widehat{\Gamma}) \vdash_{\mathsf{CFA}} \mathbf{fn}_\pi\ x \texttt{=>} e_0 : \theta_G(\theta_0\ \alpha_x) \xrightarrow{\theta_G\ \beta_0} \theta_G\ \widehat{\tau}_0$$

which is the desired result.

The case $\mathbf{fun}_\pi\ f\ x \texttt{=>} e_0$. We shall use the notation already established in the clause for $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, \mathbf{fun}_\pi\ f\ x \texttt{=>} e_0)$. So let θ_G be a ground validation of $\theta_1(\theta_0\ \widehat{\Gamma})$, $\theta_1(\theta_0\ \alpha_x) \xrightarrow{\theta_1(\theta_0\ \beta_0)} \theta_1\ \widehat{\tau}_0$, and $(\theta_1\ C_0) \cup \{\theta_1(\theta_0\ \beta_0) \supseteq \{\pi\}\}$. Then $\theta_G \circ \theta_1$ is a ground validation of $\theta_0\ \widehat{\Gamma}$, $\theta_0\ \alpha_x \xrightarrow{\theta_0\ \beta_0} \widehat{\tau}_0$ and C_0. Since $\theta_1\ \widehat{\tau}_0 = \theta_1(\theta_0\ \alpha_0)$ by Fact 5.17 we also have that $\theta_G \circ \theta_1$ is a ground validation of $\theta_0\ \widehat{\Gamma}$, $\theta_0\ \alpha_x \xrightarrow{\theta_0\ \beta_0} \theta_0\ \alpha_0$ and C_0. Hence we can apply the induction hypothesis and get:

$$\theta_G(\theta_1(\theta_0\ (\widehat{\Gamma}[f \mapsto \alpha_x \xrightarrow{\beta_0} \alpha_0][x \mapsto \alpha_x]))) \vdash_{\mathsf{CFA}} e_0 : \theta_G(\theta_1\ \widehat{\tau}_0)$$

Since $\theta_1\ \widehat{\tau}_0 = \theta_1(\theta_0\ \alpha_0)$ and $\theta_G \models (\theta_1\ C_0) \cup \{\theta_1(\theta_0\ \beta_0) \supseteq \{\pi\}\}$ we get $\theta_G(\theta_1(\theta_0\ \beta_0)) \supseteq \{\pi\}$ so we can apply the rule [*fun*] of Table 5.2 and get

$$\theta_G(\theta_1(\theta_0\ \widehat{\Gamma})) \vdash_{\mathsf{CFA}} \mathbf{fun}_\pi\ f\ x \texttt{=>} e_0 : \theta_G(\theta_1(\theta_0\ \alpha_x)) \xrightarrow{\theta_G(\theta_1(\theta_0\ \beta_0))} \theta_G(\theta_1\ \widehat{\tau}_0)$$

and this is the desired result.

The case $e_1\ e_2$. We shall use the notation already established in the clause for $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e_1\ e_2)$. So let θ_G be a ground validation of $\theta_3(\theta_2(\theta_1\ \widehat{\Gamma}))$ and $\theta_3\ \alpha$ and $\theta_3\ (\theta_2\ C_1) \cup \theta_3\ C_2$. Let θ'_G be a *ground extension* of θ_G upon $\theta_3(\theta_2\ \widehat{\tau}_1)$ and $\theta_3\ \widehat{\tau}_2$. Then $\theta'_G \circ \theta_3 \circ \theta_2$ is a ground validation of $\theta_1\ \widehat{\Gamma}$, $\widehat{\tau}_1$ and C_1. Hence we can apply the induction hypothesis to e_1 and get:

$$\theta'_G(\theta_3(\theta_2(\theta_1\ \widehat{\Gamma}))) \vdash_{\mathsf{CFA}} e_1 : \theta'_G(\theta_3(\theta_2\ \widehat{\tau}_1))$$

Similarly $\theta'_G \circ \theta_3$ is a ground validation of $\theta_2(\theta_1\ \widehat{\Gamma})$, $\widehat{\tau}_2$ and C_2. Hence we can apply the induction hypothesis to e_2 and get:

$$\theta'_G(\theta_3(\theta_2(\theta_1\ \widehat{\Gamma}))) \vdash_{\mathsf{CFA}} e_2 : \theta'_G(\theta_3\ \widehat{\tau}_2)$$

Since $\theta_3(\theta_2\ \widehat{\tau}_1) = (\theta_3\ \widehat{\tau}_2) \xrightarrow{\theta_3\ \beta} (\theta_3\ \alpha)$ follows from Fact 5.17 we can use the rule [*app*] of Table 5.2 and get

$$\theta'_G(\theta_3(\theta_2(\theta_1\ \widehat{\Gamma}))) \vdash_{\mathsf{CFA}} e_1\ e_2 : \theta'_G(\theta_3\ \alpha)$$

which is equivalent to $\theta_G(\theta_3(\theta_2(\theta_1\ \widehat{\Gamma}))) \vdash_{\mathsf{CFA}} e_1\ e_2 : \theta_G(\theta_3\ \alpha)$ and this is the desired result.

The cases `if` e_0 `then` e_1 `else` e_2, `let` x `=` e_1 `in` e_2 and e_1 *op* e_2 are analogous. ∎

Syntactic completeness. It is not enough to show that $\mathcal{W}_{\mathsf{CFA}}$ is syntactically sound: an algorithm that always fails will indeed be a syntactically sound implementation of our analysis. We shall therefore be interested in a result saying that any judgement of the Annotated Type System can in fact be obtained by the algorithm:

Theorem 5.21
Assume that $\widehat{\Gamma}$ is a simple type environment and $\theta'\ \widehat{\Gamma} \vdash_{\mathsf{CFA}} e : \widehat{\tau}'$ holds for some ground substitution θ' that covers $\widehat{\Gamma}$. Then there exists $\widehat{\tau}$, θ, C and θ_G such that

- $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e) = (\widehat{\tau}, \theta, C)$,
- θ_G is a ground validation of $\theta\ \widehat{\Gamma}$, $\widehat{\tau}$ and C,
- $\theta_G \circ \theta = \theta'$ except on fresh type and annotation variables (as created by $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e)$), and
- $\theta_G\ \widehat{\tau} = \widehat{\tau}'$.

Proof The proof is by induction on the shape of the inference tree; since the Annotated Type System of Table 5.2 is syntax directed this means that the proof follows the syntactic structure of e. Without loss of generality we may assume that θ' is not defined on type and annotation variables that are freshly generated in the call $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e)$.

The case c. We have $\theta'\ \widehat{\Gamma} \vdash_{\mathsf{CFA}} c : \widehat{\tau}'$ and $\widehat{\tau}' = \tau_c$. Clearly $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, c) = (\tau_c, id, \emptyset)$ so it suffices to set $\theta_G = \theta'$ and clearly $\theta_G\ \widehat{\tau}' = \widehat{\tau}'$.

The case x. We have $\theta'\ \widehat{\Gamma} \vdash_{\mathsf{CFA}} x : \widehat{\tau}'$ because $\widehat{\tau}' = \theta'(\widehat{\Gamma}(x))$. Clearly $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, x) = (\widehat{\Gamma}(x), id, \emptyset)$ so it suffices to set $\theta_G = \theta'$.

The case $\mathtt{fn}_\pi\ x$ `=>` e_0. We have

$$\theta'\ \widehat{\Gamma} \vdash_{\mathsf{CFA}} \mathtt{fn}_\pi\ x \texttt{ => } e_0 : \widehat{\tau}'_x \xrightarrow{\{\pi\} \cup \varphi'} \widehat{\tau}'_0$$

where φ' is an annotation (i.e. it does not contain any annotation variables) and from Table 5.2 we get that also:

$$(\theta'\ \widehat{\Gamma})[x \mapsto \widehat{\tau}'_x] \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}'_0$$

Let now α_x be a fresh type variable and note that $\alpha_x \notin dom(\theta')$. Define θ'' by

$$\theta''\ \zeta = \begin{cases} \widehat{\tau}'_x & \text{if } \zeta = \alpha_x \\ \theta'\ \zeta & \text{otherwise} \end{cases}$$

where ζ can either be a type variable or an annotation variable. Then we also have:

$$\theta''(\widehat{\Gamma}[x \mapsto \alpha_x]) \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}'_0$$

By the induction hypothesis there exists $\widehat{\tau}_0$, θ_0, C_0 and θ'_G such that:

$$\begin{array}{l} \mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}[x \mapsto \alpha_x], e_0) = (\widehat{\tau}_0, \theta_0, C_0), \\ \theta'_G \text{ is a ground validation of } (\theta_0\ \widehat{\Gamma})[x \mapsto \theta_0(\alpha_x)], \widehat{\tau}_0, \text{ and } C_0, \\ \theta'_G \circ \theta_0 = \theta'' \text{ except on fresh type and annotation variables} \\ \qquad\qquad \text{created by } \mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}[x \mapsto \alpha_x], e_0) \\ \theta'_G\ \widehat{\tau}_0 = \widehat{\tau}'_0 \end{array}$$

Next let β_0 be a fresh annotation variable and define:

$$\theta_G\ \zeta = \begin{cases} \{\pi\} \cup \varphi' & \text{if } \zeta = \beta_0 \\ \theta'_G\ \zeta & \text{otherwise} \end{cases}$$

Then we get:

$$\begin{array}{l} \theta_G \text{ is a ground validation of } \theta_0\ \widehat{\Gamma}, (\theta_0\ \alpha_x) \xrightarrow{\beta_0} \widehat{\tau}_0 \text{ and } C_0 \cup \{\beta \supseteq \{\pi\}\}, \\ \theta_G \circ \theta_0 = \theta' \text{ except on fresh type and annotation variables} \\ \qquad\qquad \text{created by } \mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, \mathtt{fn}_\pi\ x \texttt{ => } e_0), \\ \theta_G(\theta_0\ \alpha_x \xrightarrow{\beta_0} \widehat{\tau}_0) = \widehat{\tau}'_x \xrightarrow{\{\pi\}\cup\varphi'} \widehat{\tau}'_0 \end{array}$$

This is the desired result.

The case $\mathtt{fun}_\pi\ f\ x \texttt{ => } e_0$. We have

$$\theta'\ \widehat{\Gamma} \vdash_{\mathsf{CFA}} \mathtt{fun}_\pi f x \texttt{ => } e_0 : \widehat{\tau}'_x \xrightarrow{\{\pi\}\cup\varphi'} \widehat{\tau}'_0$$

where φ' does not contain annotation variables and according to Table 5.2 this is because:

$$(\theta'\ \widehat{\Gamma})[f \mapsto \widehat{\tau}'_x \xrightarrow{\{\pi\}\cup\varphi'} \tau'_0][x \mapsto \widehat{\tau}'_x] \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}'_0$$

Let α_x, α_0 and β_0 be fresh type and annotation variables and note that they are not in $dom(\theta')$. Define θ'' by:

$$\theta''\ \zeta = \begin{cases} \widehat{\tau}'_x & \text{if } \zeta = \alpha_x \\ \{\pi\} \cup \varphi' & \text{if } \zeta = \beta_0 \\ \widehat{\tau}'_0 & \text{if } \zeta = \alpha_0 \\ \theta'\ \zeta & \text{otherwise} \end{cases}$$

Then we also have:

$$\theta''(\widehat{\Gamma}[f \mapsto \alpha_x \xrightarrow{\beta_0} \alpha_0][x \mapsto \alpha_x]) \vdash_{\mathsf{CFA}} e_0 : \widehat{\tau}_0'$$

By the induction hypothesis there exists $\widehat{\tau}_0$, θ_0, C_0 and θ_G' such that:

$$\begin{array}{l}
\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}[f \mapsto \alpha_x \xrightarrow{\beta_0} \alpha_0][x \mapsto \alpha_x], e_0) = (\widehat{\tau}_0, \theta_0, C_0) \\
\theta_G' \text{ is a ground validation of } \theta_0(\widehat{\Gamma}[f \mapsto \alpha_x \xrightarrow{\beta_0} \alpha_0][x \mapsto \alpha_x]), \widehat{\tau}_0, \text{ and } C_0 \\
\theta_G' \circ \theta_0 = \theta'' \text{ except on fresh type and annotation variables} \\
\qquad\qquad \text{created by } \mathcal{W}_{\mathsf{CFA}}(\cdots, e_0) \\
\theta_G' \ \widehat{\tau}_0 = \widehat{\tau}_0'
\end{array}$$

Since $\theta_G'(\theta_0\ \alpha_0) = \theta''\ \alpha_0 = \widehat{\tau}_0' = \theta_G'\ \widehat{\tau}_0$ it follows from Fact 5.17 that there exists θ_1 and θ_G such that $\mathcal{U}_{\mathsf{CFA}}(\theta_0\ \alpha_0, \widehat{\tau}_0) = \theta_1$ and $\theta_G' = \theta_G \circ \theta_1$. Hence

$$\begin{array}{l}
\theta_G \text{ is a ground validation of } \theta_1(\theta_0\ \widehat{\Gamma}), \theta_1(\theta_0\ \alpha_x) \xrightarrow{\theta_1(\theta_0\ \beta_0)} \theta_1\ \widehat{\tau}_0, \\
\qquad\qquad \text{and } (\theta_1\ C_0) \cup \{\theta_1(\theta_0\beta_0) \supseteq \{\pi\}\} \\
\theta_G \circ \theta_1 \circ \theta_0 = \theta' \text{ except on fresh type and annotation variables} \\
\qquad\qquad \text{created by } \mathcal{W}_{\mathsf{CFA}}(\cdots, \mathbf{fun}_\pi\ f\ x \texttt{=>} e_0) \\
\theta_G(\theta_1(\theta_0\ \alpha_x) \xrightarrow{\theta_1(\theta_0\ \beta_0)} \theta_1(\theta_0\ \alpha_0)) = \widehat{\tau}_x' \xrightarrow{\{\pi\} \cup \varphi'} \widehat{\tau}_0'
\end{array}$$

and this is the desired result.

The case $e_1\ e_2$. We have

$$\theta'\ \widehat{\Gamma} \vdash_{\mathsf{CFA}} e_1\ e_2 : \widehat{\tau}_0'$$

and according to Table 5.2 this is because:

$$\begin{array}{l}
\theta'\ \widehat{\Gamma} \vdash_{\mathsf{CFA}} e_1 : \widehat{\tau}_2' \xrightarrow{\varphi'} \widehat{\tau}_0' \\
\theta'\ \widehat{\Gamma} \vdash_{\mathsf{CFA}} e_2 : \widehat{\tau}_2'
\end{array}$$

By the induction hypothesis applied to e_1 there exists $\widehat{\tau}_1$, θ_1, C_1 and θ_G^1 such that:

$$\begin{array}{l}
\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e_1) = (\widehat{\tau}_1, \theta_1, C_1) \\
\theta_G^1 \text{ is a ground validation of } \theta_1\ \widehat{\Gamma}, \widehat{\tau}_1, \text{ and } C_1 \\
\theta_G^1 \circ \theta_1 = \theta' \text{ except on fresh type and annotation variables} \\
\qquad\qquad \text{created by } \mathcal{W}_{\mathsf{CFA}}(\theta'\ \widehat{\Gamma}, e_1) \\
\theta_G^1\ \widehat{\tau}_1 = \widehat{\tau}_2' \xrightarrow{\varphi'} \widehat{\tau}_0'
\end{array}$$

Then we have

$$\theta_G^1(\theta_1\ \widehat{\Gamma}) \vdash_{\mathsf{CFA}} e_2 : \widehat{\tau}_2'$$

so by the induction hypothesis applied to e_2 there exists $\widehat{\tau}_2$, θ_2, C_2 and θ_G^2 such that:

$$\begin{array}{l}
\mathcal{W}_{\mathsf{CFA}}(\theta_1\ \widehat{\Gamma}, e_2) = (\widehat{\tau}_2, \theta_2, C_2) \\
\theta_G^2 \text{ is a ground validation of } \theta_2(\theta_1\ \widehat{\Gamma}), \widehat{\tau}_2 \text{ and } C_2 \\
\theta_G^2 \circ \theta_2 = \theta_G^1 \text{ except on fresh type and annotation variables} \\
\qquad\qquad \text{created by } \mathcal{W}_{\mathsf{CFA}}(\cdots, e_2) \\
\theta_G^2\ \widehat{\tau}_2 = \widehat{\tau}_2'
\end{array}$$

It then follows that:

$$\begin{array}{l}
\theta_G^2 \text{ is a ground validation of } \theta_2(\theta_1\ \widehat{\Gamma}), \theta_2\ \widehat{\tau}_1, \theta_2\ C_1, \widehat{\tau}_2, \text{ and } C_2 \\
\theta_G^2 \circ \theta_2 \circ \theta_1 = \theta' \text{ except on fresh type and annotation variables} \\
\qquad\qquad \text{created by } \mathcal{W}_{\mathsf{CFA}}(\cdots, e_1) \text{ and } \mathcal{W}_{\mathsf{CFA}}(\cdots, e_2) \\
\theta_G^2(\theta_2\ \widehat{\tau}_1) = \widehat{\tau}_2' \xrightarrow{\varphi'} \widehat{\tau}_0' \\
\theta_G^2\ \widehat{\tau}_2 = \widehat{\tau}_2'
\end{array}$$

Next let α and β be fresh and define:

$$\theta_G^3\ \zeta = \begin{cases} \widehat{\tau}_0' & \text{if } \zeta = \alpha \\ \varphi' & \text{if } \zeta = \beta \\ \theta_G^2\ \zeta & \text{otherwise} \end{cases}$$

Since $\theta_G^3(\theta_2\ \widehat{\tau}_1) = \theta_G^2(\theta_2\ \widehat{\tau}_1) = \widehat{\tau}_2' \xrightarrow{\varphi'} \widehat{\tau}_0' = \theta_G^2\ \widehat{\tau}_2 \xrightarrow{\varphi'} \widehat{\tau}_0' = \theta_G^3(\widehat{\tau}_2 \xrightarrow{\beta} \alpha)$ it follows from Fact 5.17 that there exists θ_3 and θ_G such that $\mathcal{U}_{\mathsf{CFA}}(\theta_2\ \widehat{\tau}_1, \widehat{\tau}_2 \xrightarrow{\beta} \alpha) = \theta_3$ and $\theta_G^3 = \theta_G \circ \theta_3$. It follows that

$$\begin{array}{l}
\theta_G \text{ is a ground validation of } \theta_3(\theta_2(\theta_1\ \widehat{\Gamma})), \theta_3\ \alpha, \text{ and } \theta_3(\theta_2\ C_1) \cup \theta_3\ C_2 \\
\theta_G \circ \theta_3 \circ \theta_2 \circ \theta_1 = \theta' \text{ except on fresh type and annotation variables} \\
\qquad\qquad \text{created by } \mathcal{W}_{\mathsf{CFA}}(\cdots, e_1\ e_2) \\
\theta_G(\theta_3\ \alpha) = \theta_G^3\ \alpha = \widehat{\tau}_0'
\end{array}$$

and this is the desired result.

The cases **if** e_0 **then** e_1 **else** e_2, **let** x **=** e_1 **in** e_2 and e_1 *op* e_2 are analogous. ∎

5.3.4 Existence of Solutions

Since $\mathcal{W}_{\mathsf{CFA}}$ generates a set of constraints the statement of syntactic soundness (Theorem 5.20) is a little weaker than usual: if the constraints cannot be solved then we cannot use the soundness result to guarantee that the result produced by $\mathcal{W}_{\mathsf{CFA}}$ can be inferred in the inference system.

This suggests showing that the constraints always have solutions; in line with previous developments in this book we shall prove a stronger result. For this let $AV(C)$ be the set of annotation variables in C.

Lemma 5.22 If $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e) = (\widehat{\tau}, \theta, C)$ and X is a finite set of annotation variables such that $X \supseteq AV(C)$, then

$$\{\theta_A \mid \theta_A \models C \wedge dom(\theta_A) = X \wedge \theta_A \text{ is an annotation substitution}\}$$

is a Moore family. ∎

Proof Let C be a finite set of constraints of the form $\beta \supseteq \varphi$ where $\varphi \in$ **SAnn** and such that $X \supseteq AV(C)$ where X is a finite set of annotation variables; it will not be of importance that C is generated by $\mathcal{W}_{\mathsf{CFA}}$. Let Y be a possibly empty

subset of the set displayed in the lemma. Each element of Y will be an annotation substitution with domain X and that satisfies C. By setting

$$\theta\ \beta = \bigcap_{\theta_A \in Y} (\theta_A\ \beta) \qquad\qquad \text{for } \beta \in X$$

we define an annotation substitution θ with $dom(\theta) = X$. For each $\beta \supseteq \varphi$ in C and θ_A in Y we have

$$\theta_A\ \beta \supseteq \theta_A\ \varphi \supseteq \theta\ \varphi$$

(since φ is monotone in any free annotation variables) and hence:

$$\theta\ \beta = \bigcap_{\theta_A \in Y} \theta_A\ \beta \supseteq \bigcap_{\theta_A \in Y} \theta\ \varphi \supseteq \theta\ \varphi$$

This establishes the result. ∎

Consider now the call $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e) = (\widehat{\tau}, \theta, C)$ and the problem of finding a ground substitution θ_G that covers $\theta\ \widehat{\Gamma}$, $\widehat{\tau}$ and C. After the statement of Theorem 5.20 we made it clear that θ_G can always be written as $\theta_A \circ \theta_T$ for an annotation substitution θ_A covering $\theta\ \widehat{\Gamma}$, $\widehat{\tau}$ and C and a type substitution θ_T covering $\theta\ \widehat{\Gamma}$ and $\widehat{\tau}$. The choice of the type substitution θ_T must be performed by the user of $\mathcal{W}_{\mathsf{CFA}}$; one possibility is to let $\theta_T\ \alpha = \texttt{int}$ for all type variables α in $\theta\ \widehat{\Gamma}$ and $\widehat{\tau}$. The existence of an annotation substitution θ_A now follows from Lemma 5.22.

Corollary 5.23 If $\mathcal{W}_{\mathsf{CFA}}(\widehat{\Gamma}, e) = (\widehat{\tau}, \theta, C)$ then there exists a ground validation θ_G of $\theta\ \widehat{\Gamma}$, $\widehat{\tau}$ and C. ∎

Proof Let θ_T be given by $\theta_T\ \alpha = \texttt{int}$ for all type variables α in $\theta\ \widehat{\Gamma}$ and $\widehat{\tau}$; clearly θ_T covers $\theta\ \widehat{\Gamma}$ and $\widehat{\tau}$. Next let X be the set of annotation variables in $\theta\ \widehat{\Gamma}$, $\widehat{\tau}$ and C; then Lemma 5.22 guarantees the existence of an annotation substitution θ_A that covers $\theta\ \widehat{\Gamma}$, $\widehat{\tau}$ and C and such that $\theta_A \models C$. Taking $\theta_G = \theta_A \circ \theta_T$ we obtain a ground validation of $\theta\ \widehat{\Gamma}$, $\widehat{\tau}$ and C. ∎

The result obtained by choosing a type substitution is only unique in case there are no type variables present in $\theta\ \widehat{\Gamma}$ and $\widehat{\tau}$. If there is at least one type variable present then the result of using θ_T displayed above will differ from the result of using θ'_T given by $\theta'_T\ \alpha = \texttt{bool}$ for all type variables α in $\theta\ \widehat{\Gamma}$ and $\widehat{\tau}$. In general a type substitution θ''_T may have $\theta''_T\ \alpha \in \widehat{\mathbf{Type}}[\tau_\alpha]$ for an arbitrary underlying type τ_α. However, in case $\widehat{\mathbf{Type}}[\tau_\alpha]$ has more than one element one is likely to prefer the least element since it has empty annotations on all function arrows (but see Exercise 5.9).

In a similar way one is likely to prefer the least annotation substitution guaranteed by Lemma 5.22. Keeping in mind that all constraints in C have the form $\beta \supseteq \{\pi\}$ for $\beta \in \mathbf{AVar}$ and $\pi \in \mathbf{Pnt}$, we simply set:

$$\theta_A\ \beta = \begin{cases} \{\pi \mid \beta \supseteq \{\pi\} \text{ is in } C\} & \text{if } \beta \in AV(C) \\ \text{undefined} & \text{otherwise} \end{cases}$$

It is immediate that $dom(\theta_A) = AV(C)$ and that $\theta_A \models C$. If also $dom(\theta) = AV(C)$ and $\theta \models C$ then it is immediate that $\forall \beta : \theta\ \beta \supseteq \theta_A\ \beta$ which we may write as $\theta \sqsupseteq \theta_A$. This shows that θ_A as constructed above is indeed the least element of the Moore family displayed in Lemma 5.22.

5.4 Effects

The Type and Effect System for Control Flow Analysis is fairly simple: it is a syntax directed system using a form of subeffecting and the annotations are just sets. Much more powerful Type and Effect Systems can be constructed by allowing subtyping, `let`-polymorphism or polymorphic recursion; the resulting analyses will be more powerful and, not surprisingly, the techniques required for the implementation will be more demanding.

Subtyping and the various notions of polymorphism can be combined but for the sake of simplicity we shall present them one at a time. We shall first present a Side Effect Analysis for an extension of the FUN language with assignments; it will use subeffecting and subtyping. Then we shall present an Exception Analysis for an extension of FUN with exceptions; it will use subeffecting, subtyping as well as polymorphism. Finally, we shall present a Region Analysis for the FUN language; it will be based on polymorphic recursion.

5.4.1 Side Effect Analysis

Syntax. Let us consider an extension of the language FUN (Section 5.1) with imperative constructs for creating *reference variables* and for accessing and updating their values:

$$e ::= \cdots \mid \texttt{new}_\pi\ x\ \texttt{:=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{!}x \mid x\ \texttt{:=}\ e_0 \mid e_1\ \texttt{;}\ e_2$$

The idea is that $\texttt{new}_\pi\ x\ \texttt{:=}\ e_1\ \texttt{in}\ e_2$ creates a new reference variable called x and initialises it to the value of e_1; the scope of the reference variable is e_2 but we shall want the creation of the reference variable to be visible also outside its scope so as to be able to determine whether or not functions may need to allocate additional memory. The value of the reference variable x can be obtained by writing $\texttt{!}x$ and it may be set to a new value by the assignment $x\ \texttt{:=}\ e_0$. The sequencing construct $e_1\ \texttt{;}\ e_2$ first evaluates e_1 (for its side effects) and then e_2.

Example 5.24 The following program computes the Fibonacci number of a positive number `x` and leaves the result in the reference variable `r`:

```
newR r:=0
in   let fib = funF f z => if z<3 then r:=!r+1
                           else f(z-1); f(z-2)
     in fib x; !r
```

The program creates a new reference variable `r` and initialises it to 0, then it defines the function `fib`, applies it to the value of `x` and returns the value of `r`. The statement `r:=!r+1` in the body of the recursive function will increment the value of `r` each time it is executed and each call of `fib x` will increase the value of `r` with the Fibonacci number of `x`. ■

The aim of the *Side Effect Analysis* is to record:

> For each subexpression which locations have been created, accessed and assigned.

So for the function `fib` in Example 5.24 the analysis will record that it accesses and assigns the reference variable created at program point R.

Semantics. Before presenting the analysis let us briefly sketch the semantics of the language. To distinguish between the various incarnations of the **new**-construct we shall introduce *locations* (or references) and, as for the imperative languages of Chapter 2, the configurations will then contain a *store* component mapping locations to their values:

$$\varsigma \in \mathbf{Store} = \mathbf{Loc} \rightarrow_{\text{fin}} \mathbf{Val}$$

The values of **Val** include the constants c, the (closed) function abstractions of the form $\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e$ and the locations $\xi \in \mathbf{Loc}$. The semantic clauses of Table 5.4 are now modified to trace the store in a left-to-right evaluation as for example in the following clause for the `let`-construct:

$$\frac{\vdash \langle e_1, \varsigma_1\rangle \longrightarrow \langle v_1, \varsigma_2\rangle \quad \vdash \langle e_2[x \mapsto v_1], \varsigma_2\rangle \longrightarrow \langle v_2, \varsigma_3\rangle}{\vdash \langle \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2, \varsigma_1\rangle \longrightarrow \langle v_2, \varsigma_3\rangle}$$

For the new constructs we then have the following axioms and rules (explained below):

$$\frac{\vdash \langle e_1, \varsigma_1\rangle \longrightarrow \langle v_1, \varsigma_2\rangle \quad \vdash \langle e_2[x \mapsto \xi], \varsigma_2[\xi \mapsto v_1]\rangle \longrightarrow \langle v_2, \varsigma_3\rangle}{\vdash \langle \mathtt{new}_\pi\ x\ \mathtt{:=}\ e_1\ \mathtt{in}\ e_2, \varsigma_1\rangle \longrightarrow \langle v_2, \varsigma_3\rangle}$$

where ξ does not occur in the domain of ς_2

$$\vdash \langle !\xi, \varsigma\rangle \longrightarrow \langle \varsigma(\xi), \varsigma\rangle$$

$$\frac{\vdash \langle e, \varsigma_1\rangle \longrightarrow \langle v, \varsigma_2\rangle}{\vdash \langle \xi\mathtt{:=}e, \varsigma_1\rangle \longrightarrow \langle v, \varsigma_2[\xi \mapsto v]\rangle}$$

$$\frac{\vdash \langle e_1, \varsigma_1\rangle \longrightarrow \langle v_1, \varsigma_2\rangle \quad \vdash \langle e_2, \varsigma_2\rangle \longrightarrow \langle v_2, \varsigma_3\rangle}{\vdash \langle e_1; e_2, \varsigma_1\rangle \longrightarrow \langle v_2, \varsigma_3\rangle}$$

In the rule for **new** we evaluate e_1, create a new location ξ which is initialised to the value of e_1, and then we syntactically replace all occurrences of x with that location so that all subsequent references to x will be to ξ. We exploit this when defining the semantics for the constructs for accessing and updating the reference. Note that the value returned by the assignment will be the value being assigned.

Annotated types. In the Side Effect Analysis a location will be represented by the program point where it could be created. We shall therefore define the *annotations* (or effects) $\varphi \in \mathbf{Ann}_{\mathsf{SE}}$ by:

$$\varphi ::= \{!\pi\} \mid \{\pi\texttt{:=}\} \mid \{\texttt{new}\pi\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset$$

The annotation $!\pi$ means that the value of a location created at π is accessed, π**:=** means that a location created at π is assigned, and **new**π that a new location has been created at π. As for the Control Flow Analysis we shall consider annotations equal modulo UCAI. We shall need a set ϖ of program points defined by

$$\varpi ::= \pi \mid \varpi_1 \cup \varpi_2 \mid \emptyset$$

also to be interpreted modulo UCAI; we shall write $\varpi = \{\pi_1, \cdots, \pi_n\}$ for a typical element.

The *annotated types* $\widehat{\tau} \in \mathbf{Type}_{\mathsf{SE}}$ are now given by:

$$\widehat{\tau} ::= \texttt{int} \mid \texttt{bool} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \mid \texttt{ref}_{\varpi}\ \widehat{\tau}$$

Here $\texttt{ref}_{\varpi}\ \widehat{\tau}$ is the type of a location created at one of the program points in ϖ and that will contain values of the annotated type $\widehat{\tau}$. As before the type environment $\widehat{\Gamma}$ will map variables to annotated types.

Example 5.25 Consider the Fibonacci program

```
newR r:=0
in   let fib = funF f z => if z<3 then r:=!r+1
                                  else f(z-1); f(z-2)
     in fib x; !r
```

of Example 5.24. The variable **r** has the annotated type $\texttt{ref}_{\{\mathsf{R}\}}$ **int** and the variable **fib** has type $\texttt{int} \xrightarrow{\{!\mathsf{R},\mathsf{R}\texttt{:=}\}} \texttt{int}$ since it maps integers to integers and whenever executed it may, as a side effect, access and update a reference created at the program point R. ∎

Typing judgements. The typing judgements for the Side Effect Analysis will be of the form:

$$\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau}\ \&\ \varphi$$

$$[con] \quad \widehat{\Gamma} \vdash_{\mathsf{SE}} c : \tau_c \;\&\; \emptyset$$

$$[var] \quad \widehat{\Gamma} \vdash_{\mathsf{SE}} x : \widehat{\tau} \;\&\; \emptyset \qquad \text{if } \widehat{\Gamma}(x) = \widehat{\tau}$$

$$[fn] \quad \frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{SE}} e_0 : \widehat{\tau}_0 \;\&\; \varphi_0}{\widehat{\Gamma} \vdash_{\mathsf{SE}} \mathtt{fn}_\pi\ x \mathtt{=>} e_0 : \widehat{\tau}_x \xrightarrow{\varphi_0} \widehat{\tau}_0 \;\&\; \emptyset}$$

$$[fun] \quad \frac{\widehat{\Gamma}[f \mapsto \widehat{\tau}_x \xrightarrow{\varphi_0} \widehat{\tau}_0][x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{SE}} e_0 : \widehat{\tau}_0 \;\&\; \varphi_0}{\widehat{\Gamma} \vdash_{\mathsf{SE}} \mathtt{fun}_\pi\ f\ x \mathtt{=>} e_0 : \widehat{\tau}_x \xrightarrow{\varphi_0} \widehat{\tau}_0 \;\&\; \emptyset}$$

$$[app] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e_1 : \widehat{\tau}_2 \xrightarrow{\varphi_0} \widehat{\tau}_0 \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{SE}} e_2 : \widehat{\tau}_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{SE}} e_1\ e_2 : \widehat{\tau}_0 \;\&\; \varphi_1 \cup \varphi_2 \cup \varphi_0}$$

$$[if] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e_0 : \mathtt{bool} \;\&\; \varphi_0 \quad \widehat{\Gamma} \vdash_{\mathsf{SE}} e_1 : \widehat{\tau} \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{SE}} e_2 : \widehat{\tau} \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{SE}} \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \widehat{\tau} \;\&\; \varphi_0 \cup \varphi_1 \cup \varphi_2}$$

$$[let] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e_1 : \widehat{\tau}_1 \;\&\; \varphi_1 \quad \widehat{\Gamma}[x \mapsto \widehat{\tau}_1] \vdash_{\mathsf{SE}} e_2 : \widehat{\tau}_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{SE}} \mathtt{let}\ x \mathtt{=} e_1\ \mathtt{in}\ e_2 : \widehat{\tau}_2 \;\&\; \varphi_1 \cup \varphi_2}$$

$$[op] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e_1 : \tau^1_{op} \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{SE}} e_2 : \tau^2_{op} \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{SE}} e_1\ op\ e_2 : \tau_{op} \;\&\; \varphi_1 \cup \varphi_2}$$

$$[deref] \quad \widehat{\Gamma} \vdash_{\mathsf{SE}} \mathtt{!}x : \widehat{\tau} \;\&\; \{\mathtt{!}\pi_1, \cdots, \mathtt{!}\pi_n\} \qquad \text{if } \widehat{\Gamma}(x) = \mathtt{ref}_{\{\pi_1,\cdots,\pi_n\}}\widehat{\tau}$$

$$[ass] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{SE}} x \mathtt{:=} e : \widehat{\tau} \;\&\; \varphi \cup \{\pi_1\mathtt{:=}, \cdots, \pi_n\mathtt{:=}\}} \quad \text{if } \widehat{\Gamma}(x) = \mathtt{ref}_{\{\pi_1,\cdots,\pi_n\}}\widehat{\tau}$$

$$[new] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e_1 : \widehat{\tau}_1 \;\&\; \varphi_1 \quad \widehat{\Gamma}[x \mapsto \mathtt{ref}_{\{\pi\}}\widehat{\tau}_1] \vdash_{\mathsf{SE}} e_2 : \widehat{\tau}_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{SE}} \mathtt{new}_\pi\ x \mathtt{:=} e_1\ \mathtt{in}\ e_2 : \widehat{\tau}_2 \;\&\; (\varphi_1 \cup \varphi_2 \cup \{\mathtt{new}\pi\})}$$

$$[seq] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e_1 : \widehat{\tau}_1 \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{SE}} e_2 : \widehat{\tau}_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{SE}} e_1\ \mathtt{;}\ e_2 : \widehat{\tau}_2 \;\&\; \varphi_1 \cup \varphi_2}$$

$$[sub] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau}' \;\&\; \varphi'} \qquad \text{if } \widehat{\tau} \leq \widehat{\tau}' \text{ and } \varphi \subseteq \varphi'$$

Table 5.9: Side Effect Analysis.

The idea is that under the assumption $\widehat{\Gamma}$, the expression e will evaluate into a value with the annotated type $\widehat{\tau}$ and that during computation the side effects expressed by φ might take place. The analysis is specified by the axioms and rules of Table 5.9.

In the clauses [*con*] and [*var*] we record that there are no side effects so we use $\emptyset$ for the overall effect. The premise of the clause [*fn*] gives the effect of the body of the function and we use that to annotate the arrow of the function type whereas we use $\emptyset$ as the overall effect of the function definition itself: no side effects can be observed by simply defining the function. A similar explanation holds for the recursive function definition, the only difference being that the assumption about f in the premise has to use the *same* effect as the one determined from the function body. In the rule [*app*] we see how the information comes together: the overall effect is what we can observe from evaluating the argument e_1, what we can observe from evaluating the argnment e_2, and what we get from evaluating the body of the function called. The rules [*if*], [*let*] and [*op*] are straightforward.

Turning to the axioms and rules involving reference variables we make sure that we only assign values of the appropriate type to the variable. Also, in each of the cases we make sure to record that a location at the relevant program point has been created, referenced or assigned. The rule [*seq*] is straightforward and the final rule [*sub*] will be explained below.

Example 5.26 The following program

```
newA x:=1
in   (newB y:=!x in (x:=!y+1; !y+3))
   + (newC x:=!x in (x:=!x+1; !x+1))
```

evaluates to 8 because both summands evaluate to 4. The first summand has type and effect

$$\texttt{int} \;\&\; \{\texttt{newB}, \texttt{!A}, \texttt{A:=}, \texttt{!B}\}$$

and the second summand has type and effect

$$\texttt{int} \;\&\; \{\texttt{newC}, \texttt{!A}, \texttt{C:=}, \texttt{!C}\}$$

because the reference variable that is updated is the local one. Hence

$$\texttt{int} \;\&\; \{\texttt{newA}, \texttt{A:=}, \texttt{!A}, \texttt{newB}, \texttt{!B}, \texttt{newC}, \texttt{C:=}, \texttt{!C}\}$$

is the type and effect of the overall program. It follows from the effect that the variable being created at B (y in the program) is never reassigned after its creation. This might suggest transforming the $\texttt{new}_\texttt{B}$-construct into a `let`-construct (i.e. `let y=!x in (x:=y+1; y+3)`). ■

Subeffecting and subtyping. The purpose of the rule [*sub*] in Table 5.9 is to ensure that we obtain a conservative extension of the underlying type system. The rule is really a combination of a separate rule for *subeffecting*

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau} \;\&\; \varphi'} \quad \text{if } \varphi \subseteq \varphi'$$

and a separate rule for *subtyping*:

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{SE}} e : \widehat{\tau}' \;\&\; \varphi} \quad \text{if } \widehat{\tau} \leq \widehat{\tau}'$$

Here $\varphi \subseteq \varphi'$ means that φ is "a subset" of φ' (modulo UCAI) as discussed in Subsection 5.2.3 (and Exercise 5.3). The ordering $\widehat{\tau} \leq \widehat{\tau}'$ on annotated types is derived from the ordering on annotations as follows:

$$\widehat{\tau} \leq \widehat{\tau} \qquad \frac{\widehat{\tau}_1' \leq \widehat{\tau}_1 \quad \widehat{\tau}_2 \leq \widehat{\tau}_2' \quad \varphi \subseteq \varphi'}{\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \leq \widehat{\tau}_1' \xrightarrow{\varphi'} \widehat{\tau}_2'} \qquad \frac{\widehat{\tau} \leq \widehat{\tau}' \quad \widehat{\tau}' \leq \widehat{\tau} \quad \varpi \subseteq \varpi'}{\mathtt{ref}_{\varpi}\ \widehat{\tau} \leq \mathtt{ref}_{\varpi'}\ \widehat{\tau}'}$$

Note that the order of the comparison is reversed for arguments to functions; we say that $\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$ is *contravariant* in $\widehat{\tau}_1$ but *covariant* in φ and $\widehat{\tau}_2$. (To familiarise oneself with this idea just pretend that the types are really logical propositions and that $\xrightarrow{\varphi}$ as well as $\leq$ mean logical implication; then it should be clear that the rule is the right one.) Also $\mathtt{ref}_{\varpi}\ \widehat{\tau}$ is covariant in ϖ and both covariant in $\widehat{\tau}$ (when the reference variable is used for accessing its value as in $!x$) and contravariant in $\widehat{\tau}$ (when the reference variable is used for assignments as in x := $\cdots$). This turns out to be essential for semantic correctness to hold.

This form of subtyping we shall call *shape conformant subtyping* because $\widehat{\tau}_1 \leq \widehat{\tau}_2$ implies that the two annotated types have the same underlying types, i.e. $\lfloor \widehat{\tau}_1 \rfloor = \lfloor \widehat{\tau}_2 \rfloor$ and hence that $\widehat{\tau}_1$ and $\widehat{\tau}_2$ have the same "shape". (There are more permissive notions of subtyping than this, and we shall return to this issue in the Concluding Remarks.)

Example 5.27 Consider the following program

```
newA x:=1
in (fn f => f (fn y => !x) + f (fn z => (x:=z; z)))
   (fn g => g 1)
```

where we have omitted the labels on the function definitions. The program evaluates to 2 because each summand evaluates to 1.

The type and effect of the two arguments to f are

$$\mathtt{int} \xrightarrow{\{!\mathsf{A}\}} \mathtt{int} \quad \& \quad \emptyset$$

$$\mathtt{int} \xrightarrow{\{\mathsf{A}:=\}} \mathtt{int} \quad \& \quad \emptyset$$

and when f has the type

$$(\texttt{int} \xrightarrow{\{!\mathsf{A},\mathsf{A}:=\}} \texttt{int}) \xrightarrow{\{!\mathsf{A},\mathsf{A}:=\}} \texttt{int}$$

the application of f to the arguments will be well-typed: we have

$$(\texttt{int} \xrightarrow{\{!\mathsf{A}\}} \texttt{int}) \leq (\texttt{int} \xrightarrow{\{!\mathsf{A},\mathsf{A}:=\}} \texttt{int})$$

$$(\texttt{int} \xrightarrow{\{\mathsf{A}:=\}} \texttt{int}) \leq (\texttt{int} \xrightarrow{\{!\mathsf{A},\mathsf{A}:=\}} \texttt{int})$$

and may use the rule for subtyping to change the types of the arguments to the type expected by f.

If instead we had only used the rule for subeffecting we would be obliged to let the two arguments to f have type and effect:

$$\texttt{int} \xrightarrow{\{\mathsf{A}:=,!\mathsf{A}\}} \texttt{int} \quad \& \quad \emptyset$$

$$\texttt{int} \xrightarrow{\{\mathsf{A}:=,!\mathsf{A}\}} \texttt{int} \quad \& \quad \emptyset$$

This is indeed possible by using the rule for subeffecting just before the rule for function abstraction. ■

The combined rule [*sub*] for subeffecting and subtyping gives rise to a conservative extension of the underlying type system. This would also have been the case if we had adopted either the rule for subeffecting or the rule for subtyping. However, had we incorporated no additional rule then this would not be the case.

Remark. One can make a distinction between "Annotated Type Systems" and "Effect Systems" but it is a subtle one and it is hardly fruitful to distinguish them in a formal way; yet intuitively there is a difference. The analysis presented in this subsection is truly an Effect System because the annotations relate neither to the input nor the output of functions; rather they relate to the internal steps of the computation. By contrast the analysis of Subsection 5.1 is an Annotated Type System because the annotations relate to intensional aspects of the semantic values: what function abstraction might be the result of evaluating the expression. ■

5.4.2 Exception Analysis

Syntax. As our next analysis we consider an *Exception Analysis*; the aim of this analysis is to determine:

> For each expression, what exceptions might result from evaluating the expression.

These exceptions may be raised by primitive operators (like division by zero) or they may be explicitly raised in the program. Furthermore, there may be the possibility of trapping an exception by means of executing an expression designed to overcome the source of the abnormal situation. To illustrate this scenario we extend the syntax of expressions in FUN (Section 5.1) as follows:

$$e ::= \cdots \mid \texttt{raise}\ s \mid \texttt{handle}\ s\ \texttt{as}\ e_1\ \texttt{in}\ e_2$$

The exception is raised by the `raise`-construct. If e_2 raises some exception s_1 then **handle** s_2 **as** e_1 **in** e_2 will trap the exception in case $s_1 = s_2$ and this means that e_1 will be executed; if $s_1 \neq s_2$ we will continue propagating the exception s_1. We take a simple-minded approach to exceptions and use strings to denote their identity.

Example 5.28 Consider the following program computing the combinatorial $\binom{x}{y}$ of the values x and y of the variables `x` and `y`:

```
let comb = fun f x => fn y =>
           if x<0 then raise x-out-of-range
           else if y<0 or y>x then raise y-out-of-range
                else if y=0 or y=x then 1
                     else f (x-1) y + f (x-1) (y-1)
in handle x-out-of-range as 0 in comb x y
```

The program raises the exception `x-out-of-range` if `x` is negative and in the body of the `let`-construct this exception is trapped and the value `0` is returned. The program raises the exception `y-out-of-range` if the value of `y` is negative or larger than the value of `x` and this exception is not trapped in the program. ∎

Semantics. Before presenting the analysis let us briefly sketch the semantics of the language. An expression can now give rise to an exception so we shall extend the set **Val** of values to include entities of the form `raise` s. The semantic clauses of Table 5.4 then have to be extended to take care of the new kind of values. For function application we shall for example add three rules

$$\frac{\vdash e_1 \longrightarrow \texttt{raise}\ s}{\vdash e_1\ e_2 \longrightarrow \texttt{raise}\ s}$$

$$\frac{\vdash e_1 \longrightarrow (\texttt{fn}_\pi\ x\ \texttt{=>}\ e_0) \quad \vdash e_2 \rightarrow \texttt{raise}\ s}{\vdash e_1\ e_2 \longrightarrow \texttt{raise}\ s}$$

$$\frac{\vdash e_1 \longrightarrow (\texttt{fn}_\pi\ x\ \texttt{=>}\ e_0) \quad \vdash e_2 \rightarrow v_2 \quad \vdash e_0[x \mapsto v_2] \longrightarrow \texttt{raise}\ s}{\vdash e_1\ e_2 \longrightarrow \texttt{raise}\ s}$$

reflecting that an exception can be raised in any of the subcomputations in the premise of rule [*app*]. Similar rules are added for the other constructs.

We can then specify the meaning of the new constructs by the following axioms and rules:

$$\vdash \texttt{raise}\ s \longrightarrow \texttt{raise}\ s$$

$$\frac{\vdash e_2 \longrightarrow v_2}{\vdash \texttt{handle}\ s\ \texttt{as}\ e_1\ \texttt{in}\ e_2 \longrightarrow v_2} \qquad \text{if } v_2 \neq \texttt{raise}\ s$$

$$\frac{\vdash e_2 \longrightarrow \texttt{raise}\ s \quad \vdash e_1 \longrightarrow v_1}{\vdash \texttt{handle}\ s\ \texttt{as}\ e_1\ \texttt{in}\ e_2 \longrightarrow v_1}$$

Note that the expression e_2 in **handle** s **as** e_1 **in** e_2 may also raise an exception other than s in which case it will be propagated out of the **handle**-construct. Similarly, the expression e_1 may raise an exception which will then be propagated out of the **handle**-construct.

Annotated types. The purpose of Exception Analysis is to determine which exceptions might be raised and not trapped in the program. We shall therefore take the annotations to be sets of exceptions. To get a more flexible type system and a more powerful analysis we shall use a polymorphic type system. This means that we shall allow the annotated types to contain type variables and also we shall allow the annotations to contain annotation variables. So the *annotations* (or effects) $\varphi \in \mathbf{Ann}_{\mathsf{ES}}$ will be given by

$$\varphi ::= \{s\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset \mid \beta$$

and the *annotated types* $\widehat{\tau} \in \mathbf{Type}_{\mathsf{ES}}$ will be given by:

$$\widehat{\tau} ::= \texttt{int} \mid \texttt{bool} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \mid \alpha$$

As usual effects will be considered equal modulo UCAI. To express the *polymorphism* concisely we introduce *type schemes*. They have the form

$$\widehat{\sigma} ::= \forall(\zeta_1, \cdots, \zeta_n).\widehat{\tau}$$

where $\zeta_1, \cdots, \zeta_n$ is a (possible empty) list of type variables and annotation variables; if the list is empty we simply write $\widehat{\tau}$ for $\forall().\widehat{\tau}$.

Example 5.29 Consider the function $\texttt{fn}_{\mathsf{F}}$ `f => fn`$_{\mathsf{X}}$ `x => f x`. We can give it the type schema

$$\forall\ \texttt{'a},\ \texttt{'b},\ \texttt{'1}.\ (\texttt{'a} \xrightarrow{\texttt{'1}} \texttt{'b}) \xrightarrow{\emptyset} (\texttt{'a} \xrightarrow{\texttt{'1}} \texttt{'b})$$

which has instances like

$$(\texttt{int} \xrightarrow{\{\texttt{x-out-of-range}\}} \texttt{int}) \xrightarrow{\emptyset} (\texttt{int} \xrightarrow{\{\texttt{x-out-of-range}\}} \texttt{int})$$

and $(\texttt{int} \xrightarrow{\emptyset} \texttt{bool}) \xrightarrow{\emptyset} (\texttt{int} \xrightarrow{\emptyset} \texttt{bool})$. ■

[*con*] $\widehat{\Gamma} \vdash_{\mathsf{ES}} c : \tau_c \;\&\; \emptyset$

[*var*] $\widehat{\Gamma} \vdash_{\mathsf{ES}} x : \widehat{\sigma} \;\&\; \emptyset \qquad \text{if } \widehat{\Gamma}(x) = \widehat{\sigma}$

[*fn*] $$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{ES}} e_0 : \widehat{\tau}_0 \;\&\; \varphi_0}{\widehat{\Gamma} \vdash_{\mathsf{ES}} \mathtt{fn}_\pi\ x \mathtt{=>} e_0 : \widehat{\tau}_x \xrightarrow{\varphi_0} \widehat{\tau}_0 \;\&\; \emptyset}$$

[*fun*] $$\frac{\widehat{\Gamma}[f \mapsto \widehat{\tau}_x \xrightarrow{\varphi_0} \widehat{\tau}_0][x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{ES}} e_0 : \widehat{\tau}_0 \;\&\; \varphi_0}{\widehat{\Gamma} \vdash_{\mathsf{ES}} \mathtt{fun}_\pi\ f\ x \mathtt{=>} e_0 : \widehat{\tau}_x \xrightarrow{\varphi_0} \widehat{\tau}_0 \;\&\; \emptyset}$$

[*app*] $$\frac{\widehat{\Gamma} \vdash_{\mathsf{ES}} e_1 : \widehat{\tau}_2 \xrightarrow{\varphi_0} \widehat{\tau}_0 \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{ES}} e_2 : \widehat{\tau}_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{ES}} e_1\ e_2 : \widehat{\tau}_0 \;\&\; \varphi_1 \cup \varphi_2 \cup \varphi_0}$$

[*if*] $$\frac{\widehat{\Gamma} \vdash_{\mathsf{ES}} e_0 : \mathtt{bool} \;\&\; \varphi_0 \quad \widehat{\Gamma} \vdash_{\mathsf{ES}} e_1 : \widehat{\tau} \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{ES}} e_2 : \widehat{\tau} \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{ES}} \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \widehat{\tau} \;\&\; \varphi_0 \cup \varphi_1 \cup \varphi_2}$$

[*let*] $$\frac{\widehat{\Gamma} \vdash_{\mathsf{ES}} e_1 : \widehat{\sigma}_1 \;\&\; \varphi_1 \quad \widehat{\Gamma}[x \mapsto \widehat{\sigma}_1] \vdash_{\mathsf{ES}} e_2 : \widehat{\tau}_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{ES}} \mathtt{let}\ x \mathtt{=} e_1\ \mathtt{in}\ e_2 : \widehat{\tau}_2 \;\&\; \varphi_1 \cup \varphi_2}$$

[*op*] $$\frac{\widehat{\Gamma} \vdash_{\mathsf{ES}} e_1 : \tau^1_{op} \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{ES}} e_2 : \tau^2_{op} \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{ES}} e_1\ op\ e_2 : \tau_{op} \;\&\; \varphi_1 \cup \varphi_2}$$

[*raise*] $\widehat{\Gamma} \vdash_{\mathsf{ES}} \mathtt{raise}\ s : \widehat{\tau} \;\&\; \{s\}$

[*handle*] $$\frac{\widehat{\Gamma} \vdash_{\mathsf{ES}} e_1 : \widehat{\tau} \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{ES}} e_2 : \widehat{\tau} \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{ES}} \mathtt{handle}\ s\ \mathtt{as}\ e_1\ \mathtt{in}\ e_2 : \widehat{\tau} \;\&\; \varphi_1 \cup (\varphi_2 \backslash \{s\})}$$

[*sub*] $$\frac{\widehat{\Gamma} \vdash_{\mathsf{ES}} e : \widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{ES}} e : \widehat{\tau}' \;\&\; \varphi'} \qquad \text{if } \widehat{\tau} \leq \widehat{\tau}' \text{ and } \varphi \subseteq \varphi'$$

[*gen*] $$\frac{\widehat{\Gamma} \vdash_{\mathsf{ES}} e : \widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{ES}} e : \forall(\zeta_1, \cdots, \zeta_n).\widehat{\tau} \;\&\; \varphi}$$
if $\zeta_1, \cdots, \zeta_n$ do not occur free in $\widehat{\Gamma}$ and φ

[*ins*] $$\frac{\widehat{\Gamma} \vdash_{\mathsf{ES}} e : \forall(\zeta_1, \cdots, \zeta_n).\widehat{\tau} \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{ES}} e : (\theta\ \widehat{\tau}) \;\&\; \varphi}$$
if θ has $dom(\theta) \subseteq \{\zeta_1, \cdots, \zeta_n\}$

Table 5.10: Exception Analysis.

Typing judgements. The typing judgements for the Exception Analysis will be of the form

$$\widehat{\Gamma} \vdash_{\mathsf{ES}} e : \widehat{\sigma} \;\&\; \varphi$$

where the type environment $\widehat{\Gamma}$ now maps variables to type schemes. As expected $\widehat{\sigma}$ denotes the type schema of e, and φ is the set of exceptions that may be raised during evaluation of e. The analysis is specified by the axioms and rules of Table 5.10. Most of the axioms and rules are straightforward modifications of those we have seen before; in the rule [*let*] the use of a type schema for the `let`-bound variable is going to give us the required polymorphism.

The axiom [*raise*] ensures that a raised exception can have any type and the effect records that the exception s might be raised. The rule [*handle*] then determines the effects of its two subexpressions and records that any exception raised by e_1 and any exception except s raised by e_2 is a possible exception for the construct. Formally, $\varphi \setminus \{s\}$ is defined as follows: $\{s\} \setminus \{s\} = \emptyset$, $\{s'\} \setminus \{s\} = \{s'\}$ if $s' \neq s$, $(\varphi \cup \varphi') \setminus \{s\} = (\varphi \setminus \{s\}) \cup (\varphi' \setminus \{s\})$, $\emptyset \setminus \{s\} = \emptyset$ and $\beta \setminus \{s\} = \beta$. (We shall consider alternative definitions shortly.)

The rule [*sub*] is the rule for *subeffecting* and *subtyping* and the ordering $\widehat{\tau} \leq \widehat{\tau}'$ on types is given by

$$\widehat{\tau} \leq \widehat{\tau} \qquad\qquad \frac{\widehat{\tau}_1' \leq \widehat{\tau}_1 \quad \widehat{\tau}_2 \leq \widehat{\tau}_2' \quad \varphi \subseteq \varphi'}{\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \leq \widehat{\tau}_1' \xrightarrow{\varphi'} \widehat{\tau}_2'}$$

as was also the case in Subsection 5.4.1.

The rules [*gen*] and [*ins*] are responsible for the *polymorphism*. The generalisation rule [*gen*] is used to construct type schemes: we can quantify over any type or annotation variable that does not occur free in the assumptions or in the effect of the construct; this rule is usually used before applications of the rule [*let*]. The instantiation rule [*ins*] can then be used to turn type schemes into annotated types: we just apply a substitution in order to replace the bound type and annotation variables with other types and annotations (possibly containing type variables and annotation variables); this rule is usually used after applications of the axiom [*var*].

Example 5.30 The program

```
let f = fn g => fn x => g x
in f (fn y => if y < 0 then raise neg else y) (3-2)
   + f (fn z => if z > 0 then raise pos else 0-z) (2-3)
```

evaluates to 2 because each of the summands evaluates to `1`.

We may analyse `f` so as to obtain the type schema

$$\forall\, {}'\mathtt{a},\ {}'\mathtt{b},\ {}'0.({}'\mathtt{a} \xrightarrow{{}'0} {}'\mathtt{b}) \xrightarrow{\emptyset} ({}'\mathtt{a} \xrightarrow{{}'0} {}'\mathtt{b})$$

and we may analyse the two functional arguments to f so as to obtain:

$$\texttt{int} \xrightarrow{\{\texttt{neg}\}} \texttt{int} \;\&\; \emptyset$$

$$\texttt{int} \xrightarrow{\{\texttt{pos}\}} \texttt{int} \;\&\; \emptyset$$

We can now take two instances of the type schema for f: to match the first argument of f we use the substitution $[\texttt{'a} \mapsto \texttt{int}; \texttt{'b} \mapsto \texttt{int}; \texttt{'0} \mapsto \{\texttt{neg}\}]$ and to match the second argument we use $[\texttt{'a} \mapsto \texttt{int}; \texttt{'b} \mapsto \texttt{int}; \texttt{'0} \mapsto \{\texttt{pos}\}]$. Hence the type and effect of each summand is

$$\texttt{int} \;\&\; \{\texttt{neg}\}$$

$$\texttt{int} \;\&\; \{\texttt{pos}\}$$

and therefore

$$\texttt{int} \;\&\; \{\texttt{neg}, \texttt{pos}\}$$

is the overall type and effect of the entire program.

If we did not avail ourselves of polymorphism we would have to rely on subeffecting and subtyping and let f have the type

$$(\texttt{int} \xrightarrow{\{\texttt{neg},\texttt{pos}\}} \texttt{int}) \xrightarrow{\emptyset} (\texttt{int} \xrightarrow{\{\texttt{neg},\texttt{pos}\}} \texttt{int})$$

which is more imprecise than the type scheme displayed above, although we would still be able to obtain $\texttt{int} \;\&\; \{\texttt{neg}, \texttt{pos}\}$ as the overall type and effect of the entire program. ■

Remark. The judgements of the Exception Analyses were of the form $\widehat{\Gamma} \vdash_{\mathsf{ES}} e : \widehat{\sigma} \;\&\; \varphi$ but nonetheless most axioms and rules of Table 5.10 have conclusions of the form $\widehat{\Gamma} \vdash_{\mathsf{ES}} e : \widehat{\tau} \;\&\; \varphi$. By changing rules [*if*], [*let*], [*raise*], [*handle*] and possibly [*sub*] one can obtain a more liberal system for Exception Analysis. We leave the details for Exercise 5.15.

In the rule [*handle*] we make use of the notation $\varphi \setminus \{s\}$ and we next defined $\beta \setminus \{s\} = \beta$ even though β might later be instantiated to $\{s\}$. Since $\beta \setminus \{s\} \subseteq \beta$ should hold for all instantiations of β, this is semantically sound (corresponding to what happens in the rule [*sub*]). To define a less approximate system it would be natural to let

$$\varphi ::= \cdots \mid \varphi \setminus \{s\}$$

and then to extend the axiomatisation of UCAI to deal with set difference. ■

5.4.3 Region Inference

Let us once again consider the language FUN as introduced in Section 5.1. The purpose of *Region Inference* is to facilitate implementing FUN in a *stack-based* as opposed to a heap-based regime as is usually the case; since the

Figure 5.1: The memory model for the stack-based implementation of FUN.

stack-based regime is quite efficient in reusing dead memory locations without relying on explicit garbage collection this might lead to an efficient implementation. But functions are statically scoped and can produce other functions as results, so it is by no means clear that a stack-based regime should be possible; the purpose of region inference is to analyse how far locally allocated data can be passed around so that the allocation of memory can be performed in an appropriate manner. This leads to a memory model for the stack-based regime where the memory is a stack of dynamic regions (`r1`, `r2`, `r3`, $\cdots$) and each dynamic region is an indexed list (or array) of values as illustrated in Figure 5.1.

Syntax. To make this feasible we shall introduce a notion of extended expressions that contain explicit information about regions. To this end we introduce *region names*, *region variables*, and static *regions*

$$\begin{array}{rcll} rn & \in & \mathbf{RName} & \text{region names} \\ \varrho & \in & \mathbf{RVar} & \text{region variables} \\ r & \in & \mathbf{Reg}_{\mathsf{RI}} & \text{regions} \end{array}$$

as follows:

$$\begin{array}{rcl} rn & ::= & \mathtt{r1} \mid \mathtt{r2} \mid \mathtt{r3} \mid \cdots \\ \varrho & ::= & \text{''}\mathtt{1} \mid \text{''}\mathtt{2} \mid \text{''}\mathtt{3} \mid \cdots \\ r & ::= & \varrho \mid rn \end{array}$$

The syntax of *extended expressions*

$$ee \in \mathbf{EExp}$$

is given by:

$$
\begin{array}{rcl}
ee & ::= & c \texttt{ at } r \mid x \mid \texttt{fn}_\pi\ x \texttt{ => } ee_0 \texttt{ at } r \mid \texttt{fun}_\pi\ f\ [\vec{\varrho}]\ x \texttt{ => } ee_0 \texttt{ at } r \mid ee_1\ ee_2 \\
 & \mid & \texttt{if } ee_0 \texttt{ then } ee_1 \texttt{ else } ee_2 \mid \texttt{let } x \texttt{ = } ee_1 \texttt{ in } ee_2 \mid ee_1\ op\ ee_2 \texttt{ at } r \\
 & \mid & ee[\vec{r}] \texttt{ at } r \mid \texttt{letregion } \vec{\varrho} \texttt{ in } ee
\end{array}
$$

Here we write $\vec{\varrho}$ for a possibly empty sequence $\varrho_1, \cdots, \varrho_k$ of region variables and similarly $\vec{r}$ for a possible empty sequence $r_1, \cdots, r_k$ of regions. The main point of the extended expressions is that if a subexpression explicitly produces a new value then it has an explicit placement component, "at r", that indicates the region in which the value is (to be) placed. The `letregion`-construct explicitly allows the deallocation of the regions, $\vec{\varrho}$, that are no longer needed and the placement construct, $ee[\vec{r}]$ at r, allows us to explicitly place a "copy" of ee (typically a recursive function) in the region r. The construct for recursive function definitions explicitly takes a sequence of region variables as parameters; intuitively, this ensures that the various incarnations of the recursive function can have their local data in different regions.

Example 5.31 The relationship between expressions and extended expressions will be clarified when presenting the typing judgements below. For now we merely state that the expression `e` given by

```
(let x = 7 in fnY y => y+x) 9
```

will give rise to the extended expression `ee`:

```
letregion ϱ1, ϱ3, ϱ4
in (let x = (7 at ϱ1)
    in (fnY y => (y+x) at ϱ2) at ϱ3) (9 at ϱ4)
```

Here the value 7 is placed in the region ϱ_1, the value of `x+y` in region ϱ_2, the function named `Y` in the region ϱ_3 and the argument 9 in region ϱ_4. The final value (which is $7 + 9 = 16$) is therefore to be found in region ϱ_2. All other regions (ϱ_1, ϱ_3, and ϱ_4) no longer serve any purpose and can be deallocated; this is made explicit by the `letregion`-construct. In the version of the program that is actually run we should replace the metavariables ϱ_1, ϱ_3 and ϱ_4 by region variables "1, "2, and "3 and furthermore the free region variable ϱ_2 should be replaced by a region name such as `r1`. ∎

Semantics. The Natural Semantics of expressions is given by Table 5.4 and we now devise a Natural Semantics for the extended expressions so as to make the role of regions clear. The transitions takes the form

$$\rho \vdash \langle ee, \varsigma \rangle \longrightarrow \langle v, \varsigma' \rangle$$

where ρ is an *environment*, ee is an extended expression, ς and ς' are stores (as in Figure 5.1) and v is an expressible value. Formally we shall use the

domains:

$$
\begin{array}{lcll}
\rho & \in & \mathbf{Env} & = \mathbf{Var}_\star \to \mathbf{EVal} \\
v & \in & \mathbf{EVal} & = \mathbf{RName} \times \mathbf{Offset} \\
o & \in & \mathbf{Offset} & = \mathbf{N} \\
\varsigma & \in & \mathbf{Store} & = \mathbf{RName} \to_{\text{fin}} (\mathbf{Offset} \to_{\text{fin}} \mathbf{SVal}) \\
w & \in & \mathbf{SVal} &
\end{array}
$$

Here $\mathbf{Var}_\star$ is the finite set of variables in the extended expression $ee_\star$ of interest, a *store* is a stack of regions (where the stack is modelled as a finitary mapping from region names), a *region* is a list of storable values (where the list is modelled as a finitary mapping from indices called offsets), and a *storable value* is given by

$$w ::= c \mid \langle x, ee, \rho \rangle \mid \langle \vec{\varrho}, x, ee, \rho \rangle$$

consisting of ordinary constants, closures, and so-called *region polymorphic closures*. In addition to the formal parameter, the body of the function, and the environment at the definition point, a region polymorphic closure also contains a list of formal region parameters that have to be instantiated whenever the function is called – in this way it is explicityly ensured that the local data can be properly placed in the store for each function call. To simplify the notation we shall view a store as an element of $\mathbf{RName} \times \mathbf{Offset} \to_{\text{fin}} \mathbf{SVal}$ and so write $\varsigma(r, o)$ for $\varsigma(r)(o)$ etc. All ordinary values are "boxed" which means that they are always placed in the store and hence an expressible value is always a pair consisting of a region name and an offset.

The semantics is defined in Table 5.11 and is explained below. We intend that an extended expression does not contain free region variables when evaluated and hence many r's have become rn's. The axiom [*con*] for constants explicitly allocates a new offset in the relevant region and places the constant in that cell. The axiom [*var*] for variables does not involve the allocation of a new offset and merely performs a lookup in the environment.

The axiom [*fn*] for function abstraction allocates a new offset where it stores an ordinary closure consisting of the formal parameter, the body of the function, and the current environment. The axiom [*fun*] for recursive functions constructs a region polymorphic closure that records the list of formal region variables to be instantiated by means of the placement construct explained below; also note that recursion is handled by updating the current environment with a reference to the function itself.

The rules [*app*], [if_1], [if_2] and [*let*] should be straightforward and the rule [*op*] for binary operations allocates a new offset for the result.

The rule [*place*] takes care of the situation where an extended expression (such as a variable or a recursive function definition) evaluates to a region

$$[con] \quad \rho \vdash \langle c \ \mathbf{at}\ rn, \varsigma\rangle \longrightarrow \langle (rn, o), \varsigma[(rn, o) \mapsto c]\rangle$$
$$\text{if } o \notin dom(\varsigma(rn))$$

$$[var] \quad \rho \vdash \langle x, \varsigma\rangle \longrightarrow \langle \rho(x), \varsigma\rangle$$

$$[fn] \quad \rho \vdash \langle (\mathbf{fn}_\pi\ x \texttt{ => } ee_0)\ \mathbf{at}\ rn, \varsigma\rangle \longrightarrow \langle (rn, o), \varsigma[(rn, o) \mapsto \langle x, ee_0, \rho\rangle]\rangle$$
$$\text{if } o \notin dom(\varsigma(rn))$$

$$[fun] \quad \rho \vdash \langle (\mathbf{fun}_\pi\ f[\vec{\varrho}]\ x \texttt{ => } ee_0)\ \mathbf{at}\ rn, \varsigma\rangle \longrightarrow \langle (rn, o), \varsigma[(rn, o) \mapsto \langle \vec{\varrho}, x, ee_0, \rho[f \mapsto (rn, o)]\rangle]\rangle$$
$$\text{if } o \notin dom(\varsigma(rn))$$

$$[app] \quad \frac{\rho \vdash \langle ee_1, \varsigma_1\rangle \longrightarrow \langle (rn_1, o_1), \varsigma_2\rangle \quad \rho \vdash \langle ee_2, \varsigma_2\rangle \longrightarrow \langle v_2, \varsigma_3\rangle \quad \rho_0[x \mapsto v_2] \vdash \langle ee_0, \varsigma_3\rangle \longrightarrow \langle v_0, \varsigma_4\rangle}{\rho \vdash \langle ee_1\ ee_2, \varsigma_1\rangle \longrightarrow \langle v_0, \varsigma_4\rangle}$$
$$\text{if } \varsigma_3(rn_1, o_1) = \langle x, ee_0, \rho_0\rangle$$

$$[if_1] \quad \frac{\rho \vdash \langle ee_0, \varsigma_1\rangle \longrightarrow \langle (rn, o), \varsigma_2\rangle \quad \rho \vdash \langle ee_1, \varsigma_2\rangle \longrightarrow \langle v_1, \varsigma_3\rangle}{\rho \vdash \langle \mathbf{if}\ ee_0\ \mathbf{then}\ ee_1\ \mathbf{else}\ ee_2, \varsigma_1\rangle \longrightarrow \langle v_1, \varsigma_3\rangle}$$
$$\text{if } \varsigma_2(rn, o) = \mathbf{true}$$

$$[if_2] \quad \frac{\rho \vdash \langle ee_0, \varsigma_1\rangle \longrightarrow \langle (rn, o), \varsigma_2\rangle \quad \rho \vdash \langle ee_2, \varsigma_2\rangle \longrightarrow \langle v_2, \varsigma_3\rangle}{\rho \vdash \langle \mathbf{if}\ ee_0\ \mathbf{then}\ ee_1\ \mathbf{else}\ ee_2, \varsigma_1\rangle \longrightarrow \langle v_2, \varsigma_3\rangle}$$
$$\text{if } \varsigma_2(rn, o) = \mathbf{false}$$

$$[let] \quad \frac{\rho \vdash \langle ee_1, \varsigma_1\rangle \longrightarrow \langle v_1, \varsigma_2\rangle \quad \rho[x \mapsto v_1] \vdash \langle ee_2, \varsigma_2\rangle \longrightarrow \langle v_2, \varsigma_3\rangle}{\rho \vdash \langle \mathbf{let}\ x \texttt{ = } ee_1\ \mathbf{in}\ ee_2, \varsigma_1\rangle \longrightarrow \langle v_2, \varsigma_3\rangle}$$

$$[op] \quad \frac{\rho \vdash \langle ee_1, \varsigma_1\rangle \longrightarrow \langle (rn_1, o_1), \varsigma_2\rangle \quad \rho \vdash \langle ee_2, \varsigma_2\rangle \longrightarrow \langle (rn_2, o_2), \varsigma_3\rangle}{\rho \vdash \langle (ee_1\ op\ ee_2)\ \mathbf{at}\ rn, \varsigma_1\rangle \longrightarrow \langle (rn, o), \varsigma_3[(rn, o) \mapsto w]\rangle}$$
$$\text{if } \varsigma_3(rn_1, o_1)\ \mathbf{op}\ \varsigma_3(rn_2, o_2) = w \text{ and } o \notin dom(\varsigma_3(rn))$$

$$[place] \quad \frac{\rho \vdash \langle ee, \varsigma_1\rangle \longrightarrow \langle (rn', o'), \varsigma_2\rangle}{\rho \vdash \langle ee[\vec{rn}]\ \mathbf{at}\ rn, \varsigma_1\rangle \longrightarrow \langle (rn, o), \varsigma_2[(rn, o) \mapsto \langle x, ee_0[\vec{\varrho} \mapsto \vec{rn}], \rho_0\rangle]\rangle}$$
$$\text{if } o \notin dom(\varsigma_2(rn)) \text{ and } \varsigma_2(rn', o') = \langle \vec{\varrho}, x, ee_0, \rho_0\rangle$$

$$[region] \quad \frac{\rho \vdash \langle ee[\vec{\varrho} \mapsto \vec{rn}], \varsigma_1[\vec{rn} \mapsto [\,]]\rangle \longrightarrow \langle v, \varsigma_2\rangle}{\rho \vdash \langle \mathbf{letregion}\ \vec{\varrho}\ \mathbf{in}\ ee, \varsigma_1\rangle \longrightarrow \langle v, \varsigma_2 \backslash\!\backslash \vec{rn}\rangle}$$
$$\text{if } \{\vec{rn}\} \cap dom(\varsigma) = \emptyset$$

Table 5.11: Natural Semantics for extended expressions.

polymorphic closure. The placement construct $ee[\vec{r}]$ **at** r then allocates a new cell in the region r and stores a copy of the region polymorphic closure in the cell except that it ensures that the list of formal region parameters is replaced by a list of the actual region names; this is an important feature for allowing each recursive call of a function to allocate its auxiliary data locally on the stack rather than being lumped together (in the heap) with data from other recursive calls.

Finally, the rule [*region*] for the `letregion`-construct allocates new unused region names to be used instead of the region variables, evaluates the enclosed extended expression, and finally deallocates the newly allocated region names; formally, $dom(\varsigma \backslash\!\backslash \vec{rn}) = dom(\varsigma) \setminus \{\vec{rn}\}$ and $\forall (rn, o) \in dom(\varsigma \backslash\!\backslash \vec{rn}) : \varsigma(rn, o) = (\varsigma \backslash\!\backslash \vec{rn})(rn, o)$.

Annotated types. When analysing the extended expressions we shall want to keep track of those regions that may be affected during evaluation: in what regions do we place (or put) data and from what regions do we access (or get) data. This is somewhat analogous to the Side Effect Analysis of Subsection 5.4.1 and will be taken care of using the effects to be defined below. Another purpose of the analysis is to keep track of the regions in which the values reside. To this end we shall say that an *extended type* is a pair

$$\widehat{\tau}@r$$

consisting of an annotated type and the region where the value resides. Formally, *annotations* (or effects), *annotated types* and *type schemes*

$$\begin{array}{lcll} \varphi & \in & \mathbf{Ann}_{\mathsf{RI}} & \text{effects} \\ \widehat{\tau} & \in & \mathbf{Type}_{\mathsf{RI}} & \text{annotated types} \\ \widehat{\sigma} & \in & \mathbf{Scheme}_{\mathsf{RI}} & \text{type schemes} \end{array}$$

are given by:

$$\begin{array}{lcl} \varphi & ::= & \{\mathsf{put}\ r\} \mid \{\mathsf{get}\ r\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset \mid \beta \\ \widehat{\tau} & ::= & \mathtt{int} \mid \mathtt{bool} \mid (\widehat{\tau}_1@r_1) \xrightarrow{\beta.\varphi} (\widehat{\tau}_2@r_2) \mid \alpha \\ \widehat{\sigma} & ::= & \forall(\alpha_1, \cdots, \alpha_n), (\beta_1, \cdots, \beta_m), [\varrho_1, \cdots, \varrho_k].\widehat{\tau} \\ & \mid & \forall(\alpha_1, \cdots, \alpha_n), (\beta_1, \cdots, \beta_m).\widehat{\tau} \end{array}$$

Here we distinguish between two kinds of type schemes: *compound* type schemes (with $[\vec{\varrho}]$) to be used for recursive functions and *ordinary* type schemes (without $[\vec{\varrho}]$). We shall write $\widehat{\tau}$ for $\forall(), ().\widehat{\tau}$ whereas we insist on writing $\forall[\ ].\widehat{\tau}$ for $\forall(), (), [\ ].\widehat{\tau}$; in this way the two kinds of type schemes can always be distinguished. The use of an annotation variable, β, in the annotation placed on the function arrow, $\beta.\varphi$, is related to the use of *simple types* in Section 5.3 and is mainly of interest for the inference algorithm; for our

present purposes $\beta.\varphi$ can be read as $\beta \cup \varphi$ although the inference algorithm will view it as the constraint $\beta \supseteq \beta \cup \varphi$.

Example 5.32 Returning to the extended expression

```
letregion ϱ1, ϱ3, ϱ4
in (let x = (7 at ϱ1)
    in (fnY y => (y+x) at ϱ2) at ϱ3) (9 at ϱ4)
```

of Example 5.31, the subexpression `7 at` ϱ_1 will have the annotated type `int@`ϱ_1, the function abstraction `(fnY y => (y+x) at` ϱ_2`) at` ϱ_3 will have the annotated type

$$((\texttt{int@}\varrho_4) \xrightarrow{\beta.\varphi} (\texttt{int@}\varrho_2))\texttt{@}\varrho_3$$

where $\varphi = \{\mathsf{get}\ \varrho_4, \mathsf{get}\ \varrho_1, \mathsf{put}\ \varrho_2\}$, and the overall expression will have the extended type `int@`ϱ_2. ∎

Typing judgements. It would be quite feasible to define typing judgements for verifying that extended expressions are correctly typed. However, the main use of region inference is to facilitate the implementation of FUN and for this reason we shall touch upon the discussion initiated in Section 1.8 and let the typing judgements also describe how to translate expressions into extended expressions. This suggests using typing judgements of the form

$$\widehat{\Gamma} \vdash_{\mathsf{RI}} e \rightsquigarrow ee : \widehat{\tau}@r \ \&\ \varphi$$

where $\widehat{\Gamma}$ is a type environment mapping variables into extended type schemes that are pairs consisting of a type scheme and a region.

The analysis is defined in Tables 5.12 and 5.13 and is explained below. The axiom [*con*] for constants inserts an explicit placement component and records the placement in the effect. The axiom [*var*] for variables is straightforward as it involves no explicit placements.

The rule [*fn*] for ordinary function abstraction transforms the body of the function and then inserts an explicit placement component for the function itself. The rule [*fun*] for recursive functions involves a restricted form of *polymorphic recursion* (excluding type variables as this would be undecidable): polymorphic recursion means that when analysing the body of the function we are allowed to use the recursive function polymorphically. This generality gives much more precise type information and is essential for the success of region inference. For conciseness the rule has been presented in a manner where the rule for ordinary functions is needed to analyse the premise; clearly the rule could have been expanded so as to be syntax directed. The rules for application, conditional, local definitions and operators are straightforward.

Then we have a rule [*sub*] for *subeffecting* and *subtyping*; clearly one could decide to dispense with subtyping in which case subeffecting could be integrated with function abstraction as illustrated in Section 5.1.

$$[con] \quad \widehat{\Gamma} \vdash_{\mathsf{RI}} c \rightsquigarrow c \ \mathbf{at}\ r : (\tau_c @ r) \ \&\ \{\mathsf{put}\ r\}$$

$$[var] \quad \widehat{\Gamma} \vdash_{\mathsf{RI}} x \rightsquigarrow x : \widehat{\sigma} \ \&\ \emptyset \qquad \text{if } \widehat{\Gamma}(x) = \widehat{\sigma}$$

$$[fn] \quad \frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_x @ r_x] \vdash_{\mathsf{RI}} e_0 \rightsquigarrow ee_0 : (\widehat{\tau}_0 @ r_0) \ \&\ \varphi_0}{\begin{array}{l}\widehat{\Gamma} \vdash_{\mathsf{RI}} \mathbf{fn}_\pi\ x \texttt{=>} e_0 \rightsquigarrow (\mathbf{fn}_\pi\ x \texttt{=>} ee_0)\ \mathbf{at}\ r : \\ \qquad ((\widehat{\tau}_x @ r_x \xrightarrow{\beta.\varphi_0} \widehat{\tau}_0 @ r_0) @ r) \ \&\ \{\mathsf{put}\ r\}\end{array}}$$

$$[fun] \quad \frac{\begin{array}{c}\widehat{\Gamma}[f \mapsto (\forall \vec{\beta}, [\vec{\varrho}].\widehat{\tau}) @ r] \vdash_{\mathsf{RI}} \mathbf{fn}_\pi\ x \texttt{=>} e_0 \rightsquigarrow \\ (\mathbf{fn}_\pi\ x \texttt{=>} ee_0)\ \mathbf{at}\ r : (\widehat{\tau} @ r) \ \&\ \varphi\end{array}}{\begin{array}{l}\widehat{\Gamma} \vdash_{\mathsf{RI}} \mathbf{fun}_\pi\ f\ x \texttt{=>} e_0 \rightsquigarrow (\mathbf{fun}_\pi\ f\ [\vec{\varrho}]\ x \texttt{=>} ee_0)\ \mathbf{at}\ r : \\ \qquad ((\forall \vec{\beta}, [\vec{\varrho}].\widehat{\tau}) @ r) \ \&\ \varphi\end{array}}$$

if $\vec{\beta}$ and $\vec{\varrho}$ do not occur free in $\widehat{\Gamma}$ and φ

$$[app] \quad \frac{\begin{array}{c}\widehat{\Gamma} \vdash_{\mathsf{RI}} e_1 \rightsquigarrow ee_1 : ((\widehat{\tau}_2 @ r_2 \xrightarrow{\beta_0.\varphi_0} \widehat{\tau}_0 @ r_0) @ r_1) \ \&\ \varphi_1 \\ \widehat{\Gamma} \vdash_{\mathsf{RI}} e_2 \rightsquigarrow ee_2 : (\widehat{\tau}_2 @ r_2) \ \&\ \varphi_2\end{array}}{\widehat{\Gamma} \vdash_{\mathsf{RI}} e_1\ e_2 \rightsquigarrow ee_1\ ee_2 : (\widehat{\tau}_0 @ r_0) \ \&\ \varphi_1 \cup \varphi_2 \cup \varphi_0 \cup \beta_0 \cup \{\mathsf{get}\ r_1\}}$$

$$[if] \quad \frac{\begin{array}{c}\widehat{\Gamma} \vdash_{\mathsf{RI}} e_0 \rightsquigarrow ee_0 : (\mathbf{bool} @ r_0) \ \&\ \varphi_0 \\ \widehat{\Gamma} \vdash_{\mathsf{RI}} e_1 \rightsquigarrow ee_1 : (\widehat{\tau} @ r) \ \&\ \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{RI}} e_2 \rightsquigarrow ee_2 : (\widehat{\tau} @ r) \ \&\ \varphi_2\end{array}}{\begin{array}{l}\widehat{\Gamma} \vdash_{\mathsf{RI}} \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \rightsquigarrow \mathbf{if}\ ee_0\ \mathbf{then}\ ee_1\ \mathbf{else}\ ee_2 : \\ \qquad (\widehat{\tau} @ r) \ \&\ \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\mathsf{get}\ r_0\}\end{array}}$$

$$[let] \quad \frac{\begin{array}{c}\widehat{\Gamma} \vdash_{\mathsf{RI}} e_1 \rightsquigarrow ee_1 : (\widehat{\sigma}_1 @ r_1) \ \&\ \varphi_1 \\ \widehat{\Gamma}[x \mapsto \widehat{\sigma}_1 @ r_1] \vdash_{\mathsf{RI}} e_2 \rightsquigarrow ee_2 : (\widehat{\tau}_2 @ r_2) \ \&\ \varphi_2\end{array}}{\widehat{\Gamma} \vdash_{\mathsf{RI}} \mathbf{let}\ x \texttt{=} e_1\ \mathbf{in}\ e_2 \rightsquigarrow \mathbf{let}\ x \texttt{=} ee_1\ \mathbf{in}\ ee_2 : (\widehat{\tau}_2 @ r_2) \ \&\ \varphi_1 \cup \varphi_2}$$

$$[op] \quad \frac{\begin{array}{c}\widehat{\Gamma} \vdash_{\mathsf{RI}} e_1 \rightsquigarrow ee_1 : (\tau^1_{op} @ r_1) \ \&\ \varphi_1 \\ \widehat{\Gamma} \vdash_{\mathsf{RI}} e_2 \rightsquigarrow ee_2 : (\tau^2_{op} @ r_2) \ \&\ \varphi_2\end{array}}{\begin{array}{l}\widehat{\Gamma} \vdash_{\mathsf{RI}} e_1\ op\ e_2 \rightsquigarrow (ee_1\ op\ ee_2)\ \mathbf{at}\ r : \\ \qquad (\tau_{op} @ r) \ \&\ \varphi_1 \cup \varphi_2 \cup \{\mathsf{get}\ r_1, \mathsf{get}\ r_2, \mathsf{put}\ r\}\end{array}}$$

Table 5.12: Region Inference Analysis and Translation (part 1).

There are two rules $[gen_1]$ and $[gen_2]$ for generalising over type variables. One applies to ordinary types and the other to compound type schemes. In both cases we could have adapted the rule so as also to generalise over (additional) region and effect variables; however, even when doing so, the type inference is quite separate from the region and effect inference.

$$[sub] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{RI}} e \rightsquigarrow ee : (\widehat{\tau}@r) \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{RI}} e \rightsquigarrow ee : (\widehat{\tau}'@r) \;\&\; \varphi'} \quad \text{if } \widehat{\tau} \leq \widehat{\tau}' \text{ and } \varphi \subseteq \varphi'$$

$$[gen_1] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{RI}} e \rightsquigarrow ee : (\widehat{\tau}@r) \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{RI}} e \rightsquigarrow ee : ((\forall \vec{\alpha}.\widehat{\tau})@r) \;\&\; \varphi} \quad \text{if } \vec{\alpha} \text{ do not occur free in } \widehat{\Gamma} \text{ and } \varphi$$

$$[gen_2] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{RI}} e \rightsquigarrow ee : ((\forall \vec{\beta}, [\vec{\varrho}].\widehat{\tau})@r) \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{RI}} e \rightsquigarrow ee : ((\forall \vec{\alpha}, \vec{\beta}, [\vec{\varrho}].\widehat{\tau})@r) \;\&\; \varphi} \quad \text{if } \vec{\alpha} \text{ do not occur free in } \widehat{\Gamma} \text{ and } \varphi$$

$$[ins_1] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{RI}} e \rightsquigarrow ee : ((\forall \vec{\alpha}, \vec{\beta}.\widehat{\tau})@r) \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{RI}} e \rightsquigarrow ee : ((\theta\, \widehat{\tau})@r) \;\&\; \varphi} \quad \text{if } \theta \text{ has } dom(\theta) \subseteq \{\vec{\alpha}, \vec{\beta}\}$$

$$[ins_2] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{RI}} e \rightsquigarrow ee : ((\forall \vec{\alpha}, \vec{\beta}, [\vec{\varrho}].\widehat{\tau})@r) \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{RI}} e \rightsquigarrow ee[\theta\vec{\varrho}] \;\mathtt{at}\; r' : ((\theta\, \widehat{\tau})@r') \;\&\; \varphi \cup \{\mathsf{get}\; r, \mathsf{put}\; r'\}} \quad \text{if } \theta \text{ has } dom(\theta) \subseteq \{\vec{\alpha}, \vec{\beta}, \vec{\varrho}\}$$

$$[region] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{RI}} e \rightsquigarrow ee : (\widehat{\tau}@r) \;\&\; \varphi}{\widehat{\Gamma} \vdash_{\mathsf{RI}} e \rightsquigarrow \mathtt{letregion}\; \vec{\varrho}\; \mathtt{in}\; ee : (\widehat{\tau}@r) \;\&\; \varphi'} \quad \text{if } \varphi' = Observe(\widehat{\Gamma}, \widehat{\tau}, r)(\varphi) \text{ and } \vec{\varrho} \text{ occurs in } \varphi \text{ but not in } \varphi'$$

Table 5.13: Region Inference Analysis and Translation (part 2).

There are two inference rules $[ins_1]$ and $[ins_2]$ for instantiating a type scheme. One is for ordinary types schemes and is invisible as far as the syntax of the extended expression is concerned. The other is for compound type schemes and is visible in the extended expression in that an explicit placement construct is introduced; additionally the effect records that the value has been accessed and placed again. Both rules would typically be used immediately after the axiom for variables.

The rule $[region]$ for the `letregion`-construct uses an auxiliary function, *Observe*, to reduce the effect to what is visible from the outside; region variables that are no longer visible can then be encapsulated within the program. The auxiliary function may be defined as follows:

$$Observe(\widehat{\Gamma}, \widehat{\tau}, r')(\{\mathsf{put}\; r\}) = \begin{cases} \{\mathsf{put}\; r\} & \text{if } r \text{ occurs in } \widehat{\Gamma}, \widehat{\tau}, \text{ or } r' \\ \emptyset & \text{otherwise} \end{cases}$$

$$Observe(\widehat{\Gamma}, \widehat{\tau}, r')(\{\mathsf{get}\; r\}) = \begin{cases} \{\mathsf{get}\; r\} & \text{if } r \text{ occurs in } \widehat{\Gamma}, \widehat{\tau}, \text{ or } r' \\ \emptyset & \text{otherwise} \end{cases}$$

$$
\begin{aligned}
\mathit{Observe}(\widehat{\Gamma},\widehat{\tau},r')(\varphi_1 \cup \varphi_2) &= \mathit{Observe}(\widehat{\Gamma},\widehat{\tau},r')(\varphi_1) \cup \mathit{Observe}(\widehat{\Gamma},\widehat{\tau},r')(\varphi_2) \\
\mathit{Observe}(\widehat{\Gamma},\widehat{\tau},r')(\emptyset) &= \emptyset \\
\mathit{Observe}(\widehat{\Gamma},\widehat{\tau},r')(\beta) &= \begin{cases} \beta & \text{if } \beta \text{ occurs in } \widehat{\Gamma},\widehat{\tau}, \text{ or } r' \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

Example 5.33 In Example 5.31 we considered the expression **e**

$$(\texttt{let x = 7 in fn}_{\mathsf{Y}}\ \texttt{y => y+x})\ \texttt{9}$$

and the extended expression **ee**:

$$
\begin{array}{l}
\texttt{letregion}\ \varrho_1,\ \varrho_3,\ \varrho_4 \\
\texttt{in}\ (\texttt{let x = (7 at}\ \varrho_1) \\
\quad \texttt{in}\ (\texttt{fn}_{\mathsf{Y}}\ \texttt{y => (y+x) at}\ \varrho_2)\ \texttt{at}\ \varrho_3)\ (\texttt{9 at}\ \varrho_4)
\end{array}
$$

One can now show that $[\,] \vdash_{\mathsf{RI}} \mathbf{e} \rightsquigarrow \mathbf{ee} : (\texttt{int@}\varrho_2)\ \&\ \{\mathsf{put}\ \varrho_2\}$. ■

5.5 Behaviours

So far the effects have had a rather simple structure in that they merely denote *sets* of atomic actions like accessing a value or raising an exception. The effects have not attempted to capture the *temporal order* of these atomic actions. Often such information would be useful, for example to check that a variable is not accessed until after it has been assigned a value. In this section we shall show how to devise a Type and Effect System where effects (called behaviours) are able to record such temporal ordering in the context of a *Communication Analysis* for a fragment of Concurrent ML.

5.5.1 Communication Analysis

Syntax. Let us consider an extension of the language FUN with constructs for generating new processes, for communicating between processes over typed channels, and for creating new channels. The syntactic category $e \in$ **Exp** of *expressions* is now given by:

$$e ::= \cdots \mid \texttt{channel}_\pi \mid \texttt{spawn}\ e_0 \mid \texttt{send}\ e_1\ \texttt{on}\ e_2 \mid \texttt{receive}\ e_0 \mid e_1; e_2 \mid ch$$

Here $\texttt{channel}_\pi$ creates a new channel identifier (denoted ch above), **spawn** e_0 generates a new parallel process that executes e_0, and **send** v **on** ch sends the value v to another process ready to receive a value by means of **receive** ch; sequential composition is as before. Channel identifiers

$$ch \in \textbf{Chan} \quad \text{channel identifiers}$$

Figure 5.2: The pipeline produced by `pipe [f1, f2] inp out`.

are created dynamically and are given by:

$$ch ::= \texttt{chan1} \mid \texttt{chan2} \mid \cdots$$

Also we shall assume that the constants, $c \in$ **Const**, not only include integers and booleans but also a special value called unit and denoted by `()`; this is the value to be returned by the `spawn` and `send` constructs.

Example 5.34 In this example we shall imagine that the expressions are extended with operations on lists: `isnil` e tests whether or not the list e is empty, `hd` e selects the first element and `tl` e selects the remainder of the list. We now define a function `pipe` that takes a list of functions, an input channel and an output channel as arguments; it constructs a pipeline of processes that apply the functions to the data arriving at the input channel and returns the results on the output channel (see Figure 5.2). It makes use of the function `node` that takes a function, an input channel and an output channel as arguments and it is defined as follows:

```
let node = fnF f => fnI inp => fnO out =>
            spawn ((funH h d => let v = receive inp
                                in send (f v) on out;
                                h d) ())
in funP pipe fs => fnI inp => fnO out =>
         if isnil fs then node (fnX x =>x) inp out
         else let ch = channelC
              in (node (hd fs) inp ch; pipe (tl fs) ch out)
```

To deal with the empty list the function produces a process that just applies the identity function (denoted `id` in Figure 5.2). ■

Semantics. We shall begin by defining the operational semantics of the sequential fragment of the language and then show how to incorporate it into the concurrent fragment. This will make use of a notion of evaluation context in order to obtain a succinct specification of the semantics. The sequential fragment is evaluated in an eager left to right manner and does not need any

$$(\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e)\ v \rightarrow e[x \mapsto v]$$

$$\mathtt{let}\ x\ \mathtt{=}\ v\ \mathtt{in}\ e \rightarrow e[x \mapsto v]$$

$$v_1\ op\ v_2 \rightarrow v \quad \text{if } v_1\ \mathbf{op}\ v_2 = v$$

$$\mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e \rightarrow (\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e)[f \mapsto (\mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e)]$$

$$\mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2 \rightarrow e_1$$

$$\mathtt{if\ false\ then}\ e_1\ \mathtt{else}\ e_2 \rightarrow e_2$$

$$v; e \rightarrow e$$

Table 5.14: The sequential semantics.

notion of a store, a state or an environment. A fully evaluated expression gives rise to a value which is a constant, a channel identifier or a function abstraction. This may be modelled by

$$v \in \mathbf{Val} \quad \text{values}$$

defined by:

$$v ::= c \mid ch \mid \mathtt{fn}_\pi\ x\ \mathtt{=>}\ e_0$$

There is no need to package the function abstraction with an environment because the semantics will treat a function application $(\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e_0)\ v$ by substituting v for x in e_0 yielding $e_0[x \mapsto v]$.

Part of the *sequential semantics* is specified in Table 5.14. In the manner of a Structural Operational Semantics it axiomatises the relation

$$e_1 \rightarrow e_2$$

for when the expression e_1 evaluates to e_2 in one step. However, it does not describe how e_1 may evaluate to e_2 as a result of a subcomponent e_{11} of e_1 evaluating to e_{12} unlike what has been the case for the Structural Operational Semantics used so far. As an example Table 5.14 cannot be used to show that $(1 + 2) + 4 \rightarrow 3 + 4$ although it can be used to show that $1 + 2 \rightarrow 3$.

To recover from this shortcoming we shall make use of *evaluation contexts*; these are expressions containing *one hole* which is written [ ]. Formally, evaluation contexts E are given by:

$$\begin{array}{rcl} E & ::= & [\,] \mid E\ e \mid v\ E \mid \mathtt{let}\ x\ \mathtt{=}\ E\ \mathtt{in}\ e \\ & \mid & \mathtt{if}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid E\ op\ e \mid v\ op\ E \\ & \mid & \mathtt{send}\ E\ \mathtt{on}\ e \mid \mathtt{send}\ v\ \mathtt{on}\ E \mid \mathtt{receive}\ E \mid E; e \end{array}$$

Here the syntax ensures that E is an expression containing exactly one hole while e is an ordinary expression without any holes. The definition of E may be read as follows: you are allowed to evaluate an expression on its own, you may evaluate the function part of an application, you may evaluate the argument part of an application only after the function part has been fully evaluated etc. We shall write $E[e]$ for the expression obtained by replacing the hole of E with e; so for example if E is $[\,] + 4$ then $E[e]$ is $e + 4$. We dispense with the detailed definition of $E[e]$ because it is straightforward: since the hole in E never occurs in the scope of a bound variable there is no risk of variable capture.

The basic idea is to stipulate that e_1 evaluates to e_2 in one step ($e_1 \Rightarrow e_2$) if there exists E, e_{10} and e_{20} such that $e_1 = E[e_{10}]$, $e_{10} \rightarrow e_{20}$ and $e_2 = E[e_{20}]$. As an example, $(1+2)+4 \Rightarrow 3+4$ by taking E to be $[\,]+4$, e_{10} to be $1+2$ and e_{20} to be 3. Note that having $E\ op\ e$ as an evaluation context corresponds to having an inference rule

$$\frac{e_{10} \rightarrow e_{20}}{e_{10}\ op\ e_2 \rightarrow e_{20}\ op\ e_2}$$

as in Table 3.3. As already said, an advantage of using evaluation contexts is that the description of the semantics often becomes more succinct; we shall benefit from this below.

The *concurrent semantics* operates on a finite pool, PP, of processes and a finite set, CP, of channels. The set of channels keeps track of those channels that have been generated so far; this allows us to evaluate $\texttt{channel}_\pi$ by generating a new channel that has never been used before. The pool of processes both keeps track of the processes spawned so far and of the expression residing on each process; this allows us to allocate new processes when an expression is spawned and to let distinct processes communicate with one another. Formally

$$p \in \mathbf{Proc} \quad \text{processes}$$

is defined by

$$p ::= \texttt{proc1} \mid \texttt{proc2} \mid \cdots$$

and we take

$$\begin{aligned} CP &\in \mathcal{P}_{\text{fin}}(\mathbf{Chan}) \\ PP &\in \mathbf{Proc} \rightarrow_{\text{fin}} \mathbf{Exp} \end{aligned}$$

where it will be the case that each $PP(p)$ is a closed expression. We shall write $PP[p : e]$ for PP' given by $dom(PP') = dom(PP) \cup \{p\}$, $PP'(p) = e$ and $PP'(q) = PP(q)$ for $q \neq p$.

The concurrent semantics is specified in Table 5.15. It is a Structural Operational Semantics that axiomatises the relation $CP_1, PP_1 \Rightarrow CP_2, PP_2$ for when one configuration CP_1, PP_1 in one step evolves into another configuration CP_2, PP_2. Thanks to the use of evaluation contexts all clauses can

[*seq*]	$CP, PP[p : E[e_1]] \Rightarrow CP, PP[p : E[e_2]]$ if $e_1 \rightarrow e_2$
[*chan*]	$CP, PP[p : E[\texttt{channel}_\pi]] \Rightarrow CP \cup \{ch\}, PP[p : E[ch]]$ if $ch \notin CP$
[*spawn*]	$CP, PP[p : E[\texttt{spawn}\ e_0]] \Rightarrow CP, PP[p : E[()]][p_0 : e_0]$ if $p_0 \notin dom(PP) \cup \{p\}$
[*comm*]	$CP, PP[p_1 : E_1[\texttt{send}\ v\ \texttt{on}\ ch]][p_2 : E_2[\texttt{receive}\ ch]] \Rightarrow CP, PP[p_1 : E_1[()]][p_2 : E_2[v]]$ if $p_1 \neq p_2$

Table 5.15: The concurrent semantics.

be written succinctly. The clause [*seq*] incorporates the sequential semantics into the concurrent semantics (and corresponds to the discussion of $e_1 \Rightarrow e_2$ above). The clause [*chan*] takes care of the allocation of channels: $\texttt{channel}_\pi$ is replaced by a fresh channel identifier *ch*. The clause [*spawn*] generates a new process, initialises it to the expression to be executed, and replaces the **spawn**-construct by the unit value. Finally, the clause [*comm*] allows synchronous communication between distinct processes: the **receive**-construct is replaced by the value being sent and the **send**-construct is replaced by the unit value.

Annotated types. The purpose of the *Communication Analysis* is to determine the communication behaviour of each expression: what channels will be allocated, what type of entities will be sent and received over channels, and what is the behaviour of the processes being generated. Furthermore, we are interested in recording the temporal order (or "causality") among these actions: what takes place before what. There are several ways to formalise this and we shall choose one where the inference system is not overly complicated (but where instead the inference algorithm presents some challenges).

To formalise our idea we introduce the following syntactic categories:

$\widehat{\tau}$	$\in$	$\mathbf{Type}_{\mathsf{CA}}$	types
φ	$\in$	$\mathbf{Ann}_{\mathsf{CA}}$	annotations (or behaviours)
r	$\in$	$\mathbf{Reg}_{\mathsf{CA}}$	regions
$\widehat{\sigma}$	$\in$	$\mathbf{Scheme}_{\mathsf{CA}}$	type schemes

Types are much as before but extended with a unit type (for the unit value) and a type for channels:

$$\widehat{\tau} ::= \alpha \mid \texttt{bool} \mid \texttt{int} \mid \texttt{unit} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \mid \widehat{\tau}\ \texttt{chan}\ r$$

Type variables, $\alpha \in \mathbf{TVar}$, are as before.

Behaviours differ from the annotations and effects used so far in that we shall not merely use union for combining them:

$$\begin{array}{rcl} \varphi & ::= & \beta \mid \Lambda \mid \varphi_1;\varphi_2 \mid \varphi_1 + \varphi_2 \mid \mathsf{rec}\beta.\varphi \\ & \mid & \widehat{\tau}\ \mathsf{chan}\ r \mid \mathsf{spawn}\ \varphi \mid r!\widehat{\tau} \mid r?\widehat{\tau} \end{array}$$

Behaviour variables, $\beta \in \mathbf{AVar}$, are as before and the behaviour Λ is used for atomic actions that do not involve communication; in a sense it corresponds to the empty set in previous annotations although it will be more intuitive to think of it as the empty string in regular expressions or as the silent action in process calculi. The behaviour $\varphi_1;\varphi_2$ says that φ_1 takes place before φ_2 whereas $\varphi_1 + \varphi_2$ indicates a choice between φ_1 and φ_2 (as will be the case for the conditional); this is reminiscent of constructs in regular expressions as well as in process algebras. The construct $\mathsf{rec}\beta.\varphi$ indicates a recursive behaviour that acts as given by φ except that any occurrence of β stands for $\mathsf{rec}\beta.\varphi$ itself; it is typically used whenever there is explicit or implicit recursion in the program.

The behaviour $\widehat{\tau}\ \mathsf{chan}\ r$ indicates that a new channel has been allocated over which entities of type $\widehat{\tau}$ can be communicated; the region r indicates the set of program points (see below) where the creation could have taken place. The behaviour $\mathsf{spawn}\ \varphi$ indicates that a new process has been generated and that it operates as described by φ. Next $r!\widehat{\tau}$ indicates that a value is sent over a channel of type $\widehat{\tau}\ \mathsf{chan}\ r$ and $r?\widehat{\tau}$ indicates that a value is received over a channel of that type; this is reminiscent of constructs in most process algebras (in particular CSP).

Regions have a bit more structure than in Subsection 5.4.3 because now we shall also be able to take the union of regions:

$$r ::= \{\pi\} \mid \varrho \mid r_1 \cup r_2 \mid \emptyset$$

Region variables, $\varrho \in \mathbf{RVar}$, are as before. In Subsection 5.4.3 the static regions were used to ensure that at run-time data would be allocated in the same dynamic regions; here regions are used to identify the set $\{\pi_1, \cdots, \pi_n\}$ of program points where a channel might have been created. (So program points now play the role of region names.) However, these regions will not appear explicitly in the syntax of expressions, types or behaviours.

Type schemes have the form

$$\widehat{\sigma} ::= \forall(\zeta_1, \cdots, \zeta_n).\widehat{\tau}$$

where $\zeta_1, \cdots, \zeta_n$ is a (possibly empty) list of type variables, behaviour variables and region variables; if the list is empty we simply write $\widehat{\tau}$ for $\forall().\widehat{\tau}$.

Example 5.35 Returning to the program of Example 5.34, the **node** function is intended to have the type schema

$$\forall {}'\mathsf{a}, {}'\mathsf{b}, {}'1, {}''1, {}''2.\ ({}'\mathsf{a} \xrightarrow{'1} {}'\mathsf{b}) \xrightarrow{\Lambda} ({}'\mathsf{a}\ \mathtt{chan}\ {}''1) \xrightarrow{\Lambda} ({}'\mathsf{b}\ \mathtt{chan}\ {}''2) \xrightarrow{\varphi} \mathtt{unit}$$
$$\text{where } \varphi = \mathsf{spawn}(\mathsf{rec}\ {}'2.\ ({}''1?{}'\mathsf{a};\ {}'1;\ {}''2!{}'\mathsf{b};\ {}'2))$$

corresponding to the function argument having type ${}'\mathsf{a} \xrightarrow{'1} {}'\mathsf{b}$, the input channel having type ${}'\mathsf{a}\ \mathtt{chan}\ {}''1$ and the output channel having type ${}'\mathsf{b}\ \mathtt{chan}\ {}''2$. When supplied with these arguments the **node** function will spawn a process that recursively will read on the input channel, execute the function supplied as its parameter, and write on the output channel – this is exactly what is expressed by φ.

The **pipe** function is intended to have the type schema

$$\forall {}'\mathsf{a}, {}'1, {}''1, {}''2.(({}'\mathsf{a} \xrightarrow{'1} {}'\mathsf{a})\ \mathtt{list} \xrightarrow{\Lambda} ({}'\mathsf{a}\ \mathtt{chan}\ ({}''1 \cup \{\mathsf{C}\}))$$
$$\xrightarrow{\Lambda} ({}'\mathsf{a}\ \mathtt{chan}\ {}''2) \xrightarrow{\varphi'} \mathtt{unit}$$
$$\text{where } \varphi' = \mathsf{rec}\ {}'2.\ (\mathsf{spawn}(\mathsf{rec}\ {}'3.\ (({}''1 \cup \{\mathsf{C}\})?{}'\mathsf{a};\ \Lambda;\ {}''2!{}'\mathsf{a};\ {}'3))$$
$$+\ {}'\mathsf{a}\ \mathsf{chan}\ \mathsf{C};\ \mathsf{spawn}(\mathsf{rec}\ {}'4.\ (({}''1 \cup \{\mathsf{C}\})?{}'\mathsf{a};\ {}'1;\ \mathsf{C}!{}'\mathsf{a};\ {}'4));\ {}'2)$$

where the first summand in the body of φ' corresponds to the **then**-branch where **node** is called with the identity function (which has behaviour Λ) and the second to the **else**-branch of the conditional. Here we see that a channel is created, a process is spawned and then the overall behaviour recurses. We shall return to these types schemes after presenting the typing rules. ■

Typing judgements. The typing judgements for the Communication Analysis will be of the form

$$\widehat{\Gamma} \vdash_{\mathsf{CA}} e : \widehat{\sigma}\ \&\ \varphi$$

where the type environment $\widehat{\Gamma}$ maps variables to type schemes (or types), $\widehat{\sigma}$ is the type scheme (or type) for the expression e, and φ is the behaviour that may arise during evaluation of e. The analysis is specified by the axiom and rules of Tables 5.16 and 5.17 and have many points in common with those we have seen before; the differences are explained below.

The axioms [*con*] and [*var*] for constants and variables differ from the similar axioms in Table 5.10 in that Λ is used instead of $\emptyset$. A similar remark holds for the two rules [*fn*] and [*fun*] for functions; note that [*fun*] does not explicitly demand φ_0 to be of the form $\mathsf{rec}\beta'.\varphi'$ although this will frequently be necessary in order to satisfy the demands (using the ordering on behaviours to be introduced below). In the rule [*app*] for function application we now use sequencing to express that we first evaluate the function part, then the argument and finally the body of the function. In the rule [*if*] for conditional we additionally use choice to express that only one of the **then**- and **else**-branches are taken. Then the rules [*let*] and [*op*] should be straightforward.

$$[con] \quad \widehat{\Gamma} \vdash_{\mathsf{CA}} c : \tau_c \;\&\; \Lambda$$

$$[var] \quad \widehat{\Gamma} \vdash_{\mathsf{CA}} x : \widehat{\sigma} \;\&\; \Lambda \qquad \text{if } \widehat{\Gamma}(x) = \widehat{\sigma}$$

$$[fn] \quad \frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{CA}} e_0 : \widehat{\tau}_0 \;\&\; \varphi_0}{\widehat{\Gamma} \vdash_{\mathsf{CA}} \mathtt{fn}_\pi\ x \mathtt{=>} e_0 : \widehat{\tau}_x \xrightarrow{\varphi_0} \widehat{\tau}_0 \;\&\; \Lambda}$$

$$[fun] \quad \frac{\widehat{\Gamma}[f \mapsto \widehat{\tau}_x \xrightarrow{\varphi_0} \widehat{\tau}_0][x \mapsto \widehat{\tau}_x] \vdash_{\mathsf{CA}} e_0 : \widehat{\tau}_0 \;\&\; \varphi_0}{\widehat{\Gamma} \vdash_{\mathsf{CA}} \mathtt{fun}_\pi\ f\ x \mathtt{=>} e_0 : \widehat{\tau}_x \xrightarrow{\varphi_0} \widehat{\tau}_0 \;\&\; \Lambda}$$

$$[app] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CA}} e_1 : \widehat{\tau}_2 \xrightarrow{\varphi_0} \widehat{\tau}_0 \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{CA}} e_2 : \widehat{\tau}_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{CA}} e_1\ e_2 : \widehat{\tau}_0 \;\&\; \varphi_1; \varphi_2; \varphi_0}$$

$$[if] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CA}} e_0 : \mathtt{bool} \;\&\; \varphi_0 \quad \widehat{\Gamma} \vdash_{\mathsf{CA}} e_1 : \widehat{\tau} \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{CA}} e_2 : \widehat{\tau} \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{CA}} \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \widehat{\tau} \;\&\; \varphi_0; (\varphi_1 + \varphi_2)}$$

$$[let] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CA}} e_1 : \widehat{\sigma}_1 \;\&\; \varphi_1 \quad \widehat{\Gamma}[x \mapsto \widehat{\sigma}_1] \vdash_{\mathsf{CA}} e_2 : \widehat{\tau}_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{CA}} \mathtt{let}\ x \mathtt{=} e_1\ \mathtt{in}\ e_2 : \widehat{\tau}_2 \;\&\; \varphi_1; \varphi_2}$$

$$[op] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CA}} e_1 : \tau^1_{op} \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{CA}} e_2 : \tau^2_{op} \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{CA}} e_1\ op\ e_2 : \tau_{op} \;\&\; \varphi_1; \varphi_2; \Lambda}$$

Table 5.16: Communication Analysis (part 1).

The axiom [*chan*] for channel creation makes sure to record the program point in the type as well as the behaviour, the rule [*spawn*] encapsulates the behaviour of the spawned process in the behaviour of the construct itself and the rules [*send*] and [*receive*] for sending and receiving values over channels indicate the order in which the arguments are evaluated and then produce the behaviour for the axiom taken. The rules [*seq*] and [*ch*] are straightforward and the rules [*gen*] and [*ins*] are much as in Table 5.10.

Example 5.36 Consider the following example program:

```
let ch = channel_A
in (send 1 on ch; send true on ch)
```

Intuitively, this program should be rejected because the channel ch is used for communicating values of two different types thereby violating type safety. To see that the program is indeed rejected, first observe that the rule [*chan*] gives:

$$[\,] \vdash_{\mathsf{CA}} \mathtt{channel}_{\mathsf{A}} : {}'\mathsf{a}\ \mathsf{chan}\ \{\mathsf{A}\} \;\&\; {}'\mathsf{a}\ \mathsf{chan}\ \{\mathsf{A}\}$$

Then the generalisation rule [*gen*] does *not* allow us to generalise over the type variable 'a since it occurs in the behaviour 'a chan {A}; it follows that

$$[chan] \quad \widehat{\Gamma} \vdash_{\mathsf{CA}} \texttt{channel}_{\pi} : \widehat{\tau}\ \texttt{chan}\ \{\pi\}\ \&\ \widehat{\tau}\ \texttt{chan}\ \{\pi\}$$

$$[spawn] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CA}} e_0 : \widehat{\tau}_0\ \&\ \varphi_0}{\widehat{\Gamma} \vdash_{\mathsf{CA}} \texttt{spawn}\ e_0 : \texttt{unit}\ \&\ \texttt{spawn}\ \varphi_0}$$

$$[send] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CA}} e_1 : \widehat{\tau}\ \&\ \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{CA}} e_2 : \widehat{\tau}\ \texttt{chan}\ r_2\ \&\ \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{CA}} \texttt{send}\ e_1\ \texttt{on}\ e_2 : \texttt{unit}\ \&\ \varphi_1; \varphi_2; r_2!\widehat{\tau}}$$

$$[receive] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CA}} e_0 : \widehat{\tau}\ \texttt{chan}\ r_0\ \&\ \varphi_0}{\widehat{\Gamma} \vdash_{\mathsf{CA}} \texttt{receive}\ e_0 : \widehat{\tau}\ \&\ \varphi_0; r_0?\widehat{\tau}}$$

$$[seq] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CA}} e_1 : \widehat{\tau}_1\ \&\ \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{CA}} e_2 : \widehat{\tau}_2\ \&\ \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{CA}} e_1; e_2 : \widehat{\tau}_2\ \&\ \varphi_1; \varphi_2}$$

$$[ch] \quad \widehat{\Gamma} \vdash_{\mathsf{CA}} ch : \widehat{\tau}\ \texttt{chan}\ r\ \&\ \Lambda \qquad \text{if } \widehat{\tau}\ \texttt{chan}\ r = \widehat{\Gamma}(ch)$$

$$[sub] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CA}} e : \widehat{\tau}\ \&\ \varphi}{\widehat{\Gamma} \vdash_{\mathsf{CA}} e : \widehat{\tau}'\ \&\ \varphi'} \qquad \text{if } \widehat{\tau} \leq \widehat{\tau}' \text{ and } \varphi \sqsubseteq \varphi'$$

$$[gen] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CA}} e : \widehat{\tau}\ \&\ \varphi}{\widehat{\Gamma} \vdash_{\mathsf{CA}} e : \forall(\zeta_1, \cdots, \zeta_n).\widehat{\tau}\ \&\ \varphi}$$
$$\text{if } \zeta_1, \cdots, \zeta_n \text{ do not occur free in } \widehat{\Gamma} \text{ and } \varphi$$

$$[ins] \quad \frac{\widehat{\Gamma} \vdash_{\mathsf{CA}} e : \forall(\zeta_1, \cdots, \zeta_n).\widehat{\tau}\ \&\ \varphi}{\widehat{\Gamma} \vdash_{\mathsf{CA}} e : (\theta\ \widehat{\tau})\ \&\ \varphi}$$
$$\text{if } \theta \text{ has } dom(\theta) \subseteq \{\zeta_1, \cdots, \zeta_n\}$$

Table 5.17: Communication Analysis (part 2).

ch can "only" be given the type 'a chan {A} and therefore the program fails to type check (because 'a cannot be equal to `int` and `bool` at the same time).

However, if the generalisation rule [*gen*] were only to require that $\zeta_1, \cdots, \zeta_n$ do not occur free in $\widehat{\Gamma}$, then it would be possible to give the channel ch the type schema ∀'a. 'a chan {A}; as a consequence we would be able to give types to the two occurrences of **send** in the body of the `let` construct and the type system would not be semantically correct. ■

The rule [*sub*] for *subeffecting* and *subtyping* looks like the one in Table 5.10 but involves a few subtleties. The ordering $\widehat{\tau} \leq \widehat{\tau}'$ on types is given by

$$\widehat{\tau} \leq \widehat{\tau} \qquad \frac{\widehat{\tau}_1' \leq \widehat{\tau}_1 \quad \widehat{\tau}_2 \leq \widehat{\tau}_2' \quad \varphi \sqsubseteq \varphi'}{\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \leq \widehat{\tau}_1' \xrightarrow{\varphi'} \widehat{\tau}_2'} \qquad \frac{\widehat{\tau} \leq \widehat{\tau}' \quad \widehat{\tau}' \leq \widehat{\tau} \quad r \subseteq r'}{\widehat{\tau}\ \texttt{chan}\ r \leq \widehat{\tau}'\ \texttt{chan}\ r'}$$

$\varphi \sqsubseteq \varphi$	$\dfrac{\varphi_1 \sqsubseteq \varphi_2 \quad \varphi_2 \sqsubseteq \varphi_3}{\varphi_1 \sqsubseteq \varphi_3}$
$\dfrac{\varphi_1 \sqsubseteq \varphi_2 \quad \varphi_3 \sqsubseteq \varphi_4}{\varphi_1 ; \varphi_3 \sqsubseteq \varphi_2 ; \varphi_4}$	$\dfrac{\varphi_1 \sqsubseteq \varphi_2 \quad \varphi_3 \sqsubseteq \varphi_4}{\varphi_1 + \varphi_3 \sqsubseteq \varphi_2 + \varphi_4}$
$\dfrac{\varphi_1 \sqsubseteq \varphi_2}{\mathsf{spawn}\ \varphi_1 \sqsubseteq \mathsf{spawn}\ \varphi_2}$	$\dfrac{\varphi_1 \sqsubseteq \varphi_2}{\mathsf{rec}\beta.\varphi_1 \sqsubseteq \mathsf{rec}\beta.\varphi_2}$
$\varphi_1 ; (\varphi_2 ; \varphi_3) \sqsubseteq (\varphi_1 ; \varphi_2) ; \varphi_3$	$(\varphi_1 ; \varphi_2) ; \varphi_3 \sqsubseteq \varphi_1 ; (\varphi_2 ; \varphi_3)$
$(\varphi_1 + \varphi_2) ; \varphi_3 \sqsubseteq (\varphi_1 ; \varphi_3) + (\varphi_2 ; \varphi_3)$	$(\varphi_1 ; \varphi_3) + (\varphi_2 ; \varphi_3) \sqsubseteq (\varphi_1 + \varphi_2) ; \varphi_3$
$\varphi \sqsubseteq \Lambda ; \varphi \qquad \Lambda ; \varphi \sqsubseteq \varphi$	$\varphi \sqsubseteq \varphi ; \Lambda \qquad \varphi ; \Lambda \sqsubseteq \varphi$
$\varphi_1 \sqsubseteq \varphi_1 + \varphi_2 \qquad \varphi_2 \sqsubseteq \varphi_1 + \varphi_2$	$\varphi + \varphi \sqsubseteq \varphi$
$\mathsf{rec}\beta.\varphi \sqsubseteq \varphi[\beta \mapsto \mathsf{rec}\beta.\varphi]$	$\varphi[\beta \mapsto \mathsf{rec}\beta.\varphi] \sqsubseteq \mathsf{rec}\beta.\varphi$
$\dfrac{\widehat{\tau} \leq \widehat{\tau}' \quad \widehat{\tau}' \leq \widehat{\tau} \quad r \subseteq r'}{\widehat{\tau}\ \mathsf{chan}\ r \sqsubseteq \widehat{\tau}'\ \mathsf{chan}\ r'}$	
$\dfrac{r_1 \subseteq r_2 \quad \widehat{\tau}_1 \leq \widehat{\tau}_2}{r_1 ! \widehat{\tau}_1 \sqsubseteq r_2 ! \widehat{\tau}_2}$	$\dfrac{r_1 \subseteq r_2 \quad \widehat{\tau}_2 \leq \widehat{\tau}_1}{r_1 ? \widehat{\tau}_1 \sqsubseteq r_2 ? \widehat{\tau}_2}$

Table 5.18: Ordering on behaviours.

and is similar to the definition in Subsection 5.4.1: $\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$ is contravariant in $\widehat{\tau}_1$ but covariant in φ and $\widehat{\tau}_2$, and $\widehat{\tau}\ \mathsf{chan}\ r$ is both covariant in $\widehat{\tau}$ (for when a value is sent) and contravariant in $\widehat{\tau}$ (for when a value is received) and it is covariant in r. The ordering $r \subseteq r'$ means that r is "a subset of" of r' (modulo UCAI) just as what was the case for the effects in Section 5.4. Finally, the ordering $\varphi \sqsubseteq \varphi'$ on behaviours in more complex than before because of the rich structure possessed by behaviours. The definition is given in Table 5.18 and will be explained below. Since the syntactic categories of types and behaviours are mutually recursive also the definitions of $\widehat{\tau} \leq \widehat{\tau}'$ and $\varphi \sqsubseteq \varphi'$ need to be interpreted recursively.

The axiomatisation of $\varphi \sqsubseteq \varphi'$ in Table 5.18 ensures that we obtain a preorder that is a congruence with respect to the operations for combining behaviours. Furthermore, sequencing is an associative operation with Λ as identity and we have a distributive law with respect to choice. It follows that choice is associative and commutative. Next the axioms for recursion allows us to unfold the rec-construct. The final three rules clarify how behaviours depend upon types and regions: $\widehat{\tau}\ \mathsf{chan}\ r$ is both contravariant and covariant in $\widehat{\tau}$ and

is covariant in r (just as was the case for the type $\widehat{\tau}$ **chan** r); $r!\widehat{\tau}$ is covariant in both r and $\widehat{\tau}$ (because a value is sent) whereas $r?\widehat{\tau}$ is covariant in r and contravariant in $\widehat{\tau}$ (because a value is received). There is no explicit law for renaming bound behaviour variables as we shall regard $\mathsf{rec}\beta.\varphi$ as being equal to $\mathsf{rec}\beta'.\varphi'$ whenever they are α-equivalent.

The Communication Analysis in Tables 5.16 and 5.17 differs from the Region Inference Analysis of Tables 5.12 and 5.13 in that there is no analogue of the rule [*region*] where an *Observe* function is used to reduce the annotation to what is visible from the outside. The reason is that the Communication Analysis is intended to record *all* the side effects (in the form of communications) that take place during computation.

Example 5.37 Returning to the program of Example 5.35 we can now verify that the **node** function can be given the type:

$$('\mathsf{a} \xrightarrow{'1} '\mathsf{b}) \xrightarrow{\Lambda} ('\mathsf{a}\ \mathbf{chan}\ ''1) \xrightarrow{\Lambda} ('\mathsf{b}\ \mathbf{chan}\ ''2) \xrightarrow{\varphi} \mathbf{unit}$$
$$\text{where } \varphi = \mathsf{spawn}(\mathsf{rec}\ '2.\ (''1?'\mathsf{a};\ '1;\ ''2!'\mathsf{b};\ '2))$$

We use the rules [*fun*] and [*sub*] of Tables 5.16 and 5.17 together with the axiom $\varphi[\beta \mapsto \mathsf{rec}\beta.\varphi] \sqsubseteq \mathsf{rec}\beta.\varphi$ of Table 5.18. Then the rule [*gen*] allows us to obtain the type schema given in Example 5.35.

Turning to the **pipe** function we first note that it can be given a type of the form

$$(('\mathsf{a} \xrightarrow{'1} '\mathsf{a})\ \mathbf{list}) \xrightarrow{\Lambda} ('\mathsf{a}\ \mathbf{chan}\ (''1 \cup \{\mathsf{C}\})) \xrightarrow{\Lambda} ('\mathsf{a}\ \mathbf{chan}\ ''2) \xrightarrow{\varphi'} \mathbf{unit}$$

where the regions for the input and local channels are "merged" because they can both be used as input channels in a call to **pipe** whereas the region for the output channel is always kept separate. The behaviour φ' is of the form

$$\begin{aligned}&\mathsf{rec}\ '2.\ (\mathsf{spawn}(\mathsf{rec}\ '3.\ ((''1 \cup \{\mathsf{C}\})?'\mathsf{a};\ \Lambda;\ ''2!'\mathsf{a};\ '3))\\ &\qquad + '\mathsf{a}\ \mathbf{chan}\ \mathsf{C};\ \mathsf{spawn}(\mathsf{rec}\ '4.\ ((''1 \cup \{\mathsf{C}\})?'\mathsf{a};\ '1;\ \mathsf{C}!'\mathsf{a};'4));\ '2)\end{aligned}$$

because in the **then**-branch the input channel for **node** has type $'\mathsf{a}\ \mathbf{chan}\ (''1 \cup \{\mathsf{C}\})$ and the output channel has type $'\mathsf{a}\ \mathbf{chan}\ ''2$, whereas in the **else**-branch the input channel for **node** has type $'\mathsf{a}\ \mathbf{chan}\ (''1 \cup \{\mathsf{C}\})$ and the output channel has type $'\mathsf{a}\ \mathbf{chan}\ \{\mathsf{C}\}$ (as well as $'\mathsf{a}\ \mathbf{chan}\ (''1 \cup \{\mathsf{C}\})$). The rule [*gen*] allows us to obtain the type schema displayed in Example 5.35. ∎

Concluding Remarks

Control Flow Analysis. The literature [49, 50, 71, 18] contains many formulations of non-standard type systems aiming at performing *Control*

Flow Analysis. The formulation presented in Section 5.1 uses a particularly simple set of techniques where types are annotated but there is no additional effect component nor is there any coverage of polymorphism or subtyping. Although there is no explicit clause for subeffecting in the manner of Section 5.4, we regard the formulation as a *subeffecting analysis* because the rules for function abstraction allow us to increase the annotation on the function arrow in much the same way as is the case in subeffecting (and in much more restricted ways than holds for subtyping; see also Exercise 5.13). References for type systems with subtyping include [59, 58, 107] as well as the more advanced [83, 160, 161] that also deal with polymorphism. To allow a general treatment of subtyping, these papers generally demand constraints to be an explicit part of the inference system unlike what was done in Section 5.1. Indeed, the formulation of Section 5.1 allows only *shape conformant subtyping*, where the underlying type system does *not* make use of any form of subtyping, and is thus somewhat simpler than *atomic subtyping*, where an ordering is imposed upon base types, and *general subtyping*, where an ordering may be imposed between arbitrary types.

The semantic correctness of the Control Flow Analysis established in Section 5.2 is expressed as a subject reduction result but formulated for a Natural Semantics [91] (rather than a Structural Operational Semantics); this approach to semantic correctness has a rather long history [114, 116, 184]. The *Moore family* result about the set of typings is inspired by the ideas of [114] and is included so as to stress the fundamental role played by partial orders in all of the approaches to program analysis considered in this book; it also relates to the study of principal types in type systems.

The development of a syntactically sound and complete algorithm for the Control Flow Analysis in Section 5.3 is based on the ideas in [90, 183, 167, 168, 169]; the call-tracking analysis of Mini Project 5.1 is based on [169]. The basic idea is to ensure that the algorithm operates on a *free algebra* by restricting annotations to be annotation variables only (the concept of *simple types*) and by recording a *set of constraints* for the meaning of the annotation variables; in our case this is particularly straightforward because the Control Flow Analysis does not deal with polymorphism. Our development differs somewhat from that of [58, 107] that deal with the more advanced notions of atomic subtyping and general subtyping. A different approach, not studied here, would be to dispense with simple types and constraints and instead use techniques for unifying types in a non-free algebra [159].

Restricted effects. The Type and Effect Systems presented in Section 5.4 all share the important property (also holding for the type system in Section 5.1) that *no type information* is recorded in the effects and that the shape of the type information *cannot be influenced* by the effects. All systems included a proper effect component and thereby illustrated the diversity of effects; some pioneering papers in Type and Effect Systems are

[88, 102, 89, 90]. At the same time we illustrated a number of design considerations to be taken into account when devising a Type and Effect System: whether or not to incorporate subeffecting, subtyping, polymorphism, polymorphic recursion, whether or not types are allowed to be influenced by effects (which is not the case in Sections 5.4 and 5.1), and whether or not constraints are an explicit part of the inference system (as is implicitly the case in Subsection 5.4.3). However, it would be incorrect to surmise that the selection of components are inherently linked to the example analysis where they were illustrated. Rather, the techniques needed for semantic correctness and for syntactic soundness and completeness depend heavily on the particular selection of components; some are straightforward to deal with whereas others are beyond state-of-the-art.

The Side Effect Analysis presented in Subsection 5.4.1 illustrated the use of subeffecting and subtyping, but did not incorporate polymorphism, there were no constraints in the inference system, and the effects did not influence the types. This system is sufficiently simple that semantic soundness may be established using the techniques of Section 5.2. If the rule for subtyping was omitted then also the techniques developed in Section 5.3 would suffice for obtaining a sound and complete inference algorithm. The presence of the rule for subtyping naturally leads to a *two stage* implementation process: first the underlying types are inferred and next the constraints on effects (or annotations) are determined [169, 174]. This works because we restrict ourselves to *shape conformant subtyping* where effects *do not influence* the type information. However, adding polymorphism to this development would dramatically increase the complexity of the development (see below).

The Exception Analysis presented in Subsection 5.4.2 illustrated the use of subeffecting, subtyping and polymorphism, but there were no constraints in the inference system, and the effects did not influence the types. Semantic soundness is a bit more complex than in Section 5.2 because of the polymorphism but the techniques of [173, 167, 168, 15] suffice. For the development of a syntactically sound and complete inference algorithm one may take the two stage approach described above [169, 174]; as before, it works because we restrict ourselves to shape conformant subtyping where effects do not influence the type information. Alternatively, one may use more powerful techniques [183, 167, 168, 127] that even allow to include type information inside effects; this amounts to an extension of the approach of Section 5.3 and will be explained below.

The Region Inference analysis presented in Subsection 5.4.3 illustrated the use of polymorphic recursion as far as the effects are concerned, there were implicitly constraints in the inference system (via the dot notation on function arrows), but still the effects cannot influence the types. The presentation is mainly based on [175] but adapted to the FUN language and the style of presentation used elsewhere in this chapter. To obtain effects that are as

small as possible the inference system uses "effect masking" (developed in [102, 167, 168]) for removing internal components of the effect: effect components that only deal with regions that are not externally visible. Semantic correctness of the inference system can be shown using the approach of [176]. For the development of a syntactically sound inference algorithm one may once more take the two stage approach described above; the first stage (ordinary type inference) is standard and the second stage is considered in [174] where algorithm $\mathcal{S}$ generates effect and region variables and algorithm $\mathcal{R}$ deals with the complications due to polymorphic recursion (for effects and regions only). The inference algorithm is proved syntactically sound but is known not to be syntactically complete; indeed, obtaining an algorithm that is syntactically sound as well as complete, seems beyond state-of-the-art.

General effects. *One way* to make effects more expressive is to *allow type information* inside the effects so that the shape of the type information *can be influenced* by the effects. This idea occurred already in [183, 167, 168] for an extended Side Effect Analysis making use of polymorphism and subeffecting (but not subtyping); this work attempted to "generalise" previous work based on the idea of expansive expressions and imperative versus applicative type variables [173, 166]. As already indicated, semantic soundness amounts to an extension of the techniques of Section 5.2 as presented in [173, 167, 168, 15].

The two stage approach no longer works for obtaining an inference algorithm because the effects are used to control the shape of the underlying types in the form of which type variables are included in a polymorphic type. This suggests extending the techniques of Section 5.3 in that special care needs to be taken when deciding the variables over which to generalise when constructing a polymorphic type. The main idea is that the algorithm needs to consult the constraints in order to determine a larger set of forbidden variables than those directly occurring in the type environment or the effect; this can be formulated as a *downwards closure* with respect to the constraint set [183, 127] or by taking a *principal solution* of the constraints into account [167, 168].

Adding subtyping to this development dramatically increases the complexity of the development. The integration of shape conformant subtyping, polymorphism and subeffecting is done in [127, 134, 15] that establishes semantic soundness and develops an inference algorithm that is proved syntactically sound; extensions of this development incorporate a syntactic completeness result (see the discussion of [13] below). This work went a long way towards integrating the techniques for polymorphism and subeffecting (but no subtyping) from Effect Systems [183, 167, 168] with the techniques for polymorphism and subtyping (but no effects) from Type Systems [83, 160, 161].

Another way to make effects more expressive is to let them contain information about the *temporal order* and causality of actions, rather than just

being an unordered set of possibilities. In Section 5.5 we considered the task of extracting *behaviours* (reminiscent of terms in a process algebra) from programs in Concurrent ML by means of a Type and Effect System; here effects (the behaviours) have structure, they may influence the type information, there are no explicit constraints in the inference system (although there are in more advanced developments [13]), and there are inference rules for subeffecting and shape conformant subtyping. These ideas first occurred in [119, 120] (not involving polymorphism) and in [131, 132, 14] (involving polymorphism); our presentation in Section 5.5 is mainly based on [131, 132] with ingredients from [119, 120]. We refer to [13] for a comprehensive account of a more ambitious development where the inference system is massaged so as to facilitate developing a syntactically sound and complete inference algorithm; this includes having explicit constraints in the inference system as is usually the case in type systems that make use of subtyping. An application to the validation of embedded systems is presented in [128].

Other developments. All of the formulations presented in this chapter have had a number of common features: to the extent that polymorphism has been incorporated it has been based on the Hindley/Milner polymorphism also found in Standard ML, there has been no subtyping involved in the underlying type system, and there has been no treatment of conjunction or disjunction types as in [19, 20, 81, 82]. Also all of the formulations have expressed *safety* properties: if a certain point is reached then certain information will hold; *liveness* properties in the form of adding annotations that indicate whether or not functions can be assumed to be total was considered in [121].

Finally, linking up with the development of Chapter 4 on Abstract Interpretation, it is possible to allow annotations to be elements of a complete lattice (that is possibly of finite height as in Monotone Frameworks) as outlined in Mini Project 5.4; s this mini project also discusses how to deal with binding time analyses (inpired by [75, 114, 123]) and security analyses (inspired by [74, 1]). In the other direction it may be profitable to describe Type and Effect Systems using the framework of Abstract Interpretation [108, 36].

Mini Project 5.5 on units and the year 2000 problem (Y2K) was inspired by [95, 46, 141].

Mini Projects

Mini Project 5.1 A Call-Tracking Analysis

Consider a Type and Effect System for Call-Tracking Analysis: it has judgements of the form

$$\widehat{\Gamma} \vdash_{\mathsf{CT}} e : \widehat{\tau} \,\&\, \varphi$$

where φ denotes the set of functions that may be called during the evaluation of e (and similarly for the annotations on function arrows).

1. Formulate an inference system with subeffecting; next add subtyping and finally add polymorphism.

Next consider the inference system with subeffecting only:

2. Modify the Natural Semantics of Table 5.4 such that semantic correctness can be stated for the analysis and prove that the result holds.
3. Devise an algorithm for Call-Tracking Analysis and prove that it is syntactically sound (and complete).

For the more ambitious: can you also deal with subtyping and/or polymorphism? ■

Mini Project 5.2 Data Structures

As in Mini Project 3.2 we shall now extend the language with more general data structures and consider how to modify the Control Flow Analysis (Table 5.2) so as to track the creation points.

Pairs. To accommodate pairs we extend the syntax as follows:

$$\begin{array}{rcl} \widehat{\tau} & ::= & \cdots \mid \widehat{\tau}_1 \times^{\varphi} \widehat{\tau}_2 \\ e & ::= & \cdots \mid \texttt{Pair}_{\pi}(e_1, e_2) \mid (\texttt{case}\ e_0\ \texttt{of}\ \texttt{Pair}(x_1, x_2)\ \texttt{=>}\ e_1) \end{array}$$

Here `Pair` is a binary constructor and the corresponding `case`-expression does not need an `or`-component as in Mini Project 3.2. As an example, consider the following program for "sorting" a pair of integers:

```
let srt = fn_X x => case x of Pair(y,z) =>
                  if y<z then x else Pair_B(z,y)
in srt(Pair_A(n,m))
```

Here the pair returned with be constructed at A if the value of `n` is smaller than the value of `m` and at B otherwise. The overall type is $\texttt{int} \times^{\{A,B\}} \texttt{int}$.

1. Modify the Control Flow Analysis of Table 5.2 to track the creation points of pairs.
2. Extend the Natural Semantics of Table 5.4 and augment the proof of semantic correctness (Theorem 5.9).
3. Extend the algorithms $\mathcal{W}_{\mathsf{CFA}}$ and $\mathcal{U}_{\mathsf{CFA}}$ and augment the proof of syntactic soundness and completeness (Theorems 5.20 and 5.21).

Lists. To accommodate lists we extend the syntax as follows:

$$\begin{array}{lcl} \widehat{\tau} & ::= & \cdots \mid \widehat{\tau}\ \texttt{list}^{\varphi} \\ e & ::= & \cdots \mid \texttt{Cons}_{\pi}(e_1, e_2) \mid \texttt{Nil}_{\pi} \mid (\texttt{case}\ e_0\ \texttt{of}\ \texttt{Cons}(x_1, x_2)\ \texttt{=>}\ e_1\ \texttt{or}\ e_2) \end{array}$$

Now perform a similar development as the one you performed for pairs.

For the more ambitious: can you give a more general treatment of algebraic types in the manner of Mini Project 3.2? ■

Mini Project 5.3 A Prototype Implementation

In this mini project we shall implement the Control Flow Analysis considered in Sections 5.1 and 5.3. As implementation language we shall choose a functional language such as Standard ML or Haskell. We can then define a suitable data type for FUN expressions as follows:

$$\begin{array}{lcl} \textsf{type}\ \textit{var} & = & \textsf{string} \\ \textsf{type}\ \textit{point} & = & \textsf{int} \\ \textsf{datatype}\ \textit{const} & = & \textsf{Num of int} \mid \textsf{True} \mid \textsf{False} \\ \textsf{datatype}\ \textit{exp} & = & \textsf{Const of}\ \textit{const} \mid \textsf{Var of}\ \textit{var} \mid \textsf{Fn of}\ \textit{point} * \textit{var} * \textit{exp} \\ & \mid & \textsf{Fun of}\ \textit{point} * \textit{var} * \textit{var} * \textit{exp} \mid \textsf{App of}\ \textit{exp} * \textit{exp} \\ & \mid & \textsf{If of}\ \textit{exp} * \textit{exp} * \textit{exp} \mid \textsf{Let of}\ \textit{var} * \textit{exp} * \textit{exp} \\ & \mid & \textsf{Op of string} * \textit{exp} * \textit{exp} \end{array}$$

Now proceed as follows:

1. Define data types for simple types and simple substitutions and implement the function $\mathcal{U}_{\mathsf{CFA}}$ of Table 5.7.
2. Define data types for simple type environments and constraint sets and implement the function $\mathcal{W}_{\mathsf{CFA}}$ of Table 5.8.
3. Define data types for types and type environments and implement a function that pretty prints the result in the manner of Subsection 5.3.4: type variables must get instantiated to `int` and annotation variables to the least solution to the constraints.

Test your implementation on selected examples. ■

Mini Project 5.4 Monotone Type Systems

Consider an instance of a *monotone structure* as defined in Subsection 3.5.2 (and motivated by the Monotone Frameworks of Section 2.3); examples include the Constant Propagation Analysis of Example 3.29.

Define an Annotated Type System with subeffecting (and possibly subtyping) for performing the analysis specified by the monotone structure (in the manner of the annotated base types of Table 1.2). Design the system such that it becomes possible to specify the following analyses:

- A *binding time analysis* where data can be static (i.e. available at compile-time) or dynamic (i.e. available at run-time), denoted S and D, respectively. Define a partial ordering by setting $\mathsf{S} \sqsubseteq \mathsf{D}$ to indicate that static data can also be used for operating upon dynamic data.
- A *security analysis* where data can be classified at several clearance levels $\mathsf{C}_1, \cdots, \mathsf{C}_k$. Define a partial ordering by setting $\mathsf{C}_1 \sqsubseteq \cdots \sqsubseteq \mathsf{C}_k$ to indicate that data at low levels of clearance can also be used at high levels of clearance.

How much of the development in Sections 5.2 and 5.3 can be adapted to this scenario? (Feel free to impose further conditions if needed, e.g. that the property space satisfies the Ascending Chain Condition.)

For the more ambitious: can you deal with both subtyping and polymorphism? ■

Mini Project 5.5 Units of Measure (and Y2K)

Even the most advanced programming languages, allowing user defined data types and enforcing strong typing, hardly ever record units of measure (for example meters, degrees Centigrade, US dollars). Each unit of measure may involve two ingredients: the *scaling factor* and the *base of origin*:

- An example of different scaling factors concerns the measure of length which can be measured in feet (FT) or in meters (M); in this case there is a simple conversion between the two units: x FT $= (x * 0.3048)$ M. A related example concerns the measure of currency which can be measured in US dollars (USD) or European currency units (EURO); here the conversion is slightly more complex: x USD $= (x * \mathit{rate})$ EURO where *rate* is the exchange rate which is likely to vary over time.
- An example of different bases of origins concerns the measure of time which can be measured relative to the birth of Christ (AD) or relative to the beginning of the 20'th Century (YY); there is a simple conversion: x YY $= (1900 + x)$ AD. The so-called year 2000 problem (Y2K) arises when there are only two digits available for x and one needs to represent a year that is not in the 20'th Century.
- An example of a measure involving both ingredients is the measure of temperature which can be measured in degrees Fahrenheit (F) and de-

grees Centigrade (C); in this case the conversion between temperatures is more complex: x F $= ((x - 32) * 5)/9$ C.

There are many reasons for why one would like to extend the programming language to be precise about the units of measurement; perhaps one needs to port the software to a new environment where a different unit of measurement is used, perhaps one is operating on data presented in a mixture of units of measurement, perhaps one wants to reduce the likelihood of incorrect computations (like adding US dollars and European currency units), or perhaps one wants to prepare for a change in data formats.

Let us assume that we are given a program that is correctly typed in some underlying type system. One approach to incorporating units into the program proceeds as follows:

1. The type system is extended to include unit annotations on numbers.
2. The program is analysed according to the extended type system; most likely a number of type errors are identified.
3. The type errors are corrected by inserting explicit conversion functions based on the above discussion.
4. It is checked that the resulting program is type correct in the extended type system.
5. Possibly data formats are changed based on the extended type information.

In this mini project we focus on the first task and for our purposes it suffices to let the syntax of the underlying types be given by:

$$\tau ::= \mathtt{num} \mid \tau \rightarrow \tau$$

A suitable extended type system might be based around the following types and annotations:

$$\begin{array}{rcl}
\hat{\tau} & ::= & \mathtt{num}^{\varphi} \mid \hat{\tau} \rightarrow \hat{\tau} \\
\varphi & ::= & \mathit{scale}\ \mathbf{BASE}\ \mathit{base} \\
\mathit{scale} & ::= & \sigma \mid \mathtt{unit} \mid \mathit{scale}.\mathit{scale} \mid \mathit{scale}^{-1} \mid \\
 & & \mathtt{feet} \mid \mathtt{meter} \mid \mathtt{usd} \mid \mathtt{euro} \mid \mathtt{year} \mid \mathtt{kelvin} \mid \cdots \\
\mathit{base} & ::= & \beta \mid \mathtt{none} \mid \mathtt{ad} \mid \mathtt{20th} \mid \mathtt{freezing} \mid \cdots
\end{array}$$

We model meters (M) as $\mathtt{meter}\,\mathbf{BASE}\,\mathtt{none}$ and feet (FT) as $\mathtt{feet}\,\mathbf{BASE}\,\mathtt{none}$; US dollars (USD) as $\mathtt{usd}\,\mathbf{BASE}\,\mathtt{none}$ and European currency units (EURO) as $\mathtt{euro}\,\mathbf{BASE}\,\mathtt{none}$; years relative to the birth of Christ (AD) as $\mathtt{year}\,\mathbf{BASE}\,\mathtt{ad}$ and years relative to the 20'th Century (YY) as $\mathtt{year}\,\mathbf{BASE}\,\mathtt{20th}$; degrees

Centigrade (C) as `kelvin BASE freezing` etc. "Ordinary" numbers without units now have the annotation `unit BASE none`.

One way to associate units with numbers is to use functions like

$$\texttt{asC} : \texttt{num}^{\texttt{unit BASE none}} \rightarrow \texttt{num}^{\texttt{kelvin BASE freezing}}$$

so that `asC 7` denotes 7 degrees Centigrade. To compute with numbers having units we shall give the arithmetic operations for multiplication and division the following polymorphic types:

$$\begin{array}{lll} * & : & \forall \sigma_1, \sigma_2, \sigma[\text{with } \sigma = \sigma_1.\sigma_2] : \\ & & \texttt{num}^{\sigma_1\ \texttt{BASE none}} \rightarrow \texttt{num}^{\sigma_2\ \texttt{BASE none}} \rightarrow \texttt{num}^{\sigma\ \texttt{BASE none}} \\ \\ / & : & \forall \sigma_1, \sigma_2, \sigma[\text{with } \sigma = \sigma_1.(\sigma_2^{-1})] : \\ & & \texttt{num}^{\sigma_1\ \texttt{BASE none}} \rightarrow \texttt{num}^{\sigma_2\ \texttt{BASE none}} \rightarrow \texttt{num}^{\sigma\ \texttt{BASE none}} \end{array}$$

This reflects the fact that multiplication and division works on relative units of measurement (i.e. having no base). One needs to be careful about the the laws imposed upon annotations so as to ensure that e.g. `meter.feet` is treated in the same way as `feet.meter` in accordance with the usual conventions of Physics; this amounts to an axiomatisation that is more refined than the UCAI axiomatisation.

Turning to the operations for addition and substraction:

$$\begin{array}{lll} + & : & \forall \sigma, \beta_1, \beta_2, \beta \left[\text{with } \beta = \left\{ \begin{array}{ll} \beta_1 & \text{if } \beta_2 = \textbf{none} \\ \beta_2 & \text{if } \beta_1 = \textbf{none} \end{array} \right.\right] : \\ & & \texttt{num}^{\sigma\ \texttt{BASE}\ \beta_1} \rightarrow \texttt{num}^{\sigma\ \texttt{BASE}\ \beta_2} \rightarrow \texttt{num}^{\sigma\ \texttt{BASE}\ \beta} \\ \\ - & : & \forall \sigma, \beta_1, \beta_2, \beta \left[\text{with } \beta = \left\{ \begin{array}{ll} \beta_1 & \text{if } \beta_2 = \textbf{none} \\ \textbf{none} & \text{if } \beta_1 = \beta_2 \\ \text{undefined} & \text{otherwise} \end{array} \right.\right] : \\ & & \texttt{num}^{\sigma\ \texttt{BASE}\ \beta_1} \rightarrow \texttt{num}^{\sigma\ \texttt{BASE}\ \beta_2} \rightarrow \texttt{num}^{\sigma\ \texttt{BASE}\ \beta} \end{array}$$

This reflects the fact that addition can be used on units of measurement where at most one has a base and that subtraction can be used to find the relative unit of measurement between two based measurements as well as to decrease a based measurement by a relative unit.

Define an annotated type system for a version of the FUN language using a selection of the ideas above. It may be helpful to allow polymorphism only for the scale annotations (using the scale variables σ) and the base annotations (using the base variables β). Try to establish a notion of semantic correctness for the annotated type system. Develop an inference algorithm and prove it syntactically sound. Finally investigate whether or not the inference algorithm is syntactically complete.

In some cases it is possible to extend this scheme with explicit rules for conversion between units; e.g. x FT $= x * 30.48$ CM which involves both a conversion between feet (FT) and meters (M) and between meters (M) and centimeters (CM). This is not always feasible (e.g. in the case of currency) and we shall leave the extension to the interested reader. ■

Exercises

Exercise 5.1 Consider the following expression:

```
let f = fnX x => x 1;
    g = fnY y => y+2;
    h = fnZ z => z+3
in (f g) + (f h)
```

Use Table 5.1 (the underlying type system) to obtain a type for this expression; what types do you use for `f`, `g` and `h`? Next use Table 5.2 (Control Flow Analysis) to obtain the annotated type of this expression; what annotated types do you use for `f`, `g` and `h`? ■

Exercise 5.2 Consider the following variation of FUN where function definitions explicitly involve type information as in $\texttt{fn}_\pi\ x : \tau_x \texttt{ => } e_0$ and $\texttt{fun}_\pi\ f : (\tau_x \to \tau_0)\ x \texttt{ => } e_0$. Modify Table 5.1 accordingly and prove that the resulting type system is deterministic: $\Gamma \vdash_{\mathsf{UL}} e : \tau_1$ and $\Gamma \vdash_{\mathsf{UL}} e : \tau_2$ imply $\tau_1 = \tau_2$. ■

Exercise 5.3 Consider the inclusion $\varphi_1 \subseteq \varphi_2$ between effects that was discussed in Subsections 5.4.1 and 5.2.3. Give an axiomatisation of $\varphi_1 \subseteq \varphi_2$ such that $\varphi_1 \subseteq \varphi_2$ holds if and only if the set of elements mentioned in φ_1 is a subset of the set of elements mentioned in φ_2. ■

Exercise* 5.4 In Example 5.4 we showed how to record the annotated type information in a form close to that considered in Chapter 3. To make this precise suppose that expressions of FUN are simultaneously labelled as in Chapters 3 and 5:

$$
\begin{array}{rcl}
e & ::= & t^\ell \\
t & ::= & c \mid x \mid \texttt{fn}_\pi\ x \texttt{ => } e_0 \mid \texttt{fun}_\pi\ f\ x \texttt{ => } e_0 \mid e_1\ e_2 \\
 & \mid & \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \mid \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \mid e_1\ op\ e_2
\end{array}
$$

Next modify Table 5.2 so as to define a judgement

$$\tilde{\rho}, \tilde{C}; \widehat{\Gamma} \vdash_{\mathsf{CFA}} e : \widehat{\tau} \tag{5.2}$$

where the idea is that

- $\tilde{C}(\ell) = \widehat{\tau}_\ell$ ensures that all judgements $\tilde{\rho}, \tilde{C}; \widehat{\Gamma}' \vdash_{\mathsf{CFA}} t^\ell : \widehat{\tau}'$ in (5.2) have $\widehat{\tau}' = \widehat{\tau}_\ell$, and
- $\tilde{\rho}(x) = \widehat{\tau}_x$ ensures that all judgements $\tilde{\rho}, \tilde{C}; \widehat{\Gamma}' \vdash_{\mathsf{CFA}} x^\ell : \widehat{\tau}'$ in (5.2) have $\widehat{\tau}' = \widehat{\tau}_x$.

Check that (5.2) holds for the expression in Example 5.4 when we take $\tilde{C}(1) = \widehat{\tau}_{\mathsf{Y}}$, $\tilde{C}(2) = \widehat{\tau}_{\mathsf{Y}} \xrightarrow{\{\mathsf{X}\}} \widehat{\tau}_{\mathsf{Y}}$, $\tilde{C}(3) = \mathtt{int}$, $\tilde{C}(4) = \widehat{\tau}_{\mathsf{Y}}$, $\tilde{C}(5) = \widehat{\tau}_{\mathsf{Y}}$, $\tilde{\rho}(\mathtt{x}) = \widehat{\tau}_{\mathsf{Y}}$ and $\tilde{\rho}(\mathtt{y}) = \mathtt{int}$. ∎

Exercise 5.5 Consider adding the following inference rule to Table 5.2 (Control Flow Analysis)

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{CFA}} e_1 : \widehat{\tau}_2 \xrightarrow{\varphi} \widehat{\tau}_0 \quad \widehat{\Gamma} \vdash_{\mathsf{CFA}} e_2 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathsf{CFA}} e_1\ e_2 : \bot_{\widehat{\tau}_0}} \quad \text{if } \varphi = \emptyset$$

where $\bot_{\widehat{\tau}_0}$ is the least element of $\widehat{\mathbf{Type}}[\tau_0]$ and $\tau_0 = \lfloor \widehat{\tau}_0 \rfloor$. Explain what this rule does and determine whether or not Theorem 5.9 (semantic correctness) continues to hold. ∎

Exercise 5.6 We shall now extend the Control Flow Analysis of Section 5.1 to include information about where functions are called. To do so modify the syntax of expressions by adding labels to all application points (in the manner of Chapter 3):

$$e ::= \cdots \mid (e_1\ e_2)^\ell$$

Also define a syntactic category of label annotations $\psi \in \mathbf{LAnn}$ by

$$\psi ::= \ell \mid \psi_1 \cup \psi_2 \mid \emptyset$$

to be interpreted modulo UCAI. Finally modify the syntax of annotated types as follows:

$$\widehat{\tau} ::= \cdots \mid \widehat{\tau}_1 \xrightarrow[\psi]{\phi} \widehat{\tau}_2$$

In this system it should be possible to type the expression

$$((\mathtt{fn}_{\mathsf{X}}\ \mathtt{x} \mathtt{=>} \mathtt{x})\ (\mathtt{fn}_{\mathsf{Y}}\ \mathtt{y} \mathtt{=>} \mathtt{y}))^1$$

(see Example 5.4) such that $\mathtt{fn}_{\mathsf{X}}\ \mathtt{x} \mathtt{=>} \mathtt{x}$ gets the type

$$(\mathtt{int} \xrightarrow[\emptyset]{\{\mathsf{Y}\}} \mathtt{int}) \xrightarrow[\{1\}]{\{\mathsf{X}\}} (\mathtt{int} \xrightarrow[\emptyset]{\{\mathsf{Y}\}} \mathtt{int})$$

indicating that it is called at the application labelled 1; more generally it should have the type

$$(\widehat{\tau} \xrightarrow[\psi_{\mathsf{Y}}]{\{\mathsf{Y}\} \cup \phi_{\mathsf{Y}}} \widehat{\tau}) \xrightarrow[\{1\} \cup \psi_{\mathsf{X}}]{\{\mathsf{X}\} \cup \phi_{\mathsf{X}}} (\widehat{\tau} \xrightarrow[\psi_{\mathsf{Y}}]{\{\mathsf{Y}\} \cup \phi_{\mathsf{Y}}} \widehat{\tau})$$

for all $\phi_{\mathsf{X}}, \phi_{\mathsf{Y}} \in \mathbf{Ann}$, $\psi_{\mathsf{X}}, \psi_{\mathsf{Y}} \in \mathbf{LAnn}$, and $\widehat{\tau} \in \widehat{\mathbf{Type}}$. Modify the analysis of Table 5.2 so as to specify this analysis. ∎

Exercise 5.7 In Section 5.2 we equipped FUN with a call-by-value semantics. An alternative would be to use a call-by-name semantics. It can be obtained as a simple modification of the semantics of Table 5.4 by changing rule [*app*] such that the argument is not evaluated before the substitution takes place and similarly changing rule [*let*]. Make these changes to Table 5.4 and show that the correctness result (Theorem 5.9) still holds for the analysis of Table 5.2.

What does that tell us about the precision of the analysis? ■

Exercise 5.8 Prove Fact 5.17 (the syntactic soundness and completeness of Robinson unification). ■

Exercise 5.9 Consider the partial ordering $\widehat{\tau} \leq \widehat{\tau}'$ on the annotated types of Section 5.1 that is defined by:

$$\widehat{\tau} \leq \widehat{\tau} \qquad\qquad \frac{\widehat{\tau}_1' \leq \widehat{\tau}_1 \quad \widehat{\tau}_2 \leq \widehat{\tau}_2' \quad \varphi \subseteq \varphi'}{\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \leq \widehat{\tau}_1' \xrightarrow{\varphi'} \widehat{\tau}_2'}$$

We shall say that this ordering treats $\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$ *covariantly* in φ and $\widehat{\tau}_2$ but *contravariantly* in $\widehat{\tau}_1$; this is in line with the subtype ordering considered in Section 5.4 but differs form the partial ordering $\widehat{\tau} \sqsubseteq \widehat{\tau}'$ considered in Section 5.2.

Show that $(\widehat{\mathbf{Type}}[\tau], \leq)$ is a complete lattice for all choices of the underlying type $\tau \in \mathbf{Type}$. Next investigate whether or not an analogue of Proposition 5.12 holds for this ordering.

Finally reconsider the decision to let $\theta''_T(\alpha)$ in Subsection 5.3.4 be the least element of $(\widehat{\mathbf{Type}}[\tau], \sqsubseteq)$; would it be preferable to let $\theta''_T(\alpha)$ be the least element of $(\widehat{\mathbf{Type}}[\tau], \leq)$ or $(\widehat{\mathbf{Type}}[\tau], \geq)$? ■

Exercise* 5.10 Suppose that

$$\mathcal{W}_{\mathsf{UL}}([\,], \texttt{fun}_{\mathsf{F}}\ f\ x \texttt{ => } e_0) = (\alpha_x \to \alpha_0, \theta)$$

where α_x and α_0 are *distinct* type variables. Let e be an arbitrary correctly typed closed expression, i.e. $[\,] \vdash_{\mathsf{UL}} e : \tau$ for some τ and show that the call

$$(\texttt{fun}_{\mathsf{F}}\ f\ x \texttt{ => } e_0)\ e$$

cannot terminate. (Hint: use Fact 5.6, Theorems 5.9, 5.20 and 5.21 and that $\mathcal{W}_{\mathsf{UL}}$ is syntactically sound.) ■

Exercise 5.11 Formulate what it means for the Side Effect Analysis of Table 5.9 to be semantically correct; this involves modifying the Natural Semantics of Table 5.4 to deal with the store and to record the side effects. (Proving the result would require quite some work.) ■

Exercise 5.12 Suppose that the language of Subsection 5.4.1 has a call-by-name semantics rather than a call-by-value semantics. Modify the Side Effect Analysis of Table 5.9 accordingly. ■

Exercise 5.13 We shall now illustrate the concept of *proof normalisation* for the Side Effect Analysis of Subsection 5.4.1; for this we shall assume that Table 5.9 does *not* include the combined rule for subeffecting and subtyping but *only* the rule for subeffecting.

For this system one can dispense with an explicit rule for subeffecting by integrating its effects into all other rules; this can be done by adding a "$\cup\varphi'$" to all effects occurring in the conclusions of all axioms and rules. Do this and argue that exactly the same judgements are provable in the two systems.

A further variation is only to incorporate "$\cup\varphi'$" where it is really needed: in all axioms and in the rules for function abstraction. Do this and argue that once more exactly the same judgements are provable in the two systems.

What you have performed amounts to *proof normalisation*: whenever one has an inference system with a number of syntax directed rules and axioms and at least one rule that is not syntax directed, it is frequently possible to restrict the use of the non syntax directed rules. In this way the structure of the inference trees comes closer to the structure of the syntax trees and this is often helpful for proving semantic correctness and is a useful starting point for developing inference algorithms. ■

Exercise 5.14 Consider the rule [*handle*] in the Exception Analysis of Table 5.10. Would the analysis still be semantically correct if we replaced it by the following two rules:

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{ES}} e_1 : \widehat{\tau} \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{ES}} e_2 : \widehat{\tau} \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{ES}} \texttt{handle}\ s\ \texttt{as}\ e_1\ \texttt{in}\ e_2 : \widehat{\tau} \;\&\; \varphi_2}$$
$$\text{if } s \notin \varphi_2 \text{ and } AV(\varphi_2) = \emptyset$$

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{ES}} e_1 : \widehat{\tau} \;\&\; \varphi_1 \quad \widehat{\Gamma} \vdash_{\mathsf{ES}} e_2 : \widehat{\tau} \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{ES}} \texttt{handle}\ s\ \texttt{as}\ e_1\ \texttt{in}\ e_2 : \widehat{\tau} \;\&\; \varphi_1 \cup (\varphi_2 \backslash \{s\})}$$
$$\text{if } s \in \varphi_2 \text{ or } AV(\varphi_2) \neq \emptyset$$

Here $s \in \varphi_2$ means $\{s\} \subseteq \varphi_2$ and $AV(\varphi_2)$ is the set of annotation variables in φ_2. ■

Exercise 5.15 Consider the Exception Analysis of Table 5.10 and change the rule [*let*] to

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{ES}} e_1 : \widehat{\sigma}_1 \;\&\; \varphi_1 \quad \widehat{\Gamma}[x \mapsto \widehat{\sigma}_1] \vdash_{\mathsf{ES}} e_2 : \widehat{\sigma}_2 \;\&\; \varphi_2}{\widehat{\Gamma} \vdash_{\mathsf{ES}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \widehat{\sigma}_2 \;\&\; \varphi_1 \cup \varphi_2}$$

and perform similar changes in [*if*], [*raise*], [*handle*] and [*sub*]; do not forget to define the meaning of $\widehat{\tau} \leq \widehat{\tau}'$. Clearly the new system is at least as powerful as Table 5.10 but is it more powerful? (Hint: Consider the places where [*gen*] and [*ins*] can be used in Table 5.10.) ■

Chapter 6

Algorithms

In previous chapters we have studied several algorithms for obtaining solutions of program analyses. In this chapter we shall explore further the similarities between the different approaches to program analysis by studying general algorithms for solving equation or inequation systems.

6.1 Worklist Algorithms

We will abstract away from the details of a particular analysis by considering equations or inequations in a set of *flow variables* for which we want to solve the system. As an illustration, in Data Flow Analysis there might be separate flow variables for the entry and exit values at each program point, whilst in Constraint Based Analysis there would be a separate flow variable for the cache at each program point and for the environment at each program variable.

Example 6.1 Consider the following WHILE program

```
if [b1]^1 then (while [b2]^2 do [x := a1]^3)
          else (while [b3]^4 do [x := a2]^5);
[x := a3]^6
```

where we leave the expressions a_i and b_i unspecified. The equations generated by the Reaching Definitions Analysis of Section 2.1.2 take the form:

$$
\begin{array}{ll}
\mathsf{RD}_{entry}(1) = X_? & \mathsf{RD}_{exit}(1) = \mathsf{RD}_{entry}(1) \\
\mathsf{RD}_{entry}(2) = \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(3) & \mathsf{RD}_{exit}(2) = \mathsf{RD}_{entry}(2) \\
\mathsf{RD}_{entry}(3) = \mathsf{RD}_{exit}(2) & \mathsf{RD}_{exit}(3) = (\mathsf{RD}_{entry}(3) \backslash X_{356?}) \cup X_3 \\
\mathsf{RD}_{entry}(4) = \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(5) & \mathsf{RD}_{exit}(4) = \mathsf{RD}_{entry}(4) \\
\mathsf{RD}_{entry}(5) = \mathsf{RD}_{exit}(4) & \mathsf{RD}_{exit}(5) = (\mathsf{RD}_{entry}(5) \backslash X_{356?}) \cup X_5 \\
\mathsf{RD}_{entry}(6) = \mathsf{RD}_{exit}(2) \cup \mathsf{RD}_{exit}(4) & \mathsf{RD}_{exit}(6) = (\mathsf{RD}_{entry}(6) \backslash X_{356?}) \cup X_6
\end{array}
$$

Here $X_\ell = \{(\mathbf{x}, \ell)\}$ and we also allow a string of subscripts on X; for example, $X_{356?} = \{(\mathbf{x}, 3), (\mathbf{x}, 5), (\mathbf{x}, 6), (\mathbf{x}, ?)\}$.

When expressed as a constraint system in the flow variables $\{\mathsf{x}_1, \cdots, \mathsf{x}_{12}\}$ it takes the form

$$\begin{array}{lclclcl}
\mathsf{x}_1 & = & X_? & \qquad & \mathsf{x}_7 & = & \mathsf{x}_1 \\
\mathsf{x}_2 & = & \mathsf{x}_7 \cup \mathsf{x}_9 & & \mathsf{x}_8 & = & \mathsf{x}_2 \\
\mathsf{x}_3 & = & \mathsf{x}_8 & & \mathsf{x}_9 & = & (\mathsf{x}_3 \backslash X_{356?}) \cup X_3 \\
\mathsf{x}_4 & = & \mathsf{x}_7 \cup \mathsf{x}_{11} & & \mathsf{x}_{10} & = & \mathsf{x}_4 \\
\mathsf{x}_5 & = & \mathsf{x}_{10} & & \mathsf{x}_{11} & = & (\mathsf{x}_5 \backslash X_{356?}) \cup X_5 \\
\mathsf{x}_6 & = & \mathsf{x}_8 \cup \mathsf{x}_{10} & & \mathsf{x}_{12} & = & (\mathsf{x}_6 \backslash X_{356?}) \cup X_6
\end{array}$$

where $\mathsf{x}_1, \cdots, \mathsf{x}_6$ correspond to $\mathsf{RD}_{entry}(1), \cdots, \mathsf{RD}_{entry}(6)$ and $\mathsf{x}_7, \cdots, \mathsf{x}_{12}$ correspond to $\mathsf{RD}_{exit}(1), \cdots, \mathsf{RD}_{exit}(6)$.

Since we are generally interested in the solution for RD_{entry}, we shall in this and subsequent examples consider the following simplified equation system:

$$\begin{array}{lcl}
\mathsf{x}_1 & = & X_? \\
\mathsf{x}_2 & = & \mathsf{x}_1 \cup (\mathsf{x}_3 \backslash X_{356?}) \cup X_3 \\
\mathsf{x}_3 & = & \mathsf{x}_2 \\
\mathsf{x}_4 & = & \mathsf{x}_1 \cup (\mathsf{x}_5 \backslash X_{356?}) \cup X_5 \\
\mathsf{x}_5 & = & \mathsf{x}_4 \\
\mathsf{x}_6 & = & \mathsf{x}_2 \cup \mathsf{x}_4
\end{array}$$

Clearly nothing is lost by these changes in representation. ■

Equations versus inequations. An apparent difference between the settings of Chapters 2 and 3 is that we solve equation systems $(x_i = t_i)_{i=1}^N$ in the former but inequation systems $(x_i \sqsupseteq t_i)_{i=1}^N$ in the latter. However, already in Section 2.2, we observed that a solution of an equation system is also a solution of the inequation system resulting from replacing all occurrences of "=" by "$\sqsupseteq$". In fact, the following inequation system (where all left hand sides are the same)

$$x \sqsupseteq t_1 \qquad \cdots \qquad x \sqsupseteq t_n$$

and the equation

$$x = x \sqcup t_1 \sqcup \cdots \sqcup t_n$$

have the same solutions: any solution of the former is also a solution of the latter and vice versa. Furthermore, the least solution of the above systems is also the least solution of

$$x = t_1 \sqcup \cdots \sqcup t_n$$

(where the x component has been removed on the right hand side). Given these observations, it should be clear that it does not much matter whether our algorithms are supposed to solve inequation or equation systems – but see Exercise 6.5 for the flexibility of inequation systems in obtaining efficient algorithms. Throughout this chapter we will concentrate on constraint systems with multiple inequations for the same left hand side.

Assumptions. We make the following assumptions:

- There is a finite *constraint system* $\mathcal{S}$ of the form

$$(x_i \sqsupseteq t_i)_{i=1}^{N}$$

  for $N \geq 1$ where the left hand sides are not necessarily distinct.

- The set $FV(t_i)$ of flow variables contained in a right hand side t_i is a subset of the finite set $X = \{x_i \mid 1 \leq i \leq N\}$.

- A solution is a total function, $\psi : X \to L$, assigning each flow variable a value in the complete lattice $(L, \sqsubseteq)$ satisfying the Ascending Chain Condition.

- The terms are interpreted with respect to solutions, $\psi : X \to L$, and we write $[\![t]\!]\psi \in L$ to represent the interpretation of t with respect to ψ.

- The interpretation $[\![t]\!]\psi$ of a term t is monotone in ψ and its value only depends on the values $\{\psi(x) \mid x \in FV(t)\}$ of the solution on the flow variables occurring in the term.

In the interest of generality, we leave the nature of the right hand sides, t_i, unspecified.

Example 6.2 The constraints used in this chapter may appear to be simpler than those used in Chapter 3 in that apparently we do not allow conditional constraints. However, conditional constraints can be dealt with by allowing the terms to contain conditionals. Consider the following expression (from Example 3.20):

$$((\texttt{fn x => x}^1)^2\ (\texttt{fn y => y}^3)^4)^5$$

Using the notation above, the constraints generated by the Constraint Based 0-CFA Analysis are:

$x_1 \supseteq x_6$	$x_2 \supseteq \{\texttt{fn x => x}^1\}$
$x_3 \supseteq x_7$	$x_4 \supseteq \{\texttt{fn y => y}^3\}$
$x_5 \supseteq \text{if } \{\texttt{fn x => x}^1\} \subseteq x_2 \text{ then } x_1$	$x_5 \supseteq \text{if } \{\texttt{fn y => y}^3\} \subseteq x_2 \text{ then } x_3$
$x_6 \supseteq \text{if } \{\texttt{fn x => x}^1\} \subseteq x_2 \text{ then } x_4$	$x_7 \supseteq \text{if } \{\texttt{fn y => y}^3\} \subseteq x_2 \text{ then } x_4$

Here x_1 to x_5 correspond to C(1) to C(5), x_6 corresponds to r(x) and x_7 corresponds to r(y). ∎

6.1.1 The Structure of Worklist Algorithms

In Chapters 2 and 3, we already presented algorithms for solving the systems which arise from Intraprocedural Data Flow Analysis and Control Flow Analysis. The common feature of those algorithms is that they both use a *worklist* to control the iteration. Some representation of the work to be done is stored in the worklist; the iteration selects a task from the worklist and removes it – the processing of the task may cause new tasks to be added to the worklist. This process is iterated until there is no more work to be done – the worklist is empty.

Operations on worklists. Our starting point in this section is an abstract variant of those previous algorithms. It is abstract because it is parameterised on the details of the worklist and on the associated operations and values:

- **empty** is the empty worklist;
- **insert**$((x \sqsupseteq t),\mathsf{W})$ returns a new worklist that is as W except that a new constraint $x \sqsupseteq t$ has been added; it is normally used as in

  $$\mathsf{W} := \mathsf{insert}((x \sqsupseteq t),\mathsf{W})$$

  so as to update the worklist W to contain the new constraint $x \sqsupseteq t$;
- **extract**(W) returns a pair whose first component is a constraint $x \sqsupseteq t$ in the worklist and whose second component is the smaller worklist obtained by removing an occurrence of $x \sqsupseteq t$; it is normally used as in

  $$((x \sqsupseteq t),\mathsf{W}) := \mathsf{extract}(\mathsf{W})$$

  so as to select and remove a constraint from W.

In its most abstract form the worklist could be viewed as a set of constraints with the following operations:

$\mathsf{empty} = \emptyset$

function insert$((x \sqsupseteq t),\mathsf{W})$
return $\mathsf{W} \cup \{x \sqsupseteq t\}$

function extract(W)
return $((x \sqsupseteq t),\mathsf{W}\backslash\{x \sqsupseteq t\})$ for some $x \sqsupseteq t$ in W

However, it may be more appropriate to regard the worklist as a multiset (thereby allowing constraints to occur more than once on the worklist), as a list with additional structure, or as a combination of other structures; the structure will be used by the function **extract** to extract the appropriate constraint. Later in this chapter we shall illustrate how a judicious choice

```
INPUT:    A system S of constraints: x_1 ⊒ t_1, ···, x_N ⊒ t_N

OUTPUT:   The least solution: Analysis

METHOD:   Step 1:  Initialisation (of W, Analysis and infl)
                   W := empty;
                   for all x ⊒ t in S do
                      W := insert((x ⊒ t),W)
                      Analysis[x] := ⊥;
                      infl[x] := ∅;
                   for all x ⊒ t in S do
                      for all x' in FV(t) do
                         infl[x'] := infl[x'] ∪ {x ⊒ t};

          Step 2:  Iteration (updating W and Analysis)
                   while W ≠ empty do
                     ((x ⊒ t),W) := extract(W);
                     new := eval(t,Analysis);
                     if Analysis[x] ⋢ new then
                        Analysis[x] := Analysis[x] ⊔ new;
                        for all x' ⊒ t' in infl[x] do
                           W := insert((x' ⊒ t'),W);

USING:    function eval(t,Analysis)
          return [t](Analysis)

          value empty

          function insert((x ⊒ t),W)
          return ···

          function extract(W)
          return ···
```

Table 6.1: The Abstract Worklist Algorithm.

of "structure" will produce worklist algorithms with good practical performance.

Abstract worklist algorithm. The *abstract worklist algorithm* is shown in Table 6.1. Since the solution is a function with finite domain, we have represented it as an array, Analysis, as we did in Section 2.4. The algorithm uses a *worklist*, W, and an auxiliary array of sets of constraints, infl, that records the constraints whose values are *influenced* by a given flow variable; after step 1 we have for each x in X:

$$\mathsf{infl}[x] = \{(x' \sqsupseteq t') \text{ in } \mathcal{S} \mid x \in FV(t')\}$$

Initially the worklist contains all constraints from $\mathcal{S}$, the influence sets are generated and the Analysis array has every flow variable set to $\perp$. During the iteration a constraint $x \sqsupseteq t$ is selected from the worklist using the extract function. If the Analysis array is assigned during the iteration, all constraints influenced by the updated variable are added to the worklist using the insert function. The algorithm terminates when the worklist equals the value empty.

Example 6.3 Consider the simplified equation system from Example 6.1. After step 1 of the abstract worklist algorithm of Table 6.1, the worklist W contains all equations and the influence sets are as follows (where we identify the equations with the flow variables on the left hand side):

	x_1	x_2	x_3	x_4	x_5	x_6
infl	$\{x_2, x_4\}$	$\{x_3, x_6\}$	$\{x_2\}$	$\{x_5, x_6\}$	$\{x_4\}$	$\emptyset$

Additionally Analysis is set to $\emptyset$ for all flow variables. We shall continue this example later. ∎

Properties of the algorithm. Despite the abstract presentation of the algorithm we can give a proof of its correctness and an upper bound on its complexity that will remain true even when more specialised representations of the worklist are used.

Given a system of constraints, $\mathcal{S} = (x_i \sqsupseteq t_i)_{i=1}^{N}$, we define a function

$$F_{\mathcal{S}} : (X \to L) \to (X \to L)$$

by:

$$F_{\mathcal{S}}(\psi)(x) = \bigsqcup\{[\![t]\!]\psi \mid x \sqsupseteq t \text{ in } \mathcal{S}\}$$

This defines a monotone function over the complete lattice $X \to L$ and it follows from Tarski's Fixed Point Theorem (see Proposition A.10) that $F_{\mathcal{S}}$ has a least fixed point, $\mu_{\mathcal{S}}$, which is the least solution to the constraints $\mathcal{S}$. Since L by assumption satisfies the Ascending Chain Condition and since X is finite it follows that also $X \to L$ satisfies the Ascending Chain Condition; therefore $\mu_{\mathcal{S}}$ is given by

$$\mu_{\mathcal{S}} = \mathit{lfp}(F_{\mathcal{S}}) = \bigsqcup_{j \geq 0} F_{\mathcal{S}}^{j}(\perp)$$

and the chain $(F_{\mathcal{S}}^{n}(\perp))_n$ eventually stabilises.

Lemma 6.4 Given the assumptions, the algorithm of Table 6.1 computes the least solution of the given constraint system, $\mathcal{S}$. ∎

Proof We write $\mathsf{Analysis}_i[x]$ to represent the value of $\mathsf{Analysis}[x]$ after the i-th iteration of the loop considered.

First we prove termination. The loops in step 1 are all for-loops; thus termination of step 1 is trivially proved. The body of the while-loop of step 2 removes an element from the worklist, it then either adds at most N elements to the worklist (if $\mathsf{Analysis}[x]$ is assigned to) or else it leaves the worklist unchanged (if $\mathsf{Analysis}[x]$ is not assigned to). If $\mathsf{Analysis}[x]$ is assigned to it gets a strictly larger value. Since L satisfies the Ascending Chain Condition, $\mathsf{Analysis}[x]$ (for each of the finitely many $x \in X$) can only change a finite number of times. Thus the worklist will eventually be exhausted.

The correctness proof is in three parts: (i) first we show that on each iteration the values in Analysis are less than or equal to the corresponding values of $\mu_{\mathcal{S}}$, (ii) then we show that $\mu_{\mathcal{S}}$ is less than or equal to Analysis at the termination of step 2, and (iii) finally we combine these results.

Part (i). We show that

$$\forall x \in X : \mathsf{Analysis}_i[x] \sqsubseteq \mu_{\mathcal{S}}(x)$$

is an invariant of the while-loop of step 2: After step 1 the invariant is trivially established. We have to show that the while-loop preserves the invariant. For each iteration of the while-loop, either there is no assignment or else for some constraint $x \sqsupseteq t$ in $\mathcal{S}$, we perform an assignment to $\mathsf{Analysis}[x]$ so that:

$$\begin{aligned}
\mathsf{Analysis}_{i+1}[x] &= \mathsf{Analysis}_i[x] \sqcup \mathsf{eval}(t, \mathsf{Analysis}_i) \\
&= \mathsf{Analysis}_i[x] \sqcup [\![t]\!](\mathsf{Analysis}_i) \\
&\sqsubseteq \mu_{\mathcal{S}}(x) \sqcup [\![t]\!](\mu_{\mathcal{S}}) \\
&\sqsubseteq \mu_{\mathcal{S}}(x) \sqcup F_{\mathcal{S}}(\mu_{\mathcal{S}})(x) \\
&= \mu_{\mathcal{S}}(x)
\end{aligned}$$

The third step follows from the induction hypothesis and the monotonicity of $[\![t]\!]$ and the fourth step follows because $x \sqsupseteq t$ is one of the constraints considered in the definition of $F_{\mathcal{S}}$. So each iteration of the while-loop preserves the invariant.

Part (ii). On termination of the loop, the worklist is empty. By contradiction we establish that

$$F_{\mathcal{S}}(\mathsf{Analysis}) \sqsubseteq \mathsf{Analysis}$$

thereby showing the reductiveness of $F_{\mathcal{S}}$ on Analysis. For the proof by contradiction suppose that $\mathsf{Analysis}[x] \not\sqsupseteq F_{\mathcal{S}}(\mathsf{Analysis})(x)$ for some $x \in X$ and further suppose that one reason is because the constraint $x \sqsupseteq t$ is not fulfilled. Consider the last time that $\mathsf{Analysis}[y]$ was assigned to, for any variable $y \in FV(t)$.

If this was in step 1 then, since all constraints are in W at the beginning of step 2, for some $i \geq 1$ it is ensured that

$$\mathsf{Analysis}_i[x] \sqsupseteq \mathsf{Analysis}_{i-1}[x] \sqcup [\![t]\!](\mathsf{Analysis}_{i-1})$$

where $\mathsf{Analysis}_{i-1}[y]$ contains the final value for y; hence $[\![t]\!](\mathsf{Analysis}_j)$ remains stable for $j \geq i-1$ showing that this case cannot apply.

It follows that Analysis[y] was last assigned in step 2. This must have been in the context of dealing with a constraint $y \sqsupseteq t'$. But then, since $(x \sqsupseteq t) \in \mathsf{infl}[y]$, the constraint $x \sqsupseteq t$ was added to the worklist and then we re-established, for some later $i \geq 1$, that

$$\mathsf{Analysis}_i[x] \sqsupseteq \mathsf{Analysis}_{i-1}[x] \sqcup [\![t]\!](\mathsf{Analysis}_{i-1})$$

As before, $[\![t]\!](\mathsf{Analysis}_j)$ remains stable for $j \geq i-1$ showing that this case cannot apply either. This completes the proof by contradiction.

Thus F_S is reductive on Analysis and by Tarski's Fixed Point Theorem (see Proposition A.10):

$$\mu_S = \mathit{lfp}(F_S) \sqsubseteq \mathsf{Analysis}$$

Part (iii). That $\mu_S = \mathsf{Analysis}$ on termination of step 2 follows from the combination of parts (i) and (ii). ∎

Assume that the size of the right hand sides of constraints is at most $M \geq 1$ and that the evaluation of a right hand side takes $O(M)$ steps; further assume that each assignment takes $O(1)$ step. Each constraint is influenced by at most M flow variables and therefore the initialisation of the influence sets takes $O(N + N \cdot M)$ steps. Writing N_x for the number of constraints in $\mathsf{infl}[x]$ we note that $\sum_{x \in X} N_x \leq M \cdot N$. Assuming that L is of finite height at most $h \geq 1$, the algorithm assigns to Analysis[x] at most h times, adding N_x constraints to the worklist, for each flow variable $x \in X$. Thus, the total number of constraints added to the worklist is bounded from above by:

$$N + (h \cdot \sum_{x \in X} N_x) \leq N + (h \cdot M \cdot N)$$

Since each element on the worklist causes a call to eval, the cost of the calls is $O(N \cdot M + h \cdot M^2 \cdot N)$. This gives an overall complexity of $O(h \cdot M^2 \cdot N)$.

6.1.2 Iterating in LIFO and FIFO

Extraction based on LIFO. The algorithm of Table 6.1 is an abstract algorithm because it does not provide the details of the worklist nor of the associated operations; a concrete algorithm is only obtained once this information is supplied. We now show that the algorithm abstracts the algorithms that we studied in Tables 2.8 and 3.7 of Chapters 2 and 3, respectively. In both of the earlier algorithms the worklist was implemented by a list that was used as a stack, i.e. in a *LIFO* manner (meaning *last-in first-out*), as specified by the operations in Table 6.2. However, the two algorithms differ in the way constraints and influence sets are represented.

Example 6.5 A constraint of the form Analysis[ℓ'] $\sqsupseteq f_\ell$(Analysis[ℓ]) is represented on the worklist W of Table 2.8 by the pair (ℓ, ℓ'); this is possible

```
empty = nil

function insert((x ⊒ t),W)
return cons((x ⊒ t),W)

function extract(W)
return (head(W), tail(W))
```

Table 6.2: Iterating in last-in first-out order (LIFO).

since each ℓ uniquely identifies the transfer function f_ℓ. The influence sets were indirectly represented through the flow, F; to be more precise, $\mathsf{infl}[\ell'] = \{(\ell', \ell'') \in F \mid \ell'' \in \mathbf{Lab}\}$.

The number N of constraints in a system generated from Intraprocedural Data Flow Analysis is proportional to the number b of elementary blocks. Furthermore, it is usual to take $M = 1$. Thus the upper bound on the complexity of the abstract algorithm specialises to $O(h \cdot b)$; for an analysis such as Reaching Definitions Analysis where h is proportional to b this gives $O(b^2)$. This agrees with the bound obtained in Example 2.30. ∎

Example 6.6 In Table 3.7 constraints of the form $\{t\} \subseteq p$, $p_1 \subseteq p$ or $\{t\} \subseteq p_2 \Rightarrow p_1 \subseteq p$ are represented on the worklist W by any one of the flow variables p_1 or p_2 occurring on the left hand side. The influence sets are represented using the edge array, E; to be more precise, $\mathsf{infl}[p] = \mathsf{E}[p]$ (viewed as a set). The initialisation of the influence sets in step 1 of the algorithm of Table 6.1 corresponds to step 2 of the algorithm in Table 3.7; also the inclusion on the worklist of all the constraints in $\mathsf{infl}[p]$ is replaced by the for-loop in step 3 of Table 3.7. Finally, note that $\mathsf{Analysis}[p]$ is written as $\mathsf{D}[p]$.

The number N of constraints in a system generated from Control Flow Analysis is $O(n^2)$ where n is the size of the expression (see the discussion about the number of constraints in Section 3.4). Also, h is bounded by n and, once again, $M = 1$. Thus the upper bound on the complexity of the abstract algorithm specialises to $O(n^3)$. This agrees with the bound obtained in Section 3.4. ∎

One disadvantage of the LIFO strategy as presented above, is that we do not check for the presence of a constraint when adding it to the worklist. Hence the worklist may evolve so as to contain multiple copies of the same constraint and this may lead to unnecessarily recalculating terms before their free variables have had much chance of getting new values. This is illustrated in the following example; obviously a remedy is to modify the LIFO strategy such that it never inserts a constraint when it is already present.

W	x_1	x_2	x_3	x_4	x_5	x_6
$[x_1, x_2, x_3, x_4, x_5, x_6]$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$[x_2, x_4, x_2, x_3, x_4, x_5, x_6]$	$X_?$	–	–	–	–	–
$[x_3, x_6, x_4, x_2, x_3, x_4, x_5, x_6]$	–	$X_{3?}$	–	–	–	–
$[x_2, x_6, x_4, x_2, x_3, x_4, x_5, x_6]$	–	–	$X_{3?}$	–	–	–
$[x_6, x_4, x_2, x_3, x_4, x_5, x_6]$	–	–	–	–	–	–
$[x_4, x_2, x_3, x_4, x_5, x_6]$	–	–	–	–	–	$X_{3?}$
$[x_5, x_6, x_2, x_3, x_4, x_5, x_6]$	–	–	–	$X_{5?}$	–	–
$[x_4, x_6, x_2, x_3, x_4, x_5, x_6]$	–	–	–	–	$X_{5?}$	–
$[x_6, x_2, x_3, x_4, x_5, x_6]$	–	–	–	–	–	–
$[x_2, x_3, x_4, x_5, x_6]$	–	–	–	–	–	$X_{35?}$
$[x_3, x_4, x_5, x_6]$	–	–	–	–	–	–
$[x_4, x_5, x_6]$	–	–	–	–	–	–
$[x_5, x_6]$	–	–	–	–	–	–
$[x_6]$	–	–	–	–	–	–
$[\,]$	–	–	–	–	–	–

Figure 6.1: Example: LIFO iteration.

Example 6.7 Continuing Examples 6.1 and 6.3, the LIFO worklist algorithm obtained from Tables 6.1 and 6.2 operates as shown in Figure 6.1. The first column is the worklist where we identify the equations by the variables on the left hand side as in Example 6.3. The remaining columns are the values of Analysis[x_i]; "–" means that the value is unchanged and hence is equal to that in the previous row. The first row of the table is the result of the initialisation of step 1; each of the remaining rows corresponds to one iteration through the loop of step 2. The improved strategy, where a constraint is never inserted when it is already present, is considered in Exercise 6.3. ■

Extraction based on FIFO. An obvious alternative to the use of a LIFO strategy is to use a *FIFO* strategy (meaning *first-in first-out*) where the list is used as a queue. Again it may be worthwhile not to insert a constraint into a worklist when it is already present. However, rather than going deeper into the LIFO and FIFO strategies, we shall embark on a treatment of more advanced insertion and extraction strategies.

6.2 Iterating in Reverse Postorder

A careful organisation of the worklist may lead to algorithms that perform better in practice than simply using the LIFO or FIFO strategies discussed

above; however, in general we will not be able to improve our estimation of the worst case complexity to reflect this.

In this section we explore the idea that changes should be propagated throughout the rest of the program before returning to re-evaluate a constraint. One way of ensuring that every other constraint is evaluated before re-evaluating the constraint which caused the change is to impose some total order on the constraints. To obtain a suitable ordering we shall impose a graph structure on the constraints (see below) and then use an iteration order based on reverse postorder (see Appendix C). This approach has been very successful for Intraprocedural Data Flow Analysis and we shall show how the Round Robin Algorithm can be obtained by fully implementing these ideas.

The graph structure of a constraint system. Given a constraint system $S = (x_i \sqsupseteq t_i)_{i=1}^N$ we can construct a *graphical representation* G_S of the dependencies between the constraints in the following way:

- there is a node for each constraint $x_i \sqsupseteq t_i$, and
- there is a directed edge from the node for $x_i \sqsupseteq t_i$ to the node for $x_j \sqsupseteq t_j$ if x_i appears in t_j (i.e. if $x_j \sqsupseteq t_j$ appears in $\mathsf{infl}[x_i]$).

This constructs a *directed graph.* Sometimes it has a *root*, i.e. a node from which every other node is reachable through a directed path (see Appendix C). This will generally be the case for constraint systems corresponding to forward analyses of WHILE programs; in the case of Example 6.1 the root is x_1. It will not in general be the case for constraint systems corresponding to backward analyses for WHILE programs nor for constraint systems constructed for Constraint Based Analysis. We therefore need a generalisation of the concept of root. One obvious remedy is to add a dummy root and enough dummy edges from the dummy root to ordinary nodes that the dummy root in fact becomes a root. A more elegant formulation, that avoids cluttering the graph with dummy nodes and edges, is to consider a *handle*, i.e. a set of nodes such that each node in the graph is reachable through a directed path starting from one of the nodes in the handle (see Appendix C). Indeed, a graph G has a root r if and only if G has $\{r\}$ as a handle. In the case of Example 6.2 a minimal handle is $\{\mathsf{x}_2, \mathsf{x}_4\}$. One can take the entire set of nodes of a graph as a handle but it is more useful to choose a minimal handle: a handle such that no proper subset is also a handle; as discussed in Appendix C, minimal handles always exist (although they need not be unique).

We can then construct a *depth-first spanning forest* from the graph G_S and handle H_S using the algorithm of Table C.1. This also produces an array, rPostorder, that associates each node (i.e. each constraint $x \sqsupseteq t$) with its number in a *reverse postorder* traversal of the spanning forest. To lighten the notation, particularly when presenting the Round Robin Algorithm later, we sometimes demand that a constraint system $(x_i \sqsupseteq t_i)_{i=1}^N$ is listed in reverse

Figure 6.2: (a) Graphical representation. (b) Depth-first spanning tree.

```
empty = (nil,∅)

function insert((x ⊒ t),(W.c,W.p))
return (W.c,(W.p ∪ {x ⊒ t}))

function extract((W.c,W.p))
if W.c = nil then
    W.c := sort_rPostorder(W.p);
    W.p := ∅
return ( head(W.c), (tail(W.c),W.p) )
```

Table 6.3: Iterating in reverse postorder.

postorder. The advantages of reverse postorder over other orderings, such as preorder and breadth-first order, are discussed in Appendix C.

Example 6.8 Figure 6.2(a) shows the graphical representation of the constraints of Example 6.1; again we use the left hand side of the equation as the name of the equation. The node for x_1 is the root of the graph. Figure 6.2(b) shows a depth-first spanning tree for the graph; the associated reverse postorder is $x_1, \cdots, x_6$. ∎

Extraction based on reverse postorder. Conceptually, we now modify step 2 of the worklist algorithm of Table 6.1 so that the iteration amounts to an *outer* iteration that contains an *inner* iteration that visits the nodes in reverse postorder.

To achieve this, without actually changing Table 6.1, we shall organise the worklist W as a pair (W.c,W.p) of two structures. The first component, W.c, is a list of *current* nodes to be visited in the current inner iteration. The

W.c	W.p	x_1	x_2	x_3	x_4	x_5	x_6
[ ]	$\{x_1, \cdots, x_6\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$[x_2, x_3, x_4, x_5, x_6]$	$\{x_2, x_4\}$	$X_?$	–	–	–	–	–
$[x_3, x_4, x_5, x_6]$	$\{x_2, x_3, x_4, x_6\}$	–	$X_{3?}$	–	–	–	–
$[x_4, x_5, x_6]$	$\{x_2, x_3, x_4, x_6\}$	–	–	$X_{3?}$	–	–	–
$[x_5, x_6]$	$\{x_2, \cdots, x_6\}$	–	–	–	$X_{5?}$	–	–
$[x_6]$	$\{x_2, \cdots, x_6\}$	–	–	–	–	$X_{5?}$	–
$[x_2, x_3, x_4, x_5, x_6]$	$\emptyset$	–	–	–	–	–	$X_{35?}$
$[x_3, x_4, x_5, x_6]$	$\emptyset$	–	–	–	–	–	–
$[x_4, x_5, x_6]$	$\emptyset$	–	–	–	–	–	–
$[x_5, x_6]$	$\emptyset$	–	–	–	–	–	–
$[x_6]$	$\emptyset$	–	–	–	–	–	–
[ ]	$\emptyset$	–	–	–	–	–	–

Figure 6.3: Example: Reverse postorder iteration.

second component, W.p, is a set of *pending* nodes to be visited in a later inner iteration. Nodes are always inserted into W.p and always extracted from W.c; when W.c is exhausted the current inner iteration has finished and in preparation for the next we must sort W.p in the reverse postorder given by rPostorder and assign the result to W.c. The details are provided by the operations of Table 6.3; here (tail(W.c),W.p) denotes the pair whose first component is the tail of the list W.c and whose second component is W.p. Clearly one could dispense with sorting the set W.p if it was implemented by a suitable data structure that allows the insertion and extraction of elements in their appropriate order. This is facilitated by a *priority queue* where each element has a priority associated with it; in our case the priority is the rPostorder number. One of the better ways of implementing a priority queue is as a *2-3 tree* and in this representation an element can be inserted or extracted in $O(\log_2 N)$ steps where the priority queue contains at most N elements.

However, the amortised complexity of our scheme is equally good. To see this note that clearly a list of N elements can be sorted in $O(N \cdot \log_2(N))$ steps. Further suppose that we use a linked list representation of lists; then inserting an element to the front of a list, as well as extracting the head of a list, can be done in constant time. Thus the overall complexity for processing N insertions and N extractions is $O(N \cdot \log_2(N))$ in both schemes.

Actually we can do somewhat better under the special circumstances of our algorithm. Recall that there are at most N constraints that are all known from the outset. Let us represent W.p as a bit vector of length N and agree that it takes constant time to access or modify a component. Then insertion into W.p is performed by setting a bit to 1 and sorting of W.p is performed

by a for-loop running through all N constraints, and recording a constraint in the sorted list if and only if the bit is 1. Then we can guarantee that at most N insertions and N extractions can be done in $O(N)$ steps; hence the complexity estimation of Section 6.1.1 still holds.

Also the overall correctness of the algorithm carries over from Lemma 6.4 because we still have an abstract worklist; simply take $\mathsf{W} = \mathsf{W.c} \cup \mathsf{W.p}$.

Example 6.9 Returning to Examples 6.1 and 6.3, we shall consider the reverse postorder $\mathsf{x}_1, \cdots, \mathsf{x}_6$ of Example 6.8. Using the algorithm of Table 6.3, we get the iterations shown in Figure 6.3. Note that we perform fewer iterations than when using the LIFO strategy (see Example 6.7). Improved versions of Table 6.3 are considered in Exercises 6.6 and 6.7. ∎

6.2.1 The Round Robin Algorithm

Now suppose that we change the above algorithm such that each time W.c is exhausted we assign it the list $[1, \cdots, N]$ rather than the potentially shorter list obtained by sorting W.p. Clearly this may lead to more evaluations of right hand sides of constraints but it simplifies some of the book-keeping details. Indeed, now our only interest in W.p is whether or not it is empty; let us introduce a boolean, change, that is false whenever W.p is empty. Also let us split the iterations into an overall outer iteration having an explicit inner iteration; each inner iteration will then be a simple iteration through all constraints in reverse postorder.

We thus arrive at the *Round Robin Algorithm* shown in Table 6.4. The algorithm starts by initialising Analysis and change (step 1). The outer iteration (step 2) is a while-loop that begins by modifying change (corresponding to what would have happened in extract); it then has an inner iteration (a for-loop) that updates Analysis by recomputing the right hand sides of constraints. The algorithm continues iterating as long as any part of the solution changes (indicated by change being set to true).

Example 6.10 Using the reverse postorder $\mathsf{x}_1, \cdots, \mathsf{x}_6$ of Example 6.8 to solve the equations of Example 6.1, the Round Robin Algorithm of Table 6.4 operates as shown in Figure 6.4. The lines marked * record the assignment of false to change at the beginning of the body of the while-loop. ∎

Theoretical properties. We first study the correctness of the Round Robin Algorithm and then establish a striking bound on its complexity in the special case of Bit Vector Frameworks (as studied in Exercise 2.9 of Chapter 2).

Lemma 6.11 Given the assumptions, the algorithm of Table 6.4 computes the least solution of the given constraint system, $\mathcal{S}$. ∎

INPUT: A system $\mathcal{S}$ of constraints: $x_1 \sqsupseteq t_1, \cdots, x_N \sqsupseteq t_N$ ordered 1 to N in reverse postorder

OUTPUT: The least solution: Analysis

METHOD: Step 1: Initialisation

```
for all x ∈ X do
    Analysis[x] := ⊥
change := true;
```

Step 2: Iteration (updating Analysis)

```
while change do
    change := false;
    for i := 1 to N do
        new := eval(t_i,Analysis);
        if Analysis[x_i] ⋢ new then
            change := true;
            Analysis[x_i] := Analysis[x_i] ⊔ new;
```

USING:

```
function eval(t,Analysis)
return [[t]](Analysis)
```

Table 6.4: The Round Robin Algorithm.

change	x_1	x_2	x_3	x_4	x_5	x_6
true	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
* false						
true	$X_?$	–	–	–	–	–
true	–	$X_{3?}$	–	–	–	–
true	–	–	$X_{3?}$	–	–	–
true	–	–	–	$X_{5?}$	–	–
true	–	–	–	–	$X_{5?}$	–
true	–	–	–	–	–	$X_{35?}$
* false						
false	–	–	–	–	–	–
false	–	–	–	–	–	–
false	–	–	–	–	–	–
false	–	–	–	–	–	–
false	–	–	–	–	–	–
false	–	–	–	–	–	–

Figure 6.4: Example: Round Robin iteration.

Proof Since the algorithm of Table 6.4 is obtained in a systematic manner from that of Table 6.1, the correctness proof can be obtained in a systematic manner from that of Lemma 6.4; we leave the details to Exercise 6.9. ∎

We shall say that the constraint system $(x_i \sqsupseteq t_i)_{i=1}^N$ is an instance of a *Bit Vector Framework* when $L = \mathcal{P}(D)$ for some finite set D and when each right hand side t_i is of the form $(x_{j_i} \cap Y_i^1) \cup Y_i^2$ for sets $Y_i^k \subseteq D$ and variable $x_{j_i} \in X$. Clearly the classical Data Flow Analyses of Section 2.1 produce constraint systems of this form (possibly after expansion of a composite constraint $x_i \sqsupseteq t_i^1 \sqcup \cdots \sqcup t_i^{k_i}$ into the individual constraints $x_i \sqsupseteq t_i^1, \cdots, x_i \sqsupseteq t_i^{k_i}$).

Consider a depth-first spanning forest T and a reverse postorder rPostorder constructed for the graph G_S with handle H_S. We know from Appendix C that the *loop connectedness* parameter $d(G_S, T) \geq 0$ is defined as the largest number of so-called back edges found on any cycle-free path of G_S. We also know that the back edges are exactly those edges that are not topologically sorted by rPostorder, i.e. for which the target of the edge does not have an rPostorder number that is strictly larger than that of the source.

Let us say that the algorithm of Table 6.4 has iterated $n \geq 1$ times if the loop of step 1 has been executed once and the while-loop of step 2 has been executed $n - 1$ times. We then have the following result:

Lemma 6.12 Under the assumptions stated above, the algorithm of Table 6.4 halts after at most $d(G_S, T) + 3$ iterations. It therefore performs at most $O((d(G_S, T) + 1) \cdot N)$ assignments. ∎

Proof If a path contains d back edges, the while-loop of step 2 takes at most $d + 1$ iterations to propagate a change throughout the path. To be specific, it takes one iteration for the value to arrive at the source of the first back edge; this follows since up to this point the nodes in the path are numbered in increasing sequence. After that, it takes one iteration of the while-loop for the value to reach the source of the next back edge, and so on. So the algorithm needs at most $d + 1$ iterations to propagate the information.

One more iteration of the while-loop suffices for detecting that there are no further changes. We also have to count one for the iteration of the for-loop in step 1. This gives an upper bound of $d + 3 \leq d(G_S, T) + 3$ on the number of iterations.

Clearly each iteration can perform at most $O(N)$ assignments or evaluations of right hand sides. This gives an upper bound of $O((d(G_S, T) + 1) \cdot N)$ on the number of assignments. ∎

For WHILE programs we know from Appendix C that the loop connectedness parameter is independent of the choice of depth first spanning forest, and hence of the choice of the reverse postorder recorded in rPostorder, and that it equals the maximal nesting depth d of while-loops. It follows that Lemma 6.12 gives an overall complexity of $O((d + 1) \cdot b)$ where b is the number of elementary blocks. We would normally expect this bound to be significantly smaller than the $O(b^2)$ obtained in Examples 2.8 and 6.5.

It is worth mentioning that an empirical study of Fortran programs once reported that the loop connectedness parameter seldom exceeds 3; this gave rise to the Folk Theorem that the Round Robin Algorithm usually has linear time complexity.

Example 6.13 The WHILE program of Example 6.1 has a loop connectedness parameter of 1. According to Lemma 6.12 the Round Robin Algorithm will perform at most 4 iterations (1 for step 1 and 3 for step 2 of Table 6.4). However, Example 6.10 managed to succeed in only 3 iterations (1 for step 1 and 2 for step 2). ■

6.3 Iterating Through Strong Components

As was said above, a careful organisation of the worklist is a key factor in obtaining algorithms having a good practical performance. Iterating through the entire system of constraints in reverse postorder (as in Section 6.2) is a first step in this direction. A further step that often pays off in practice is to identify the so-called *strong components* in the system of constraints and to process them one by one; the processing of each strong component means iterating through the constraints of that strong component in reverse postorder (in the manner of Section 6.2).

Strong Components. Once again, we exploit the graph structure on constraints introduced in Section 6.2. Recall that a graph is *strongly connected* if every node is reachable from every other node (see Appendix C). The strong components of a graph are its maximal strongly connected subgraphs. The strong components partition the nodes in the graph (Fact C.3). The interconnections between components can be represented by a *reduced graph*: each strong component is represented by a node in the reduced graph and there is an edge from one strong component to another if there is an edge in the original graph from some node in the first strong component to a node in the second strong component and provided that the two strong components are not the same. The reduced graph is always a DAG, i.e. a directed, acyclic graph (Lemma C.5). As a consequence, the strong components can be linearly ordered in *topological order*: $SC_1 \leq SC_2$ (where SC_1 and SC_2 are nodes in the reduced graph) whenever there is an edge from SC_1 to SC_2; such a topological order can be obtained by constructing a reverse postorder for the reduced graph (as follows from Corollary C.11).

Example 6.14 Consider once again the equation system of Example 6.1. Its strong components and the reduced graph are shown in Figure 6.5. There are two possible topological orderings of the strong components. One is $\{x_1\}, \{x_2, x_3\}, \{x_4, x_5\}, \{x_6\}$ and the other is $\{x_1\}, \{x_4, x_5\}, \{x_2, x_3\}, \{x_6\}$.

Figure 6.5: (a) Strong components. (b) Reduced graph.

INPUT:	A graph partitioned into strong components
OUTPUT:	srPostorder

METHOD:	scc := 1;
	for each scc in topological order do
	  rp := 1;
	  for each $x \sqsupseteq t$ in the strong component scc
	  in local reverse postorder do
	  srPostorder[$x \sqsupseteq t$] := (scc,rp);
	  rp := rp + 1
	  scc := scc + 1;

Table 6.5: Pseudocode for constraint numbering.

In this example each strong component was an outermost loop. This holds in general for both forward and backward flow graphs for programs in the WHILE language. ■

For each constraint we need to record both the strong component it occurs in and its number in the local reverse postorder for that strong component. We shall do so by means of a numbering srPostorder that to each constraint $x \sqsupseteq t$ assigns a pair (scc,rp) consisting of the number scc of the strong component and the number rp of its reverse postorder numbering inside that strong component. When srPostorder[$x \sqsupseteq t$] $= (scc, rp)$ we shall write fst(srPostorder[$x \sqsupseteq t$]) for scc, and snd(srPostorder[$x \sqsupseteq t$]) for rp. One way to obtain srPostorder is using the algorithm of Table 6.5.

The Algorithm. The basic method of the new algorithm is that strong components are visited in topological order with nodes being visited in reverse postorder within each strong component.

Conceptually, we now modify step 2 of the worklist algorithm of Table 6.1 so that the iteration amounts to *three* levels of iteration; the *outermost level* deals with the strong components one by one; the *intermediate level* performs a number of passes over the constraints in the current strong component; and the *inner level* performs one pass in reverse postorder over the appropriate constraints.

To achieve this, without actually changing Table 6.1, we shall again organise the worklist W as a pair (W.c,W.p) of two structures. The first component, W.c, is a list of *current* nodes to be visited in the current inner iteration. The second component, W.p, is a set of *pending* nodes to be visited in a later intermediate or outer iteration. Nodes are always inserted into W.p and always extracted from W.c; when W.c is exhausted the current inner iteration has finished and in preparation for the next we must extract a strong component from W.p, sort it and assign the result to W.c. The details are provided by the operations of Table 6.6.

Intuitively, an inner iteration ends when W.c is exhausted, an intermediate iteration ends when scc gets a higher value than last time it was computed, and the outer iteration ends when both W.c and W.p are exhausted.

Again the lists can be organised as priority queues. This time the priority of $x \sqsupseteq t$ is the srPostorder information about the strong component, *scc*, and the local reverse postorder number, *rp*; rather than directly using pairs, (*scc*, *rp*), it may be helpful to use a linearised representation with numbers such that the lexicographic ordering on the pairs corresponds to the arithmetic ordering on the numbers.

Example 6.15 Consider the ordering $\{x_1\}, \{x_2, x_3\}, \{x_4, x_5\}, \{x_6\}$ of the strong components of Example 6.14. The algorithm iterating through strong components (Table 6.6) produces the walkthrough of Figure 6.6 when solving the system. Note that even on this small example the algorithm performs slightly better than the others (ignoring the cost of maintaining the worklist). Improved versions of Table 6.6 are considered in Exercises 6.11 and 6.12. ∎

It is also possible to split the set W.p of pending constraints into two collections: W.pc for those that relate to the current strong component and W.pf for those that relate to future strong components. (Since the reduced graph is a DAG there cannot be any constraints relating to earlier strong components.) We leave these details to Exercise 6.12.

Once more the overall correctness of the algorithm carries over from Lemma 6.4; also the estimation of the complexity carries over from Section 6.1.1.

```
empty = (nil,∅)

function insert((x ⊒ t),(W.c,W.p))
return (W.c,(W.p ∪ {x ⊒ t}))

function extract((W.c,W.p))
local variables: scc, W_scc
if W.c = nil then
    scc := min{fst(srPostorder[x ⊒ t]) | (x ⊒ t) ∈ W.p};
    W_scc := {(x ⊒ t) ∈ W.p | fst(srPostorder[x ⊒ t]) = scc};
    W.c := sort_srPostorder(W_scc);
    W.p := W.p \ W_scc;
return ( head(W.c), (tail(W.c),W.p) )
```

Table 6.6: Iterating through strong components.

W.c	W.p	x_1	x_2	x_3	x_4	x_5	x_6
[ ]	$\{x_1,\cdots,x_6\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
[ ]	$\{x_2,\cdots,x_6\}$	$X_?$	–	–	–	–	–
[x_3]	$\{x_3,\cdots,x_6\}$	–	$X_{3?}$	–	–	–	–
[ ]	$\{x_2,\cdots,x_6\}$	–	–	$X_{3?}$	–	–	–
[x_3]	$\{x_4,x_5,x_6\}$	–	–	–	–	–	–
[ ]	$\{x_4,x_5,x_6\}$	–	–	–	–	–	–
[x_5]	$\{x_5,x_6\}$	–	–	–	$X_{5?}$	–	–
[ ]	$\{x_4,x_5,x_6\}$	–	–	–	–	$X_{5?}$	–
[x_5]	$\{x_6\}$	–	–	–	–	–	–
[ ]	$\{x_6\}$	–	–	–	–	–	–
[ ]	$\emptyset$	–	–	–	–	–	$X_{35?}$

Figure 6.6: Example: Strong component iteration.

Concluding Remarks

Iterative solvers. Worklist algorithms have a long history; an early presentation of a general worklist algorithm for Data Flow Analysis may be found in [96]. A number of special purpose iterative algorithms are presented in [69]; this covers versions for forward Data Flow Analysis problems (based on reverse postorder traversal of the depth-first spanning forest) and backward Data Flow Analysis problems (based on postorder traversal of the depth-first spanning forest); it also covers versions in so-called "integrated form" (roughly meaning that constraints are placed on the worklist) and "segregated form" (roughly meaning that flow variables are placed on the

worklist); it also deals with the Round Robin Algorithm. A theoretical study of Round Robin Algorithms may be found in [92]. The empirical study of the loop connectedness parameter for Fortran programs, giving rise to the Folk Theorem that the Round Robin Algorithm operates in linear time in practice, is contained in [98].

The use of strong components to speed up iterative algorithms has been considered by a number of authors (e.g. [77, 87]). The treatment in Section 6.3 and Exercise 6.12 is based on [77] where the worklists W.c, W.pc, and W.pf are represented by priority queues currentQ, pendingQ, and futureQ, respectively. We refer to [77] for examples for which iteration over strong components performs better than either the basic worklist algorithm or the Round Robin Algorithm. There are many ways to implement priority queues as balanced search trees; the use of *2-3 trees* is explained in [4].

Local solvers. Suppose that the set of constraints is very large but that we are only interested in the values of a small subset of the flow variables. In this situation the classical worklist solvers may compute information in which we have no interest; hence it may be better to perform a *local fixed point computation* where one finds a *partial solution* for the flow variables of interest (and of course all flow variables which directly or indirectly influence them). For best performance the local fixed point solvers perform a dynamic determination of what is essentially the strong components of the constraints of interest.

One such algorithm is the top down solver [29]; this is a general fixed point algorithm that was originally developed for program analysis tools for logic programming languages. The algorithm proceeds top down; as it meets a new flow variable, it attempts to produce a solution for that flow variable. Its practical performance is very good and is competitive with the more sophisticated versions of the worklist algorithm.

Another algorithm is the worklist based local fixed point solver [51, 54] that uses *time stamps* to impose an ordering on the elements of the worklist. The worklist is now a maximum priority queue – the priority of an entry being given by its time stamp. Elements are selected from the worklist in an order which gives a run-time approximation to strong components (as with the top down solver).

A further refinement is the *differential* worklist based local fixed point solver [52, 53]. The differential worklist algorithm aims to minimise the amount of recomputation that occurs when a value changes; it does this by only recomputing the actual sub-terms influenced by the changed value and by only computing the incremental change due to the difference between the old and new values. This algorithm compares well in practice with a number of special purpose algorithms reported in the literature. (See Exercise 6.5 for a key insight.)

Efficient algorithms for restricted forms of systems. Elimination methods have been proposed as an alternative to the iterative approaches for solving systems. Most of this development has concentrated on the kind of systems generated for Data Flow Analysis. Ryder and Paull [146] provide a survey of different elimination methods.

A very successful elimination method is structural analysis [154]. The first phase of structural analysis performs a postorder search of the graph representing the constraint system in order to identify control structures; here postorder is with respect to a depth-first spanning forest for the graph and as a consequence, deeper nested control structures are identified earlier. Having identified a control structure, its nodes are reduced to a single node to produce a new, derived graph. This process terminates when there is only one node. The result of the first phase is a control tree which records the reduction sequence; this can be viewed as a technique for translating machine code into high-level programs (where the primitives are those control structures considered in structural analysis). The second phase of structural analysis then traverses the control tree in order to solve the constraint system; this takes the form of a bottom-up traversal followed by a top-down traversal. The bottom-up traversal associates a transfer function with each node in the control tree; the transfer function describes the effect of analysing that part of the program. The top-down traversal evaluates the transfer functions upon appropriate arguments. The second phase of structural analysis essentially amounts to high-level Data Flow Analysis [144] which again has some of the flavour of Section 1.6.

An early elimination algorithm for Data Flow Analysis was based on interval analysis [11]. An interval can briefly be described as a single entry cycle together with an acyclic extension. The algorithm performs in a similar way to structural analysis. The first phase constructs a derived sequence of graphs that is usually coarser than the control tree produced by structural analysis. The second phase consists of a backward traversal of the derived sequence followed by a forward traversal.

We conclude by mentioning another elimination method based on path algebras [170, 171]. For intraprocedural analyses, path expressions amount to regular expressions, and some very fast algorithms exist for solving analyses over such structures.

Practical systems. A number of generic tools for implementing program analyses have been constructed. The analyses are specified using a specification language; in some systems this is a special purpose language (e.g. a functional language) whereas in others it is the implementation language of the tool (e.g. C++). The tools then provide one or more algorithms for solving equations or constraints; the details of these algorithms depend on the scope of the tools; often the tools are restricted to certain classes of languages (e.g. they might not support languages with dynamic dispatch) or

certain classes of analyses (e.g. they may not support interprocedural analyses). Internally, some of the systems work on abstract syntax trees, others work on flow graphs and yet others work directly on the equations or constraints.

The specification languages of tools like *Spare* [177], *System Z* [185] and *PAG* [104] are based on eager functional languages and they support high-level definitions of complete lattices and the associated analysis functions; Spare and System Z are inspired by a denotational approach to program analysis (see e.g. [117]) whereas PAG is closer to the traditional Data Flow Analysis approach. The above tools build on the ideas of Abstract Interpretation by allowing the user to control the complexity of the analyses by specifying operations related to widening operators (see Section 4.2 and Exercise 6.13). Other program analysis tools are based on other specification mechanisms like graph rewriting [17] or modal logic [97].

To improve the efficiency of the generated analysers, some of the tools can take advantage of special properties of the analyses. An example is PAG [104] which supports several implementations of sets including bit vectors, AVL trees (see [4]) and BDD's [26] (binary decision diagrams). Another example is *Sharlit* [172] which provides an equation solver based on elimination techniques for Bit Vector Frameworks.

BANE [8] is a general constraint solver for a very general class of *set constraints* [7] (see the Concluding Remarks to Chapter 3). The constraints are sufficiently general that the system has been used successfully for implementing both Constraint Based Analyses and Type Systems. It is written in Standard ML but also admits a simple textual interface in which a system of constraints can be specified.

Mini Projects

Mini Project 6.1 Comparison of Algorithms

Recall that a *Bit Vector Framework* (see Section 6.2 and Exercise 2.9) is a special case of a Monotone Framework with

- $L = (\mathcal{P}(D), \sqsubseteq)$ where D is a finite set and $\sqsubseteq$ is either $\subseteq$ or $\supseteq$, and
- $\mathcal{F} = \{f : \mathcal{P}(D) \to \mathcal{P}(D) \mid \exists Y_f^1, Y_f^2 \subseteq D : \forall Y \subseteq D : f(Y) = (Y \cap Y_f^1) \cup Y_f^2\}$

Implement a system that will accept a description of a Bit Vector Framework and a description of a constraint system that is an instance of a Bit Vector Framework and that will produce the least solution. It should be possible to

direct the system to use (i) the Round Robin Algorithm (see Table 6.4), or the worklist algorithm of Table 6.1 based on (ii) last-in first-out (see Table 6.2 and Exercise 6.3), (iii) reverse postorder (see Table 6.3 and Exercise 6.6) and (iv) strong components with local reverse postorders (see Table 6.6 and Exercise 6.11).

Design suitable experiments to allow an empirical comparison of the worst-case and average-case performance of the four algorithms.

For the more ambitious: Generalise your system to accept descriptions of Monotone Frameworks and more general instances of constraint systems. Apply the system to the equations and constraints generated in Chapters 2 and 3. ■

Mini Project 6.2 Algorithms for Conditional Constraints

Consider the following simplified form of set constraints (as discussed in the Concluding Remarks to Chapter 3) where the terms on the right hand sides of the constraints are given by:

$$\begin{array}{rcl} t & ::= & x \mid \bot \mid c \mid t_1 \sqcup t_2 \mid t_1 \sqcap t_2 \mid \\ & & \mathsf{if}\ t_1 \neq \bot\ \mathsf{then}\ t_2 \mid \mathsf{if}\ c \sqsubseteq t_1\ \mathsf{then}\ t_2 \end{array}$$

Here x stands for a flow variable, c for a primitive constant (like $\{\texttt{fn x => x}^1\}$ of Example 6.2), and $\mathsf{if} \cdots \mathsf{then} \cdots$ denotes a conditional constraint as considered in Example 6.2.

1. Verify that the assumptions of this chapter still hold; in particular, that the evaluation of terms is still monotone.
2. Observe that once a condition becomes true, it remains so. Modify the LIFO worklist algorithm (see Tables 6.1 and 6.2) such that it simplifies constraints as conditions become true.
3. State and prove a correctness result for the new algorithm. What can you say about its complexity?
4. Modify the new algorithm to implement the optimisation suggested by Exercise 6.3.
5. Investigate the application of these ideas to the material of Chapter 3.

For the more ambitious: Explore the extent to which similar optimisations are possible for the other worklist algorithms of this chapter. Give particular attention to the maintenance of the ordering on the constraints; what are the implications for the complexity of the modified algorithms? ■

Exercises

Exercise 6.1 In Example 6.7 the result of step 1 produced the worklist $\mathsf{W} = [\mathsf{x}_1, \cdots, \mathsf{x}_6]$. Redo the example assuming that step 1 instead produced the worklist $\mathsf{W} = [\mathsf{x}_6, \cdots, \mathsf{x}_1]$. Compare the number of iterations. ■

Exercise 6.2 Consider the program `((fn x => x) (fn y => y))` from Example 6.2. Give a walkthrough of the worklist algorithm of Table 6.1 to solve these constraints using the LIFO extraction strategy of Table 6.2. ■

Exercise 6.3 Modify the LIFO extraction strategy of Table 6.2 so that a constraint is never inserted into the worklist W when it is already present. Then redo Example 6.7 using the modified strategy and compare the number of iterations. ■

Exercise 6.4 In Appendix C we explained that the set $Dom(n)$ of dominators of a node n in a graph (N, A) with handle H can be described as the greatest solution of the following equation system:

$$Dom(n) = \begin{cases} \{n\} & \text{if } n \in H \\ \{n\} \cup \bigcap_{(n',n) \in A} Dom(n') & \text{otherwise} \end{cases}$$

(This can be regarded as an intersection based analysis in the sense of Section 2.1.) Develop a worklist algorithm for computing dominator sets. ■

Exercise 6.5 Consider a constraint system

$$(x_i \sqsupseteq t_i^1 \sqcup \cdots \sqcup t_i^M)_{i=1}^N$$

where each t_i^j has size $O(1)$ and can be evaluated in $O(1)$ steps. Show that the worklist algorithm can solve this system in $O(h \cdot M^2 \cdot N)$ steps. Next show that the constraint system is equivalent to the constraint system

$$((x_i \sqsupseteq t_i^j)_{j=1}^M)_{i=1}^N$$

in the sense that they have the same solutions. Finally show that the worklist algorithm can solve this system in $O(h \cdot M \cdot N)$ steps. ■

Exercise 6.6 Modify the reverse postorder extraction strategy of Table 6.3 so that a constraint is never inserted into W.p when it is already present in either W.p or W.c. Then redo Example 6.9 using the modified strategy and compare the number of iterations. ■

Exercise 6.7 In Section 6.2 we discussed using two priority queues for the worklist algorithm based on reverse postorder: W.c for the constraints to be considered in the current "inner" iteration and W.p for those to be considered in the next "inner" iteration.

Develop a concrete algorithm based on these ideas so that as few operations as possible need to be performed. Explore the idea of adding a constraint $x' \sqsupseteq t'$ to W.c or W.p depending on how rPostorder$[x' \sqsupseteq t']$ compares to rPostorder$[x \sqsupseteq t]$ where $x \sqsupseteq t$ is the constraint just considered in step 2 of Table 6.1. The objective should be to keep W.c as large as possible without destroying the iteration order. Then redo Example 6.9 and compare the number of iterations with the result from Exercise 6.6. ■

Exercise 6.8 Consider the equations used to introduce Available Expressions Analysis in Example 2.5 of Chapter 2; when expressed as a constraint system in the flow variables $\{x_1, \cdots, x_{10}\}$ it takes the form

$$\begin{array}{lclcrcl} x_1 & = & \emptyset & & x_6 & = & x_1 \cup \{\mathsf{a+b}\} \\ x_2 & = & x_6 & & x_7 & = & x_2 \cup \{\mathsf{a*b}\} \\ x_3 & = & x_7 \cap x_{10} & & x_8 & = & x_3 \cup \{\mathsf{a+b}\} \\ x_4 & = & x_8 & & x_9 & = & x_4 \backslash \{\mathsf{a+b}, \mathsf{a*b}, \mathsf{a+1}\} \\ x_5 & = & x_9 & & x_{10} & = & x_5 \cup \{\mathsf{a+b}\} \end{array}$$

where $x_1, \cdots, x_5$ correspond to $\mathsf{AE}_{entry}(1), \cdots, \mathsf{AE}_{entry}(5)$ and $x_6, \cdots, x_{10}$ correspond to $\mathsf{AE}_{exit}(1), \cdots, \mathsf{AE}_{exit}(5)$.

Draw the graph of this system as described in Section 6.2. Use the depth-first search algorithm of Appendix C to assign a reverse postorder numbering to the nodes in the graph. Give a walkthrough of the Round Robin Algorithm to produce a solution to the system. ■

Exercise 6.9 Complete the details of the proof of Lemma 6.11 concerning the correctness of the Round Robin Algorithm. ■

Exercise 6.10 Draw the graph of the constraint system in Exercise 6.8. Identify the strong components of the graph (see Appendix C) and write down a srPostorder numbering for the nodes in the graph. Use the algorithm of Section 6.3 to solve the system. ■

Exercise 6.11 Modify the strong component extraction strategy of Table 6.6 so that a constraint is never inserted into W.p when it is already present in either W.p or W.c. Then redo Example 6.15 using the modified strategy and compare the number of iterations. ■

Exercise 6.12 In Section 6.3 we discussed using two priority queues for the worklist algorithm based on strong components: W.c for the constraints to be considered in the current "inner" iteration, W.pc for those to be considered in the next "inner" iteration, and W.pf for those to be considered in future "outer" iterations.

Develop a concrete algorithm based on these ideas so that as few operations as possible need to be performed. Explore the idea of adding a constraint $x' \sqsupseteq t'$ to W.c, W.pc or W.pf depending on how srPostorder$[x' \sqsupseteq t']$ compares to srPostorder$[x \sqsupseteq t]$ where $x \sqsupseteq t$ is the constraint just considered in step 2 of Table 6.1. The objective should be to keep W.c and W.pc as large as possible without destroying the iteration order. Then redo Example 6.15 and compare the number of iterations with the result from Exercise 6.11. ■

Exercise 6.13 Modify the worklist algorithm of Table 6.1 to use a *widening operator* (see Section 4.2) over L. Does the widening operator over L give rise to a widening operator over $X \to L$? Prove that the resulting worklist algorithm computes an upper approximation to the least solution of the constraint system. Also prove that it always terminates when L is a complete lattice (even when it does not satisfy the Ascending Chain Condition). ■

Exercise 6.14 A monotone function f is *fast* whenever $f \circ f \sqsubseteq f \sqcup id$ and a Monotone Framework $(L, \mathcal{F})$ is fast whenever all functions in $\mathcal{F}$ are fast. Notice that if a monotone function is idempotent then it is also fast; use this to show that all Bit Vector Frameworks (see Exercise 2.9) are fast.

It is frequently possible to approximate a function by a fast function. Let f be a monotone function and define the following sequence (for $n \geq 0$):

$$f^{\langle n \rangle} = (f \sqcup id)^n$$

Assuming that there exists a number i_f such that $f^{\langle i_f \rangle} = f^{\langle i_f + 1 \rangle}$ we define the *fastness closure* of f to be:

$$\bar{f} = f^{\langle i_f \rangle}$$

Show that $f^{\langle i_f \rangle} = f^{\langle j \rangle}$ for $j \geq i_f$ and conclude that the the fastness closure is well-defined. Next prove that $\bar{f}$ is idempotent and hence fast. Finally show that $\bar{f} \sqsupseteq f \sqcup id$ and that they are equal when f is fast and distributive (i.e. additive).

Suppose that f is distributive (and hence monotone) and consider the functional F defined by

$$F(g) = id \sqcup g \circ f$$

as might arise for a simple while-loop where the body is described by f. Prove that

$$F^{n+1}(\bot) = (f \sqcup id)^n = \bigsqcup_{j=0}^{n} f^j$$

holds for $n \geq 0$. Conclude that if $\bar{f}$ is the fastness closure of f then $lfp(F) = \bar{f}$. This shows that for fast Distributive Frameworks (including all Bit Vector Frameworks) there is a simple non-iterative way of solving data flow equations for programs with a simple loop structure. ■

Appendix A

Partially Ordered Sets

Partially ordered sets and complete lattices play a crucial role in program analysis and in this appendix we shall summarise some of their properties. We review the basic approaches for how to construct complete lattices from other complete lattices and state the central properties of partially ordered sets satisfying the Ascending Chain and Descending Chain Conditions. We then review the classical results about least and greatest fixed points.

A.1 Basic Definitions

Partially ordered set. A *partial ordering* is a relation $\sqsubseteq: L \times L \rightarrow \{true, false\}$ that is reflexive (i.e. $\forall l : l \sqsubseteq l$), transitive (i.e. $\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$), and anti-symmetric (i.e. $\forall l_1, l_2 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$). A *partially ordered set* $(L, \sqsubseteq)$ is a set L equipped with a partial ordering $\sqsubseteq$ (sometimes written $\sqsubseteq_L$). We shall write $l_2 \sqsupseteq l_1$ for $l_1 \sqsubseteq l_2$ and $l_1 \sqsubset l_2$ for $l_1 \sqsubseteq l_2 \wedge l_1 \neq l_2$.

A subset Y of L has $l \in L$ as an *upper bound* if $\forall l' \in Y : l' \sqsubseteq l$ and as a *lower bound* if $\forall l' \in Y : l' \sqsupseteq l$. A *least upper bound* l of Y is an upper bound of Y that satisfies $l \sqsubseteq l_0$ whenever l_0 is another upper bound of Y; similarly, a *greatest lower bound* l of Y is a lower bound of Y that satisfies $l_0 \sqsubseteq l$ whenever l_0 is another lower bound of Y. Note that subsets Y of a partially ordered set L need not have least upper bounds nor greatest lower bounds but when they exist they are unique (since $\sqsubseteq$ is anti-symmetric) and they are denoted $\bigsqcup Y$ and $\bigsqcap Y$, respectively. Sometimes $\bigsqcup$ is called the *join operator* and $\bigsqcap$ the *meet operator* and we shall write $l_1 \sqcup l_2$ for $\bigsqcup\{l_1, l_2\}$ and similarly $l_1 \sqcap l_2$ for $\bigsqcap\{l_1, l_2\}$.

Complete lattice. A *complete lattice* $L = (L, \sqsubseteq) = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ is a partially ordered set $(L, \sqsubseteq)$ such that all subsets have least upper bounds

Figure A.1: Two complete lattices.

as well as greatest lower bounds. Furthermore, $\bot = \bigsqcup \emptyset = \bigsqcap L$ is the *least element* and $\top = \bigsqcap \emptyset = \bigsqcup L$ is the *greatest element.*

Example A.1 If $L = (\mathcal{P}(S), \subseteq)$ for some set S then $\sqsubseteq$ is $\subseteq$ and $\bigsqcup Y = \bigcup Y$, $\bigsqcap Y = \bigcap Y$, $\bot = \emptyset$ and $\top = S$. If $L = (\mathcal{P}(S), \supseteq)$ then $\sqsubseteq$ is $\supseteq$ and $\bigsqcup Y = \bigcap Y, \bigsqcap Y = \bigcup Y$, $\bot = S$ and $\top = \emptyset$.

Hence $(\mathcal{P}(S), \subseteq)$ as well as $(\mathcal{P}(S), \supseteq)$ are complete lattices. In the case where $S = \{1, 2, 3\}$ the two complete lattices are shown on Figure A.1; these drawings are often called Hasse diagrams. Here a line "going upwards" from some l_1 to some l_2 means that $l_1 \sqsubseteq l_2$; we do not draw lines that follow from reflexivity or transitivity of the partial ordering. ∎

Lemma A.2 For a partially ordered set $L = (L, \sqsubseteq)$ the claims

(i) L is a complete lattice,

(ii) every subset of L has a least upper bound, and

(iii) every subset of L has a greatest lower bound

are equivalent. ∎

Proof Clearly (i) implies (ii) and (iii). To show that (ii) implies (i) let $Y \subseteq L$ and define

$$\bigsqcap Y \quad = \quad \bigsqcup \{l \in L \mid \forall l' \in Y : l \sqsubseteq l'\} \tag{A.1}$$

and let us prove that this indeed defines a greatest lower bound. All the elements of the set on the right hand side of (A.1) are lower bounds of Y so clearly (A.1) defines a lower bound of Y. Since any lower bound of Y will be in the set it follows that (A.1) defines the greatest lower bound of Y. Thus (i) holds.

To show that (iii) implies (i) we define $\bigsqcup Y = \bigsqcap\{l \in L \mid \forall l' \in Y : l' \sqsubseteq l\}$. Arguments analogous to those above show that this defines a least upper bound and that (i) holds. ∎

Moore family. A *Moore family* is a subset Y of a complete lattice $L = (L, \sqsubseteq)$ that is closed under greatest lower bounds: $\forall Y' \subseteq Y : \bigsqcap Y' \in Y$. It follows that a Moore family always contains a least element, $\bigsqcap Y$, and a greatest element, $\bigsqcap \emptyset$, which equals the greatest element, $\top$, from L; in particular, a Moore family is never empty.

Example A.3 Consider the complete lattice $(\mathcal{P}(S), \subseteq)$ of Figure A.1 (a). The subsets

$$\{\{2\}, \{1,2\}, \{2,3\}, \{1,2,3\}\} \text{ and } \{\emptyset, \{1,2,3\}\}$$

are both Moore families, whereas neither of

$$\{\{1\}, \{2\}\} \text{ and } \{\emptyset, \{1\}, \{2\}, \{1,2\}\}$$

are. ∎

Properties of functions. A function $f : L_1 \to L_2$ between partially ordered sets $L_1 = (L_1, \sqsubseteq_1)$ and $L_2 = (L_2, \sqsubseteq_2)$ is *surjective* (or *onto* or *epic*) if

$$\forall l_2 \in L_2 : \exists l_1 \in L_1 : f(l_1) = l_2$$

and it is *injective* (or 1–1 or *monic*) if

$$\forall l, l' \in L_1 : f(l) = f(l') \Rightarrow l = l'$$

The function f is *monotone* (or *isotone* or *order-preserving*) if

$$\forall l, l' \in L_1 : l \sqsubseteq_1 l' \quad \Rightarrow \quad f(l) \sqsubseteq_2 f(l')$$

It is an *additive* function (or a *join morphism*, sometimes called a *distributive* function) if

$$\forall l_1, l_2 \in L_1 : f(l_1 \sqcup l_2) = f(l_1) \sqcup f(l_2)$$

and it is called a *multiplicative* function (or a *meet morphism*) if

$$\forall l_1, l_2 \in L_1 : f(l_1 \sqcap l_2) = f(l_1) \sqcap f(l_2)$$

The function f is a *completely additive* function (or a *complete join morphism*) if for all $Y \subseteq L_1$:

$$f(\bigsqcup\nolimits_1 Y) = \bigsqcup\nolimits_2 \{f(l') \mid l' \in Y\} \text{ whenever } \bigsqcup\nolimits_1 Y \text{ exists}$$

and it is *completely multiplicative* (or a *complete meet morphism*) if for all $Y \subseteq L_1$:

$$f(\bigsqcap_1 Y) = \bigsqcap_2 \{f(l') \mid l' \in Y\} \text{ whenever } \bigsqcap_1 Y \text{ exists}$$

Clearly $\bigsqcup_1 Y$ and $\bigsqcap_1 Y$ always exist when L_1 is a complete lattice; when L_2 is not a complete lattice the above statements also require the appropriate least upper bounds and greatest lower bounds to exist in L_2. The function f is *affine* if for all *non-empty* $Y \subseteq L_1$

$$f(\bigsqcup_1 Y) = \bigsqcup_2 \{f(l') \mid l' \in Y\} \text{ whenever } \bigsqcup_1 Y \text{ exists (and } Y \neq \emptyset)$$

and it is *strict* if $f(\bot_1) = \bot_2$; note that a function is completely additive if and only if it is both affine and strict.

Lemma A.4 If $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ and $M = (M, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ are complete lattices and M is finite then the three conditions

(i) $\gamma : M \to L$ is monotone,

(ii) $\gamma(\top) = \top$, and

(iii) $\gamma(m_1 \sqcap m_2) = \gamma(m_1) \sqcap \gamma(m_2)$ whenever $m_1 \not\sqsubseteq m_2 \wedge m_2 \not\sqsubseteq m_1$

are jointly equivalent to $\gamma : M \to L$ being completely multiplicative. ∎

Proof First note that if γ is completely multiplicative then (i), (ii) and (iii) hold. For the converse note that by monotonicity of γ we have $\gamma(m_1 \sqcap m_2) = \gamma(m_1) \sqcap \gamma(m_2)$ also when $m_1 \sqsubseteq m_2 \vee m_2 \sqsubseteq m_1$. We then prove by induction on the (finite) cardinality of $M' \subseteq M$ that:

$$\gamma(\bigsqcap M') = \bigsqcap \{\gamma(m) \mid m \in M'\} \tag{A.2}$$

If the cardinality of M' is 0 then (A.2) follows from (ii). If the cardinality of M' is larger than 0 then we write $M' = M'' \cup \{m''\}$ where $m'' \notin M''$; this ensures that the cardinality of M'' is strictly less than that of M'; hence:

$$\begin{aligned}
\gamma(\bigsqcap M') &= \gamma((\bigsqcap M'') \sqcap m'') \\
&= \gamma(\bigsqcap M'') \sqcap \gamma(m'') \\
&= (\bigsqcap \{\gamma(m) \mid m \in M''\}) \sqcap \gamma(m'') \\
&= \bigsqcap \{\gamma(m) \mid m \in M'\}
\end{aligned}$$

This proves the result. ∎

Lemma A.5 A function $f : (\mathcal{P}(D), \subseteq) \to (\mathcal{P}(E), \subseteq)$ is *affine* if and only if there exists a function $\varphi : D \to \mathcal{P}(E)$ and an element $\varphi_\emptyset \in \mathcal{P}(E)$ such that

$$f(Y) = \bigcup \{\varphi(d) \mid d \in Y\} \cup \varphi_\emptyset$$

The function f is completely additive if and only if additionally $\varphi_\emptyset = \emptyset$. ∎

Proof Suppose that f is of the form displayed and let $\mathcal{Y}$ be a non-empty set; Then

$$\begin{aligned}
\bigcup\{f(Y) \mid Y \in \mathcal{Y}\} &= \bigcup\{\bigcup\{\varphi(d) \mid d \in Y\} \cup \varphi_\emptyset \mid Y \in \mathcal{Y}\} \\
&= \bigcup\{\bigcup\{\varphi(d) \mid d \in Y\} \mid Y \in \mathcal{Y}\} \cup \varphi_\emptyset \\
&= \bigcup\{\varphi(d) \mid d \in \bigcup \mathcal{Y}\} \cup \varphi_\emptyset \\
&= f(\bigcup \mathcal{Y})
\end{aligned}$$

showing that f is affine.

Next suppose that f is affine and define $\varphi(d) = f(\{d\})$ and $\varphi_\emptyset = f(\emptyset)$. For $Y \in \mathcal{P}(D)$ let $\mathcal{Y} = \{\{d\} \mid d \in Y\} \cup \{\emptyset\}$ and note that $Y = \bigcup \mathcal{Y}$ and $\mathcal{Y} \neq \emptyset$. Then

$$\begin{aligned}
f(Y) &= f(\bigcup \mathcal{Y}) \\
&= \bigcup(\{f(\{d\}) \mid d \in Y\} \cup \{f(\emptyset)\}) \\
&= \bigcup(\{\varphi(d)) \mid d \in Y\} \cup \{\varphi_\emptyset\}) \\
&= \bigcup\{\varphi(d) \mid d \in Y\} \cup \varphi_\emptyset
\end{aligned}$$

so f can be written in the required form. The additional statement about completely additivity is straightforward. ∎

An *isomorphism* from a partially ordered set $(L_1, \sqsubseteq_1)$ to a partially ordered set $(L_2, \sqsubseteq_2)$ is a *monotone* function $\theta : L_1 \to L_2$ such that there exists a (necessarily unique) monotone function $\theta^{-1} : L_2 \to L_1$ with $\theta \circ \theta^{-1} = id_2$ and $\theta^{-1} \circ \theta = id_1$ (where id_i is the identity function over L_i, $i = 1, 2$).

A.2 Construction of Complete Lattices

Complete lattices can be combined to construct new complete lattices. We shall first see how to construct products and then two kinds of function spaces.

Cartesian product. Let $L_1 = (L_1, \sqsubseteq_1)$ and $L_2 = (L_2, \sqsubseteq_2)$ be partially ordered sets. Define $L = (L, \sqsubseteq)$ by

$$L = \{(l_1, l_2) \mid l_1 \in L_1 \wedge l_2 \in L_2\}$$

and

$$(l_{11}, l_{21}) \sqsubseteq (l_{12}, l_{22}) \text{ iff } l_{11} \sqsubseteq_1 l_{12} \wedge l_{21} \sqsubseteq_2 l_{22}$$

It is then straightforward to verify that L is a partially ordered set. If additionally each $L_i = (L_i, \sqsubseteq_i, \bigsqcup_i, \bigsqcap_i, \bot_i, \top_i)$ is a complete lattice then so is $L = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ and furthermore

$$\bigsqcup Y = (\ \bigsqcup_1\{l_1 \mid \exists l_2 : (l_1, l_2) \in Y\}\ ,\ \bigsqcup_2\{l_2 \mid \exists l_1 : (l_1, l_2) \in Y\}\ )$$

and $\bot = (\bot_1, \bot_2)$ and similarly for $\bigsqcap Y$ and $\top$. We often write $L_1 \times L_2$ for L and call it the *cartesian product* of L_1 and L_2.

A variant of the cartesian product called the *smash product* is obtained if we require that all the pairs (l_1, l_2) of the lattice satisfy $l_1 = \bot_1 \Leftrightarrow l_2 = \bot_2$.

Total function space. Let $L_1 = (L_1, \sqsubseteq_1)$ be a partially ordered set and let S be a set. Define $L = (L, \sqsubseteq)$ by

$$L = \{f : S \to L_1 \mid f \text{ is a total function}\}$$

and

$$f \sqsubseteq f' \text{ iff } \forall s \in S : f(s) \sqsubseteq_1 f'(s)$$

It is then straightforward to verify that L is a partially ordered set. If additionally $L_1 = (L_1, \sqsubseteq_1, \bigsqcup_1, \bigsqcap_1, \bot_1, \top_1)$ is a complete lattice then so is $L = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ and furthermore

$$\bigsqcup Y = \lambda s. \bigsqcup_1 \{f(s) \mid f \in Y\}$$

and $\bot = \lambda s. \bot_1$ and similarly for $\bigsqcap Y$ and $\top$. We often write $S \to L_1$ for L and call it the *total function space* from S to L_1.

Monotone function space. Again let $L_1 = (L_1, \sqsubseteq_1)$ and $L_2 = (L_2, \sqsubseteq_2)$ be partially ordered sets. Now define $L = (L, \sqsubseteq)$ by

$$L = \{f : L_1 \to L_2 \mid f \text{ is a monotone function}\}$$

and

$$f \sqsubseteq f' \text{ iff } \forall l_1 \in L_1 : f(l_1) \sqsubseteq_2 f'(l_1)$$

It is then straightforward to verify that L is a partially ordered set. If additionally each $L_i = (L_i, \sqsubseteq_i, \bigsqcup_i, \bigsqcap_i, \bot_i, \top_i)$ is a complete lattice then so is $L = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ and furthermore

$$\bigsqcup Y = \lambda l_1. \bigsqcup_2 \{f(l_1) \mid f \in Y\}$$

and $\bot = \lambda l_1. \bot_2$ and similarly for $\bigsqcap Y$ and $\top$. We often write $L_1 \to L_2$ for L and call it the *monotone function space* from L_1 to L_2.

A.3 Chains

The ordering $\sqsubseteq$ on a complete lattice $L = (L, \sqsubseteq)$ expresses when one property is better (or more precise) than another property. When performing a program analysis we will typically construct a sequence of elements in L and it is the general properties of such sequences that we shall study now. In the next section we will be more explicit and consider the sequences obtained during a fixed point computation.

Chains. A subset $Y \subseteq L$ of a partially ordered set $L = (L, \sqsubseteq)$ is a *chain* if

$$\forall l_1, l_2 \in Y : (l_1 \sqsubseteq l_2) \vee (l_2 \sqsubseteq l_1)$$

Thus a chain is a (possibly empty) subset of L that is totally ordered. We shall say that it is a *finite chain* if it is a finite subset of L.

A sequence $(l_n)_n = (l_n)_{n \in \mathbf{N}}$ of elements in L is an *ascending chain* if

$$n \leq m \Rightarrow l_n \sqsubseteq l_m$$

Writing $(l_n)_n$ also for $\{l_n \mid n \in \mathbf{N}\}$ it is clear that an ascending chain also is a chain. Similarly, a sequence $(l_n)_n$ is a *descending chain* if

$$n \leq m \Rightarrow l_n \sqsupseteq l_m$$

and clearly a descending chain is also a chain.

We shall say that a sequence $(l_n)_n$ *eventually stabilises* if and only if

$$\exists n_0 \in \mathbf{N} : \forall n \in \mathbf{N} : n \geq n_0 \Rightarrow l_n = l_{n_0}$$

For the sequence $(l_n)_n$ we write $\bigsqcup_n l_n$ for $\bigsqcup\{l_n \mid n \in \mathbf{N}\}$ and similarly we write $\bigsqcap_n l_n$ for $\bigsqcap\{l_n \mid n \in \mathbf{N}\}$.

Ascending Chain and Descending Chain Conditions. We shall say that a partially ordered set $L = (L, \sqsubseteq)$ has finite *height* if and only if all chains are finite. It has finite height *at most* h if all chains contain at most $h+1$ elements; it has finite height h if additionally there is a chain with $h+1$ elements. The partially ordered set L satisfies the *Ascending Chain Condition* if and only if all ascending chains eventually stabilise. Similarly, it satisfies the *Descending Chain Condition* if and only if all descending chains eventually stabilise. These concepts are related as follows:

Lemma A.6 A partially ordered set $L = (L, \sqsubseteq)$ has finite height if and only if it satisfies both the Ascending and Descending Chain Conditions. ∎

Proof First assume that L has finite height. If $(l_n)_n$ is an ascending chain then it must be a finite chain and hence eventually stabilise; thus L satisfies the Ascending Chain Condition. In a similar way it is shown that L satisfies the Descending Chain Condition.

Next assume that L satisfies the Ascending Chain Condition as well as the Descending Chain Condition and consider a chain $Y \subseteq L$. We shall prove that Y is a finite chain. This is obvious if Y is empty so assume that it is not. Then also $(Y, \sqsubseteq)$ is a non-empty partially ordered set satisfying the Ascending and Descending Chain Conditions.

As an auxiliary result we shall now show that

$$\text{each non-empty subset } Y' \text{ of } Y \text{ contains a least element} \tag{A.3}$$

Figure A.2: Two partially ordered sets.

To see this we shall construct a descending chain $(l'_n)_n$ in Y' as follows: first let l'_0 be an arbitrary element of Y'. For the inductive step let $l'_{n+1} = l'_n$ if l'_n is the least element of Y'; otherwise we can find $l'_{n+1} \in Y'$ such that $l'_{n+1} \sqsubseteq l'_n \wedge l'_{n+1} \neq l'_n$. Clearly $(l'_n)_n$ is a descending chain in Y; since Y satisfies the Descending Chain Condition the chain will eventually stabilise, i.e. $\exists n'_0 : \forall n \geq n'_0 : l'_n = l'_{n'_0}$ and the construction is such that $l'_{n'_0}$ is the least element of Y'.

Returning to the main proof obligation we shall now construct an ascending chain $(l_n)_n$ in Y. Using (A.3) each l_n is chosen as the least element of the set $Y \setminus \{l_0, \cdots, l_{n-1}\}$ as long as the latter set is non-empty, and this yields $l_{n-1} \sqsubseteq l_n \wedge l_{n-1} \neq l_n$; when $Y \setminus \{l_0, \cdots, l_{n-1}\}$ is empty we set $l_n = l_{n-1}$, and since Y is non-empty we know that $n > 0$. Thus we have an ascending chain in Y and using the Ascending Chain Condition we have $\exists n_0 : \forall n \geq n_0 : l_n = l_{n_0}$. But this means that $Y \setminus \{l_0, \cdots, l_{n_0}\} = \emptyset$ since this is the only way we can achieve that $l_{n_0+1} = l_{n_0}$. It follows that Y is finite. ∎

Example A.7 The partially ordered set of Figure A.2 (a) satisfies the Ascending Chain Condition but does not have finite height; the partially ordered set of Figure A.2 (b) satisfies the Descending Chain Condition but does not have finite height. ∎

One can show that each of the three conditions finite height, ascending chain, and descending chain, is preserved under the construction of cartesian product: if L_1 and L_2 satisfies one of the conditions then $L_1 \times L_2$ will also satisfy that condition. The construction of total function spaces $S \to L$ only preserves the conditions of L if S is finite and the construction of monotone function spaces $L_1 \to L_2$ does not in general preserve the conditions.

An alternative characterisation of complete lattices satisfying the Ascending Chain Condition is given by the following result:

Lemma A.8 For a partially ordered set $L = (L, \sqsubseteq)$ the conditions

(i) L is a complete lattice satisfying the Ascending Chain Condition, and

(ii) L has a least element, $\perp$, and binary least upper bounds and satisfies the Ascending Chain Condition

are equivalent. ∎

Proof It is immediate that (i) implies (ii) so let us prove that (ii) implies (i). Using Lemma A.2 it suffices to prove that all subsets Y of L have a least upper bound $\bigsqcup Y$. If Y is empty clearly $\bigsqcup Y = \perp$. If Y is finite and non-empty then we can write $Y = \{y_1, \cdots, y_n\}$ for $n \geq 1$ and it follows that $\bigsqcup Y = (\cdots(y_1 \sqcup y_2) \sqcup \cdots) \sqcup y_n$.

If Y is infinite then we construct a sequence $(l_n)_n$ of elements of L: let l_0 be an arbitrary element y_0 of Y and given l_n take $l_{n+1} = l_n$ in the case where $\forall y \in Y : y \sqsubseteq l_n$ and take $l_{n+1} = l_n \sqcup y_{n+1}$ in the case where some $y_{n+1} \in Y$ satisfies $y_{n+1} \not\sqsubseteq l_n$. Clearly this sequence is an ascending chain. Since L satisfies the Ascending Chain Condition it follows that the chain eventually stabilises, i.e. there exists n such that $l_n = l_{n+1} = \cdots$. This means that $\forall y \in Y : y \sqsubseteq l_n$ because if $y \not\sqsubseteq l_n$ then $l_n \neq l_n \sqcup y$ and we have a contradiction. So we have constructed an upper bound for Y. Since it is actually the least upper bound of the subset $\{y_0, \cdots, y_n\}$ of Y it follows that it is also the least upper bound of Y. ∎

A related result is the following:

Lemma A.9 For a complete lattice $L = (L, \sqsubseteq)$ satisfying the Ascending Chain Condition and a total function $f : L \to L$, the conditions

(i) f is additive, i.e. $\forall l_1, l_2 : f(l_1 \sqcup l_2) = f(l_1) \sqcup f(l_2)$, and

(ii) f is affine, i.e. $\forall Y \subseteq L, Y \neq \emptyset : f(\bigsqcup Y) = \bigsqcup\{f(l) \mid l \in Y\}$

are equivalent and in both cases f is a monotone function. ∎

Proof It is immediate that (ii) implies (i): take $Y = \{l_1, l_2\}$. It is also immediate that (i) implies that f is monotone since $l_1 \sqsubseteq l_2$ is equivalent to $l_1 \sqcup l_2 = l_2$.

Next suppose that f satisfies (i) and let us prove (ii). If Y is finite we can write $Y = \{y_1, \cdots, y_n\}$ for $n \geq 1$ and

$$f(\bigsqcup Y) = f(y_1 \sqcup \cdots \sqcup y_n) = f(y_1) \sqcup \cdots \sqcup f(y_n) \sqsubseteq \bigsqcup\{f(l) \mid l \in Y\}$$

If Y is infinite then the construction of the proof of Lemma A.8 gives $\bigsqcup Y = l_n$ and $l_n = y_n \sqcup \cdots \sqcup y_0$ for some $y_i \in Y$ and $0 \leq i \leq n$. We then have

$$f(\bigsqcup Y) = f(l_n) = f(y_n \sqcup \cdots \sqcup y_0) = f(y_n) \sqcup \cdots \sqcup f(y_0) \sqsubseteq \bigsqcup\{f(l) \mid l \in Y\}$$

Furthermore

$$f(\bigsqcup Y) \sqsupseteq \bigsqcup\{f(l) \mid l \in Y\}$$

follows from the monotonicity of f. This completes the proof. ∎

A.4 Fixed Points

Reductive and extensive functions. Consider a monotone function $f : L \to L$ on a complete lattice $L = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$. A *fixed point* of f is an element $l \in L$ such that $f(l) = l$ and we write

$$Fix(f) = \{l \mid f(l) = l\}$$

for the set of fixed points. The function f is *reductive at* l if and only if $f(l) \sqsubseteq l$ and we write

$$Red(f) = \{l \mid f(l) \sqsubseteq l\}$$

for the set of elements upon which f is reductive; we shall say that f itself is *reductive* if $Red(f) = L$. Similarly, the function f is *extensive at* l if and only if $f(l) \sqsupseteq l$ and we write

$$Ext(f) = \{l \mid f(l) \sqsupseteq l\}$$

for the set of elements upon which f is extensive; we shall say that f itself is *extensive* if $Ext(f) = L$.

Since L is a complete lattice it is always the case that the set $Fix(f)$ will have a greatest lower bound in L and we denote it by $lfp(f)$:

$$lfp(f) = \bigsqcap Fix(f)$$

Similarly, the set $Fix(f)$ will have a least upper bound in L and we denote it by $gfp(f)$:

$$gfp(f) = \bigsqcup Fix(f)$$

We then have the following result, known as *Tarski's Fixed Point Theorem*, showing that $lfp(f)$ is the *least fixed point* of f and that $gfp(f)$ is the *greatest fixed point* of f:

Proposition A.10
Let $L = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ be a complete lattice. If $f : L \to L$ is a monotone function then $lfp(f)$ and $gfp(f)$ satisfy:

$$\begin{array}{rcll} lfp(f) & = & \bigsqcap Red(f) & \in \; Fix(f) \\ gfp(f) & = & \bigsqcup Ext(f) & \in \; Fix(f) \end{array}$$

Proof To prove the claim for $lfp(f)$ we define $l_0 = \bigsqcap Red(f)$. We shall first show that $f(l_0) \sqsubseteq l_0$ so that $l_0 \in Red(f)$. Since $l_0 \sqsubseteq l$ for all $l \in Red(f)$ and f is monotone we have

$$f(l_0) \sqsubseteq f(l) \sqsubseteq l \text{ for all } l \in Red(f)$$

Figure A.3: Fixed points of f.

and hence $f(l_0) \sqsubseteq l_0$. To prove that $l_0 \sqsubseteq f(l_0)$ we observe that $f(f(l_0)) \sqsubseteq f(l_0)$ showing that $f(l_0) \in Red(f)$ and hence $l_0 \sqsubseteq f(l_0)$ by definition of l_0. Together this shows that l_0 is a fixed point of f so $l_0 \in Fix(f)$. To see that l_0 is least in $Fix(f)$ simply note that $Fix(f) \subseteq Red(f)$. It follows that $lfp(f) = l_0$.

The claim for $gfp(f)$ is proved in a similar way. ∎

In denotational semantics it is customary to iterate to the least fixed point by taking the least upper bound of the sequence $(f^n(\bot))_n$. However, we have not imposed any continuity requirements on f (e.g. that $f(\bigsqcup_n l_n) = \bigsqcup_n(f(l_n))$ for all ascending chains $(l_n)_n$) and consequently we cannot be sure to actually reach the fixed point. In a similar way one could consider the greatest lower bound of the sequence $(f^n(\top))_n$. One can show that

$$\begin{aligned} \bot \sqsubseteq f^n(\bot) \sqsubseteq \bigsqcup_n f^n(\bot) &\sqsubseteq lfp(f) \\ &\sqsubseteq gfp(f) \sqsubseteq \bigsqcap_n f^n(\top) \sqsubseteq f^n(\top) \sqsubseteq \top \end{aligned}$$

as is illustrated in Figure A.3; indeed all inequalities (i.e. $\sqsubseteq$) can be strict (i.e. $\sqsubset$). However, if L satisfies the Ascending Chain Condition then there exists n such that $f^n(\bot) = f^{n+1}(\bot)$ and hence $lfp(f) = f^n(\bot)$. (Indeed any monotone function f over a partially ordered set satisfying the Ascending Chain Condition is also continuous.) Similarly, if L satisfies the Descending Chain Condition then there exists n such that $f^n(\top) = f^{n+1}(\top)$ and hence $gfp(f) = f^n(\top)$.

Remark (*for readers familiar with ordinal numbers*). It is possible always to obtain $lfp(f)$ as the limit of an ascending (transfinite) sequence but one may have to iterate through the ordinals. To this effect define $f^{\uparrow\kappa} \in L$ for an ordinal κ by the equation

$$f^{\uparrow\kappa} = f(\bigsqcup_{\kappa'<\kappa} f^{\uparrow\kappa'})$$

and note that for a natural number n we have $f^{\uparrow n} = f^{n+1}(\bot)$. Then $lfp(f) = f^{\uparrow\kappa}$ whenever κ is a cardinal number strictly greater than the cardinality of L, e.g. κ may be taken to be the cardinality of $\mathcal{P}(L)$. A similar construction allows to obtain $gfp(f)$ as the limit of a descending (transfinite) chain. ∎

Concluding Remarks

For more information on partially ordered sets consult a text book (e.g. [43]).

Appendix B

Induction and Coinduction

We begin by reviewing a number of techniques for conducting inductive proofs. We then motivate the concept of coinduction and finally formulate a general proof principle for coinduction. This makes heavy use of Tarski's Fixed Point Theorem (Proposition A.10).

B.1 Proof by Induction

Mathematical induction. Perhaps the best known induction principle is that of *mathematical induction.* To prove that a property, $Q(n)$, holds for all natural numbers, n, we establish

$$Q(0)$$
$$\forall n : Q(n) \Rightarrow Q(n+1)$$

and conclude

$$\forall n : Q(n)$$

Formally, the correctness of mathematical induction can be related to the fact that each natural number is either 0 or the successor, $n+1$, of some other natural number, n. Thus the proof principle reflects the way the natural numbers are constructed.

Structural induction. Mathematical induction allows us to perform induction on the *size* of any structure for which a notion of size can be defined; this is just a mapping from the structure into the natural numbers. As an example consider an algebraic data type given by

$$d \in \mathbf{D}$$
$$d ::= \mathsf{Base} \mid \mathsf{Con}_1(d) \mid \mathsf{Con}_2(d, d)$$

where Base is a base case, Con_1 is a unary constructor and Con_2 is a binary constructor. To prove that a certain property, $Q(d)$, holds for all elements, d, of D we can define a size measure:

$$\begin{aligned} \mathit{size}(\mathsf{Base}) &= 0 \\ \mathit{size}(\mathsf{Con}_1(d)) &= 1 + \mathit{size}(d) \\ \mathit{size}(\mathsf{Con}_2(d_1, d_2)) &= 1 + \mathit{size}(d_1) + \mathit{size}(d_2) \end{aligned}$$

and then proceed by mathematical induction on $\mathit{size}(d)$ to prove $Q(d)$.

Alternatively we can conceal the mathematical induction within a principle of *structural induction*: we must then show

$$\begin{aligned} & Q(\mathsf{Base}) \\ & \forall d : Q(d) \Rightarrow Q(\mathsf{Con}_1(d)) \\ & \forall d_1, d_2 : Q(d_1) \wedge Q(d_2) \Rightarrow Q(\mathsf{Con}_2(d_1, d_2)) \end{aligned}$$

from which we conclude

$$\forall d : Q(d)$$

Once again the proof principle reflects the way the data are constructed.

Induction on the shape. Now suppose that Base represents 0, that $\mathsf{Con}_1(d)$ represents $d + 1$, and that $\mathsf{Con}_2(d_1, d_2)$ represents $d_1 + d_2$. We can then define a Natural Semantics

$$d \rightarrow n$$

for evaluating d into the number, n, it represents:

$$\begin{array}{ll} [\mathit{base}] & \mathsf{Base} \rightarrow 0 \\ [\mathit{con}_1] & \dfrac{d \rightarrow n}{\mathsf{Con}_1(d) \rightarrow n + 1} \\ [\mathit{con}_2] & \dfrac{d_1 \rightarrow n_1 \quad d_2 \rightarrow n_2}{\mathsf{Con}_2(d_1, d_2) \rightarrow n_1 + n_2} \end{array}$$

This defines a notion of evaluation trees, $d \overset{\nabla}{\rightarrow} n$: there is one base case ($[\mathit{base}]$) and two constructors ($[\mathit{con}_1]$ and $[\mathit{con}_2]$). Again we can perform induction on the size of the evaluation trees but as above it is helpful to conceal the mathematical induction within a principle of *induction on the shape* of inference trees: we must show

$$Q(\mathsf{Base} \rightarrow 0)$$

$$\forall (d \overset{\nabla}{\rightarrow} n) : Q(d \overset{\nabla}{\rightarrow} n) \Rightarrow Q\left(\frac{d \overset{\nabla}{\rightarrow} n}{\mathsf{Con}_1(d) \rightarrow n + 1}\right)$$

$$\forall (d_1 \xrightarrow{\nabla} n_1), (d_2 \xrightarrow{\nabla} n_2) : Q(d_1 \xrightarrow{\nabla} n_1) \wedge Q(d_2 \xrightarrow{\nabla} n_2) \Rightarrow$$

$$Q\left(\frac{d_1 \xrightarrow{\nabla} n_1 \quad d_2 \xrightarrow{\nabla} n_2}{\mathsf{Con}_2(d_1, d_2) \rightarrow n_1 + n_2}\right)$$

from which we conclude

$$\forall (d \xrightarrow{\nabla} n) : Q(d \xrightarrow{\nabla} n)$$

As is to be expected, the proof principle once again reflects the way evaluation trees are constructed.

Course of values induction. All of the above induction principles have been constructive in the sense that we establish a predicate for the base cases and then show that it is maintained by all constructors. A variant of mathematical induction with a different flavour requires proving

$$\forall n : (\forall m < n : Q(m)) \Rightarrow Q(n)$$

from which we conclude

$$\forall n : Q(n)$$

Here the base case is dealt with in the same manner as the induction step. This induction principle is called *course of values induction.*

Well-founded induction. Course of values induction is an instance of a very powerful induction principle called *well-founded induction.* Given a partially ordered set $(D, \preceq)$, the partial ordering is a *well-founded ordering* if there is no infinite decreasing sequence

$$d_1 \succ d_2 \succ d_3 \succ \cdots$$

where $d \succ d'$ means $d' \preceq d \wedge d \neq d'$ – this amounts to the Descending Chain Condition studied in Appendix A. The principle of well-founded induction then says: if we show

$$\forall d : (\forall d' \prec d : Q(d')) \Rightarrow Q(d)$$

we may then conclude

$$\forall d : Q(d)$$

(The proof of correctness of this principle is along the lines of the proof of (A.3) in Lemma A.6 and can be found also in the literature referenced below.)

B.2 Introducing Coinduction

To explain the difference between induction and coinduction, and to motivate the need for coinductive methods, let us consider a small example. Consider the program

$$\texttt{if } f(27, m) \texttt{ then } \text{“something good”} \texttt{ else } \text{“something bad”}$$

where f is a function from pairs of natural numbers (i.e. pairs of non-negative integers) to truth values.

We want to ensure that the program never undertakes to do "something bad". Since the value of m is not known it is not feasible to prove that $f(27, m) \neq \mathit{false}$ by merely evaluating $f(27, m)$; we therefore need to perform some kind of proof. For this it is natural to define the predicate Q_f as follows

$$\mathsf{Q}_f(n) \text{ iff } \forall m : f(n, m) \neq \mathit{false}$$

where it is implicit that $m, n \geq 0$.

Perhaps the most obvious approach is to use mathematical induction to prove $\forall n : \mathsf{Q}_f(n)$. This amounts to proving

$$\begin{array}{l} \mathsf{Q}_f(0) \\ \forall n : \mathsf{Q}_f(n) \Rightarrow \mathsf{Q}_f(n+1) \end{array}$$

and then concluding

$$\forall n : \mathsf{Q}_f(n)$$

from which the desired $\mathsf{Q}_f(27)$ follows.

An alternative presentation of essentially the same idea is to establish the validity of the axiom and rule

$$\mathsf{Q}_f(0) \qquad\qquad \frac{\mathsf{Q}_f(n)}{\mathsf{Q}_f(n+1)}$$

and then deduce that

$$\forall n : \mathsf{Q}_f(n)$$

Here the basic steps in the mathematical induction have been couched in terms of an *inductive definition* of the predicate Q_f.

The approach outlined above works nicely for the function f_0 defined by

$$\begin{array}{rcl} f_0(0, m) & = & \mathit{true} \\ f_0(n+1, m) & = & f_0(n, m) \end{array}$$

but what about the functions f_1, f_2 and f_3 defined by

$$\begin{array}{rcl} f_1(0, m) & = & f_1(0, m) \\ f_1(n+1, m) & = & f_1(n, m) \end{array} \qquad \begin{array}{rcl} f_2(0, m) & = & \mathit{true} \\ f_2(n+1, m) & = & f_2(n+1, m) \end{array}$$

$$\begin{array}{rcl} f_3(0, m) & = & f_3(0, m) \\ f_3(n+1, m) & = & f_3(n+1, m) \end{array}$$

where $f_i(27, m)$ never terminates? Intuitively, they should be acceptable in the sense that "something bad" never happens. However, we cannot prove this by induction because we cannot establish the base case (for f_1 and f_3) and/or we cannot establish the inductive step (for f_2 and f_3).

An intuitive argument for why f_i is acceptable might go as follows: assume that all occurrences of f_i on the right hand sides of the above definitions satisfy Q_i; then it follows that also the f_i on the left hand side does. Hence f_i satisfies Q_i, i.e. $\forall n : \mathsf{Q}_i(n)$. This sounds very dangerous: we assume the desired result in order to prove it. However, with due care and given the proper definition of Q_i, this is a valid proof: it is a coinductive proof.

Obtaining a functional. Let us rewrite the defining clauses of f_i into clauses for Q_i so as to clarify the relationship between when Q_i holds on the left hand side and on the right hand side of the definitions of f_i:

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{Q}_0(0) & \text{iff} & \mathit{true} & \mathsf{Q}_1(0) & \text{iff} & \mathsf{Q}_1(0) \\ \mathsf{Q}_0(n+1) & \text{iff} & \mathsf{Q}_0(n) & \mathsf{Q}_1(n+1) & \text{iff} & \mathsf{Q}_1(n) \\ \mathsf{Q}_2(0) & \text{iff} & \mathit{true} & \mathsf{Q}_3(0) & \text{iff} & \mathsf{Q}_3(0) \\ \mathsf{Q}_2(n+1) & \text{iff} & \mathsf{Q}_2(n+1) & \mathsf{Q}_3(n+1) & \text{iff} & \mathsf{Q}_3(n+1) \end{array} \tag{B.1}$$

Here the clauses for Q_0 look just like our principle for mathematical induction whereas the others involve some amount of circularity. To make this evident let us rewrite the above as

$$\mathsf{Q}_i \quad = \quad \mathcal{Q}_i(\mathsf{Q}_i) \tag{B.2}$$

where

$$\begin{array}{rcl@{\qquad}rcl} \mathcal{Q}_0(Q')(0) & = & \mathit{true} & \mathcal{Q}_1(Q')(0) & = & Q'(0) \\ \mathcal{Q}_0(Q')(n+1) & = & Q'(n) & \mathcal{Q}_1(Q')(n+1) & = & Q'(n) \\ \mathcal{Q}_2(Q')(0) & = & \mathit{true} & \mathcal{Q}_3(Q')(0) & = & Q'(0) \\ \mathcal{Q}_2(Q')(n+1) & = & Q'(n+1) & \mathcal{Q}_3(Q')(n+1) & = & Q'(n+1) \end{array} \tag{B.3}$$

Clearly Q_i satisfies (B.1) if and only if it satisfies (B.2) with $\mathcal{Q}_i$ as in (B.3).

It is immediate that each $\mathcal{Q}_i$ is a monotone function on the complete lattice

$$(\mathbf{N} \to \{\mathit{true}, \mathit{false}\}, \sqsubseteq)$$

of predicates where $Q_1 \sqsubseteq Q_2$ means that $\forall n : Q_1(n) \Rightarrow Q_2(n)$ and where the least element $\bot$ is given by $\forall n : \bot(n) = \mathit{false}$ and the greatest element $\top$ is given by $\forall n : \top(n) = \mathit{true}$. Using Tarski's Fixed Point Theorem (Proposition A.10) it follows that each $\mathcal{Q}_i$ has a least fixed point $\mathit{lfp}(\mathcal{Q}_i)$ and a greatest fixed fixed point $\mathit{gfp}(\mathcal{Q}_i)$; these are possibly different predicates in $(\mathbf{N} \to \{\mathit{true}, \mathit{false}\}, \sqsubseteq)$.

We frequently refer to $\mathcal{Q}_i$ as a *functional* by which we mean a function whose argument and result are themselves functions (or that are more elaborate structures containing functions in them).

Least fixed point. Let us begin by looking at the least fixed points. It follows from Appendix A that

$$\bigsqcup_k \mathcal{Q}_0^k(\bot) \sqsubseteq \mathit{lfp}(\mathcal{Q}_0)$$

and given that the clauses for $\mathcal{Q}_0(Q)$ only use a finite number of Q's on the right hand sides (in fact zero or one), $\mathcal{Q}_0$ satisfies a continuity property that ensures that

$$\bigsqcup_k \mathcal{Q}_0^k(\bot) = \mathit{lfp}(\mathcal{Q}_0)$$

This is good news because our previous proof by mathematical induction essentially defines the predicate $\bigsqcup_k \mathcal{Q}_0^k(\bot)$: $\mathcal{Q}_0^k(\bot)(n)$ holds if and only if at most k axioms and rules suffice for proving $\mathrm{Q}_0(n)$. Thus it would seem that a proof by induction "corresponds" to taking the least fixed point of $\mathcal{Q}_0$.

Next let us look at $\mathcal{Q}_3$. Here

$$\mathit{lfp}(\mathcal{Q}_3) = \bot$$

because $\mathcal{Q}_3(\bot) = \bot$ so that $\bot$ is a fixed point. This explains why we have $\mathit{lfp}(\mathcal{Q}_3)(27) = \mathit{false}$ and why an inductive proof will not work for establishing $\mathrm{Q}_3(27)$. Somewhat similar arguments can be given for $\mathcal{Q}_1$ and $\mathcal{Q}_2$.

Greatest fixed point. Let us next look at the greatest fixed points. Here

$$\mathit{gfp}(\mathcal{Q}_3) = \top$$

because $\mathcal{Q}_3(\top) = \top$ so that $\top$ is a fixed point. This explains why we have $\mathit{gfp}(\mathcal{Q}_3)(27) = \mathit{true}$ and thus provides a formal underpinning for our belief that f_3 will not cause any harm in the example program. Somewhat similar arguments can be given for $\mathcal{Q}_1$ and $\mathcal{Q}_2$.

Also for $\mathcal{Q}_0$ it will be the case that $\mathit{gfp}(\mathcal{Q}_0)(27) = \mathit{true}$. This is of course not surprising since $\mathit{lfp}(\mathcal{Q}_0)(27) = \mathit{true}$ and $\mathit{lfp}(\mathcal{Q}_0) \sqsubseteq \mathit{gfp}(\mathcal{Q}_0)$. However, it is more interesting to note that for $\mathcal{Q}_0$ there is no difference between the inductive and the coinductive approach (unlike what is the case for $\mathcal{Q}_1$, $\mathcal{Q}_2$ and $\mathcal{Q}_3$):

$$\mathit{lfp}(\mathcal{Q}_0) = \mathit{gfp}(\mathcal{Q}_0)$$

because mathematical induction on n suffices for proving that $\mathit{lfp}(\mathcal{Q}_0)(n) = \mathit{gfp}(\mathcal{Q}_0)(n)$.

Remark. To the mathematically inclined reader we should point out that the fact that $\mathit{lfp}(\mathcal{Q}_0) = \mathit{gfp}(\mathcal{Q}_0)$ is related to Banach's Fixed Point Theorem: a contractive operator on a complete metric space has a unique fixed point. Contractiveness of $\mathcal{Q}_0$ (as opposed to $\mathcal{Q}_1$, $\mathcal{Q}_2$ and $\mathcal{Q}_3$) follows because the clause for $\mathcal{Q}_0(Q)(n)$ only performs calls to Q on arguments smaller than n. ■

B.3 Proof by Coinduction

Consider once again the algebraic data type

$$d \in \mathbf{D}$$
$$d ::= \mathsf{Base} \mid \mathsf{Con}_1(d) \mid \mathsf{Con}_2(d, d)$$

with one base case, one unary constructor and one binary constructor. Next consider the definition of a predicate

$$\mathsf{Q} : \mathbf{D} \to \{\mathit{true}, \mathit{false}\}$$

by means of clauses of the form:

$$\begin{array}{rcl} \mathsf{Q}(\mathsf{Base}) & \text{iff} & \cdots \\ \mathsf{Q}(\mathsf{Con}_1(d)) & \text{iff} & \cdots \mathsf{Q}(d') \cdots \\ \mathsf{Q}(\mathsf{Con}_2(d_1, d_2)) & \text{iff} & \cdots \mathsf{Q}(d'_1) \cdots \mathsf{Q}(d'_2) \cdots \end{array}$$

We can use this as the basis for defining a functional $\mathcal{Q}$ by cases as in:

$$\begin{array}{rcl} \mathcal{Q}(Q')(\mathsf{Base}) & = & \cdots \\ \mathcal{Q}(Q')(\mathsf{Con}_1(d)) & = & \cdots Q'(d') \cdots \\ \mathcal{Q}(Q')(\mathsf{Con}_2(d_1, d_2)) & = & \cdots Q'(d'_1) \cdots Q'(d'_2) \cdots \end{array}$$

We note that

$$(\mathbf{D} \to \{\mathit{true}, \mathit{false}\}, \sqsubseteq)$$

is a complete lattice under the ordering given by $Q_1 \sqsubseteq Q_2$ if and only if $\forall d : Q_1(d) \Rightarrow Q_2(d)$. We also

$$\text{assume that } \mathcal{Q} \text{ is monotone}$$

and this means that e.g. a clause like "$\mathsf{Q}(\mathsf{Con}_1(d))$ iff $\neg\mathsf{Q}(d)$" will not be acceptable. From Proposition A.10 it follows that $\mathcal{Q}$ has a least as well as a greatest fixed point.

Induction (or least fixed point). Consider first the inductive definition:

$$\mathsf{Q} = \mathit{lfp}(\mathcal{Q}) \tag{B.4}$$

This is more commonly written as:

$$\frac{\cdots}{\mathsf{Q}(\mathsf{Base})} \qquad \frac{\cdots \mathsf{Q}(d') \cdots}{\mathsf{Q}(\mathsf{Con}_1(d))} \qquad \frac{\cdots \mathsf{Q}(d'_1) \cdots \mathsf{Q}(d'_2) \cdots}{\mathsf{Q}(\mathsf{Con}_2(d_1, d_2))} \tag{B.5}$$

It is often the case that each rule only has a *finite* number of calls to Q and then the two definitions are equivalent: the predicate in (B.5) amounts

to $\bigsqcup_k \mathcal{Q}^k(\bot)$ and by a continuity property as discussed above, this agrees with the predicate of (B.4). A *proof by induction* then simply amounts to establishing the validity of the axioms and rules in (B.5). Such a proof has a very constructive flavour: we take nothing for granted and only believe what can be demonstrated to hold. This proof strategy is often used to reason about semantics because the semantics of a program should not allow any spurious behaviour that is not forced by the semantics.

Coinduction (or greatest fixed point). Consider next the *coinductive definition*

$$\mathsf{Q} = gfp(\mathcal{Q})$$

A *proof by coinduction* then amounts to using the proof rule

$$\frac{Q' \sqsubseteq \mathcal{Q}(Q')}{Q' \sqsubseteq \mathsf{Q}} \qquad \left(\text{i.e. } \frac{Q' \sqsubseteq \mathcal{Q}(Q')}{Q' \sqsubseteq gfp(\mathcal{Q})}\right)$$

as follows from the formula for $gfp(\mathcal{Q})$ given in Proposition A.10. So to prove $\mathsf{Q}(d)$ one needs to

$$\begin{array}{l} \text{find some } Q' \text{ such that} \\ \quad Q'(d) \\ \quad \forall d' : Q'(d') \Rightarrow \mathcal{Q}(Q')(d') \end{array}$$

Such a proof has a very optimistic flavour: we can assume everything we like as long as it cannot be demonstrated that we have violated any facts. It is commonly used for checking that a specification holds because the specification should not forbid some behaviour unless explicitly called for.

It sometimes saves a bit of work to use the derived proof rule

$$\frac{Q' \sqsubseteq \mathcal{Q}(\mathsf{Q} \sqcup Q')}{Q' \sqsubseteq \mathsf{Q}}$$

To see that this is a valid proof rule suppose that $Q' \sqsubseteq \mathcal{Q}(\mathsf{Q} \sqcup Q')$. By definition of Q we also have $\mathsf{Q} \sqsubseteq \mathcal{Q}(\mathsf{Q})$ and by monotonicity of $\mathcal{Q}$ this gives $\mathsf{Q} \sqsubseteq \mathcal{Q}(\mathsf{Q} \sqcup Q')$. Hence

$$\mathsf{Q} \sqcup Q' \sqsubseteq \mathcal{Q}(\mathsf{Q} \sqcup Q')$$

and $\mathsf{Q} \sqcup Q' \sqsubseteq \mathsf{Q}$ follows by definition of Q. It is then immediate that $Q' \sqsubseteq \mathsf{Q}$ as required.

Clearly this explanation can be generalised to algebraic data types with additional base cases and constructors, and from predicates to relations.

Example B.1 Consider the relations

$$\begin{array}{l} \mathsf{R}_1 : D_{11} \times D_{12} \to \{\mathit{true}, \mathit{false}\} \\ \mathsf{R}_2 : D_{21} \times D_{22} \to \{\mathit{true}, \mathit{false}\} \end{array}$$

defined by:

$$\begin{array}{l} \mathsf{R}_1 = \mathcal{R}_1(\mathsf{R}_1, \mathsf{R}_2) \\ \mathsf{R}_2 = \mathcal{R}_2(\mathsf{R}_1, \mathsf{R}_2) \end{array}$$

This is intended to define

$$(\mathsf{R}_1, \mathsf{R}_2) = \mathit{gfp}(\mathcal{R})$$

where $\mathcal{R}(R_1', R_2') = (\mathcal{R}_1(R_1', R_2'), \mathcal{R}_2(R_1', R_2'))$ is assumed to be monotone. Next write $R' \sqcup R''$ for the relation defined by

$$d_1 \ (R' \sqcup R'') \ d_2 \ \text{ iff } \ (d_1 \ R' \ d_2) \vee (d_1 \ R'' \ d_2)$$

and write $R' \sqsubseteq R''$ for the truth value defined by:

$$R' \sqsubseteq R'' \ \text{ iff } \ \forall d_1, d_2 : d_1 \ R' \ d_2 \Rightarrow d_1 \ R'' \ d_2$$

We then have the following version of the coinduction principle: we establish

$$\begin{array}{lcl} R_1' & \sqsubseteq & \mathcal{R}_1(R_1', R_2') \\ R_2' & \sqsubseteq & \mathcal{R}_2(R_1', R_2') \end{array}$$

and conclude:

$$R_1' \sqsubseteq \mathsf{R}_1 \text{ and } R_2' \sqsubseteq \mathsf{R}_2$$

By analogy with the previous discussion, it sometimes saves a bit of work only to show

$$\begin{array}{lcl} R_1' & \sqsubseteq & \mathcal{R}_1(\mathsf{R}_1 \sqcup R_1', \mathsf{R}_2 \sqcup R_2') \\ R_2' & \sqsubseteq & \mathcal{R}_2(\mathsf{R}_1 \sqcup R_1', \mathsf{R}_2 \sqcup R_2') \end{array}$$

because we also have

$$\begin{array}{lclclcl} \mathsf{R}_1 & \sqsubseteq & \mathcal{R}_1(\mathsf{R}_1, \mathsf{R}_2) & \sqsubseteq & \mathcal{R}_1(\mathsf{R}_1 \sqcup R_1', \mathsf{R}_2 \sqcup R_2') \\ \mathsf{R}_2 & \sqsubseteq & \mathcal{R}_2(\mathsf{R}_1, \mathsf{R}_2) & \sqsubseteq & \mathcal{R}_2(\mathsf{R}_1 \sqcup R_1', \mathsf{R}_2 \sqcup R_2') \end{array}$$

and this allows us to conclude

$$R_1' \sqsubseteq \mathsf{R}_1 \text{ and } R_2' \sqsubseteq \mathsf{R}_2$$

using the definition of $(\mathsf{R}_1, \mathsf{R}_2) = \mathit{gfp}(\mathcal{R})$. ∎

Example B.2 For an example of a somewhat different flavour consider the universe

$$\{0,1,2\}^\infty = \{0,1,2\}^\star \cup \{0,1,2\}^\omega$$

where $\{0,1,2\}^\star$ consists of finite strings $(a_i)_{i=1}^n = a_1 \cdots a_n$ of symbols $a_i \in \{0,1,2\}$ and where $\{0,1,2\}^\omega$ consists of infinite strings $(a_i)_{i=1}^\infty = a_1 \cdots a_n \cdots$ of symbols $a_i \in \{0,1,2\}$. Concatenation of strings is defined as follows

$$\begin{array}{lcll}
(a_i)_{i=1}^n\ (b_j)_{j=1}^m & = & (c_k)_{k=1}^{n+m} & \text{where} \quad c_k = \begin{cases} a_k & \text{if } k \le n \\ b_{k-n} & \text{if } k > n \end{cases} \\
(a_i)_{i=1}^n\ (b_j)_{j=1}^\infty & = & (c_k)_{k=1}^\infty & \text{where} \quad c_k = \begin{cases} a_k & \text{if } k \le n \\ b_{k-n} & \text{if } k > n \end{cases} \\
(a_i)_{i=1}^\infty\ (b_j)_{j=1}^{\cdots} & = & (c_k)_{k=1}^\infty & \text{where} \quad c_k = a_k
\end{array}$$

(where $\cdots$ denotes "m" or "∞" in the last formula).

Next consider a context free grammar with start symbol S and productions:

$$S \to 0 \mid 1 \mid 0\ S \mid 1\ S$$

It can be rephrased as an inductive definition

$$0 \in S \qquad 1 \in S \qquad \frac{x \in S}{0x \in S} \qquad \frac{x \in S}{1x \in S}$$

for defining a subset $S \subseteq \{0,1,2\}^\infty$. If an *inductive* interpretation is taken, as is usually the case for context free grammars, the set S is $\{0,1\}^+$ that consists of all strings $(a_i)_{i=1}^n$ that are finite and nonempty (so $n > 0$) and without any occurrence of 2's (i.e. $a_i \in \{0,1\}$).

If a *coinductive* interpretation is taken the set S becomes $\{0,1\}^+ \cup \{0,1\}^\omega$ where also all infinite strings of 0's and 1's occur. To see this note that the empty string cannot be in S because it is not of one of the forms 0, 1, $0x$ or $1x$. Also note that no string containing 2 can be in S: a string containing 2 can be written $a_1 \cdots a_n 2x$ where $a_i \in \{0,1\}$ and we proceed by induction; for $n = 0$ the result is immediate (since none of 0, 1, $0x$ or $1x$ starts with a 2) and for $n > 0$ we note that if $a_1 \cdots a_n 2x \in S$ then $a_2 \cdots a_n 2x \in S$ and by the induction hypothesis this cannot be the case. Finally note that there is no way to exclude infinite strings of 0's and 1's to be in S because they can always be written in one of the forms $0x$ or $1x$ (where $x \in \{0,1\}^\omega$).

Now suppose we add the production

$$S \to S$$

corresponding to the rule:

$$\frac{x \in S}{x \in S}$$

When interpreted *inductively* this does not change the set S; it is still $\{0,1\}^+$. However, when interpreted *coinductively* the set S changes dramatically; it is now $\{0,1,2\}^\infty$ and thus includes the empty string as well as strings containing 2. To see this simply note that any string x in $\{0,1,2\}^\infty$ can be written as $x = x$ and hence no string can be excluded from being in S. ∎

Concluding Remarks

For more information on induction principles consult a text book (e.g. [16]). It is harder to find good introductory material on coinduction; one possibility is to study the work on strong bisimulation in CCS (e.g. Chapter 4 of [106]).

Appendix C

Graphs and Regular Expressions

C.1 Graphs and Forests

Directed graphs. A *directed graph* (or *digraph*) $G = (N, A)$ consists of a finite set N of *nodes* (or *vertices*) and a set $A \subseteq N \times N$ of *edges* (or *arcs*). An edge (n_1, n_2) has source n_1 and target n_2 and is said to go from n_1 to n_2; if $n_1 = n_2$ it is a *self-loop*. We shall sometimes refer to a directed graph simply as a *graph*.

A *directed path* (or just *path*) from a node n_0 to a node n_m is a sequence of edges $(n_0, n_1), (n_1, n_2), \cdots, (n_{m-2}, n_{m-1}), (n_{m-1}, n_m)$ where the target of the edge (n_{i-1}, n_i) equals the source of the edge (n_i, n_{i+1}). The path is said to have source n_0, target n_m and length $m \geq 0$; if $m = 0$ the path is said to be trivial and then consists of the empty sequence of edges. The concatenation of the path $(n_0, n_1), \cdots, (n_{m-1}, n_m)$ with the path $(n_m, n_{m+1}), \cdots, (n_{k-1}, n_k)$ is the path $(n_0, n_1), \cdots, (n_{k-1}, n_k)$. We say that n' is reachable from n whenever there is a (possibly trivial) directed path from n to n'.

Example C.1 Let S be a statement from the WHILE language of Chapter 2. The forward flow graph $(\mathit{labels}(S), \mathit{flow}(S))$ and the backward flow graph $(\mathit{labels}(S), \mathit{flow}^R(S))$ are both directed graphs. Next let **Lab** be a finite set of labels and $F \subseteq \mathbf{Lab} \times \mathbf{Lab}$ a flow relation in the manner of Section 2.3. Then $G = (\mathbf{Lab}, F)$ is a directed graph. ∎

Example C.2 Let $e_\star$ be a program in the FUN language of Chapter 3 and let $N_\star$ be the finite set of nodes p of the form $\mathsf{C}(\ell)$ or $\mathsf{r}(x)$ where $\ell \in \mathbf{Lab}_\star$ is a label in the program and $x \in \mathbf{Var}_\star$ is a variable in the program. Let C be

a set of constraints of the form considered in Section 3.4 and let $A_\star$ contain the edge (p_1, p_2) for each constraint $p_1 \subseteq p_2$ in C and the edges (p, p_2) and (p_1, p_2) for each constraint $\{t\} \subseteq p \Rightarrow p_1 \subseteq p_2$ in C. Then $G = (N_\star, A_\star)$ is a directed graph. ∎

In keeping with the notation used in Chapter 2 we shall usually write a path $p = (n_0, n_1), (n_1, n_2), \cdots, (n_{m-2}, n_{m-1}), (n_{m-1}, n_m)$ as the sequence $p = [n_0, n_1, n_2, \cdots, n_{m-2}, n_{m-1}, n_m]$ of nodes visited. In this notation the sequence of nodes is never empty and the trivial path from n_0 to n_0 is written $[n_0]$ whereas a self-loop is written $[n_0, n_0]$. The concatenation of the path $[n_0, n_1, \cdots, n_{m-1}, n_m]$ with the path $[n_m, n_{m+1}, \cdots, n_{k-1}, n_k]$ is the path $[n_0, n_1, \cdots, n_{k-1}, n_k]$ (where n_m does not occur twice).

We shall define the set of paths from n to n' as follows:

$$paths_\bullet(n, n') = \{[n_0, \cdots, n_m] \mid m \geq 0 \wedge n_0 = n \wedge n_m = n' \wedge \forall i < m : (n_i, n_{i+1}) \in A\}$$

Cycles. A *cycle* is a non-trivial path from a node to itself (and may take the form of a self-loop); the set of cycles from a node n to itself is defined as follows:

$$cycles(n) = \{[n_0, \cdots, n_m] \mid m \geq 1 \wedge n_0 = n \wedge n_m = n \wedge \forall i < m : (n_i, n_{i+1}) \in A\}$$

and we observe that $cycles(n) = \{p \in paths_\bullet(n, n) \mid p \neq [n]\}$. A cycle $p = [n_0, n_1, \cdots, n_m]$ is *multiple entry* if it contains two distinct nodes (not merely distinct indices), $n_i \neq n_j$, such that there are (not necessarily distinct) nodes n'_i and n'_j external to the cycle with $(n'_i, n_i), (n'_j, n_j) \in A$.

A graph that contains no cycles (so $\forall n \in N : cycles(n) = \emptyset$) is said to be *acyclic*. A directed, acyclic graph is often called a *DAG*.

A *topological sort* of a directed graph is a total ordering of the nodes such that if (n, n') is an edge then n is ordered strictly before n'. A directed graph has a topological sort if and only if it is acyclic.

Strongly connected components. Two nodes n and n' are said to be strongly connected whenever there is a (possibly trivial) directed path from n to n' and a (possibly trivial) directed path from n' to n. Defining

$$\mathcal{SC} = \{(n, n') \mid n \text{ and } n' \text{ are strongly connected}\}$$

we obtain a binary relation $\mathcal{SC} \subseteq N \times N$.

Fact C.3 $\mathcal{SC}$ is an equivalence relation. ∎

Proof Reflexivity follows from the fact that there is a trivial path from any node to itself. Symmetry follows immediately from the definition. Transitivity follows

from observing that if p_{12} is a path from n_1 to n_2 and p_{23} is a path from n_2 to n_3 then $p_{12}p_{23}$ is a path from n_1 to n_3. ■

The equivalence classes of $\mathcal{SC}$ are called the *strong components* (or *strongly connected components*) of the graph $G = (N, A)$. A graph is said to be *strongly connected* whenever it contains exactly one strongly connected component.

Example C.4 In uniquely labelled statements of the WHILE language of Chapter 2, the nodes corresponding to the outermost loop in any nested sequence of loops would constitute a strongly connected component; this is true for both forward and backward flow graphs. ■

The interconnections between strong components can be represented by the *reduced graph.* Each strongly connected component is represented by a node in the reduced graph and there is an edge from one node to another distinct node if and only if there is an edge from some node in the first strongly connected component to a node in the second in the original graph. Hence, the reduced graph contains no self-loops.

Lemma C.5 For any graph G the reduced graph is a DAG. ■

Proof Suppose, by way of contradiction, that the reduced graph contains a cycle $[SC_0, \cdots, SC_m]$ where $SC_m = SC_0$. Since we already observed that the reduced graph contains no self-loops this means that the cycle contains distinct nodes SC_i and SC_j and we may without loss of generality assume that $i < j$.

That there is an edge from SC_k to SC_{k+1} in the reduced graph ($0 \leq k < m$) means that there is an edge (n_k, n'_{k+1}) in the original graph from a node $n_k \in SC_k$ to a node $n'_{k+1} \in SC_{k+1}$. It is also immediate that there is a path p_k in the original graph from n'_k to n_k and a path p'_0 from n'_m to n_0.

We can now construct a path $(n_i, n'_{i+1})p_{i+1} \cdots (n_{j-1}, n'_j)p_j$ in the original graph from n_i to n_j and a path $(n_j, n'_{j+1})p_{j+1} \cdots (n_{m-1}, n'_m)p'_0(n_0, n'_1)p_1 \cdots (n_{i-1}, n'_i)p_i$ from n_j to n_i. But then n_i and n_j should be in the same equivalence class thereby contradicting our assumption that $SC_i \neq SC_j$. ■

Handles and roots. A *handle* for a directed graph $G = (N, A)$ is a set $H \subseteq N$ of nodes such that for all nodes $n \in N$ there exists a node $h \in H$ such that there is a (possibly trivial) directed path from h to n. A *root* is a node $r \in N$ such that for all nodes $n \in N$ there is a (possibly trivial) directed path from r to n. If r is a root for $G = (N, A)$ then $\{r\}$ is handle for G; in fact, $\{n\}$ is a handle for G if and only if n is a root. Conversely, if H is a handle for $G = (N, A)$ and $r \notin N$ then r is a root for the graph $(N \cup \{r\}, A \cup \{(r, h) \mid h \in H\})$; we sometimes refer to r as a dummy root and to each (r, h) as a dummy edge.

A directed graph $G = (N, A)$ always has a handle since one can use $H = N$ as the handle. A handle H is minimal if no proper subset is a handle, that is, if $H' \subset H$ then H' is not a handle. One can show that minimal handles always exist since the set of nodes is finite. The minimal handle is a singleton if and only if the graph has a root; in this case we say that the graph is rooted.

Example C.6 Let S be a statement from the WHILE language of Chapter 2. Then $init(S)$ is a root for $(labels(S), flow(S))$ and $\{init(S)\}$ is a minimal handle. Furthermore, $final(S)$ is a handle for the backward flow graph $(labels(S), flow^R(S))$. ■

A path from a handle H to a node n is a path from a node $h \in H$ to n; similarly for a path from the root r to a node n. Given a handle H, possibly in the form of a root r (corresponding to $H = \{r\}$), the set of paths from H to n is defined by:

$$path_\bullet^H(n) = \bigcup\{paths_\bullet(h, n) \mid h \in H\}$$

When H is implicit we write $path_\bullet(n)$ for $path_\bullet^H(n)$.

Forests and trees. A node n in a graph $G = (N, A)$ is said to have *in-degree* m if the set of predecessors, $\{n' \mid (n', n) \in A\}$, has cardinality m. Similarly, one can define the concept of *out-degree*.

A *forest* (or unordered, directed forest) is a directed, acyclic graph $G = (N, A)$ where nodes have in-degree at most 1. One can show that the set of nodes with in-degree 0 constitutes a minimal handle for the forest. If $(n, n') \in A$ then n is said to be the *parent* of n' and n' is a said to be a *child* of n; *ancestor* and *descendant* are the reflexive, transitive closures of the parent and child relations, respectively. The node n is a *proper* ancestor of n' if it is an ancestor and $n \neq n'$ and similarly for proper descendants.

A *tree* (or unordered, directed tree) is a forest that has a root; in other words, a tree is a forest where exactly one node has in-degree 0 and this is then the root. Given a forest (N, A) with minimal handle H, each node $h \in H$ will be a root for the tree consisting of all the nodes $n \in N$ reachable from h and all the edges $(n_1, n_2) \in E$ whose source and target are both in the tree; thus a forest can be viewed as a set of trees.

Sometimes we will be interested in *ordered forests* and *ordered trees* which are forests and trees where additionally all children of a node are linearly ordered (unlike what is normally the case for graphs, forests and trees).

Dominators. Given a directed graph $G = (N, A)$ with handle H, possibly in the form of a root r (corresponding to $H = \{r\}$), a node n' is said to be a *dominator* of a node n whenever every (possibly trivial) path from H to n contains n'. We also say that n' *dominates* n. As a consequence of the

INPUT:	A directed graph (N, A) with k nodes and handle H
OUTPUT:	(1) A DFSF $\mathsf{T} = (N, A_T)$, and (2) a numbering rPostorder of the nodes indicating the reverse order in which each node was last visited and represented as an element of array $[N]$ of int

METHOD:	i := k;
	mark all nodes of N as unvisited;
	let A_T be empty;
	while unvisited nodes in H exists do
	  choose a node h in H;
	  DFS(h);

USING:	procedure DFS(n) is
	  mark n as visited;
	  while $(n, n') \in A$ and n' has not been visited do
	  add the edge (n, n') to A_T;
	  DFS(n');
	  rPostorder[n] := i;
	  i := i − 1;

Table C.1: The DFSF Algorithm.

definition of paths, the only dominator of an element of a minimal handle (such as a root) is the element itself.

For any node n the set of dominators can be specified as the greatest solution to the following equation:

$$Dom(n) = \begin{cases} \{n\} & \text{if } n \in H \\ \{n\} \cup \bigcap\{Dom(n') \mid (n', n) \in A\} & \text{otherwise} \end{cases}$$

The node n' *properly dominates* the node n if the two nodes are distinct and n' dominates n. The node n' *directly dominates* the node n if it is the "closest" proper dominator of n; that is $n' \in Dom(n) \backslash \{n\}$ and for all $n'' \in Dom(n) \backslash \{n\}$ we have $n'' \in Dom(n')$.

C.2 Reverse Postorder

Spanning forests. A *spanning forest* for a graph is a forest, with the same nodes as the graph and a subset of the edges of the graph (see below) as

Figure C.1: A flow graph.

edges. The algorithm of Table C.1 non-deterministically constructs a *depth-first spanning forest* (abbreviated *DFSF*). In parallel with constructing the forest, the algorithm also generates a numbering of the nodes which is the reverse of the order in which the nodes are last visited in the construction of the tree; this ordering is called the *reverse postorder* and in the algorithm it is represented as an array indexed by the nodes of the graph. If the DFSF is a tree it is called a *depth-first spanning tree* (abbreviated *DFST*). Note that the algorithm does not specify which unvisited node to select at each stage; consequently, the depth-first spanning forest for a graph is not unique.

Given a spanning forest one can categorise the edges in the original graph as follows:

- *Tree edges*: edges present in the spanning forest.
- *Forward edges*: edges that are not tree edges and that go from a node to a proper descendant in the tree.
- *Back edges*: edges that go from descendants to ancestors (including self-loops).
- *Cross edges*: edges that go between nodes that are unrelated by the ancestor and descendant relations.

The algorithm of Table C.1 ensures that cross edges always go from nodes visited later (i.e. with lower numbers in the reverse postorder) to nodes visited earlier (i.e. with higher numbers).

Example C.7 To illustrate the operation of the algorithm, consider the graph in Figure C.1 with root b_1. The algorithm may produce the tree shown

Figure C.2: A DFSF for the graph in Figure C.1.

in Figure C.2. The reverse postorder of the nodes is $b_1, b_3, S_3, b_2, S_1, S_2$ and the second number annotating each node in Figure C.2 reflects this ordering. The edges from S_2 to b_2 and from S_3 to b_3 in Figure C.1 are back edges. The edge from b_3 to S_1 is a cross edge. ■

Example C.8 Let S be a uniquely labelled statement from WHILE and consider the forward flow graph $(\mathit{labels}(S), \mathit{flow}(S))$ with root $\mathit{init}(S)$. First consider a while loop **while** b' **do** S' in S. It gives rise to one tree edge and one or more back edges; more precisely as many back edges as there are elements in $\mathit{final}(S')$. Next consider a conditional **if** b' **then** S'_1 **else** S'_2 in S that is not the last statement of S and not the last statement of the body of a while loop in S. It gives rise to three tree edges and one or more cross edges; more precisely there will be one fewer cross edges than there are elements in $\mathit{final}(S'_1) \cup \mathit{final}(S'_2)$. Note that no forward edges can arise for statements in WHILE; however, an extension of WHILE with a one-branch conditional **if** b **then** S can give rise to forward edges. ■

Properties. We now establish some properties of reverse postorder. The first lemma shows that the node at the source of a back edge comes later than the target (unless it is a self-loop).

Lemma C.9 Let $G = (N, A)$ be a directed graph, T a depth-first spanning forest of G and rPostorder the associated ordering computed by the algorithm of Table C.1. An edge $(n, n') \in A$ is a back edge if and only if rPostorder$[n] \geq$ rPostorder$[n']$ and is a self-loop if and only if rPostorder$[n]$ = rPostorder$[n']$. ■

Proof The statement about self-loops is immediate. Next let $(n, n') \in A$.

($\Rightarrow$): If (n, n') is a back edge then n is a descendant of n' in T. Consequently, n' is visited before n in the depth first search and the call DFS(n') is pending during the entire call of DFS(n) and consequently rPostorder$[n] \geq$ rPostorder$[n']$.

($\Leftarrow$): If rPostorder[n] $\geq$ rPostorder[n'] then, either n is a descendant of n' or n' is in a subtree that was constructed later than the subtree that n appears in. In the second case, since $(n, n') \in A$, it would be a cross edge going from a node with a high reverse postorder number to a node with a low reverse postorder number – since our algorithm does not admit such cross edges, n must be a descendant of n' and thus (n, n') is a back edge. ∎

Next, we establish that every cycle in a directed graph must contain at least one back edge.

Corollary C.10 Let $G = (N, A)$ be a directed graph, T a depth-first spanning forest of G and rPostorder the associated ordering computed by the algorithm of Table C.1. Any cycle of G contains at least one back edge. ∎

Proof A cycle is a path $[n_0, \cdots, n_m]$ with $n_0 = n_m$ and $m \geq 1$. Then, since rPostorder[n_0] = rPostorder[n_m], we must have rPostorder[n_i] $\geq$ rPostorder[n_{i+1}] for some $0 \leq i < m$. Thus Lemma C.9 applies. ∎

The ordering rPostorder topologically sorts the depth-first spanning forest.

Corollary C.11 Let $G = (N, A)$ be a directed graph, T a depth-first spanning forest of G and rPostorder the associated ordering computed by the algorithm of Table C.1. Then rPostorder topologically sorts T as well as the forward and cross edges. ∎

Proof By Lemma C.9, for any edge (n, n'), rPostorder[n] < rPostorder[n'] if and only if the edge is not a back edge, that is, if and only if the edge is a tree edge in T or a forward edge or a cross edge. Thus rPostorder topologically sorts T as well as the forward and cross edges. ∎

Loop connectedness. Let $G = (N, A)$ be a directed graph with handle H. The *loop connectedness* parameter of G with respect to a depth-first spanning forest T constructed by the algorithm of Table C.1, is the largest number of back edges found in any cycle-free path of G; we write $d(G, T)$, or $d(G)$ when T is clear from the context, to denote the loop connectedness parameter of a graph. The value of $d(G)$ for the graph in Figure C.1 is 1.

Let a dominator-back edge be an edge (n_1, n_2) where the target n_2 dominates the source n_1. A dominator-back edge (n_1, n_2) is a back edge regardless of the choice of spanning forest T because the path from H to n_1 in T must contain n_2 (given that n_2 dominates n_1). The graph shown in Figure C.3 shows that there may be more back edges than there are dominator-back edges; indeed, it has root r and no dominator-back edges, but any spanning forest T will characterise one of the edges between n_1 and n_2 as a back edge.

The literature contains many definitions of the concept of reducible graph. We shall say that a directed graph $G = (N, A)$ with handle H is *reducible*

Figure C.3: An irreducible graph.

if and only if the graph $(N, A \setminus A_{db})$ obtained by removing the set A_{db} of dominator-back edges is acyclic and still has H as handle. The simplest example of an irreducible graph is shown in Figure C.3.

Reducible graphs are of interest because for a reducible graph G with handle H and an arbitrary depth-first spanning forest T, it will be the case (as shown below) that an edge is characterised as a back edge with respect to T if and only if it is a dominator-back edge. It follows that the loop connectedness parameter is then independent of the choice of depth-first spanning forest.

Lemma C.12 Let $G = (N, A)$ be a reducible graph with handle H, T a depth first spanning forest for G and H, and rPostorder the associated ordering computed by the algorithm of Table C.1. Then an edge is a back edge if and only if it is a dominator-back edge. ∎

Proof We already argued that a dominator-back edge necessarily is a back edge with respect to T. Next suppose by way of contradiction that there is a back edge (n_s, n_t) that is not a dominator-back edge. It is clear that the tree edges (i.e. the edges in T) are not back edges and hence not dominator-back edges and therefore they all occur in the graph $(N, A \setminus A_{db})$ constructed above. Next let T' be the graph obtained by adding the edge (n_s, n_t) to T. Then also all edges in T' occur in the graph $(N, A \setminus A_{db})$. The path in T' from n_t to n_s, followed by the back edge (n_s, n_t), constitutes a cycle in T' and hence in $(N, A \setminus A_{db})$. This contradicts the acyclicity of $(N, A \setminus A_{db})$ and hence the reducibility of G. ∎

Example C.13 Once more let S be a uniquely labelled statement from WHILE and consider the flow graph $(\mathit{labels}(S), \mathit{flow}(S))$ with root $\mathit{init}(S)$. It is reducible: Each while loop can be entered only at the test node which then dominates all the nodes for the body of the loop and hence all the back edges introduced by the while loop are also dominator-back edges. Clearly the graph consisting of tree edges, forward edges and cross edges will be acyclic.

The loop connectedness parameter equals the maximum number of back edges (and hence dominator-back edges) of a cycle-free path in the flow graph; it

follows that it is equal to the maximum number of nested while loops in the program. ∎

Another interesting property of rPostorder for reducible graphs is the following result.

Corollary C.14 Let $G = (N, A)$ be a reducible graph with handle H. Any cycle-free path in G beginning with a node in the handle, is monotonically increasing by the ordering rPostorder computed by the algorithm of Table C.1. ∎

Proof Any such path must contain no dominator-back edges and thus by Lemma C.12 is either a path in the depth-first spanning forest or a sequence of paths from the forest connected by forward or cross edges. The result then follows from Corollary C.11. ∎

Other orders. As we have seen, reverse postorder numbers the nodes in the reverse of the order in which they were last visited in the construction of the depth-first spanning forest. Two commonly used alternative orderings on nodes are *preorder* and *breadth-first order*; as an example, the preorder for the tree in Figure C.2 is $b_1, b_2, S_2, S_1, b_3, S_3$ and the breadth-first order is $b_1, b_2, b_3, S_2, S_1, S_3$. Like reverse postorder, both orders topologically sort the depth-first spanning forest. They also topologically sort forward edges but they do not necessarily topologically sort cross edges. (For preorder, consider the edge from b_3 to S_1 in the example; for breadth-first order, modify the example to have an additional node between b_1 and b_3.) This observation makes reverse postorder the better choice for a number of iterative algorithms.

C.3 Regular Expressions

An *alphabet* is a finite and non-empty set Σ of symbols; we shall assume that it is disjoint from the set $\{\Lambda, \emptyset, (,), +, \cdot, *\}$ of special symbols. A *regular expression* over Σ is any expression constructed by the following inductive definition:

1. Λ and $\emptyset$ are (so-called atomic) regular expressions. For any $a \in \Sigma$, a is a (so-called atomic) regular expression.

2. If R_1 and R_2 are regular expressions then $(R_1 + R_2)$, $(R_1 \cdot R_2)$ and $(R_1)^*$ are (so-called compound) regular expressions.

The above definition constructs expressions that are fully parenthesised; by placing a precedence on the operators so that '$*$' binds more tightly than '$\cdot$', and '$\cdot$' binds more tightly than '$+$', most parentheses can be omitted.

A string $w \in \Sigma^*$ is a sequence of symbols from Σ. A language L over Σ is a set of strings, i.e. $L \subseteq \Sigma^*$. The language defined by a regular expression R is $\mathcal{L}[R]$ defined by:

$$\begin{aligned}
\mathcal{L}[\Lambda] &= \{\Lambda\} \\
\mathcal{L}[\emptyset] &= \emptyset \\
\mathcal{L}[a] &= \{a\} \text{ for all } a \in \Sigma \\
\mathcal{L}[R_1 + R_2] &= \mathcal{L}[R_1] \cup \mathcal{L}[R_2] \\
\mathcal{L}[R_1 \cdot R_2] &= \mathcal{L}[R_1] \cdot \mathcal{L}[R_2] \\
\mathcal{L}[R_1^*] &= \bigcup_{k=0}^{\infty} (\mathcal{L}[R_1])^k
\end{aligned}$$

where $L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$, $L^0 = \{\Lambda\}$ and $L^{i+1} = L^i \cdot L$. Two regular expressions R_1 and R_2 are *equivalent*, denoted $R_1 = R_2$, if their languages are equal, that is $\mathcal{L}[R_1] = \mathcal{L}[R_2]$.

A *homomorphism* from Σ_1^* to Σ_2^* is a function $h : \Sigma_1 \to \Sigma_2^*$ that is extended to operate on words $h(a_1 \cdots a_m) = h(a_1) \cdots h(a_m)$ and on languages $h(L) = \{h(w) \mid w \in L\}$. It may be extended to operate on regular expressions:

$$\begin{aligned}
h(\emptyset) &= \emptyset \\
h(\Lambda) &= \Lambda \\
h(a) &= \begin{cases} b_1 \cdot \ldots \cdot b_m & \text{if } h(a) = b_1 \cdots b_m \wedge m > 0 \\ \Lambda & \text{if } h(a) = \Lambda \end{cases} \\
h(R_1 + R_2) &= h(R_1) + h(R_2) \\
h(R_1 \cdot R_2) &= h(R_1) \cdot h(R_2) \\
h(R^*) &= h(R)^*
\end{aligned}$$

If R is a regular expression over Σ_1 then it is immediate that $h(R)$ is a regular expression over Σ_2 and that $\mathcal{L}(h(R)) = h(\mathcal{L}(R))$.

Concluding Remarks

There are a number of good books on graph theoretic concepts; our presentation is based on [48], Chapter 5 of [4], Chapters 3 and 4 of [69], Chapter 10 of [5] and Chapter 7 of [110]. For more information on regular languages consult a text book such as [76].

Index of Notation

$\square \twoheadrightarrow \square$, 214, 219, **220**
$\square \to \square$, 341
$\square \to_{\text{fin}} \square$, VIII
$\square : \square \to \square$, 17, 18, **20**
$\square \leq \square$, 324, 329, **347**
$\square \models \square$, 309
$\square \nabla \square$, 226
$\square \mid \square$, 154, 286
$[\![\square]\!]\square$, 367
$\square \sim \square$, 62
$\square \sqsubseteq \square$, 298, 348
$\square \sqsupseteq \square$, 367
$\square \subseteq \square$, 298
$\square \supseteq \square$, 308
$\square \vdash \square : \square$, 22
$\square \vdash \square : \square \,\&\, \square$, 24
$\square \vdash \square \rightsquigarrow \square$, 211, **219**
$\square \vdash \square \rhd \square$, 27, 212, **219**
$\square \vdash \square \to \square$, 154
$\square \vdash_\star \square \to \square$, 86
$\lfloor \square \rfloor$, 287
$\diamond$, 105

α, 15, 300, 306
α_η, 237
β, 216, 306
$\beta \supseteq \varphi$, 308
$\beta_1 \twoheadrightarrow \beta_2$, 220
β_R, 217
β_η, 237
Δ, 91, 96, 97, 99, **100**, **191**, **230**
δ, 91, 191
$[\delta, \ell]$, 96
$\lceil \delta, \ell \rceil_k$, 98, 193
$\widehat{\delta}$, 187
η, 237
$\eta[V]$, 246
Γ, 22, 285
$\Gamma \vdash e : \tau$, 22
$\Gamma \mid X$, 286
$\widehat{\Gamma} \vdash e : \widehat{\tau} \,\&\, \varphi$, 24
$\widehat{\Gamma}$, 24, 287, 307, 321
γ, 15
γ_η, 237
ι, 69
ι_E^ℓ, 66, 69
$\iota_\cdot$, 186
$\widehat{\iota}$, 96, 97, 99, 100
Λ, 344
λ, VIII
ϕ_ℓ^{SA}, 116
φ, 24, 287, 306, 321, 327, 335, 343
$\varphi \sqsubseteq \varphi'$, 348
$\varphi_1 \sqsubseteq \varphi_2$, 298
$\varphi_1 \subseteq \varphi_2$, 298
π, 284
ψ, 367
ρ, 86, 153, 333
$\rho \vdash_\star \langle S, \varsigma \rangle \to \langle S', \varsigma' \rangle$, 86
$\rho \vdash_\star \langle S, \varsigma \rangle \to \varsigma'$, 86
$\rho \mid X$, 154
$\widehat{\rho}$, 11, 144, 187, 191
$\rho_\star$, 86
ϱ, 331
$\vec{\varrho}$, 332
Σ, 17
σ, 54, 105
$\sigma_1 \sim_V \sigma_2$, 62
$\widehat{\sigma}$, 327, 335, 343
ς, 86, 244, 320, 333
$\varsigma[M]$, 244

ς_η, 246
τ, 285, 300
$\widehat{\tau} \leq \widehat{\tau}'$, 324, 329, 347
$\widehat{\tau}_1 \sqsubseteq \widehat{\tau}_2$, 298
$\widehat{\tau}$, 24, 287, 306, 321, 327, 335, 343
θ, 301
$\theta_A \models C$, 309
ξ, 86, 105, 320
ζ, 327

$\mathcal{A}$, 55, 108
$\mathcal{A}_{\mathsf{CP}}$, 73
a, 3
AE_{entry}, 40
AE_{exit}, 40
AExp$_\star$, 39
AExp, 3
AExp$(\cdot)$, 39
AHeap, 111, 114
ALoc, 110
ALoc(H), 111
ALoc(S), 111
Analysis$^{=}$, 69
Analysis$^{\sqsubseteq}$, 69
Analysis, 369
Analysis, 65
Ann$_{\mathsf{CA}}$, 343
Ann, 287
Ann$_{\mathsf{ES}}$, 327
Ann$_{\mathsf{RI}}$, 335
Ann$_{\mathsf{SE}}$, 321
AState, 110, 114
AType, 300
AV, 101, 317
AVar, 306
$\mathcal{B}$, 55, 108
b, 3
BExp, 3
BExp$(\cdot)$, 39
`bind`-construct, 87, 154
$[B]^\ell$, 37
Blocks$_\star$, 38
blocks$_\star$, 84
blocks, 36, 83
C, 308
$\widehat{\mathsf{C}}$, 11, 144, 187, 193
$\mathcal{C}_\star$, 174
c, 142, 284
$\widehat{\mathbf{Cache}}$, 144, 187, 193
ce, 191
CEnv, 191
ch, 339
Chan, 339
clear, 52
`close`-construct, 153
Const, 142, 284
CP, 102
CP, 342
CS, 14
cycles, 418
$D_\star$, 82
$\widehat{\mathsf{D}}$, 187
$\widehat{d}$, 187
d_c, 182
d_{op}, 183
Data, 182
$\widehat{\mathbf{Data}}$, 187
$\widehat{\mathbf{DCache}}$, 187
def, 52
$\widehat{\mathbf{DEnv}}$, 187
DOM, 13
dom, 86, 154, 286, 301
DU, 54
du, 53
E, 69, 341
$e_1 \rightarrow e_2$, 341
e, 142, 283, 339
ee, 331
EExp, 331
empty, 368
Env, 86, 153, 333
$\widehat{\mathbf{Env}}$, 144, 187, 191
EVal, 333
Exp, 142, 283, 339
Ext, 223, 402
extract, 368
$\mathcal{F}$, 68
$\widehat{\mathcal{F}}$, 91
$\mathcal{F}_{\mathsf{CP}}$, 73

$[f]^n_\Delta$, 230
$\widehat{f^1_{\ell_c}}$, 93, 96, 98, 100, 101
$\widehat{f^2_{\ell_c,\ell_r}}$, 93, 96, 98, 100, 101
$\widehat{f_\ell}$, 91
$\widehat{f_{\ell_n}}$, 92
$\widehat{f_{\ell_x}}$, 92
f^n_∇, 226
f^{SA}_ℓ, 115, 116
$f_\cdot$, 186
f_ℓ, 68
f_p, 212, 219
f^{CP}_ℓ, 73
$final_\star$, 84
final, 36, 83
Fix, 223, 402
$flow_\star$, 84
flow, 37, 83
$flow^R$, 38
FV, 143, 367
gen_{AE}, 40
gen_{LV}, 50
gen_{RD}, 43
gen_{VB}, 46
gfp, 223, 402
H, 111, 114
$\varkappa$, 106
Heap, 106
IAV, 101
id, 68, 302
ie, 154
IExp, 154
inf, 222
infl, 369
$init_\star$, 84
init, 36, 83
insert, 368
inter-flow$_\star$, 84
Interval, 221
is, 112, 114
IsShared, 114
it, 154
ITerm, 154
$\mathsf{JUDG}_{\mathsf{CFA}}[\Gamma \vdash_{\mathsf{UL}} e : \tau]$, 299
$kill_{\mathsf{AE}}$, 40
$kill_{\mathsf{LV}}$, 49
$kill_{\mathsf{RD}}$, 43
$kill_{\mathsf{VB}}$, 46
$kill_x$, 117
$kill_{x.sel}$, 125
$\widehat{L}$, 91
ℓ, 3, 142
l^ϕ_n, 225
$\mathbf{Lab}_\star$, 38, 171
Lab, 3, 142
$labels_\star$, 84
labels, 37, 83
lfp^Δ_∇, 230
lfp_∇, 227
lfp, 223, 402
Loc, 86, 105, 320
$\mathsf{LV}^=$, 60
$\mathsf{LV}^\subseteq$, 60
LV_{entry}, 50
LV_{exit}, 50
MFP, 74
MOP, 79
MVP, 90
$\mathcal{N}$, 55
n_X, 110
$n_\emptyset$, 110
n, 3
Num, 3
o, 333
Observe, 338
Offset, 333
Op, 142, 284
$\widehat{\mathsf{op}}$, 182
op, 142, 284
$\mathbf{op}_a$, 55, 73
op_a, 4
$\mathbf{op}_b$, 55
op_b, 4
$\mathbf{op}_r$, 55
op_r, 4
$P_\star$, 82
$\wp$, 107
$p \vdash l_1 \rhd l_2$, 212, 219

$p \vdash v_1 \rightsquigarrow v_2$, 211, 219
Parity, 244
path, 79
Pnt, 284
PP, 342
Proc, 342
$R \twoheadrightarrow R$, 214
$R_1 \twoheadrightarrow R_2$, 219
R_β, 217
R, 214
$\mathcal{R}$, 158
$\vec{r}$, 332
r, 331, 343
ran, 86
Range, 248
$\mathsf{RD} \vdash S \rhd S'$, 27
RD, 5
RD_{entry}, 43
RD_{exit}, 43
Red, 223, 402
$\mathbf{Reg}_{\mathsf{CA}}$, 343
$\mathbf{Reg}_{\mathsf{RI}}$, 331
rn, 331
RName, 331
rPostorder, 375
RVar, 331
$S : \mathsf{RD}_1 \rightarrow \mathsf{RD}_2$, 18
$S : \Sigma \xrightarrow[\mathsf{RD}]{X} \Sigma$, 20
$S : \Sigma \rightarrow \Sigma$, 17
$S_\star$, 38
S, 3
S, 110, 114
$\mathcal{S}$, 367
SAnn, 306
$\mathbf{Scheme}_{\mathsf{CA}}$, 343
$\mathbf{Scheme}_{\mathsf{RI}}$, 335
Sel, 104
sel, 104
SG, 114
Shape, 115
Sign, 238
SRD, 13
State, 54, 105
$\widehat{\mathbf{State}}_{\mathsf{CP}}$, 72
Stmt, 3
Store, 86, 320, 333
SType, 306
sup, 222
SVal, 333
$[\![t]\!]\psi$, 367
t, 142, 367
TEnv, 285
$\widehat{\mathbf{TEnv}}$, 287
$\mathbf{Term}_\star$, 171
Term, 142
tr, 13
Trace, 13
$\mathbf{TrVar}^?_\star$, 134
TVar, 300, 306
Type, 285
$\mathbf{Type}_{\mathsf{CA}}$, 343
$\mathbf{Type}_{\mathsf{ES}}$, 327
$\mathbf{Type}_{\mathsf{RI}}$, 335
$\mathbf{Type}_{\mathsf{SE}}$, 321
$\widehat{\mathbf{Type}}[\tau]$, 298
$\widehat{\mathbf{Type}}$, 287
$\mathcal{U}_{\mathsf{CFA}}$, 307
$\mathcal{U}_{\mathsf{UL}}$, 302, 305
UD, 54
ud, 53
use, 52
$\mathcal{V}$, 159
$\widehat{v}$, 144, 182, 187, 191
v, 153, 293, 333, 341
Val, 153, 293, 341
$\widehat{\mathbf{Val}}_d$, 182
$\widehat{\mathbf{Val}}$, 144, 187, 191
$\mathbf{Var}_\star$, 38, 171
Var, 3, 142, 284
VB_{entry}, 47
VB_{exit}, 47
vpath, 90
$\mathcal{W}_{\mathsf{CFA}}$, 309
W, 74
$\mathcal{W}_{\mathsf{UL}}$, 301
w, 333
$(x_i \sqsupseteq t_i)_{i=1}^N$, 367

Index

abstract 0-CFA, 147
abstract cache, 144, 193
abstract data value, 182
abstract environment, 144, 191
abstract heap, 111
Abstract Interpretation, 13, 211
abstraction function, 15, 234
abstract location, 109, 110
abstract reachability component, 202
abstract state, 110
abstract summary location, 110
abstract syntax, 4, 142
abstract value, 144, 191
abstract worklist algorithm, 369
acceptable 0-CFA, 146
acyclic, 418
additive, 395
adjoint, 239
adjunction, 16, 236
affine, 396
alphabet, 426
ancestor, 420
annotated type, 287, 321, 327, 335
annotated type environment, 287
Annotated Type System, 18, 283
annotation, 287, 321, 327, 335
annotation substitution, 309
annotation variable, 306
arc, 417
Array Bound Analysis, 221, 247
ascending chain, 399
Ascending Chain Condition, 67, 399
assigned variable, 101
assumption set, 99, 198
atomic subtyping, 350
augmented type, 300
Available Expressions Analysis, 39

back edge, 422
backward analysis, 66, 85
BANE, 387
base of origin, 356
basic block, 136
behaviour, 344
behaviour variable, 344
binding time analysis, 356
Bit Vector Framework, 137, 380, 387

call-by-name, 207
call-by-result, 82
call-by-value, 82
Call-Tracking Analysis, 23
called procedure, 102
call string, 96, 196
cartesian product, 398
Cartesian Product Algorithm, 196
chain, 399
channel, 339
channel identifier, 339
channel pool, 342
Chaotic Iteration, 25
child, 420
closure, 153
Closure Analysis, 199
Code Motion, 54
coinductive definition, 150, 165, 412
collecting semantics, 13, 266, 277
combination operator, 67
Communication Analysis, 339, 343
compatible shape graph, 114
complete lattice, 393

completely additive, 395
completely multiplicative, 396
complete path, 89
concrete syntax, 4
concretisation function, 15, 234
conditional constraint, 12, 173, 367
conservative extension, 291
Constant Folding, 27, 72, 132
Constant Propagation Analysis, 72, 132, 186, 212, 265, 266
constraint, 173, 308
constraint based 0-CFA, 174
Constraint Based Analysis, 141
constraint based approach, 8, 10, 11
constraint system, 60, 69, 367
context-insensitive, 95, 150, 190
context-sensitive, 95, 190
context environment, 191
context information, 91, 191
contravariant, 324
Control Flow Analysis, 10, 213, 287, 288
correctness relation, 62, 158, 214, 241, 260
course of values induction, 407
covariant, 324
cover, 301, 309
CPA, 196
cross edge, 422
cycle, 418

DAG, 418
data array, 179
Data Flow Analysis, 5, 35
Dead Code Elimination, 49, 53, 132
Definition-Use chain, 52
definition clear path, 52
Denotational Semantics, 223, 273
depth-first spanning forest, 375, 422
depth-first spanning tree, 422
derivation sequence, 57
descendant, 420
descending chain, 399
Descending Chain Condition, 68, 399
Detection of Signs Analysis, 91, 183
DFSF, 422
DFST, 422
digraph, 417
directed graph, 375, 417
directed path, 417
directly dominates, 421
direct product, 254
direct tensor product, 255
distributive, 69, 395
Distributive Framework, 69
dominates, 420
dominator, 420
du-chain, 52, 132
duality, 273
dynamic dispatch problem, 84, 142

edge, 417
edge array, 179
effect, 24, 321, 327, 335
Effect System, 18
elementary block, 3, 36
embellished Monotone Framework, 91
environment, 86, 153, 332
equality of annotations, 290
equational approach, 5
equation system, 60, 65, 66, 69, 366
equivalent, 427
evaluation context, 341
eventually stabilises, 399
Exception Analysis, 325
expression, 142, 283, 339
extended expression, 331
extended type, 335
extensive, 223, 402
extraction function, 237, 246
extremal label, 69
extremal value, 69, 186

faint variable, 136
fast, 391

fastness closure, 391
FIFO, 374
final label, 36
finite chain, 399
first-in first-out, 374
first-order analysis, 212
fixed point, 223, 402
flow, 37, 69
flow-insensitive, 101, 150
flow-sensitive, 101
flow graph, 37
flow variable, 365
forest, 420
forward analysis, 66, 85, 115
forward edge, 422
free algebra, 306
free variable, 143
fresh type variable, 302
functional, 409
functional composition, 247

Galois connection, 16, 234
Galois insertion, 242
generalisation, 329, 337
generalised Monotone Framework, 262
general subtyping, 350
generated by, 217
graph, 417
graph formulation of the constraints, 177, 375
greatest element, 394
greatest fixed point, 223, 402
greatest lower bound, 393
ground substitution, 301, 309
ground validation, 309

handle, 375, 419
heap, 106
height, 399
hoisting, 46

in-degree, 420
independent attribute analysis, 138, 249
induced analysis, 16, 258
induction on the shape, 406
inductive definition, 165, 408
inequation system, 366
influence set, 369
initial label, 36
injective, 395
instance, 69, 186, 262
instantiation, 329, 338
instrumented semantics, 133
intermediate expression, 154
intermediate term, 154
interprocedural analysis, 82, 196
interprocedural flow, 84, 142
intraprocedural analysis, 82
isolated entries, 39
isolated exits, 39
isomorphism, 397

join semi-lattice, 67

k-CFA, 191, 198

label, 37
label consistent, 39
last-in first-out, 372
latent effect, 24
lazy, 207
least element, 394
least fixed point, 223, 261, 402
least upper bound, 393
LIFO, 372
live variable, 49
Live Variables Analysis, 49
location, 86, 105, 320
logical relation, 214, 219
loop connectedness, 380, 424
lower adjoint , 239
lower bound, 393

materialise, 121
mathematical induction, 405
Maximal Fixed Point, 74
may analysis, 66, 115
Meet Over all Paths, 78

Meet over all Valid Paths, 89
MFP solution, 74
model intersection property, 162
Modified Post Correspondence Problem, 79
monotone, 395
Monotone Framework, 68, 391
monotone function space, 253, 398
monotone structure, 186, 275, 355
Monotone Type System, 355
monovariant, 190
Moore family, 136, 162, 172, 215, 299, 395
MOP solution, 78
multiple entry, 418
multiplicative, 395
must analysis, 66, 115
MVP solution, 89, 90

narrowing operator, 230
Natural Semantics, 292
node, 417
non-free algebra, 306
non-trivial expression, 39

offset, 333
open system, 149, 182
optimal analysis, 17, 261
ordered forest, 420
ordered tree, 420
out-degree, 420

PAG, 387
parent, 420
partially ordered set, 393
partial ordering, 393
path, 78, 417
pointer expression, 104
polymorphic recursion, 336
polymorphism, 327, 329
polynomial k-CFA, 195, 198
polyvariant, 190
procedure call graph, 102
process, 339
process pool, 342
program point, 284
proof by coinduction, 412
proof by induction, 412
proof normalisation, 312, 362
proper, 420
properly dominates, 421
property space, 67

Reaching Definitions Analysis, 4, 43
reduced graph, 381, 419
reduced product, 256
reduced tensor product, 256
reducible, 424
reduction operator, 244
reductive, 223, 402
reference variable, 319
region, 331, 333, 344
Region Inference, 330
region name, 331
region polymorphic closure, 333
region variable, 331
regular expression, 426
relational analysis, 138, 250
representation function, 216, 237, 242, 260, 264
reverse flow, 38
reverse postorder, 375, 422
root, 375, 419
Round Robin Algorithm, 378

safe approximation, 168
scaling factor, 356
second-order analysis, 212
security analysis, 356
selector name, 104
self-loop, 417
semantically reaching definitions, 13
semantic correctness, 295
semantics based, 3, 29
semantics directed, 3, 29
sequential composition, 247
Set Based Analysis, 198
set constraints, 201, 387

sets of states analysis, 265
Shape Analysis, 104, 274
shape conformant subtyping, 324, 350
shape graph, 109, 114
sharing information, 112
Sharlit, 387
Side Effect Analysis, 320
simple annotation, 306
simple substitution, 307
simple type, 306
simple type environment, 307
smash product, 398
Spare, 387
state, 54, 105
storable value, 333
store, 86, 320, 333
strict, 396
strong component, 381, 419
strongly connected, 381, 419
structural induction, 406
Structural Operational Semantics, 54, 153
subeffecting, 288, 324, 329, 336, 347
subject reduction result, 158
substitution, 301
subsumption rule, 19, 21
subtyping, 324, 329, 336, 347
surjective, 395
syntactic completeness, 308, 314
syntactic soundness, 308, 312
syntax directed 0-CFA, 169
System Z, 387

Tarski's Fixed Point Theorem, 402
temporal order, 339
tensor product, 252, 271
term, 142
topological sort, 418
total function space, 252, 398
trace, 13, 133
transfer function, 68
transition, 55
tree, 420
tree edge, 422
type, 22, 285, 343
Type and Effect System, 17, 283
type environment, 22, 285
type reconstruction, 301, 309
type schema, 327, 335, 344
type substitution, 308
type variable, 300, 306
typing judgement, 286, 288, 321, 329, 336, 345

UCAI, 291, 306
ud-chain, 52, 132, 145
underlying type, 287
underlying type system, 284
unification procedure, 302, 305, 307
uniform k-CFA, 191, 193, 198
unit, 356
upper adjoint, 239
upper bound, 393
upper bound operator, 225
upper closure operator, 270
Use-Definition chain, 52, 145

valid path, 89
value, 153, 293, 341
vertex, 417
Very Busy Expressions Analysis, 46

well-founded induction, 407
well-founded ordering, 407
widening operator, 226, 267, 391
worklist, 74, 179, 368, 369

Y2K, 356

Bibliography

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proc. POPL '99*, pages 147–160. ACM Press, 1999.

[2] O. Agesen. The cartesian product algorithm. In *Proc. ECOOP'95*, volume 952 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 1995.

[3] O. Agesen, J. Palsberg, and M. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. In *Proc. ECOOP'93*, volume 707 of *Lecture Notes in Computer Science*. Springer, 1993.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[6] A. Aiken. Set constraints: Results, applications and future directions. In *Proc. Second Workshop on the Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 326–335. Springer, 1994.

[7] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2):79–111, 1999.

[8] A. Aiken, M. Fähndrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proc. Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 78–96. Springer, 1998.

[9] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proc. FPCA '93*, pages 31–41. ACM Press, 1993.

[10] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Proc. POPL '94*, pages 163–173. ACM Press, 1994.

[11] F. E. Allen and J. A. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, 1976.

[12] B. Alpern, M. M. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proc. POPL '88*, pages 1–11. ACM Press, 1988.

[13] T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency.* Imperial College Press, 1999.

[14] T. Amtoft, F. Nielson, and H.R. Nielson. Type and behaviour reconstruction for higher-order concurrent programs. *Journal of Functional Programming*, 7(3):321–347, 1997.

[15] T. Amtoft, F. Nielson, H.R. Nielson, and J. Ammann. Polymorphic subtyping for effect analysis: The dynamic semantics. In *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 172–206. Springer, 1997.

[16] A. Arnold and I. Guessarian. *Mathematics for Computer Science.* Prentice Hall International, 1996.

[17] U. Assmann. How to uniformly specify program analysis and transformation. In *Proc. CC '96*, volume 1060 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1996.

[18] A. Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proc. ICFP '97*, pages 1–10. ACM Press, 1997.

[19] P. N. Benton. Strictness logic and polymorphic invariance. In *Proc. Second International Symposium on Logical Foundations of Computer Science*, volume 620 of *Lecture Notes in Computer Science*, pages 33–44. Springer, 1992.

[20] P. N. Benton. Strictness properties of lazy algebraic datatypes. In *Proc. WSA '93*, volume 724 of *Lecture Notes in Computer Science*, pages 206–217. Springer, 1993.

[21] S. K. Biswas. A demand-driven set-based analysis. In *Proc. POPL '97*, pages 372–385. ACM Press, 1997.

[22] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Control flow analysis for the π-calculus. In *Proc. CONCUR'98*, number 1466 in Lecture Notes in Computer Science, pages 84–98. Springer, 1998.

[23] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Static analysis of processes for no read-up and no write-down. In *Proc. FOSSACS'99*, number 1578 in Lecture Notes in Computer Science, pages 120–134. Springer, 1999.

[24] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 10:407–435, 1992.

[25] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 1993.

[26] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *Computing Surveys*, 24(3), 1992.

[27] G. L. Burn, C. Hankin, and S. Abramsky. Strictness Analysis for Higher-Order Functions. *Science of Computer Programming*, 7:249–278, 1986.

[28] W. Charatonik and L. Pacholski. Set constraints with projections are in NEXPTIME. In *Proc. FOCS '94*, pages 642–653, 1994.

[29] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for prolog. *ACM TOPLAS*, 16(1):35–101, 1994.

[30] D. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Proc. PLDI '90*, pages 296–310. ACM Press, 1990.

[31] C. Colby. Analyzing the communication topology of concurrent programs. In *Proc. PEPM '95*, pages 202–214. ACM Press, 1995.

[32] C. Colby. Determining storage properties of sequential and concurrent programs with assignment and structured data. In *Proc. SAS '95*, volume 983 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 1995.

[33] A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. In *Proc. SAS '95*, Lecture Notes in Computer Science, pages 100–117. Springer, 1995.

[34] A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. *ACM TOPLAS*, 19(1):7–47, 1997.

[35] P. Cousot. Semantics Foundation of Program Analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice Hall International, 1981.

[36] P. Cousot. Types as abstract interpretations. In *Proc. POPL '97*, pages 316–331. ACM Press, 1997.

[37] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. POPL '77*, pages 238–252. ACM Press, 1977.

[38] P. Cousot and R. Cousot. Static determination of dynamic properties of generalised type unions. In *Conference on Language Design for Reliable Software*, volume 12(3) of *ACM SIGPLAN Notices*, pages 77–94, 1977.

[39] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. POPL '79*, pages 269–282, 1979.

[40] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Proc. PLILP '92*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer, 1992.

[41] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proc. POPL '78*, pages 84–97. ACM Press, 1978.

[42] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadek. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, 1991.

[43] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, 1990.

[44] A. Deutsch. On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher Order Functional Specifications. In *Proc. POPL '90*, pages 157–169. ACM Press, 1990.

[45] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proc. PLDI '94*, pages 230–241. ACM Press, 1994.

[46] P. H. Eidorf, F. Henglein, C. Mossin, H. Niss, M. H. Sørensen, and M. Tofte. AnnoDomini: From type theory to year 2000 conversion tool. In *Proc. POPL '99*, pages 1–14. ACM Press, 1999.

[47] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. PLDI '94*, pages 242–256. ACM Press, 1994.

[48] S. Even. *Graph Algorithms*. Pitman, 1979.

[49] K.-F. Faxén. Optimizing lazy functional programs using flow inference. In *Proc. SAS '95*, volume 983 of *Lecture Notes in Computer Science*, pages 136–153. Springer, 1995.

[50] K.-F. Faxén. Polyvariance, polymorphism, and flow analysis. In *Proc. Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 260–278. Springer, 1997.

[51] C. Fecht and H. Seidl. An even faster solver for general systems of equations. In *Proc. SAS '96*, volume 1145 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 1996.

[52] C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *Proc. ESOP '98*, volume 1381 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 1998.

[53] C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. *Nordic Journal of Computing*, 5:304–329, 1998.

[54] C. Fecht and H. Seidl. A faster solver for general systems of equations. *Science of Computer Programming*, 35(2):137–161, 1999.

[55] C. N. Fischer and Jr. R. J. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings, 1988.

[56] C. Flanagan and M. Felleisen. Well-founded touch optimizations for futures. Technical Report Rice COMP TR94-239, Rice University, 1994.

[57] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *Proc. POPL '95*, pages 209–220. ACM Press, 1995.

[58] Y.-C. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proc. TAPSOFT '89*, volume 352 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1989.

[59] Y.-C. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73:155–175, 1990.

[60] K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In *Proc. ICFP '97*, pages 38–51. ACM Press, 1997.

[61] R. Ghiya and L. Hendren. Connection analysis: a practical interprocedural analysis for C. In *Proc. of the eight workshop on languages and compilers for parallel computing*, 1995.

[62] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in C. In G. Kahn, editor, *Proc. POPL '96*, pages 1–15. ACM Press, 1996.

[63] R. Giacobazzi and F. Ranzato. Compositional optimization of disjunctive abstract interpretations. In *Proc. ESOP '96*, volume 1058 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 1996.

[64] R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Science of Computer Programming*, 32:177–210, 1998.

[65] R. Giegerich, U. Möncke, and R. Wilhelm. Invariance of approximative semantics with respect to program transformations. In *Proc. GI - 11. Jahrestagung*, volume 50 of *Informatik Fachberichte*, pages 1–10. Springer, 1981.

[66] P. Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.

[67] P. Granger. Static Analysis of Linear Congruence Equalities among Variables of a Program. In *Proc. TAPSOFT '91*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192. Springer, 1991.

[68] R. R. Hansen, J. G. Jensen, F. Nielson, and H. R. Nielson. Abstract interpretation of mobile ambients. In *Proc. SAS '99*, Lecture Notes in Computer Science. Springer, 1999.

[69] M. S. Hecht. *Flow Analysis of Computer Programs.* North Holland, 1977.

[70] N. Heintze. Set-based analysis of ML programs. In *Proc. LFP '94*, pages 306–317, 1994.

[71] N. Heintze. Control-flow analysis and type systems. In *Proc. SAS '95*, volume 983 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 1995.

[72] N. Heintze and J. Jaffar. A decision procedure for a class of Herbrand set constraints. In *Proc. LICS '90*, pages 42–51, 1990.

[73] N. Heintze and J. Jaffar. An engine for logic program analysis. In *Proc. LICS '92*, pages 318–328, 1992.

[74] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with Secrecy and Integrity. In *Proc. POPL '98*, pages 365–377. ACM Press, 1998.

[75] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In *Proc. ESOP '94*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer, 1994.

[76] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[77] S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24:679–694, 1987.

[78] S. Jagannathan and S. Weeks. Analyzing Stores and References in a Parallel Symbolic Language. In *Proc. LFP '94*, pages 294–305, 1994.

[79] S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Proc. POPL '95*. ACM Press, 1995.

[80] S. Jagannathan and A. Wright. Effective flow analysis for avoiding run-time checks. In *Proc. SAS '95*, volume 983 of *Lecture Notes in Computer Science*, pages 207–224. Springer, 1995.

[81] T. P. Jensen. Strictness analysis in logical form. In *Proc. FPCA '91*, volume 523 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 1991.

[82] T. P. Jensen. Disjunctive strictness analysis. In *Proc. LICS '92*, pages 174–185, 1992.

[83] M. P. Jones. A theory of qualified types. In *Proc. ESOP '92*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 1992.

[84] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice Hall International, 1981.

[85] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proc. POPL '82*, pages 66–74. ACM Press, 1982.

[86] N. D. Jones and F. Nielson. Abstract Interpretation: a Semantics-Based Tool for Program Analysis. In *Handbook of Logic in Computer Science volume 4*. Oxford University Press, 1995.

[87] M. Jourdan and D. Parigot. Techniques for improving grammar flow analysis. In *Proc. ESOP '90*, volume 432 of *Lecture Notes in Computer Science*, pages 240–255. Springer, 1990.

[88] P. Jouvelot. Semantic Parallelization: a practical exercise in abstract interpretation. In *Proc. POPL '87*, pages 39–48, 1987.

[89] P. Jouvelot and D. K. Gifford. Reasoning about continuations with control effects. In *Proc. PLDI '89*, ACM SIGPLAN Notices, pages 218–226. ACM Press, 1989.

[90] P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Proc. POPL '91*, pages 303–310. ACM Press, 1990.

[91] G. Kahn. Natural semantics. In *Proc. STACS'87*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[92] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23:158–171, 1976.

[93] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[94] M. Karr. Affine Relationships among Variables of a Program. *Acta Informatica*, 6(2):133–151, 1976.

[95] A. Kennedy. Dimension types. In *Proc. ESOP '94*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 1994.

[96] G. Kildall. A Unified Approach to Global Program Optimization. In *Proc. POPL '73*, pages 194–206. ACM Press, 1973.

[97] M. Klein, J. Knoop, D. Koschützki, and B. Steffen. DFA & OPT-METAFrame: A toolkit for program analysis and optimisation. In *Proc. TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 422–426. Springer, 1996.

[98] D. E. Knuth. An empirical study of Fortran programs. *Software — Practice and Experience*, 1:105–133, 1971.

[99] W. Landi and B. G. Ryder. Pointer-Induced Aliasing: A Problem Classification. In *Proc. POPL '91*, pages 93–103. ACM Press, 1991.

[100] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proc. PLDI '92*, pages 235–248. ACM Press, 1992.

[101] J. Larus and P. Hilfinger. Detecting conflicts between structure accesses. In *Proc. PLDI '88*, pages 21–34. ACM Press, 1988.

[102] J. M. Lucassen and D. K. Gifford. Polymorphic effect analysis. In *Proc. POPL '88*, pages 47–57. ACM Press, 1988.

[103] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks - a unified model. *Acta Informatica*, 28(2):121–163, 1990.

[104] F. Martin. Pag – an efficient program analyzer generator. *Journal of Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[105] F. Masdupuy. Using Abstract Interpretation to Detect Array Data Dependencies. In *Proc. International Symposium on Supercomputing*, pages 19–27, 1991.

[106] R. Milner. *Communication and Concurrency.* Prentice Hall International, 1989.

[107] J. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.

[108] B. Monsuez. Polymorphic types and widening operators. In *Proc. Static Analysis (WSA '93)*, volume 724 of *Lecture Notes in Computer Science*, pages 267–281. Springer, 1993.

[109] R. Morgan. *Building an Optimising Compiler.* Digital Press, 1998.

[110] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997 (third printing).

[111] F. Nielson. *Abstract Interpretation using Domain Theory.* PhD thesis, University of Edinburgh, Scotland, 1984.

[112] F. Nielson. Program Transformations in a denotational setting. *ACM TOPLAS*, 7:359–379, 1985.

[113] F. Nielson. Tensor Products Generalize the Relational Data Flow Analysis Method. In *Proc. 4th Hungarian Computer Science Conference*, pages 211–225, 1985.

[114] F. Nielson. A formal type system for comparing partial evaluators. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Proc. Partial Evaluation and Mixed Computation*, pages 349–384. North Holland, 1988.

[115] F. Nielson. Two-Level Semantics and Abstract Interpretation. *Theoretical Computer Science — Fundamental Studies*, 69:117–242, 1989.

[116] F. Nielson. The typed λ-calculus with first-class processes. In *Proc. PARLE'89*, volume 366 of *Lecture Notes in Computer Science*, pages 355–373. Springer, 1989.

[117] F. Nielson. Semantics-directed program analysis: a tool-maker's perspective. In *Proc. Static Analysis Symposium (SAS)*, number 1145 in Lecture Notes in Computer Science, pages 2–21. Springer, 1996.

[118] F. Nielson and H. R. Nielson. Finiteness Conditions for Fixed Point Iteration. In *Proc. LFP '92*, pages 96–108. ACM Press, 1992.

[119] F. Nielson and H. R. Nielson. From CML to process algebras. In *Proc. CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 493–508. Springer, 1993.

[120] F. Nielson and H. R. Nielson. From CML to its process algebra. *Theoretical Computer Science*, 155:179–219, 1996.

[121] F. Nielson and H. R. Nielson. Operational semantics of termination types. *Nordic Journal of Computing*, pages 144–187, 1996.

[122] F. Nielson and H. R. Nielson. Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis. In *Proc. POPL '97*. ACM Press, 1997.

[123] F. Nielson and H. R. Nielson. A prescriptive framework for designing multi-level lambda-calculi. In *Proc. PEPM'97*, pages 193–202. ACM Press, 1997.

[124] F. Nielson and H. R. Nielson. The flow logic of imperative objects. In *Proc. MFCS'98*, number 1450 in Lecture Notes in Computer Science, pages 220–228. Springer, 1998.

[125] F. Nielson and H. R. Nielson. Flow logics and operational semantics. *Electronic Notes of Theoretical Computer Science*, 10, 1998.

[126] F. Nielson and H. R. Nielson. Interprocedural control flow analysis. In *Proc. ESOP '99*, number 1576 in Lecture Notes in Computer Science, pages 20–39. Springer, 1999.

[127] F. Nielson, H.R. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The algorithm. In *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 207–243. Springer, 1997.

[128] H. R. Nielson, T. Amtoft, and F. Nielson. Behaviour analysis and safety conditions: a case study in CML. In *Proc. FASE '98*, number 1382 in Lecture Notes in Computer Science, pages 255–269. Springer, 1998.

[129] H. R. Nielson and F. Nielson. Bounded fixed-point iteration. *Journal of Logic and Computation*, 2(4):441–464, 1992.

[130] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992. (An on-line version may be available at `http://www.imm.dtu.dk/~riis/Wiley_book/wiley.html`).

[131] H. R. Nielson and F. Nielson. Higher-Order Concurrent Programs with Finite Communication Topology. In *Proc. POPL '94*. Springer, 1994.

[132] H. R. Nielson and F. Nielson. Communication analysis for Concurrent ML. In F. Nielson, editor, *ML with Concurrency*, Monographs in Computer Science, pages 185–235. Springer, 1997.

[133] H. R. Nielson and F. Nielson. Flow logics for constraint based analysis. In *Proc. CC '98*, volume 1383 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 1998.

[134] H.R. Nielson, F. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 141–171. Springer, 1997.

[135] L. Pacholski and A. Podelski. Set constraints: A pearl in research on constraints. In *Proc. Third International Conference on the Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 549–561. Springer, 1997.

[136] J. Palsberg. Closure analysis in constraint form. *ACM TOPLAS*, 17 (1):47–62, 1995.

[137] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems.* Wiley, 1994.

[138] H. D. Pande and B. G. Ryder. Data-flow-based virtual function resolution. In *Proc. SAS '96*, volume 1145 of *Lecture Notes in Computer Science*, pages 238–254. Springer, 1996.

[139] J. Plevyak and A. A. Chien. Precise concrete type inference of object-oriented programs. In *Proc. OOPSLA '94*, 1994.

[140] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981.

[141] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Proc. POPL '99*, pages 119–132. ACM Press, 1999.

[142] J. H. Reif and S. A. Smolka. Data Flow Analysis of Distributed Communicating Processes. *International Journal of Parallel Programming*, 19(1):1–30, 1990.

[143] J. Reynolds. Automatic computation of data set definitions. In *Information Processing*, volume 68, pages 456–461. North Holland, 1969.

[144] B. K. Rosen. High-level data flow analysis. *Communications of the ACM*, 20(10):141–156, 1977.

[145] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proc. PLDI '95*, pages 13–22. ACM Press, 1995.

[146] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):275–316, 1986.

[147] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Proc. TAPSOFT '95*, volume 915 of *Lecture Notes in Computer Science*, pages 651–665, 1995.

[148] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proc. POPL '96*, pages 16–31. ACM Press, 1996.

[149] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, 1998.

[150] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. POPL '99*, pages 105–118. ACM Press, 1999.

[151] D. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proc. POPL '98*, pages 38–48. ACM Press, 1998.

[152] P. Sestoft. Replacing function parameters by global variables. Master's thesis, Department of Computer Science, University of Copenhagen, Denmark, 1988.

[153] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proc. POPL '97*, pages 1–14. ACM Press, 1997.

[154] M. Sharir. Structural Analysis: a New Approach to Flow Analysis in Optimising Compilers. *Computer Languages*, 5:141–153, 1980.

[155] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*. Prentice Hall International, 1981.

[156] O. Shivers. Control flow analysis in Scheme. In *Proc. PLDI '88*, volume 7 (1) of *ACM SIGPLAN Notices*, pages 164–174. ACM Press, 1988.

[157] O. Shivers. Data-flow analysis and type recovery in Scheme. In P.Lee, editor, *In Topics in Advanced Language Implementation*, pages 47–87. MIT Press, 1991.

[158] O. Shivers. The semantics of Scheme control-flow analysis. In *Proc. PEPM '91*, volume 26 (9) of *ACM SIGPLAN Notices*. ACM Press, 1991.

[159] J. H. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7:207–274, 1989.

[160] G. S. Smith. Polymorphic type inference with overloading and subtyping. In *Proc. TAPSOFT '93*, volume 668 of *Lecture Notes in Computer Science*, pages 671–685. Springer, 1993.

[161] G. S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[162] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. POPL '96*, pages 32–41. ACM Press, 1996.

[163] D. Stefanescu and Y. Zhou. An equational framework for the flow analysis of higher order functional programs. In *Proc. LFP '94*, pages 318–327, 1994.

[164] B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21:115–239, 1993.

[165] J. Stransky. A lattice for abstract interpretation of dynamic (lisp-like) structures. *Information and Computation*, 1990.

[166] J.-P. Talpin and P. Jouvelot. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[167] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Proc. LICS '92*, pages 162–173, 1992.

[168] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[169] Y.-M. Tang. *Control-Flow Analysis by Effect Systems and Abstract Interpretation.* PhD thesis, Ecole des Mines de Paris, 1994.

[170] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.

[171] R. E. Tarjan. A unified approach to path programs. *Journal of the ACM*, 28(3):577–593, 1981.

[172] S. Tjiang and J. Hennessy. Sharlit – a tool for building optimizers. In *Proc. PLDI '92.* ACM Press, 1992.

[173] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[174] M. Tofte and L. Birkedal. A region inference algorithm. *ACM TOPLAS*, 20(3):1–44, 1998.

[175] M. Tofte and J.-P. Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proc. POPL '94*, pages 188–201. ACM Press, 1994.

[176] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132:109–176, 1997.

[177] G. V. Venkatesh and C. N. Fischer. Spare: A development environment for program analysis algorithms. *IEEE Transactions on Software Engineering*, 1992.

[178] J. Vitek, R. N. Horspool, and J. S. Uhl. Compile-Time Analysis of Object-Oriented Programs. In *Proc. CC '92*, volume 641 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 1992.

[179] A. B. Webber. Program analysis using binary relations. In *Proc. PLDI '97*, volume 32 (5) of *ACM SIGPLAN Notices*, pages 249–260. ACM Press, 1997.

[180] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM TOPLAS*, pages 181–210, 1991.

[181] R. Wilhelm and D. Maurer. *Compiler Design.* Addison-Wesley, 1995.

[182] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. PLDI '95*, pages 1–12. ACM Press, 1995.

[183] A. K. Wright. Typing references by effect inference. In *Proc. ESOP '92*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491. Springer, 1992.

[184] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

[185] K. Yi and W. L. Harrison III. Automatic generation and management of interprocedural program analyses. In *Proc. POPL '93*, pages 246–259. ACM Press, 1993.

Principles of Program Analysis

NIELSON
NIELSON
HANKIN

Program analysis concerns static techniques for computing reliable approximate information about the dynamic behaviour of programs. Applications include compilers (for code improvement), software validation (for detecting errors in algorithms or breaches of security) and transformations between data representation (for solving problems such as the Y2K problem). This book is unique in giving an overview of the four major approaches to program analysis: data flow analysis, constraint based analysis, abstract interpretation, and type and effect systems. The presentation demonstrates the extensive similarities between the approaches; this will aid the reader in choosing the right approach and in enhancing it with insights from the other approaches. The book covers basic semantic properties as well as more advanced algorithmic techniques. The book is aimed at M. Sc. and Ph. D. students but will be valuable also for experienced researchers and professionals.

ISBN 3-540-65410-0

9 783540 654100

springeronline.com