Static Analysis of Dynamic Languages

Magnus Madsen

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Static Analysis of Dynamic Languages

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Magnus Madsen
March 2, 2015

Abstract

Dynamic programming languages are highly popular and widely used. JavaScript is often called the lingua franca of the web and it is the de facto standard for client-side web programming. On the server-side the PHP, Python and Ruby languages are prevalent. What these languages have in common is an expressive power which is not easily captured by any static type system. These, and similar dynamic languages, are often praised for their ease-of-use and flexibility. Unfortunately, this dynamism comes at a great cost: The lack of a type system implies that most errors are not discovered until run-time. Thus, in the worst-case, these bugs are not uncovered before they are encountered by real users of the system. A further cost is limited tool support: For instance, integrated development environments with code completion, code navigation and automatic refactorings are widely available for languages with static type systems, such as Java and C♯, but the same features are rarely available for dynamic languages such as JavaScript.

The aim of this thesis is to investigate techniques for improving the tool-support for dynamic programming languages without imposing any artificial restrictions on the behaviour of these languages. A common theme is the reliance on static program analysis to over-approximate the behaviour of programs written in these languages. Specifically, the use of whole-program dataflow analysis. The research challenge of this line of work is the adaption of existing- and invention of new dataflow analysis techniques to tackle the nature of dynamic programming languages.

Resumé

Dynamiske programmeringssprog er meget populære og udbredte. JavaScript bliver ofte kendetegnet som lingua franca af internettet og er de facto standarden for klient-side programmering. På server-siden er programmeringssprogene PHP, Python og Ruby velanvendte. Det som disse sprog har tilfælles er en udtrykskraft som ikke nemt kan tæmmes af noget statisk type system. Disse, og lignende dynamiske sprog, bliver ofte rost for deres nemhed og fleksibilitet. Desværre kommer denne dynamisme ofte med en høj pris: Manglen på et type system betyder at de fleste fejl ikke bliver opdaget før programmet afvikles. Det vil sige, at i det værste tilfælde, bliver de ikke fundet før rigtige brugere af systemet støder på dem. En yderligere omkostning er begrænset værktøjsunderstøttelse: Eksempelvis er integrerede udviklingsmiljøer med kode færdiggørelse, kode navigation og omstrukturering vidt tilgængelige for sprog med statiske type systemer, som fx Java og C♯, men de samme funktionaliteter er sjældent tilgængelige for dynamiske sprog som f.eks. JavaScript.

Målet med denne afhandling er at undersøge teknikker for at forbedre værktøjsunderstøttelsen for dynamiske programmeringssprog uden at pålægge kunstige begrænsninger på opførslen af disse sprog. Et gennemgående tema er brugen af a statisk program analyse til at over-approksimere opførslen af programmer skrevet i disse sprog. Specifikt, brugen af af hel-programs datastrømningsanalyse. Den forskningsmæssige udfordring, med dette arbejde, er anvendelsen og opfindelsen af nye datastrømningsanalyse teknikker til at tackle naturen af dynamiske programmeringssprog.

Acknowledgements

"The Cosmos is all that is or ever was or ever will be."

— Carl Sagan

I would like to express my thanks to the following people for their help, inspiration and moral support during my PhD studies:

Anders Møller, my advisor, for introducing me to static analysis, for our many good discussions and for all the opportunities given to me. Ondřej Lhoták, my advisor at the University of Waterloo, for ensuring a pleasant and productive stay abroad. Olivier Danvy introducing me to functional programming, for his moral support and for his insights and perspectives on life. Lars Aarge and Jesper Buus Nielsen, my PhD support group, for their guidance and support. Asger Feldhaus, my office mate, for our many good discussions and for the time we watched a lawn mower achieve its dreams of flying [YouTube, 2015]. Mathias Schwarz for being a good travel companion during my early years as a PhD student. Esben Andreasen for many interesting and fruitful discussions. And a thanks to the rest of my colleagues and friends; Casper Jensen, Ian Zerny, Jakob Thomsen, Jan Midtgaard, Jesper Nielsen and Casper Kejlberg-Rasmussen. Finally, I would like to thank my study group; Heidi Hyldegaard, Matthias Ingesman and Thomas Plousgaard without whom the early years at computer science would never have been the same!

About the Author. Magnus Madsen was born in 1986 in Aarhus, Denmark. It was at HTX Teknisk Gymnasium that he first become interested in math and science. Magnus has accepted a post doctoral position at the University of Waterloo, Canada, starting September 2015 under the supervision of Ondřej Lhoták. Until then, he is doing an internship at Samsung Research, California, under the supervision of Frank Tip.

Contents

Abstract i

Resumé ii

Acknowledgements iii

Contents iv

I Overview 1

1 Introduction 2
1.1 Background 3
1.2 Thesis Statement 4
1.3 Methodology 4
1.4 Outline 6

2 Dynamic Languages 7
2.1 Static- vs. Dynamic Typing 9
2.2 Common Features 12
2.3 A Formal Calculus 14

3 Static Analysis 20
3.1 Abstractions 21
3.2 Dimensions of Static Analysis 22
3.3 Trade-offs in Static Analysis 22

4 Asynchronous Events 25
4.1 Semantics 26
4.2 Event-related Bugs 28

4.3 Static Analysis 29

5 Dynamic Fields **31**
5.1 Semantics 32
5.2 Static Analysis 33

6 Analysis of Libraries **38**
6.1 Use Analysis 40

7 The Browser Environment **44**
7.1 The Document Object Model 45
7.2 Events & Event Listeners 47

8 Sparse Dataflow Analysis **48**
8.1 Static Single Assignment-form 49
8.2 Construction of SSA-form 51

9 FLIX: Programming with Fixedpoints **58**
9.1 Datalog and its Limitations 60
9.2 The FLIX Language 62
9.3 Safety Properties 65
9.4 Verification 66

10 Conclusions **68**

II Papers **71**

11 Static Analysis of Event-based JavaScript Applications **72**
11.1 Introduction 73
11.2 Motivating Examples 74
11.3 Language 77
11.4 Beyond Call Graphs 82
11.5 Analysis Framework 86
11.6 Evaluation 90
11.7 Related Work 96
11.8 Conclusion 98

12 String Analysis for Dynamic Field Access **99**
12.1 Introduction 100
12.2 Related Work 103
12.3 String Domains 105
12.4 Evaluation 114
12.5 Conclusion 120

13 Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries 121
13.1 Introduction . . . 122
13.2 Analysis Challenges . . . 126
13.3 Overview . . . 128
13.4 Techniques . . . 133
13.5 Experimental Evaluation . . . 143
13.6 Related Work . . . 151
13.7 Conclusions . . . 152

14 Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications 153
14.1 Introduction . . . 155
14.2 Challenges . . . 159
14.3 The TAJS Analyzer . . . 163
14.4 Modeling the HTML DOM and Browser API . . . 164
14.5 Evaluation . . . 170
14.6 Related Work . . . 176
14.7 Conclusion . . . 177

15 Sparse Dataflow Analysis with Pointers and Reachability 179
15.1 Introduction . . . 180
15.2 A Basic Analysis Framework . . . 182
15.3 Sparse Analysis . . . 185
15.4 Implementation and Evaluation . . . 193
15.5 Related Work . . . 196
15.6 Conclusion . . . 197

16 A Framework for Declarative Specification of Static Analyses 198
16.1 Introduction . . . 199
16.2 Verification . . . 214
16.3 Solver . . . 216
16.4 Case Studies . . . 217
16.5 Related Work . . . 223
16.6 Conclusion . . . 224

Bibliography 225

Part I

Overview

1

Introduction

"The limits of my language are the limits of my world."

— Ludwig Wittgenstein

Programming languages are the tools software developers use to create programs. Programs, as seen by humans, are rich in their capabilities and appear almost magical to their users. Yet at the same time, programs, as seen from the machine, are merely primitive instructions to be thoughtlessly executed. Programming languages is the bridge which connects these two worlds. Programming languages enable software developers to express their intent in rich abstractions and high-level constructs which are translated into low-level instructions for the unsophisticated machine.

Programming languages come in all kinds of shapes and sizes. There is no universally adopted programming language. Some languages are general purpose, usable for any kind of programming and make few assumptions about the programs the developers write. Other languages are domain-specific and tailored for a specific kind of programming. Programming languages differ in their paradigm (e.g. imperative, functional, declarative, . . .), their feature set (e.g. availability of higher-order functions), their execution model (interpreted vs. compiled), their syntax (C-style, S-expression, indention, . . .), their typing discipline (e.g. statically vs. dynamically-typed), and so on. All these choices permit a huge design space for programming languages and, perhaps as result, there is an abundance of programming languages.

Although programming languages are plentify, the design of new programming languages is thriving. The last years have seen numerous new languages being created and implemented: Ceylon, Kotlin, Swift, Rust, Go, Dart, TypeScript and Flow, to name just a few. These are not hobby projects undertaken by a few people in their spare time nor are they research projects created by academics. They are, in fact, business projects being undertaken by large IT companies, including Apple, Facebook, Google, and Microsoft. Programming language design is a big business.

1.1 Background

The *dynamic programming languages* are a special class of programming languages celebrated for their flexibility, expressive power and the few restrictions they impose on the programmer. On the surface these traits appear useful, but they also have drawbacks. The lack of restrictions leave more room for the programmer to make mistakes and complicate the construction of tools for these languages. To better understand the philosophy behind dynamic languages, let us consider a small program fragment:

```
42 + if c then 21 else false
```

What can be said about this program? A developer who is familiar with statically-typed languages, like Java, would probably say that the program is ill-typed since the `if-then-else` expression may evaluate to either a boolean or number. Thus, he or she would argue, the program should be rejected by the compiler. However, a developer who is familiar dynamic languages, like JavaScript, might argue that the program is perfectly fine as long as the condition is always true.

The Java developer might then continue and ask how we are to compute the sum of `42` and `false` if the condition is false. The JavaScript developer might reply that the addition operator ignores non-number values, coerces boolean to some numeric value, works as string concatenation or simply throws the boolean away and returns `42`. Whatever the solution, the important point is that the program does not crash, but continues running.

Many dynamic programming languages follow this paradigm of *don't crash, but keep executing*. The simple idea is that operations, which may be nonsensical in other languages, should be given some semantics, and that the program should continue execution whenever possible. The proponents argue that all that really matters, in the end, is that the program works under the given circumstances (e.g. a particular input), i.e. it is unimportant that the program might fail in some situation, if that situation is *perceived* not to occur. Opponents argue that the developer cannot foresee all situations and that program execution should be stopped as soon as possible to prevent harm and to aid debugging.

The dynamic programming languages derive their name from being in contrast to the *statically-typed* programming languages. That is, the word "dynamic" refers to the lack of a static type system. Dynamic programming languages still have types, but they are checked at run-time, not compile-time. The benefit is that programmers can write any program and not have it rejected by the type-checker. The downside is that type errors occur at run-time and little or no type information is available for developer tools.

To overcome these issues, this thesis proposes the use of *static analysis* technique to reason about programs written in dynamic programming languages. A static analysis is a tool which executes the program "in the abstract", i.e. approximates all possible behaviour of the program. The computed approximation is then used to detect bugs or to provide information to developer tools in lieu of type information.

We keep the terms type system and static analysis separate, although it could be argued that a type system is a kind of static analysis. We adopt the view that a type system restricts the programs which are accepted by the compiler; it is the responsibility of the developer to correct type errors until the type checker is satisfied. A static analysis, on the other hand, takes the program "as-is" and never rejects it outright.

1.2 Thesis Statement

We can now succinctly state the claim of this thesis:

> **Thesis Statement:** We can use static analysis techniques to reason about dynamic programming languages to achieve the same benefits, in terms of correctness, safety and tool-support, which are readily available for statically-typed languages.

In other words, the purpose of this thesis is to develop static analysis techniques which are useful in compilers, linters and integrated development environments (IDEs) for bug finding, code auto-completion, code navigation and refactoring targeting programs written in dynamic programming languages. The research challenge is to design and implement these techniques, and experimentally validate their usefulness.

1.3 Methodology

We briefly discuss some points about the research methodology commonly used in the area of static analysis. A large body of work, in this line of study, follows a four-step method:

- **Problem:** The researcher identifies a specific challenge or problem faced by software developers. For example, to verify that a program satisfies its specification, to ensure that a program is free of certain errors, to help programmers debug or refactor the code. A classic challenge is to ensure that a program is free of memory errors, e.g. buffer overflows or double frees.

- **Scope:** The researcher describes, informally or formally, a model of computation which allows the specific problem. The purpose of the model is twofold: First, the model allows the researcher to define exactly when the problem occurs. Second, the model allows the researcher to abstract a way from irrelevant detail. A common method is to choose a small programming calculus which has the necessary constructs for the original problem to occur while remaining realistic. Sometimes the model is specified informally, e.g. by simply referring to all "C-style programming languages" or formally with a mathematical semantics of the language. In continuation of the example above, the model should define when a buffer overflow or double free occurs.

- **Solution:** The researcher designs and implements a static analysis which can identify the problem of interest. The implementation is often a combination of existing techniques together with new abstracts for modeling the specific problem of interest. The researcher makes the hypothesis that the static analysis tool is able to detect the problem in real-world programs.

- **Evaluation:** The researcher applies the static analysis tool to a collection of real-world programs to determine if it can find the original problem. Experimentation is necessary since programs are written by humans and hard to capture by any model. Thus, the best measure of whether an analysis "works" is to apply it to real programs. The evaluation often includes previous solutions or alternative solutions to demonstrate why the new technique is worthwhile.

Of course not every research inquiry proceeds along these lines. In some cases the result is of a more theoretical nature and can be proven mathematically. In other cases the research challenge is a *meta* question about the theory of static analysis itself and may not be motivated by a particular use of static analysis.

In the big picture, research on static analysis is not merely a quest of understanding computation and helping programmers write better applications, but the mere existence of these techniques raises important philosophical questions in computer ethics. The paper *When Formal Systems Kill: Computer Ethics and Formal Methods* [Abramson and Pike, 2011] provides a good introduction to the question of the *moral obligations* of software developers to ensure correctness of their programs.

1.4 Outline

This thesis is divided into two parts: An *overview* and a *paper* part. The paper part comprises six conference papers; four published papers and two in preparation. One of the four published papers is a student paper written by Esben Andreasen and the author. The overview part consists of an introduction to each paper. As shown in the table below, there is a one-to-one correspondence between the introductory chapters and the papers:

Overview	Paper
Chapter 4	Static Analysis of Event-based JavaScript Applications.
Chapter 5	String Analysis for Dynamic Field Access (*CC '14*).
Chapter 6	Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries (*FSE/ESEC '13*).
Chapter 7	Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications (*FSE/ESEC '11*).
Chapter 8	Sparse Dataflow Analysis with Pointers and Reachability (*SAS '14*).
Chapter 9	A Declarative Framework for Specification of Static Analyses.

Table 1.1: Correspondence between introductory chapters and papers.

Each chapter aims to introduce the research challenge, present the key insights in the proposed solution and describe related work. As such, the full details of the techniques and experimental evaluation is deferred to the papers. In some cases the introductory chapters provide additional information which is not in the original paper.

As evidenced by the title of the papers, much of the current work has been carried out in the context of the JavaScript programming language. Although the analyses have been implemented for JavaScript and experimentally evaluated on JavaScript programs, the techniques have been developed with a broader scope in mind, and we believe they are applicable to other dynamic programming languages with similar features.

A summary of the main contributions is available in Chapter 10.

2

Dynamic Languages

"To know another language is to have second soul."

— Charlemagne

What exactly is a dynamic programming language? Benjamin Pierce classifies programming languages along two axes: whether they are *safe* or *unsafe*, and whether they are *dynamically* or *statically* checked [Pierce, 2002]. A *safe* language is a language which protects its abstractions. For instance, that the run-time cannot be corrupted by writing past the end of some memory region, that uninitialized memory is not read, that lexically scoped variables cannot be changed outside their scope, and so on. A *statically checked* language is a language which ensures the safety of its abstractions at compile-time. For instance, by checking that all variables are initialized before they are used. A safe language, which is dynamically checked, must enforce these safety properties by performing checks at run-time. The table below classifies some existing languages along these axes:

	Statically checked	Dynamically checked
Safe	Java, Standard ML, etc.	JavaScript, Python, etc.
Unsafe	C, C++, etc.	

Table 2.1: Classification of programming languages [Pierce, 2002].

As an example, Java is a safe and (mostly) statically-checked language. Its type system ensures, for example, that variables and fields are assigned values of the correct type, that methods are invoked on receiver objects that understand them, that methods are called with arguments of the expected type, that checked-exceptions are caught, and so on. These properties are enforced at compile-time. However, a few properties are deferred to run-time. For example, dynamic casts and array accesses are checked at run-time [Bodík et al., 2000; Sridharan and Bodík, 2006; Winther, 2011]. Static elimination of array

bounds checking is a difficult problem and an active area of research [Qian et al., 2002; Kowshik et al., 2002; Würthinger et al., 2007].

As another example, JavaScript is a safe and dynamically checked language. JavaScript is interpreted and has no notion of compile-time. Thus, the run-time must check every operation to ensure its safety. For example, returning `undefined` when accessing a non-existent array element.

In contrast, C and C++ are unsafe and statically checked languages. In C and C++ the run-time can be corrupted by the program. This has lead to many security bugs. A particular menace is buffer overflows where program data is overwritten by malicious input which is then executed [One, 1996; Cowan et al., 1998, 2003]. The static checking provided by these languages is incomplete and only detects certain bugs and errors; no guarantees are provided by the compiler.

Robert Harper has a different view on dynamic programming languages. He argues that dynamic programming languages are merely a special case of statically typed languages restricted to a single recursive type [Harper, 2012]. This single recursive type, or *unitype*, permits any operation and defers the checking of its safety to the run-time.

We will, for the purpose of this thesis, not require an exact definition of dynamic programming languages, but instead focus on the features provided by these languages and how they interact with static analysis techniques.

Dynamic programming languages should not be confused with *scripting languages*. Although the two are often used interchangeably. The term *scripting* describes the intended *use* of the language; a (shell) scripting language is used to glue other software programs together. Other types of languages are *system* languages for low-level programming, *query* languages for databases, *application* languages for general-purpose applications, and so on.

As claimed previously, dynamic programming languages are popular and widely used. Table 2.2 shows the ten most popular programming languages according to the Tiobe Index and the LangPop websites.

The rankings are based on information collected on the internet. For example, the LangPop website measures five metrics:

- The number of Google search results for `"<language> programming"`.
- The number of Google search results for `"filetype:<extension>"`.
- The number of Craiglist jobs for `"<language> programmer"`.
- Data from GitHub. A large repository of open source code.
- Data from OpenHub. An index of open source projects.

Rank	TIOBE.com	LangPop.com
1st	C	C
2nd	Java	Java
3rd	Objective-C	PHP
4th	C++	JavaScript
5th	C#	C++
6th	PHP	Python
7th	Python	Shell
8th	JavaScript	Ruby
9th	Perl	Objective-C
10th	VB.Net	C#

Table 2.2: Language popularity. Dynamic languages are in grey.

Ofcourse this does not necessarily indicate that a programming language is widely used. For example, a controversial or difficult language might lead to more discussion online. Furthermore, the data from GitHub and OpenHub might not be representative since not all code bases are open source.

The survey show that C and Java are the two most popular languages. More interestingly, roughly half of the most popular languages are dynamic languages, as shown in grey. These languages are, in approximate order, PHP, JavaScript, Python, Shell, Ruby and Perl. JavaScript is, as mentioned before, the most popular client-side web programming language. PHP is a server-side web programming language. Python and Ruby are application languages which are also frequently used for scripting and server-side programming. Perl and Shell are scripting languages.

2.1 Static- vs. Dynamic Typing

We have previously hinted at the advantages and disadvantages of static type systems. We now discuss these in greater detail.

Advantages of Static Type Systems

Absence of Type Errors. The primary purpose of type systems is to protect the abstractions of the language and to detect type errors at compile-time. Informally, this is often stated as *well-typed programs do not go wrong.* In type theory this is often restated as progress + preservation = safety. Progress means that well-typed terms are either values or can be evaluated further (i.e. at least one evaluation rule is applicable). Preservation means that if a

well-typed term takes a step (i.e. an evaluation rule is applied) then the result continues to be well-typed. Together these properties imply that a well-typed program reduces to a value or diverges (i.e. loops forever).

Refactoring. Refactoring is the process of restructuring a program while preserving its semantics, i.e. the program source code is changed, but the behaviour of the program is the same. Types are useful for finding the places in the source code where one change requires another change. For instance, if a programmer renames a class and re-runs the compiler, then the compiler will report all the places where the old class name was used. In the same way tool-supported refactoring, i.e. tools which automatically perform the refactoring, are dependent on type information.

Auto-completion. Integrated development environments (IDEs) often offer programmer assistance in the form of auto-complete, i.e. the action where the programmer types in some text and the IDE completes the remaining text or suggests possible completions. In some cases auto-complete is simple a syntactic matter (e.g. to auto-complete the `class` keyword), but in other cases auto-complete requires knowledge about the program. A most typical case is when the programmer is trying to access a field or invoke a method on an object and wants to inspect the available fields and methods on that object. Types provides exactly this information.

Performance. Run-time performance is improved by static type systems. Specifically, static types enable a compiler to generate more efficient code by elimination of redundant type tests and run-time checks which are otherwise required to ensure safety. For example, consider the function below:

```
function sum(x, y)
  return x + y;
```

If the type of `x` and `y` are unknown to the compiler then it cannot know whether the `+` operation is integer addition, floating-point addition or string concatenation. As consequence, the compiler must emit code for all three cases and inspect the types at run-time.

Modularity. Types help enforce modularity. Existential, universal and subtyping all help hide implementation details from the client of a module. In object-orientated programming interfaces define the boundary between application and libary code. Modularity enable library developers to change and restructure their code without affecting their clients. The type system ensures that developers, whether by mistake or negligence, do not tightly couple their code to the internals of a module.

Documentation. Type annotations, i.e. explicit types in the program source code, are useful as documentation. For instance, function argument– and return types specify the expected input and output of a function. Annotated types, unlike regular comments, are checked by the compiler and consequently always up-to-date.

Disadvantages of Static Type Systems

Type Slack. Most type systems aim for soundness, i.e. if the program is well-typed then it is safe. As a consequence, type systems must be conservative; if they cannot prove a program safe, then it must rejected. This means that the compiler may unfairly reject some programs which are perfectly fine, but cannot be typed. This unfair rejection is refered to as "type slack" and may be confusing to programmers who know their programs to be correct, but cannot get them to pass the type checker.

Program Completeness. A common assumption of type systems and compilers is that the program is complete: All used symbols must be declared and have an implementation. Incomplete programs will not pass the type checker and as a consequence cannot be run. This limits the ability of the programmer to experiment. Dynamic languages, on the other hand, seem more suited for agile development where the programmer is continuously running an incomplete program.

Lack of Open Modules/Classes. Another common assumption of type systems is that modules and classes are invariant, i.e. they do not change over time. As a consequence, it is hard or impossible to extend libraries in ways that where not anticipated by the library developer. For example, in Java it is impossible to add fields or methods to an existing class. In some cases the decorator design pattern can be used. The decorator pattern wraps an existing class inside a new class which acts as a proxy while providing additional functionality. Unfortunately, for this to work, the library designer must have anticipated such uses: The class to be decorated must be an interface or a non-final class, and the library must not rely on reference equality. Recent work has focused overcoming these limitations with *aspect-orientated programming* [Kiczales et al., 1997] or *extension methods* [Schildt, 2010], but these are still very limited compared to what is possible in dynamic languages. For instance, in JavaScript any object can be extended with new fields and methods on-the-fly. Ruby, as another example, allows class definitions to be re-opened and modified at run-time.

Annotation Burden. In languages without type inference, the programmer must manually specify type annotations. Although type annotation are useful as documentation (see above), they can be verbose and tedious to write. In Java, for example, it is not uncommon to see code like:

```
Person person = new Person();
```

which effectively contains the same information thrice. Languages with type inference, such as Standard ML and Haskell, and to a degree Scala, limit this burden. In Scala, for example, the same code can be written as:

```
val person = new Person();
```

Unfortunately, type inference is undecidable for many type systems [Tiuryn, 1990; Wells, 1999; Pierce, 2002].

2.2 Common Features

We briefly survey some of the commonly found features in dynamic programming languages. We focus our attention on the JavaScript, PHP, Python and Ruby languages. Our aim is *not* to provide a comprehensive overview of all features available in these languages, but instead to give some intuition about these languages and the challenges they pose to static analysis.

Impurity. The languages allow, if not encourage, the use of side-effects. Local variables are re-assignable. Data structures allocated in the heap are mutable. Library functions frequently mutate their arguments. Impurity implies that a static analysis cannot use equational reasoning: A function, given the same set of arguments, may return different results.

Globals. Global state is pervasive. JavaScript and PHP each have a global namespace where data can be stored. In JavaScript the global namespace is an object where fields can be added or removed dynamically. For instance, it is possible to completely redefine or delete the `Math` object. More obscurely, PHP supports directives such as `disable_functions` and `disable_classes` which can be tuned to prohibit the use of certain functions or classes. Global state is challenging for static analysis techniques since the behaviour of a piece of code cannot be determined locally. This is one of the primary motivations for whole-program static analyses.

Objects. The languages are object-orientated. PHP, Python and Ruby are class-based, whereas JavaScript is prototype-based. In JavaScript objects are essentially maps from keys (strings) to values. Fields are keys which may be added or removed on-the-fly. Ruby supports *opening* a class to a add new fields

and methods. Python supports a form of on-the-fly class decoration. A static analysis must approximate the unbounded heap by possibly collapsing several concrete objects into abstract objects. In languages with static type systems it is common to use the class of objects to guide the object abstraction. However, for dynamic programming languages no such information is available.

Higher-order Functions. The languages support higher-order functions, i.e. function values which can be assigned to variables, passed as arguments to functions and returned from functions. Higher-order functions complicate static analysis since the control-flow becomes dependent on the dataflow. Higher-order functions enable *functional programming*, but due to the prevalent use of side-effects, and the lack of types, most "functions" do not behave as functions, but as procedures.

Reflection. Reflection is form of meta-programming which allow a program to inspects its own structure at run-time. For instance, reflection is often used to execute functions and methods, where the receiver object or method name is not known statically, but has to be computed at run-time. Another example, which is available in all four languages, is the powerful `eval`-construct which parses and executes a string. The `eval` construct is often (mis)-used to construct code on-the-fly although most use cases could be solved with simpler means. Reflection is challenging for static analyses since it obscures the behaviour of the program or at least requires the static analysis to reason precisely about string values.

Ubiquitous Strings. The languages, lacking type systems, also lack rich datatypes. Instead of algebraic data types, these languages rely *special* integers or strings as arguments to functions and as return values. A favorite example, from JavaScript, is the `canPlayType` function introduced as part of HTML5 Media which returns either the empty string, `"maybe"` or `"probably"` depending on whether the supplied media can be played. The design of these libraries leads developers to use strings everywhere, and as a result, static analysis tools are forced to reason precisely about strings. Robert Harper argues against what he calls *boolean blindness* [Harper, 2012], in dynamic languages *string blindness* is a curse ten-times worse.

Coercions. Coercions, sometimes called type conversions, is the translation of a value of one type into a value of another type. Coercions are available in most languages and often used to translate numbers into strings and vice versa. Implicit coercions are coercions which are applied automatically, by the run-time or compiler, whenever the type of a value does not match the

expected type. A typical use case of implicit coercions is in equality tests, but some languages go further and provide them in other situations. Python and Ruby have coercions for numeric values. PHP has more advanced coercions and JavaScript has even more idiosyncratic coercions. For instance, JavaScript has the following coercion:

```
!new Boolean(false) = false
```

which happens because the boxed boolean object (with the internal value of `false`) is coerced to `true` since any non-null and non-undefined value is `true`. Coercions are tricky for static analyses since programmers frequently rely on them, but on the other hand, they are also a source of many bugs: A static analysis cannot easily know whether a coercion was intended or not.

2.3 A Formal Calculus

We study the dynamic programming languages using a small programming language calculus. The calculus is based on the lambda calculus extended with (functional) objects and a mutable heap. In the following chapters we extend the calculus with features from dynamic programming languages and show how they are handled by static analysis techniques. We call the base language presented in this chapter $\lambda_{\texttt{base}}$.

The design of calculus is motivated by the features discussed in the previous section and is based on a minimal JavaScript calculus [Guha et al., 2010]. As mentioned earlier, most of the present work is set in the context of JavaScript, but the techniques are more broadly applicable.

Syntax

The syntax of $\lambda_{\texttt{base}}$ is shown in Figure 2.1. The grammar consists of three syntactic categories: Constants *Cst*, values *Val* and expressions *Exp*.

Constants and Values. The constants are booleans (`true`, `false`), numbers (`1.0`, `3.14`, `42.0`), strings (`"abc", "xyz"`) and the special `null` value. The values are the constants combined with lambda expressions and objects. An object is a finite map from string keys to values. We will write objects as $\{f : v\}$ to denote an object which has a field named f with the value v. We will write $o(f)$ for the value of the field f. All values, including objects, are *functional* meaning that they are *immutable*: A value cannot be changed, but we can create a new value from an existing value. As discussed soon, the heap is the only mutable component.

$$
\begin{array}{rcll}
c \in \mathit{Cst} & = & \mathit{bool} \mid \mathit{num} \mid \mathit{str} \mid \texttt{null} & \text{[constant]} \\
\\
v \in \mathit{Val} & = & c & \text{[constant]} \\
& \mid & a & \text{[address]} \\
& \mid & \lambda\, x.\, e & \text{[lambda]} \\
& \mid & \{f : v, \cdots\} & \text{[object]} \\
\\
e \in \mathit{Exp} & = & v & \text{[value]} \\
& \mid & x & \text{[variable]} \\
& \mid & e \;\; e & \text{[call]} \\
& \mid & e = e & \text{[assignment]} \\
& \mid & \texttt{let}\;\; x = e \;\;\texttt{in}\;\; e & \text{[binding]} \\
& \mid & \texttt{if}\;\; e \;\;\texttt{then}\;\; e \;\;\texttt{else}\;\; e & \text{[if-then-else]} \\
& \mid & e.f & \text{[field load]} \\
& \mid & e.f = e & \text{[field store]} \\
& \mid & \texttt{ref}\, e & \text{[address of]} \\
& \mid & \texttt{deref}\, e & \text{[value at]} \\
& \mid & \odot\, e & \text{[unary]} \\
& \mid & e \oplus e & \text{[binary]} \\
\\
\odot & = & \text{is a set of unary operators.} \\
\oplus & = & \text{is a set of binary operators.} \\
x \in \mathit{Var} & = & \text{is a finite set of variable names.} \\
f \in \mathit{Fld} & = & \text{is a finite set of field names.} \\
a \in \mathit{Addr} & = & \text{is an infinite set of memory addresses.} \\
\lambda \in \mathit{Lam} & = & \text{is the set of lambda expressions.}
\end{array}
$$

Figure 2.1: The syntax of the base language λ_{base}. The syntax is derived from three syntactic categories: The constants *Cst* are booleans, numbers, strings and the special `null` value. The values *Val* are the constants together with addresses, lambda expressions and objects. The expressions *Exp* are variables, values, function calls, local assignments, conditional branches, field reads and writes, heap references and dereferences, unary and binary operations.

Expressions. We briefly explain the semantics of each expression in λ_{base}:

- **Values:** A value expression v embeds a value into an expression. A value expression is fully evaluated.
- **Variables:** A variable expression x is a symbolic for a value which has not yet been substituted.
- **Assignments:** An assignment expression $e_1 = e_2$ evaluates the left-hand-side e_1 to an address, evaluates the right-hand-side e_2 to a value and modifies the heap such that the address points to the value.
- **Let:** A let expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ evaluates the expression e_1 to a value v and substitutes all free occurrences of x in e_2 with v.
- **Call:** A call expression $e_1\ \ e_2$ evaluates e_1 to a lambda expression $\lambda\, x.\, e_3$ and the argument e_2 to a value v. Then replaces all free occurrences of x in e_3 with v.
- **If-then-else:** An if-then-else expression $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ evaluates e_1 to a boolean value v. If v is `true` then evaluation proceeds for e_2, otherwise for e_3.
- **Field Read/Load:** A field read (also called a load) expression $e.f$ evaluates e to an object and extracts the field with name f. The term is stuck if the field does not exist on the object.
- **Field Write/Store:** A field write (also called a store) expression $e_1.f = e_2$ evaluates e_1 to an object o, e_2 to a value and returns the object o updated with the field f mapping to the value v.
- **Heap Reference:** A heap reference expression $\texttt{ref}\ e$ evaluates e to a value v, allocates a new memory address a, mutates the heap s.t. a points to v and returns a.
- **Heap Dereference:** A heap dereference expression $\texttt{deref}\ e$ evaluates e to an address a and returns the value of a in the heap. The term is stuck if the address does exist in the heap.
- **Unary and Binary Operations:** A unary expression $\odot$ evaluates e to a value v and applies the operator $\odot$ to v. Similarly, a binary expression $e_1 \oplus e_2$ evaluates e_1 and e_2 to values v_1 and v_2 and applies the binary operator $\oplus$ to the values v_1 and v_2.

$$h \in \mathit{Heaps} \;=\; \mathit{Addr} \hookrightarrow \mathit{Val}$$
$$\langle h, e \rangle \in \mathit{States} \;=\; \mathit{Heap} \times \mathit{Exp}$$

Figure 2.2: The abstract machine of λ_{base}.

The λ_{base} language is large enough to contain significant challenges, yet the grammar is small enough to fit on one page. As stated before, the language is based on the work of Guha et al. [2010], and their paper shows how the entire JavaScript language can be compiled into λ_{base}.

Semantics

We have informally described the semantics of λ_{base}. A more precise description is given by a small-step operational semantics of the language λ_{base} together with an abstract machine. We will give some examples of the semantics, but otherwise refer to the reader to [Pierce, 2002; Guha et al., 2010; Harper, 2012] for more details.

Abstract Machine. We formalize the semantics of λ_{base} on the abstract machine shown in Figure 2.2. A configuration of the machine has a heap component σ and an expression e component. The transition relation $\hookrightarrow$, defined as a set of inference rules, describes how the machine moves from one state to another. The initial state is the program e combined with the empty heap.

If, through a series of transitions, the machine ends up in a configuration where the expression component has been reduced to a value then the machine has successfully halted. Otherwise the machine is stuck. Not all expressions reduce to a value. For example, the omega combinator applied to itself diverges:

$$(\lambda\, x.\, x\, x)\, (\lambda\, x.\, x\, x)$$

We give a few examples of how to specify the transition relation:

Example: If-then-else. We can specify the semantics of the well-known if-then-else expression using two rules: E-IF-TRUE and E-IF-FALSE. The E-IF-TRUE, shown below, specifies that if we have a configuration with some heap σ and the expression `if true then` e_1 `else` e_2 then we can make a transition to a new configuration with the same heap σ and the e_1 expression. The E-IF-FALSE is defined in analogous way.

We use *evaluation contexts* to specify when evaluation is allowed to proceed inside an expression [Felleisen et al., 2009]. Informally, an evaluation context specifies a "hole" in the grammar where evaluation may proceed.

$$\frac{}{\langle \sigma, \texttt{if true then } e_1 \texttt{ else } e_2 \rangle \hookrightarrow \langle \sigma, e_1 \rangle} \quad \text{[E-IF-TRUE]}$$

$$\frac{}{\langle \sigma, \texttt{if false then } e_1 \texttt{ else } e_2 \rangle \hookrightarrow \langle \sigma, e_2 \rangle} \quad \text{[E-IF-FALSE]}$$

Figure 2.3: Evaluation rules for if-then-else expression.

In the case of `if-then-else` expressions the evaluation contexts is:

$$E = \texttt{if } E \texttt{ then } e \texttt{ else } e$$

The recursion bottoms out with the special □ hole term (assumed to be part of the grammar for E). The above evaluation context specifies that when we encounter an `if-then-else` expression we are allowed to open up its conditional, reduce it to a value and put the expression back together with the reduced value. Thus, to evaluate an `if-then-else` expression we first fully evaluate its conditional and then apply either E-IF-TRUE or E-IF-FALSE.

Example: References. We can specify the semantics of the reference and dereference expressions using two rules: E-REF and E-DEREF together with two evaluation contexts. The E-REF rule specifies that to evaluate `ref` v we must allocate a fresh address a, i.e. an address which is not already in the heap, update the heap such that a points to v and return the newly allocated address a. The E-DEREF rule is similar, but instead retrieves the value from the heap. If the address is not allocated in the heap then the term is stuck.

$$\frac{a \notin \text{dom}(\sigma)}{\langle \sigma, \texttt{ref } v \rangle \hookrightarrow \langle \sigma[a \mapsto v], a \rangle} \quad \text{[E-REF]} \qquad \frac{\sigma(a) = v}{\langle \sigma, \texttt{deref } a \rangle \hookrightarrow \langle \sigma, v \rangle} \quad \text{[E-DEREF]}$$

Figure 2.4: Evaluation rules for address-of and value-at.

The evaluation contexts:

$$E = \texttt{ref } E \mid \texttt{deref } E$$

enable reduction to continue under the `ref` and `deref` operations.

Evaluation Contexts. We need to connect the evaluation contexts to the semantic rules. To this, we introduce the reflexive and transitive closure of $\hookrightarrow$. The transitive closure $\hookrightarrow^\star$ captures that if $\langle \sigma_1, e_1 \rangle$ may reduce to $\langle \sigma_n, e_n \rangle$

through some steps $\langle \sigma_1, e_1 \rangle \hookrightarrow \cdots \langle \sigma_i, e_i \rangle \cdots \hookrightarrow \langle \sigma_n, e_n \rangle$ then $\langle \sigma_1, e_1 \rangle \hookrightarrow^\star \langle \sigma_n, e_n \rangle$. The $\hookrightarrow^\star$ relation is also referred to as the "big-step" evaluation relation [Nielson and Nielson, 1992].

$$\frac{\langle \sigma, e \rangle \hookrightarrow^\star \langle \sigma', e' \rangle}{\langle \sigma, E[e] \rangle \hookrightarrow \langle \sigma', E[e'] \rangle} \qquad \text{[E-CONTEXT]}$$

Figure 2.5: The evaluation context rule.

The evaluation context rule E-CONTEXT specifies that if $\langle \sigma, e \rangle$ can reduce to $\langle \sigma', e' \rangle$ through a number of steps then we implicitly have a rule which allows us to take an evaluation context and substitute e for e', where there is a hole in the context, while preserving the new heap.
To put it more concretely, if we have the configuration:

$$\langle \sigma, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \rangle$$

and $\langle \sigma, e_1 \rangle$ can reduce to $\langle \sigma', e_1' \rangle$ then there is a transition to:

$$\langle \sigma, \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3 \rangle$$

by plugging the hole in the evaluation context for the if-then-else with e_1'.

3

Static Analysis

"Testing can show the presence of errors, but not their absence."

— Edsger W. Dijkstra

We can ask many interesting questions about programs: Does a given program terminate? Does a given program contain bugs? Does a given program contain security vulnerabilities? Does a given program satisfy its specification? Unfortunately, most real-world programming languages are Turing-complete, and as a consequence any non-trivial property is undecidable [Martin, 1991].

All these questions are important, but let us, for the moment, focus on the question of whether a given program contains bugs. Testing has been proposed as a technique to detect and prevent bugs. Testing is easy: Simply exercise the program, either manually or automatically, and observe if any bugs are encountered. However, as illustrated by the quote, testing can only show the presence of bugs, never their absence. That is, we can test a program as much as we want, but this does not *prove* that the program is bug-free.

A *static analysis* is an automatic technique which aims to *prove* specific properties of programs. A static analysis *does not* solve undecidable problems, but carefully sidesteps them. The key insight is to approximate programs into a system where the property of interest is decideable. An approximation is either *over-approximate* or *under-approximate*. An over-approximate analysis aims to cover all possible executions of the program, and is said to be *sound* if when the approximation is bug-free, then the program is bug-free. On the other hand, an under-approximate analysis is said to be *complete* if when the approximation contains a bug, then the program actually contains that bug. The specific choice of approximation is frequently called the analysis abstraction. This leads us to the research challenge of static analysis:

> **Research Challenge**: The design of analysis abstractions which are useful for proving the properties of interest while balancing the trade-offs between soundness, precision and performance.

Precision is a measure of how closely the approximation matches the actual behaviour of the program. A precise analysis mimicks the program and contains little *spurious flow*, i.e. artifacts of the approximation which cannot actually occur in the program. Spurious flow causes spurious warnings and must be kept minimal for the analysis to remain useful to the programmer, who must manually inspect the warnings. A driving goal of static analysis is the design of clever approximations to reduce spurious warnings. The difficult trade-off between soundness, precision and performance is discussed later.

We will not give an introduction to the details of static analysis, but instead refer the reader to the literature and the book *Principles of Program Analysis* [Nielson et al., 1999]. Instead, we focus on key terminology used in the thesis and discuss some meta issues of static analysis.

3.1 Abstractions

A static analysis maps the program from the concrete world into the abstract world. The concrete world, i.e. the concrete run-time and semantics, permits infinitely many execution paths. In contrast, the abstract run-time and semantics must be finite to ensure termination of the static analysis.

For example, the run-time of λ_{base}, shown in Figure 2.2, permits an infinitely large heap and as a consequence an infinite state space. A static analysis must approximate, i.e. project, this infinite space down into a finite space. Hence the pigeon hole principle comes into play:

> **The Pigeon Hole Principle:** If there are n pigeon holes, but there are $n + 1$ pigeons, then two pigeons must share the same hole.

In other words, it is a *mathematical necessity* that when the concrete heap is abstracted two (or more) concrete memory addresses must *share* the same value. The immediate question is then: How should we combine concrete values? The solution is to introduce lattice elements, i.e. a set of *abstract values*, which can represent multiple concrete values. For example, we can use the idea of *odd* and *even* numbers to represent all natural numbers, i.e. *even* represents the infinite set $2, 4, 6, \ldots$.

The abstract interpretation framework provides a formal technique to obtain an abstract run-time and semantics from a given lattice together with the concrete run-time and semantics [Cousot and Cousot, 1977]. Although static analysis rest on the foundations of the abstract interpretation framework, in this thesis we take an informal approach and focus more on the ideas of the abstractions, instead of their rigorous formalization. Not to say that such work is without merit!

3.2 Dimensions of Static Analysis

We briefly discuss some relevant dimensions of static analysis:

We are primarily interested in dataflow analyses. A dataflow analysis tracks the flow of primitive values through the program. A dataflow analysis be flow-sensitive or flow-insensitive, a flow-sensitive analysis respects the order of statements, i.e. the control-flow of the program, whereas a flow-insensitive analysis does not consider the order between statements [Aho, 2003].

We extend the dataflow analysis with points-to information to approximate the heap. A points-to analysis determines which addresses in the heap every expression can refer to, and the points-to relationships inside the heap. Two common points-to analyses are the equality-based vs. the subset-based. The equality-based points-to analysis, sometimes refered to as Steengaard's analysis, considers all assignments to be bi-directional and is very fast [Steensgaard, 1996]. The subset-based points to analysis, sometimes refered to as Anderson's analysis, is based on subset constraint and is much more precise than equality-based points-to analysis, but also much slower [Andersen, 1994]. We are primarily interested in subset-based points-to analysis.

We are interested in whole-program analyses which are inter-procedural, i.e. which track dataflow, including points-to information, across procedure boundaries. This is in contrast to intra-procedural analyses, which consider each procedure in isolation [Pnueli, 1981].

The interest in these heavy-weight static analysis techniques, e.g. flow-sensitivity, subset-based points-to analysis and inter-procedural analysis, stems from the challenging environment of JavaScript.

3.3 Trade-offs in Static Analysis

We are usually interested in three parameters when evaluating or comparing static analyses: soundness, precision and performance. Other important factors are the correctness of the implementation, its software architecture, the choice of implementation languages and so on.

Soundness. The *soundness* of a static analysis refers to whether the analysis is able to capture all possible behaviour of the program under analysis. If an analysis is sound, and reports that some behaviour is impossible, then truly that behaviour must be impossible. However, a sound analysis may report that some behaviour is possible, although it is in fact impossible. Thus, for the purposes of safety, a sound analysis can be counted on: If the analysis says the program is bug-free then it truly is bug-free. If, on the other hand, the analysis reports a bug then there is potentially a bug.

Precision. The *precision* of a static analysis refers to its ability to accurately capture the behaviour of the program. A very precise analysis will closely mimick the execution of the concrete program, whereas a less precise analysis may include a lot of spurious behaviour which is not possible in the program. Spurious behaviour leads to spurious warnings produced by the analysis. A spurious warning occurs when the analysis believes the program can encounter an error, but that error is in fact impossible, and simply an artifact of the analysis.

Performance. The *performance* of an analysis is how quickly the analysis is able to reach the fixpoint. In algorithmics it is common to describe the time and memory usage of algorithms using asymptotic bounds. However, in the field of static analysis, these bounds turn out to be rarely useful. The reason is that programs are, at least usually, written by humans, and humans program in a certain style. For example, although statements in most languages may be nested arbitrarily deep, we should not expect the nesting depth to increase as the program length increases: A program may be a million lines of code, but it will never have a thousand nested if-then-else statements. In fact, it seems more reasonable to assume that the nesting depth – although in principle unbounded – is in fact a small constant. Another example, assume a points-to graph for n objects. If the transitive closure of the graph is fully connected then it has $\mathcal{O}(n^2)$ edges and takes $\mathcal{O}(n^3)$ time to compute in a straightforward way. Yet, points-to analyses scale to million of lines of code. How can this be? The answer is that, for carefully engineered analyses, the worst-case is never encountered for real programs.

Discussion. The research challenge in the design and implementation of any static analysis is to achieve soundness and a usable trade-off between precision and performance. Soundness is often an engineering issue: Making sure that all corner cases have been handled correctly. This usually involves a lot of work, but is theoretically possible. Although recently, several researchers have proposed a notion of *soundy* [Livshits et al., 2015]. A soundy analysis is unsound, but for all practical purposes sound, and could be made fully sound with sufficient engineering and legwork. Precision and performance, perhaps surprisingly, has no close relationship. In fact there are two opposing forces influencing the precision and performance of an analysis:

- An increase in precision means that the analysis has to track more information and to apply more expensive operations to that information slowing the analysis down. On the other hand,

- An increase in precision means that the analysis may have less spurious information and thus *reduce* the amount work.

In some cases the increase in precision is sufficient to offset its cost. In other cases it is not. Thus the trade-off between precision and performance must be studied empirically.

4

Asynchronous Events

"Anything that can possibly go wrong, will go wrong."

— Murphy's Law

Asynchronous events is a programming language mechanism which enable interactive and reactive applications. Events are external occurences which the program react to, one by one, as they arrive. Events may be caused by the user (e.g. when the user clicks the mouse, presses a key on the keyboard, etc.) or by the system (e.g. when a disk read completes, when a new socket connection is established, etc.).

In an event-based system the control-flow of the program is not immediately available from the program syntax. Events may cause the program switch between *event listeners* at pre-determined points in the execution. In this chapter we focus on event-based programs written in the popular NodeJS framework [NodeJS, 2015].

NodeJS is framework and run-time for running JavaScript applications on the server-side. NodeJS provides file system and network APIs which are not available in plain JavaScript. NodeJS is event-based and most APIs require the programmer to use event listeners. For example, to read a file the programmer registers an event listener which is executed once the file has been read into memory.

We want to capture the essence of asynchronous programming in NodeJS with an extension of our $\lambda_{\texttt{base}}$ calculus. To do so, we make several design decision motivated by the implementation of NodeJS:

- Execution is single-threaded and non-preemptive, i.e. there is no parallelism and an event listener cannot be interrupted. Specifically, every event listener must run to completion before another event listener is scheduled for execution.

- Scheduling of event listeners is non-deterministic: If multiple event listeners are pending execution, we cannot know which one will be scheduled first.
- Event listeners are associated with objects and event names. An event listener is scheduled for execution when the event is emitted on its object. Multiple event listeners may be registered for the same event and on same object. If so, the event listeners are scheduled in registration order.
- If an event listener registers another event listener, for the current event, that listener will not be executed until the event is emitted again.
- The event names are drawn from a statically known finite set. In NodeJS event names are strings, but for simplicitely we assume that the event names are statically known.

We summarize the research challenge of event-based systems as:

> **Research Challenge:** The design of static analysis techniques which can precisely track the control– and data-flow in the presence of indirect calls due to events and event listeners.

4.1 Semantics

We introduce three new terms into the base language λ_{base} to obtain an event-based language. The new language is named λ_{event}:

- **Listen:** $e_1.\texttt{listen}(\tau, e_2)$: An expression which registers an event listener for the event name τ on an object. The expression e_1 is the receiver object on which the registration occurs and the expression e_2 is the event listener (which must evaluate to a lambda). Preserves the registration order of previously registered listeners.
- **Emit:** $e_1.\texttt{emit}(\tau)$: Emits an event on an object which causes all listeners registered on that object and for that event to be scheduled for execution. If multiple listeners are present for the event then they are scheduled in registration order. The expression e_1 is the address of the object, τ is the event name and e_2 is the argument passed to the event listener(s).
- **Loop:** $\bullet$: An expression which represents the event loop, i.e. the situation when the call stack is empty and scheduled event listeners are allowed to execute.

Figure 4.1 shows the extended grammar.

$$
\begin{array}{rcll}
e \in \mathit{Exp} & = & \bullet & \text{[no-op]} \\
& | & e.\texttt{listen}\,(\tau, e) & \text{[attach listener]} \\
& | & e.\texttt{emit}\,(\tau, e) & \text{[emit event]} \\
\\
\tau \in \mathit{Event} & = & \text{is a finite set of event names.} &
\end{array}
$$

Figure 4.1: Syntax of λ_{event}.

Runtime. We augment the runtime of the base language λ_{base} by adding two new components. The first component is a map to track the registered listeners on an object. The second component is a queue of event listeners pending execution. The new runtime is shown below:

$$
\begin{array}{rcl}
\sigma \in \mathit{Heaps} & = & \mathit{Addr} \rightarrow \mathit{Val} \\
\vartheta \in \mathit{Listeners} & = & \mathit{Addr} \times \mathit{Event} \rightarrow \mathit{Lam}^{\star} \\
\pi \in \mathit{Queues} & = & \mathit{Addr} \times \mathit{Event} \rightarrow (\mathit{Lam} \times \mathit{Val})^{\star} \\
s \in \mathit{States} & = & \mathit{Heap} \times \mathit{Listener} \times \mathit{Queue} \times \mathit{Exp}
\end{array}
$$

Figure 4.2: Runtime of λ_{event}.

The *listeners* component is a map from addresses and events to a sequence of registered event listeners. The *queues* component is a map from addresses and events to a sequence of pending event listeners together with their argument.

Evaluation Rules. The semantics of λ_{event} is defined by the addition of three new evaluation rules.

$$
\frac{o \in \mathit{Addr} \qquad f = \lambda\,(x \cdots)\; e \qquad \vartheta' = \vartheta[(o, \tau) \mapsto f :: \vartheta(o, \tau)]}{\langle \sigma, \vartheta, \pi, o.\texttt{listen}(\tau, f) \rangle \hookrightarrow \langle \sigma, \vartheta', \pi, \texttt{null} \rangle} \quad \text{[E-LISTEN]}
$$

The E-LISTEN rule specifies that a listen expression evaluates to the `null` value and updates the map of registered listeners by pre-pending the event listener.

$$
\frac{o \in \mathit{Addr} \qquad \vartheta(o, \tau) = \lambda_1 :: \cdots :: \lambda_n \qquad \pi' = \pi[o \mapsto (\lambda_1, v) :: \cdots :: (\lambda_n, v) :: \pi(v)]}{\langle \sigma, \vartheta, \pi, o.\text{emit}(\tau, v) \rangle \hookrightarrow \langle \sigma, \vartheta, \pi', \texttt{null} \rangle} \quad \text{[E-EMIT]}
$$

The E-LISTEN rules specifies that an emit expression evaluates to `null` and schedules the event listeners registered for the emitted event.

$$\frac{o \in \mathit{Addr} \qquad \pi(o,\tau) = (\lambda_1, v_1) :: \cdots :: (\lambda_n, v_n) \qquad e = \lambda_n(v_n); \ldots; \lambda_1(v_1) \qquad \pi' = \pi[(o,\tau) \mapsto \mathit{Nil}]}{\langle \sigma, \vartheta, \pi, \bullet \rangle \hookrightarrow \langle \sigma, \vartheta, \pi', e; \bullet \rangle} \quad \text{[E-LOOP]}$$

The E-LOOP rules specifies that when the call stack is empty, as indicated by the • term, the pending event listeners, for a non-determistic event and object, are scheduled for execution by extracting them from the queue and sequencing them as the current expression.

Evaluation Contexts. We add two evaluation contexts for the `listen` expression and one for the `emit` expression. The contexts for the `listen` expression allow us to evaluate the receiver object expression and then the event listener expression, before evaluating the listener expression itself. The contexts for the `emit` expression are similar.

$$E = E.\texttt{listen}\,(\tau, e) \mid v.\texttt{listen}\,(\tau, E) \mid E.\texttt{emit}\,(\tau, e) \mid v.\texttt{emit}\,(\tau, E)$$

4.2 Event-related Bugs

We briefly remark on some of the possible bugs related to event-based systems:

- **Dead Emits**: A *dead emit* occurs when an emit expression does not cause any event listener to be scheduled. A typical cause of this bug is emitting the wrong event or emitting it on the wrong object. The NodeJS API makes these mistakes more common by using similar sounding event names, e.g. `connect` vs. `connection`[1].
- **Dead Listeners**: A *dead listener* occurs when an event listener is registered on an object for some event, but that event is never emitted after the registration. As before, the event listener may be registered on the wrong object or for the wrong event.

  A program may have a dead listener independently of a dead emit.
- **Data Race**: A *data race* occurs when two event listeners communicate via shared state. For example, one event listener may write a piece of global state, which is then read by the second event listener. If the write is required to happen before the read can succeed then there must be a may-happen-before relationship between the write and read, and no other possible relationships.

[1] http://nodejs.org/api/net.html

4.3 Static Analysis

We wish to design a static analysis which can reason about events and event listeners to detect bugs like the ones discussed in the previous section.

Abstract Runtime

The abstract runtime is similar to the one described in the previous chapter. The new design choices involve how to abstract the listeners and event queue. A straightforward choice, motivated in more detail in the accompanying paper, is to abstract the list of event listeners with a set of listeners, and similarly for the event queue. This abstraction loses the individual ordering between listeners registered for the *same* event and on the *same* object, but can still preserve the order between different events or listeners based on the choice of context policy as described next.

Context Policies

We describe three different policies for separating dataflow based on the control-flow due to event listeners. These policies are similar to context sensitivity, or more precisely, trace partitioning [Mauborgne and Rival, 2005].

Baseline Analysis. The *baseline* analysis merges all dataflow in the event loop. After execution of an event listener, its abstract state is merged into the abstract state immediately before the event loop. Thus, the baseline analysis models all possible interleavings of event listener executions. As a consequence, the benefits of flow sensitive are lost, as the baseline analysis cannot track any correlation between event listeners.

Event Sensitive Analysis. The *event sensitive* analysis separates dataflow based on which events have been emitted. Specifically, two abstract states are only merged, in the event loop, if they have seen the same set of emitted events. The event sensitive analysis is precise, but expensive since the number of contexts is exponential in the number of emitted events.

Listener Sensitive Analysis. The *listener sensitive* analysis separates dataflow based which event listeners are registered. Specifically, two abstract states are only merged, in the event loop, if they share the same set of registered listeners. The strength of the listener sensitive analysis is that it can track dependecies between listeners which register other listeners.

Discussion. The event and listener sensitive analysis are *incomparable* in precision and performance. There exists a program where the event sensitive analysis is strictly more precise than the listener sensitive analysis, and vice versa. The two analysis can be combined in a natural way to obtain an event *and* listener sensitive analysis at the cost of performance. Which analysis is the better suited depends on the specific problem of interest.

Summary. We have presented a framework for static analysis of event-based JavaScript applications. We have described three different context sensitive policies: baseline, event– and listener sensitive. The baseline analysis ignores any dependencies between event listener registrations and assumes that events can happen in any order. The event sensitive analysis separates dataflow based on previously emitted events. The listener sensitive analysis separates dataflow based on the set of registered listeners. Experimental results show that the event– and listener sensitive analyses are more precise than the baseline analysis and necessary to avoid many spurious warnings. Furthermore, the listener sensitive analysis is significantly faster than the event sensitive analysis, while maintaing comparable precision.

5

Dynamic Fields

"Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy."

— Alan Perlis

In object-oriented programming languages, with statically-checked types, every object is a member of a specific class. The class is a template which describes the fields and methods available on every object of that class. Importantly, the declared fields and methods are *static*, i.e. they do not change during the lifetime of the object: A field cannot be added nor removed. Only the value of a field may change. For example, if a class `Animal` has a method `eat()` then all `Animal` objects have an `eat()` method throughout their lifetimes.

In many dynamic programming languages, which lack statically-checked types, classes are changeable at run-time. For example, Ruby and PHP allows fields to be added at run-time. Other dynamic languages, such as JavaScript and Lua, completely lack classes: Every object is simply a map from keys to values. This "objects as dictionaries"-style of programming allow field names of objects, which are typically strings, to be computed at run-time [ECMA, 2015; Lua, 2015].

As a consequence, the shape of objects are no longer known statically. Instead, a static analysis which wishes to track dataflow through objects must perform string analysis to approximate the fields and field accesses of objects. This leads us to the research challenge:

> **Research Challenge:** The design of abstract string domains to reason precisely about strings used in dynamic field accesses.

There are at least two issues to consider: (a) What abstraction is useful for the objects themselves, and (b) what abstraction is useful for strings used in dynamic field accesses. In this work, we focus on the latter, and use a simple object abstraction for the former.

5.1 Semantics

We begin by extending the base language $\lambda_{\texttt{base}}$ with dynamic field accesses. The language is extended with two new expressions: a dynamic field load and a dynamic field store. The new language is called $\lambda_{\texttt{dyn}}$.

Dynamic Load. A dynamic field load is an expression of the form $e_1[e_2]$ where e_1 is the *base* or *receiver* object and e_2 is the *field expression*. Intuitively, the semantics of a dynamic field load evaluates the base expression to an object o, evaluates the field expression to a field (string) f and returns the value of $o[f]$. For simplicity, we let the term be stuck if the field does not exist on the object.

Dynamic Store. A dynamic field store is an expression of the form $e_1[e_2] = e_3$ where e_1 is the *base* or *receiver* object, e_2 is the *field expression* and e_3 is the *value expression*. Intuitively, the semantics of a dynamic field store is to evaluate the base expression e_1 to an object o, the field expression e_2 to a field (i.e. a string value) f, the value expression e_3 to a value v and then return an updated object with the new mapping $o[f \mapsto v]$, replacing any existing mapping.

Evaluation Rules. The E-LOAD and E-STORE rules capture the new semantics. They are similar to the original E-LOAD and E-STORE, except that they explicitly require the field name to be a string [Lua, 2015]. Otherwise the expression is stuck. In some languages (e.g. Lua) any value is allowed as a key, whereas in other languages (e.g. JavaScript) all keys must be strings, and non-string keys are automatically coerced to strings [ECMA, 2015].

$$\frac{o = \{f : v \cdots\} \quad f \in \mathit{str}}{\langle \sigma, o[f] \rangle \hookrightarrow \langle \sigma, v \rangle}\ [\text{E-LOAD}] \qquad \frac{f \in \mathit{str}}{\langle \sigma, o[f] = v \rangle \hookrightarrow \langle \sigma, o[f \mapsto v] \rangle}\ [\text{E-STORE}]$$

Evaluation Contexts. We introduce two new evaluation contexts for dynamic field load expressions:

$$E[e] \quad \text{and} \quad E = v[E]$$

The contexts show that we must evaluate the base object to a value before evaluating the field expression. For the dynamic field store expressions, we introduce three new contexts:

$$E[e] = e \quad \text{and} \quad v[E] = e \quad \text{and} \quad v[v] = E$$

The contexts show that we must evaluate the base object first, then the field expression and finally the value expression.

Example. The small program fragment below, which consists of an object literal, a dynamic field access, and an if-then-else expression:

$$\{x : 21, y : 42, z : 84\}\,[\texttt{if } c \texttt{ then "x" else "y"}]$$

evaluates to 21 if c is `true`, and to 42 is c is `false`. If the condition c is indeterminate to a static analysis then the analysis may conclude that either `"x"` or `"y"` is read from the object. However, depending on the string abstraction, the join of `"x"` or `"y"` may lose all information and conclude that any field could be read. If so, the analysis has to read `"z"` too. The primary purpose of this work is to come up with string domains which reduce the amount of spuriously read and written fields.

Correlation Tracking. A related problem is *correlated field access* which occurs when dynamic field access is combined with the ability to iterate over the field names of objects [Sridharan et al., 2012]. The program fragment below illustrates how all fields of the source object is copied to the target object:

```
function copy(src, dst) {
    for (var f in src) {
        dst[f] = src[f];
    }
}
```

In JavaScript, the above programming pattern occur frequently. The jQuery library is notoriously known for this [Andreasen and Møller, 2014]. The issue here, for a static analysis, is not the strings being used in the dynamic read and write, but their dependency. If the above code fragment is analyzed naïvely by a flow-sensitive points-to analysis it will lose information about the variable f and conclude that the value of any field on the destination object could point to the value of any field of the source object. Correlation tracking is a proposed static analysis technique which explicitly keeps tracks of correlated dynamic reads and writes to preserve precision in the situations such as the above [Sridharan et al., 2012].

5.2 Static Analysis

We aim to accurately track the fields read and written by dynamic field accesses as illustrated by the first example. To achieve this, we require two things: First, a new abstraction of objects, which can handle the scenario when we do not know what field is being written to. Second, a new string abstraction which is able to reduce the number of spuriously read and written fields.

Object Abstraction

In the base analysis the field names were statically known and abstract objects were simply a map from field names to abstract values. In the extended language this is no longer possible: The set of field names is no longer guaranteed to be finite.

We adopt a new abstraction: Every abstract object is a pair of two components $\langle m, d \rangle$. The m component is, as before, a map from field names to abstract values. The d component is a *default* abstract value which represents the value of all field names not in the domain of m. For example, if the abstract object is $\langle \{a : 21, b : 42\}, \texttt{true} \rangle$ then the concrete value of the field a is 21 whereas the concrete value of the field c would be `true` (since c is not in the domain of m).

We introduce two helper functions for reading and writing fields of abstract objects: `Read` and `Write`. The abstract semantics uses these functions whenever a dynamic field access occurs:

Reading a Field. The `Read` function takes an abstract object $\langle m, d \rangle$ and an abstract field name f, and returns an abstract value:

$$\texttt{Read}(\langle m, d \rangle, f) = \begin{cases} m(f) & \text{if } f \text{ is exact and } f \in \text{dom}(m) \\ \texttt{match}(m, f) \sqcup d & \text{otherwise} \end{cases}$$

Intuitively, the first case corresponds to the concrete semantics when the field name is known and the field is present on the object. The second case uses the helper function `match` to read all fields on m which may equal f and joins the result with the default field f. The `match` is defined as the least upper bound on the values of all the field names which maybe equal the given field name:

$$\texttt{match}(m, f) = \bigsqcup \{m(k) \mid k \in \text{dom}(m) \text{ and } m \mathrel{\widehat{=}} f\}$$

where $m \mathrel{\widehat{=}} f$ holds if m may equal f.

Writing a Field. The `Write` function is conceptually similar to the `Read` function. `Write` takes an abstract object $\langle m, d \rangle$, an abstract field name f, the value to write v and returns a new abstract object where the matching fields have been updated:

$$\texttt{Write}(\langle m, d \rangle, f, v) = \begin{cases} \langle m[f \mapsto v], d \rangle & \text{if } f \text{ is exact} \\ \langle \texttt{update}(m, f, v), d \sqcup v \rangle & \text{otherwise} \end{cases}$$

The first case, as before, corresponds to the concrete semantics when the exact field name is known. The second case uses the helper function `update` to

update all matching fields with the value:

$$\texttt{update}(m, f, v) = \lambda k. \begin{cases} m(k) & \text{if } k \mathrel{\widehat{\neq}} f \\ m(k) \sqcup v & \text{otherwise} \end{cases}$$

The `update` function returns a new map (as a lambda term, but possible to implement as a regular map) where fields which match the given field name f are updated to contain the join of the old and new value.

As these definitions highlight, the precision of reads and writes is influenced by the precision of abstract equality $\widehat{=}$.

String Analysis

The challenges posed by dyanmic field accesses where first identified in relation to points-to analysis JavaScript: The GATEKEEPER tool, the first points-to analysis for JavaScript, completely ignored dynamic field accesses and resorted to run-time checks to ensure the soundness of the technique [Guarnieri and Livshits, 2009]. The Type Analysis for JavaScript (TAJS) tool, a whole-program dataflow analysis for JavaScript, split the default value described into *two* default values; one for numerical and one for non-numerical fields [Jensen et al., 2009]. As mentioned previously, correlation tracking was introduced to specifically handle the programming pattern where fields are copied from one object to another [Sridharan et al., 2012]. The JSAI tool, a whole-program dataflow analysis for JavaScript similiar to TAJS, has hardcoded extra precision for certain field names (e.g. `toString`) [Kashyap et al., 2014].

We are interested in light-weight string abstractions which are useful for dynamic field access in points-to analysis. For this purpose, we identify the following design requirements:

- The important string operations are equality and concatenation, together with operations for computing the partial order and least upper bound.
- The abstract string domain should be immediately usable in off-the-shelf points-to analysis.
- The abstract string domain should be efficient and compact; ideally requiring only $\mathcal{O}(1)$ space and time.
- Precision should degrade gracefully, and not abruptly.

String Domains

Based on the design criteria above, we experimented with twelve different abstract string domains, including five previously known and seven new. We briefly discuss a few of the more interesting ones.

The Constant Propagation Domain. The constant propagation domain is the commonly used string abstraction in static analyses of JavaScript [Jensen et al., 2009; Kashyap et al., 2014]. The constant propagation lattice tracks a single concrete string. The great weakness of simple constant propagation, in case of dynamic field access, is that its precision does not degrade gracefully: The least upper bound of two different strings is the top element, which represents all possible concrete strings.

The Prefix– & Suffix Character Domain. The idea behind the prefix– & suffix character domain is to keep track of the *first* and *last* character symbol of the string. For example, for the string `"abc"` the lattice will track the prefix character `"a"` and the suffix character `"c"`. If the two concrete strings `"abc"` and `"aabb"` are joined the lattice will still track the prefix character `"a"`, but have no information about the suffix character. Information about prefix and suffix is easy to maintain under concatenation or when computing the least upper bound.

The Character Inclusion Domain. The idea behind the character inclusion lattice is to keep track of which characters *may* or *must* occur inside the string without tracking their positions. For example, the two strings `"abc"` and `"cba"` both contain the same characters and would be represented by the same lattice element. On the other hand, the two strings s_1 = `"aabb"` and s_2 = `"abcd"` would be represented as: $s_1^{\text{may}} = s_1^{\text{must}} = \{a, b\}$ and $s_2^{\text{may}} = s_2^{\text{must}} = \{a, b, c, d\}$ and their join would be:

$$s_j^{\text{may}} = \{a, b, c, d\} \qquad s_j^{\text{must}} = \{a, b\}$$

The strength of the character inclusion domain is that its precision degrades gracefully. Since the character inclusion domain does not maintain any positional information, concatenation is straightforward to implement: The may and must sets are simply joined.

The String Hash Domain. The idea behind the string hash domain is to separate different strings based on *hashing*. The string hash domain is parameterized by a bucket size k and a hash function. An element of the string hash lattice is a set of hash values. For example, if we fix $k = 64$ and choose the simple hash function $h(s) = \bar{s} \mod k$, where $\bar{s}$ is the sum of all character symbols in s, then the string `"a"` would be 33 and `"m"` would be 27, since their ASCII value are 97 and 155, respectively. The least upper bound of the two strings would then be the set $\{27, 33\}$ encoded as a 64 bit-vector.

Summary. We have discussed several abstract domains for static analysis of strings used in dynamic field accesses. We have focused our attention on domains which are compact, efficient and easy to implement in points-to analyses. Experimental evaluation have shown that a combination of the character inclusion– and string hash domains achieve significantly greater precision than traditionally used constant propagation at cost to analysis performance.

6

Analysis of Libraries

"To understand a program you must become both the machine and the program."

— Alan Perlis

Applications are typically not created from scratch. Software libraries help speed-up development and avoid duplication of commonly used functionality. Unfortunately, libraries, and especially those for JavaScript, often complicate static analysis due to a range of factors:

- A library may be partially or fully implemented in another programming language. For example, web browsers, PhoneGap and NodeJS all provide functionally which is implemented as part of the run-time and for which no JavaScript source code exists [Jensen et al., 2011].
- A library may be vastly larger than the program. In this case it is wasteful to apply the static analysis to the program and library, since the analysis would spend most of its time in the library code, rather than in the program code [Ali and Lhoták, 2012, 2013].
- A library may be significantly more difficult to analyze than the program. For example, many JavaScript libraries are notoriously known for their use of reflection [Schäfer et al., 2013a; Andreasen and Møller, 2014].
- The source code of a library may not be available due to commercial reasons or it may only be available in an obfuscated and minified form [Gopan and Reps, 2007; Balakrishnan et al., 2008]. Obfuscated code is often more difficult for static analyzers [Curtsinger et al., 2011].
- A library may be incrementally loaded, i.e. streamed from the server-side to client-side, meaning that obtaining the whole source code requires the program to be exercised [Guarnieri and Livshits, 2010].

As illustrated above, much research effort has been invested in techniques for static analysis of large libraries. The techniques range from using hand-crafted models [Jensen et al., 2011] to clever heuristics for inferring library behaviour [Ali and Lhoták, 2012, 2013], to full-blown very precise analyses tailored for specific libraries, e.g. the very popular jQuery library [Sridharan et al., 2012; Schäfer et al., 2013a; Andreasen and Møller, 2014].

We develop a light-weight *inference* technique to discover the flow of applications objects through a library. The technique relies on simple interface descriptions of the library (commonly called stubs) and a new technique called *use analysis*. The idea is to ignore the library code completely, which may not be available anyway, and observe how objects are used by the application code to infer missing dataflow:

> **Use Analysis:** A heuristic for recovering missing dataflow facts, due to missing library code, by observing how applications objects are used in the application code.

As said before, use analysis relies on simple interface descriptions, also commonly called *stubs*. A simple stub for a small fragment of the Windows 8 JavaScript API might look like:

```
Windows.Storage.Stream.FileOutputStream = function () {};
Windows.Storage.Stream.FileOutputStream.prototype = {
  close = function () {},
  flushAsync = function () {},
  writeAsync = function () {}
}
```

The stub specifies that there is an object called `FileOutputStream` which is available on the object `Windows.Storage.Stream`. The object has three fields `close`, `flushAsync` and `writeAsync` which are functions. The stub does not specify the implementation, i.e. the behaviour of the functions. The `close` function returns `undefined`, which is the default for JavaScript, and thus its implementation is sufficient. However, both the `flushAsync` and `writeAsync` functions return a `Promise` object, which is an object with the stub:

```
WinJS.Promise = function () {};
WinJS.Promise.prototype = {
  then = function () {},
  done = function () {},
  // ...
}
```

We remark that these interface descriptions are easy to obtain from documentation or from a scrape of the initial JavaScript heap.

We can now succintly state the challenge of use analysis:

> **Research Challenge:** The design of analysis techniques to account for indirect dataflow in libraries, without analysis of the library code or specification of the library behaviour.

In simultaneous work, Karim Ali and Ondřej Lhoták investigated call-graph construction for Java programs without analysis of the large Java class library [Ali and Lhoták, 2012]. The key insight of their work is the *separate compilation assumption* which informally states that the class library was compiled separately from the application code, and as a consequence, the library cannot be dependent on any types of application. Using this observation, the authors were able to design a *sound* call-graph construction algorithm for Java programs, without analysis of the class library. Unfortunately, the separate compilation assumption does not hold for JavaScript.

6.1 Use Analysis

If we run a points-to analysis on the application code and the simple interface descriptions for the library we obtain a partial points-to graph. This graph has "dead flow" due to missing dataflow facts from the empty function bodies in the interface descriptions. The purpose of use analysis is to recover this missing information. We consider three cases where dead flow can occur:

- **Dead Load:** A *dead load* occurs when the points-to set returned by a field read is empty.
- **Dead Argument:** A *dead argument* occurs when the points-to set of an argument to a user function is empty.
- **Dead Return:** A *dead return* occurs when the points-to set returned by a library function is empty.

Example. We illustrate use analysis with a simple example:

```
var e = document.getElementById("canvas");
var ctx = e.getContext("2d");
var area = e.height * e.width;
```

The call to `document.getElementById` is a library call. The object returned by the function is unspecified since it is a stub. However, notice that `e` is used as the base object in three field reads `getContext`, `height` and `width`. Use analysis identifies that the call is dead return, injects a *sentinel* (placeholder) object into the points-to graph, re-runs the analysis, discovers that `getContext`, `height` and `width` are dead loads, and concludes that the `HTMLCanvasElement` is the only object in heap which has the three fields. Notice that this implicitely assumes that the three fields where spelled correctly by the programmer.

Algorithm. Use Analysis works in the following way:

1. An initial points-to graph is constructed. The points-to graph has dead flow due to interface descriptions with empty function bodies.

2. The analysis identifiers the locations where flow has died and injects so-called "sentinel" objects into the points-to graph.

3. The points-to analysis is restarted/resumed with the injected sentinels.

4. Requirements on the sentinels, i.e. dead loads, are collected and each sentinel is unified with a set of *matching objects*. The sentinels are removed.

5. The steps 1–4 are repeated until the fixed point is reached.

Termination is ensured since once a dead load is discover and unified it never becomes a dead load again, and there is only a finite number of field loads in the program.

Unification. In the fourth step the sentinel objects are unified with a set of matching objects. How are these objects chosen? There are several possible strategies. A simple strategy is to observe that if a field f is read from the sentinel then any object which has a f field should be unified with the sentinel. A more precise strategy is to say that if there field loads f_1 and f_2 then the sentinel is only unified with objects which have *both* a f_1 and f_2 field.

Unification for Arrays. Unification for arrays benefit from special consideration. Consider the following program fragment:

```
for (var i = 0; i < persons.length; i++) {
    var fst = persons[i].firstName;
    var lst = persons[i].lastName;
    // ...
}
```

Assume that `persons` is an unknown object returned from the library. The object is used as an array, as evident from the access to `length` and numeric fields, but this does not in itself tell us what the object is. However, by "skipping one level" of unification, we can introduce another sentinel for the i'th array access and observe how this object is used. By doing this, we can discover dead loads for the `firstName` and `lastName` fields, indicating that the objects in the array might be `Person` objects.

Unification for Prototypes. Another situation which is worth special consideration is prototype chains. Consider the following program fragment:

```
function toString(firstName, lastName) {
    return firstName.toString() + " " + lastName.toString();
}
```

If the function `toString` is passed into the library and then any calls sites are hidden from the analysis, i.e. the arguments `firstName` and `lastName` will be dead. If so, unification will kick in, and discover that the `toString` field on both arguments is read. However, all JavaScript objects implicitely have a toString field inherited from `Object[[Proto]]`! In this case regular unification will conclude that `firstName` and `lastName` could point to any object in the entire abstract heap. This makes the points-to graph impractically large. A better solution is to detect this specific situation and perform a different kind of unification: Instead of resolving the points-to set of `firstName` and `lastName` it is sufficient to conclude that the (internal) prototype of these fields is `Object[[Proto]]` this prevents the call graph from exploding while still resolving to the two calls to `toString`.

Unsoundness. Use analysis is *fundamentally* unsound. Use analysis cannot, without the library source code or a formal specification, soundly infer dataflow going into- and out-of the library. The only sound statement is the *useless* statement that *every* object may flow *anywhere*. The reason is two fold: First, the library may access any user object by reflection on the global object and cause that object to flow anywhere. Second, use analysis implicitly assumes that fields read from an object should be present, but this assumption may not always hold. As a consequence we cannot provide an elegant statement like the *separate compilation assumption* [Ali and Lhoták, 2012]. Instead, we can offer a few observations about well-behaved libraries:

- A library should not add or remove fields from user objects.
- A library should not mutate user objects, e.g. by overwriting fields.
- A library may add fields to the global object.
- User and library objects should have a static set of fields.

Unsoundness does not make the analysis useless. There are practical uses do not require soundness, e.g. auto-completion and code navigation. Furthermore, a manual study of unsoundness found that only 4 out of 200 call sites were unsound, i.e. had too missing call targets.

Summary. We have presented a technique called *use analysis* which allow points-to analysis of JavaScript applications which use libraries without analyzing the library code itself. Use analysis recovers dead flow by observing how objects are used by the code. Experimental results show that use analysis improves call-graph resolution (compared to an analysis which completely ignores the library) and its unsoundness is limited in practice.

7

The Browser Environment

"Inside every large program, there is a small program trying to get out."

— Tony Hoare

Today JavaScript is used as general-purpose programming language in a variety of applications. JavaScript is used for server-side applications [NodeJS, 2015], as a query language in databases [MongoDB, 2015], inside PDF documents [Adobe, 2015] and on smart phones [Tizen, 2015]. However, the original and dominant use of JavaScript is client-side programming on webpages.

A typical webpage is composed of four things: A HTML document which contains the text and the structure of the page, a CSS stylesheet which describe the visual appearance of the page, a collection of images, fonts and other resources used on the page, and JavaScript code to make the page interactive.

JavaScript enable programmers to write interactive applications which change the webpage on-the-fly and communicate with the server-side without reloading the entire page. Specifically, the browser environment allow programmers to listen for user events, e.g. mouse clicks, to perform requests over the network, e.g. AJAX and WebSockets [Mozilla, 2015], and to interact with the Document Object Model, which is an API for interfacing with the representation of the HTML [W3C, 2004, 2012].

A static analysis which targets JavaScript code running in the web browser must provide a model of these APIs. In this chapter, we consider two challenges related to modeling the web browser:

> **Researh Challenge I:** The design of analysis abstractions to model the HTML document and its operations.
>
> **Research Challenge II:** The design of analysis abstractions to model events in the web browser.

We now discuss these two challenges in turn.

```
<html>
    <head>
        <title>...</title>
    </head>
    <script type="text/javascript">
    function init(){
        var i = document.getElementById("image");
        i.url = "...";
        i.addEventListener("click", click);
    }
    function click() {
        var i = document.getElementById("image");
        i.url = "...";
    }
    </script>
    <body onload="init">
        <h1>Hello World!</h1>
        <p>Welcome to my web page.<p>
        <img id="image" src="..."/>
    </body>
</html>
```

Figure 7.1: A simple web page with embedded JavaScript.

7.1 The Document Object Model

The JavaScript interface for interacting with the HTML document is described in the Document Object Model (DOM). The HTML document is represented as a rooted tree with several operations for querying and mutating the tree.

Figure 7.1 shows an example of an HTML document with embedded JavaScript. The HTML document is arranged as rooted tree. The `<html>` tag is root. It has three child elements: The `<head>`, `<script>` and `<body>` elements. The `<script>` element contains JavaScript code. The `<body>` element contains three nested elements: A headline `<h1>`, paragraph `<p>` and `<img>` element. The image element has the unique identifier `image` specified by the `id` attribute.

The Document Object Model specifies various ways to navigate a HTML document [W3C, 2004]:

- **Parent Pointer**: Every HTML element object have field pointing to the parent element object called `parentElement`.The `parentElement` of the paragraph `<p>` element is the body `<body>` element.
- **Children Pointers**: Every HTML element object has a field `children` which is an array of its immediate child elements. The fields `firstChild` and `lastChild` point to the first and last child elements, respectively.

- **Global Queries**: The `document.getElementById` function allow an element in the HTML document to be found by its unique identifier. For example, the image `<img>` element has the unique identifier `"image"` and is found by `document.getElementById("image")`.
- **Local Queries**: Every HTML element object support the queries `getElementByTagName` and `getElementByClassName` which return all the elements of the given tag name or class name beneath the HTML element in the tree, respectively.

Furthermore, the HTML document is changable by the JavaScript code. Specifically, the following operations are available:

- **Insertion**: Every HTML element object has methods `insertBefore` and `appendChild` with obvious semantics.
- **Replace**: Every HTML element object has a method `replaceChild` to substitute one HTML element for another.
- **Deletion**: Every HTML object has a method `removeChild` to remove an HTML element.

Furthermore, there is the notorious `innerHTML` field which replaces a HTML element with an HTML fragment represented as a string.

We make the following important observations which guide us in the design of a simple approximation of the HTML document:

- The HTML document is rarely navigated by following parent and child pointers. Instead programmers prefer to use the query methods `getElementById` and `getElementBy` to find HTML elements.
- The shape of HTML element objects is often sufficient to detect bugs or verify their absence. For example, although a document may contain two image elements, every image element has `height` and `width` fields.

These two critical observations enable a simple and straightforward abstraction: We collapse all concrete HTML elements into an abstract object for every kind of HTML element (headline, paragraph, image, etc.). We maintain a mapping, initially build from the original HTML document, from HTML ids, tags and classes to these abstract HTML elements.

Consider the small program fragment below:

```
var e = document.getElementById("id");
var area = e.height * e.width;
```

If, by a programmer mistake, the `id` HTML element is a paragraph `<p>` element, and not an image `<img>` element as expected, the analysis can report that the paragraph element does not have `height` and `width` fields.

In some cases the analysis cannot accurately capture what HTML elements may occur. For instance, to ensure soundness, the `parentElement` field must point to all abstract HTML element objects. But since the use of `parentElement` is rare, at least in the programs we looked at, this does not seem to be an issue.

7.2 Events & Event Listeners

Event-based programs and their challenges have already been extensively discussed. In many ways that work supersedes this work, with respect to events. We briefly touch upon some of the challenges of the browser-based environment in relation to events.

The two central aspects are how to model browser-based and user-based events. As it turns out, one of the most important events is the `onload` event which triggers when the page has finished loading and is often used by programmers to run initialization logic. The user-based events occur based on actions taken by the user. For example, when user presses the mouse of clicks on an element. Based on this, we design a simple model of events, where the event listeners for the `onload` events are executed first, followed by a non-deterministic event loop which models the user by triggered various user-based events (e.g. a mouse click).

However, this work made no attempt to track any dependecies between event listeners, as done in the work on NodeJS.

Summary. We have presented a framework for static analysis of JavaScript web applications. The framework uses a simple model of the HTML DOM where every node of the same shape are collapsed into a single abstract summary object. The model explicitly tracks the mapping between HTML identifiers and abstract HTML element objects to allow precise treatment of queries such as `getElementById` and `getElementByTagName`. Furthermore, the framework simulates events caused by the user interacting with the page by non-deterministically emitting browser events.

8

Sparse Dataflow Analysis

> *"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."*
>
> — Antoine de Saint Exupry

We have seen, in the previous chapters, how static analysis techniques can handle several features in dynamic programming languages. In this chapter, and the following, we investigate how to implement these analyses.

A major challenge is the efficiency and scalability of whole-program static analyses. Especially challenging is the combination flow-sensitivity and subset-based points-to analysis [Hind, 2001]. A typical flow-sensitive analysis must maintain an entire abstract state, which consists of the entire abstract heap, for every program point. This leads us to the research challenge:

> **Research Challenge:** Efficient and scalable implementation of flow-sensitive dataflow analyses which embed points-to information.

A key observation is that it is unnecessary to have a complete representation of the entire abstract state at every program point. Instead sparse dataflow analysis techniques have focused on the construction of intermediate representations which use a partial representation of the abstract state at every program point. The insight is to identify *data dependencies* between different program points. Specifically, to identify *definition sites*, which are program points where a part of the abstract state is defined, and *use sites* which are program points where part of the abstract state is used [Oh et al., 2012]. Using this information definition– and use sites can be linked directly together using *def-use* edges, and dataflow propagated directly from source to target.

A popular and widely used intermediate representation, for these data dependencies, is static single assignment-form (SSA-form) [Cytron et al., 1989; Briggs et al., 1998], but other representations have also been proposed [Ferrante et al., 1987; Johnson and Pingali, 1993; Ramalingam, 2002].

We briefly outline the key differences between traditional dense and sparse dataflow analysis:

- **Traditional Dataflow Analysis:** A traditional *dense* analysis propagates dataflow facts along the edges of the control-flow graph [Kildall, 1973; Aho, 2003]. A *forwards* analysis propagates dataflow facts from a control-flow graph node to its successors, whereas a *backwards* analysis propagates dataflow facts to its predecessors. Every program point, represented by a control-flow graph node, maintains an entire abstract state. Immutable data structures and clever algorithms can reduce some of the memory overhead of dense dataflow analysis [Cousot et al., 2007], but these techniques are not always applicable nor sufficient.

- **Sparse Dataflow Analysis:** A *sparse* analysis propagates dataflow facts along def-use edges bypassing intermediate edges in the control-flow graph [Cytron et al., 1989; Choi et al., 1991; Oh et al., 2012]. Furthermore, in contrast to the dense analysis, every program point only has a partial representation of the abstract state. A sparse analysis may be *semi-sparse* or *fully-sparse* depending on whether it allows some redundancies. A semi-sparse analysis may be more efficient than a fully-sparse analysis since the computation cost to ensure full sparseness may outweigh its savings. Most sparse analysis techniques have focused on forwards dataflow problems, and there has been relatively little work on backwards analysis [Choi et al., 1991].

In this work, we develop a sparseness technique for forwards dataflow problems based on static single assignment-form. As the "devil is in the details" we switch from a semantic to a graph-based point-of-view. Specifically, we will no longer work with an expression-orientated language, but instead work on a control-flow graph composed of atomic statements.

8.1 Static Single Assignment-form

As mentioned before, static single assignment-form (SSA-form) is a popular intermediate representation built on top of the control-flow graph. We must introduce some terminology before we can describe the important property of SSA-form and how to construct SSA-form given a control-flow graph.

Control-Flow Graph. A control-flow graph (CFG), for a function λ, is a directed graph $G = (V, E)$ where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. The set of vertices is the set of atomic statements of the program

and the set edges is the control-flow graph edges connecting statements together [Aho, 2003]. The predecessors and successors of a vertex v is the set of incoming and outgoing edges of that vertex, respectively. Every control-flow graph has a designated entry vertex $s_{\text{entry}} \in V$ (which has no predecessors) and a designated exit vertex $s_{\text{exit}} \in V$ (which has no succesors). We assume that every vertex is reachable from the entry vertex, and that from every vertex the exit vertex is reachable. The control-flow graph can be constructed from the abstract syntax tree of the program [Aho, 2003].

Dominance. A vertex v is said to *dominate* a vertex u if every path from s_{entry} to u contains v. A vertex v *strictly* dominates u if $v \neq u$. The *immediate* dominator of u is the dominator closest to u [Cooper et al., 2001].

Dominator Tree. A *dominator tree* is a tree where the root is s_{entry} and the children of each node are immediately dominated by it, i.e. the unique parent of each node is its immediate dominator [Cooper et al., 2001]. Intuitively, a path in the dominator tree tells us which vertices we must encounter on any path in the CFG from the s_{entry} vertex to the target vertex.

Dominance Frontiers & Frontier Edges. The dominance frontier *DF*(v) of a vertex v is the set vertices $y \in V$ such that v dominates a predecessor, x, of y but not y itself [Cytron et al., 1989; Cooper et al., 2001]. The edge $x \to y$ is called the *frontier edge*. Intuitively, the dominance frontier of a vertex marks where its "influence" ends, e.g. x is dominated by v, but y is not. The *iterated dominance frontier* (*IDF*) is the transitive closure of the dominance frontier:

- *DF*$(v) \subseteq$ *IDF*(v).
- $\forall u \in$ *IDF*$(v) : DF(u) \subseteq$ *IDF*(v)

Definition and Use-sites. A vertex d is called a *definition site* for some abstract address a if the statement associated with that vertex may write to that address. Similarly, a vertex u is a *use site* for some abstract address a if the statement associated with that vertex may read from that address. In the context of $\lambda_{\texttt{base}}$ these correspond to the `ref` and `deref` expressions. We assume, without loss of generality, that a vertex is never both a definition- and use site for the same address, except for phi-sites as explained next.

Def-Use Edge. A *def-use edge* is an edge, not necessarily part of the CFG, which connects a definition site directly to a use site. A sequence of def-use edges $e_1, e_2, \ldots$ where $e_i \to e_{i+1}$ is a def-use edge is called a *def-use chain*.

Static Single Assignment-form. We can now state the important property of static single assignment-form:

> **SSA-Property:** A CFG is in *static single assignment-form* iff every use site is dominated by a single unique definition site.

A CFG may contain multiple definitions of the same *address*, but when these definitions meet in a merge point in the CFG the SSA-property is violated. To handle such cases, so-called phi-nodes or phi-sites, are inserted. A phi-site acts both as a definition– and use site and reestablishes the SSA-property. We can ensure that the SSA-property is satisfied by inserting phi-sites along the iterated dominance frontier of every definition [Cytron et al., 1989].

An equivalent formulation of SSA, and the one we will use, is to maintain a set of def-use edges. Using def-use edges is convenient since it represents the data dependencies directly.

Example: Static Single Assignment-form. Figure 8.1 shows an example of static single assignment-form. The left and right part show the same control-flow graph. The variable x is defined at the `A`, `D`, `E`, `G` and `H` vertices, and used at the `C` vertex. The left part shows the control-flow graph in static single assignment-form where the variable x is suitably renamed at every phi-site to ensure the SSA-property. For example, at `G` the definitions x_2 and x_3 are merged and a new definition x_4 is created. The right part shows an isomorphic representation based on def-use edges (shown as dashed arrows). For example, the def-use edge `A` $\rightarrow$ `C` represents that the definition at `A` is used at `C`.

8.2 Construction of SSA-form

Traditional SSA Construction. We can construct SSA-form given a control-flow graph and the set of definition– and use sites. The algorithm is straightforward and relies on dominance information. Assume, without loss of generality, that there is only one address. Given a CFG G the key steps are:

1. Construct the dominator tree for G.
2. Create phi-sites along the IDF of every definition site in G.
3. Recursively visit the dominator tree starting from the root while maintaining the current dominator. Perform each of the following, in order:

    a) If the vertex is a definition site then set the current dominator to the definition site.

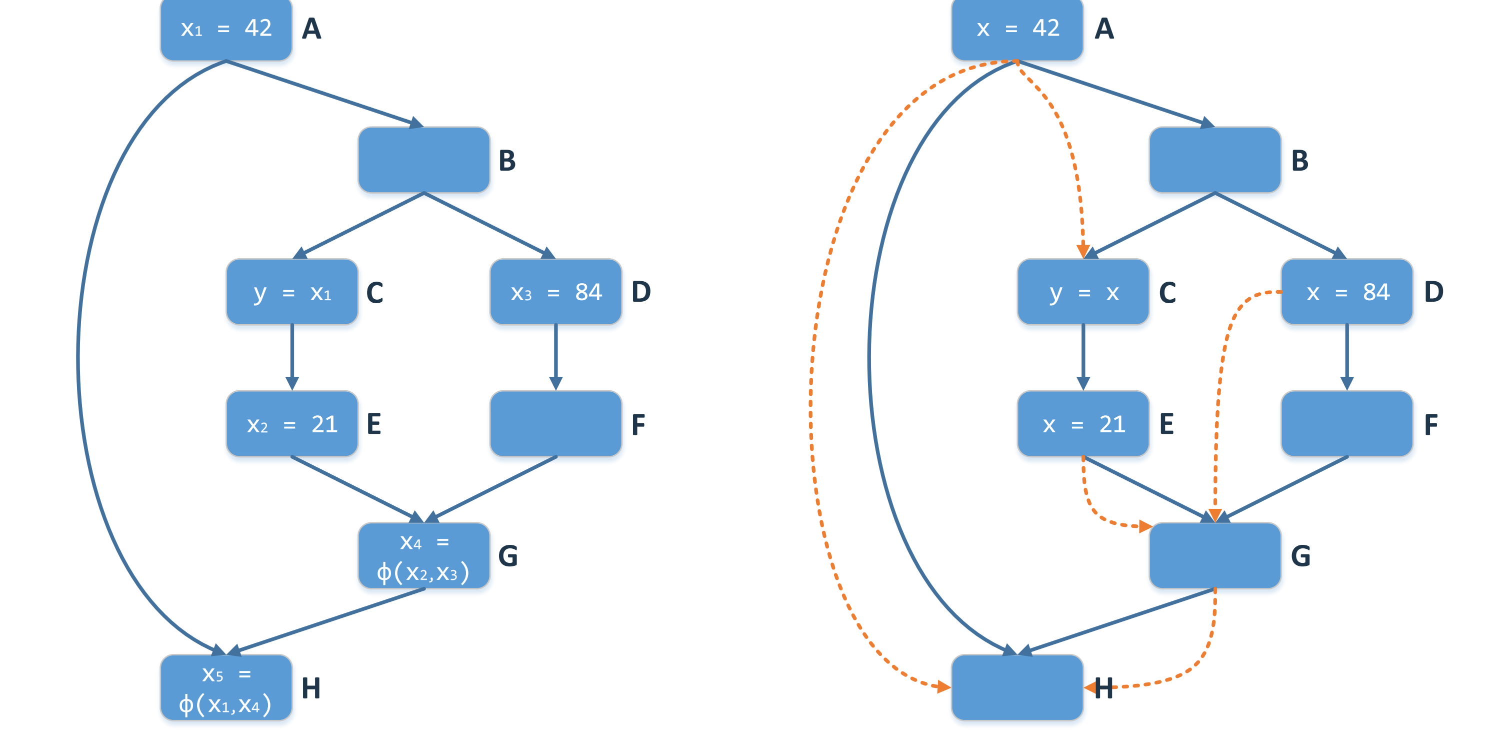


Figure 8.1: An example of a control-flow graph in static single assignment-form. The figure shows that the variable x is defined at the vertices `A`, `D`, `E`, `G`, `H` and used at the vertex `C`. The left part shows the traditional SSA-form where phi-sites have been introduced and the variable renamed. The right part shows an isomorphic representation with dashed def-use edges.

b) If the vertex is a use site then introduce a def-use edge from the current dominator to the vertex.
c) If the vertex has a dominance frontier edge then introduce a def-use edge from the current dominator to the phi-site (i.e. the target of the frontier edge).
d) Recurse on each child of the vertex.

As seen, the traditional SSA construction algorithm requires a single pass through the dominator tree.

Heap SSA

Heap-SSA, or Array-SSA, is an extension of SSA to dynamically allocated memory [Knobe and Sarkar, 1998; Sarkar and Knobe, 1998]. The challenge with heap-SSA is that the definition- and use sites are not statically known from the control-flow graph: The definition and use-sites depends on points-to information. Thus, we have the chicken-and-egg problem:

> **Research Challenge of heap-SSA:** We want to *use* heap-SSA for efficient propagation of dataflow facts during a dataflow analysis (which incorporates a points-to analysis), but to *construct* heap-SSA we need points-to information.

In fact this is a broader challenge of sparse analysis techniques: The sparse intermediate representation depends on the information computed by the analysis, but such information is not readily available before reaching the analysis fixed point.
We consider two approaches to overcome this problem:

Pre-analysis SSA Construction. The first idea is perform a *pre-analysis* to discover points-to information (specifically, the definition- and use sites) and then use this information to construct the heap-SSA *once*. The constructed SSA-form is then used during the main analysis to sparsely propagate dataflow facts [Hardekopf and Lin, 2009; Oh et al., 2012].

The advantage of this strategy is that it is conceptually simple and the SSA construction algorithm can be used as-is. However, there are also drawbacks: We must design two analyses, the pre-analysis and the main analysis. Ideally, the pre-analysis should (1) be fast to reduce overall analysis time, and (2) compute precise definition- and use sites to ensure sparseness. The more precise the pre-analysis, the more sparse the heap-SSA. Unfortunately, the design space for a pre-analysis is huge; we can use *any* analysis as the pre-analysis, but we cannot know, which one offers the best trade-off between time spent in pre-analysis and timed saved by sparseness in the main analysis.

On-the-fly SSA Construction. The second idea is to perform incremental SSA construction on-the-fly as new definition- and use sites are discovered [Staiger-Stöhr, 2013]. The advantage of this approach is twofold: First, there is no need to design, implement and test a pre-analysis. Second, the precision of the main analysis is used which yields the "optimal" SSA, i.e. there is a one-to-one correspondence between the definition- and use sites discovered by the main analysis and those used in the SSA-form. The disadvantage is that incremental maintenance of the SSA may be costly.

The information used by the traditional SSA construction algorithm is the control-flow graph, the dominator tree and the definition– and use sites. In our new setting, the control-flow graph and dominator tree remain static, the only information which is changing is the definition– and use sites. Thus, our task is to develop a new incremental SSA construction algorithm for when new definition– and use sites are discovered.

We will consider a slightly extended problem: We want our on-the-fly heap-SSA construction algorithm to support *branch prunning* or *dynamic reachability*, i.e. if the dataflow analysis *knows* that a certain branch in the program is unrealizable then the SSA-form should take that into account [Wegman and Zadeck, 1991]. Perhaps confusingly, we will use the term *reachability* when discussing branches which are realizable according to the dataflow analysis.

On-the-fly construction of heap-SSA

We design the on-the-fly heap-SSA construction algorithm as a blackbox with three commands:

- $\texttt{NewDef}(s, a)$: Informs the SSA construction algorithm that the control-flow graph vertex s is a new *definition site* for abstract address a.
- $\texttt{NewUse}(s, a)$: Informs the SSA construction algorithm that the control-flow graph vertex s is a new *use site* for abstract address a.
- $\texttt{Continue}(s_{\texttt{src}}, s_{\texttt{dst}})$: Informs the SSA construction algorithm that the control-flow graph edge $(s_{\texttt{src}}, s_{\texttt{dst}})$ is realizable, i.e. dataflow can flow on this edge.

Furthermore, we can at any time ask for the current def-use edges. As the analysis progresses it executes these commands and queries the current def-use edges to propagate dataflow. At any point in time, the def-use edges satisfy the SSA-property for the currently known definition– and use sites.

We now describe the on-the-fly SSA construction algorithm in greater detail. The algorithm is a fixed point computation on a small constraint system. Resolving one constraint may violate other constraints and require further

resolution. The three operations `NewDef`, `NewUse` and `Continue` each give rise to a new constraint which must then be resolved. An operation may not necessarily cause any change, if for example, a `NewUse` is executed for the same vertex, then only the first time will have an affect.

Internally the blackbox maintains three data structures:

- $\mathcal{D}$ is the currently known definition sites stored in data structure such that they can be efficiently queried based on dominance information. A key development, described in the paper, is the choice of data structure.
- $\mathcal{F}$ is a map of *frontier definitions*, i.e. a map from every dominance frontier edge to the currently dominating definition on that edge. The $\mathcal{F}$-map is used when frontier edges become reachable.
- $\mathcal{DU}$ is the current set of def-use edges.

The constraints express properties on the maps $\mathcal{D}$, $\mathcal{DU}$ and $\mathcal{F}$. Here $\mathcal{D}$ and $\mathcal{F}$ grow monotonically during the fixed point computation, whereas $\mathcal{DU}$ may have edges removed. However, the same def-use edge is never re-introduced and there are only a limited number of def-use edges that can ever be added, thus the algorithm terminates.

- `NewDef`$(d \in V, a \in$ `Address`$)$. Expresses that a new definition, for address a, must be created at d, all def-use chains bypassing d must be "cut" and the definition must be propagated to all dominance frontiers (which is ensured by adding the `Forward` constraint).

1 **if** the definition d is not present in $\mathcal{D}$.
2   **let** d' be the unique dominating definition at d.
3   **for** every def-use edge (d', a, u) for any u.
4   **if** u is dominated by d OR u is in the dominance frontier of d.
5   Remove the def-use edge (d', a, u).
6   Add the def-use edge (d, a, u).
7   Propagate dataflow along (d, a, u).
8   Add the constraint `Forward`(u, a).

- `NewUse`$(u \in V, a \in$ `Address`$)$. Expresses that a def-use chain, for address a, must be introduced for the dominating definition at u.

1 **let** d be the unique dominating definition at u according to $\mathcal{F}$.
2 Add the def-use edge (d, a, u).
3 Propagate dataflow along the edge (d, a, u).

- `NewPhi`$(d \in V, a \in$ `Address`$)$. Expresses that a phi site must be created at d, def-use chains introduced for all *reachable* frontier edges and phi sites created at the dominance frontiers of d.

```
1  for each predecessor s_0 of d.
2      if the edge (s_0, d) is reachable.
3          if a is available on the edge (s_0, d) (as determined by F).
4              let d' be the unique dominating definition at s_0.
5              Introduce the def-use edge (d', a, d).
6              Propagate dataflow along (d', a, d).
7          else
8              Add the constraint NewUse(d, a).
9  Add the constraint NewDef(d, a).
```

The inner if-statement checks for the case when an edges carries no definition. In Figure 8.1, for example, the edge `A` $\to$ `H` is not a frontier edge, and carries no definition according to $\mathcal{F}$.

- `Continue`$(s_{\texttt{src}}, s_{\texttt{dst}})$. Expresses that the control-flow graph edge $(s_{\texttt{src}}, s_{\texttt{dst}})$ is now reachable and definitions must be propagated across it.

```
1  for each address a on the edge (s_src, s_dst) (as determined by F).
2      Add the constraint NewPhi(s_dst, a).
3  for each address a available at s_dst, but no on the edge (s_src, s_dst).
4      Add the constraint NewPhi(s_dst, a).
```

- `Forward`$(d \in V, a \in$ `Address`$)$. Expresses that a definition site, for address a, must be introduced at all dominance frontiers of d.

```
1  for each frontier edge (s_src, s_dst) of d.
2      if a is not on the frontier edge (s_src, s_dst).
3          Add a to the addresses on the frontier edge (s_src, s_dst).
4          if (s_src, s_dst) is a reachable edge.
5              Add the constraint NewPhi(s_dst, a).
```

Propagation: The phrase *propagate dataflow along* (d, a, u) instructs the analysis to propagate the value available *immediately after* the program point d to the program point *immediately before* u for address a. If the value is changed then the analysis must propagate it along further def-use chains (which are known) and add any statements to the worklist where the abstract value changed.

Future Work. We briefly outline two directions for future work.

- **Super-graph SSA.** The *super-graph* of a program is the control-flow graph obtained by stitching, i.e. inlining, the control-flow graphs of the individual functions at their call sites [Reps et al., 1995]. We can imagine an on-the-fly heap-SSA construction algorithm which operates at the level of the super-graph. An SSA-form on the super-graph would allow definition- and use sites to be connected across functions bypassing any superfluous phi-sites inserted at function entry and exit statements. The main challenge is that the super-graph depends on the control- and dataflow of the program, and is itself discovered on-the-fly. As a consequence the SSA construction algorithm must be adapted to work on an evolving control-flow graph, and as a consequence, with changing dominance information.

- **Minimal SSA.** Minimal SSA is a variant of SSA with the requirement that every phi-site is *live*, i.e. every phi-site must be directly or transitively connected to a non-phi use-site [Cytron et al., 1989; Briggs et al., 1998]. The construction of minimal SSA relies on liveness information to eliminate superfluous phi-sites. We can envision an on-the-fly heap-SSA construction algorithm combined with an on-the-fly heap liveness analysis to construct minimal heap-SSA on-the-fly. A precise heap liveness analysis may be too expensive in practice. However, *any* liveness analysis is usable: The precision of the liveness analysis determines the minimality of the SSA-form. Recent work has shown, in a non-sparse, non-SSA setting, that liveness analysis can be used to speed-up points-to analysis [Khedker et al., 2012].

Summary. We have presented a technique for sparse dataflow propagation based on on-the-fly construction of heap-based static single assignment-form. The def-use edges of heap-SSA allow dataflow facts to be propagated directly from definition sites to uses site modulo any phi-sites. The new SSA construction algorithm improves upon existing algorithms by taking control-flow reachability into account. Experiments have shown that the performance of the on-the-fly SSA construction algorithm is competitive with pre-analysis based techniques.

9

FLIX: Programming with Fixedpoints

"A language that doesn't affect the way you think about programming, is not worth knowing."

— Alan Perlis

We have seen, in the previous chapters, how static analysis techniques are used to approximate the behaviour of programs written in dynamic programming languages. We now turn to the question of how to implement such analyses.

A straightforward solution is to implement the abstract domains and abstract semantics in a general-purpose programming language. The implementation effectively becomes an interpreter for the abstract semantics [Cousot and Cousot, 1977]. The correctness of the implementation is immediate from the one-to-one correspondence with the mathematical rules. Unfortunately, such implementations tend to be horribly slow since the mathematical rules often contain computational redundancies.

Many of these redundancies can be eliminated through clever observations and careful rule rewriting [Bancilhon et al., 1985]. The literature is rich with optimization techniques for static analyses. To highlight just a few:

- Sparse dataflow propagation techniques which use compact representations of abstract states and data- dependencies to limit memory usage and dataflow propagation [Ramalingam, 2002; Hardekopf and Lin, 2009; Oh et al., 2012].

- On-demand dataflow algorithms which given a query attempt to compute the minimal amount information necessary to answer the query. These techniques often rely on both forwards and backwards propagation [Duesterwald et al., 1995; Horwitz et al., 1995].

- Function summaries which are computed once for every function and then applied at every call site of the function [Pnueli, 1981; Reps et al., 1995; Xie and Aiken, 2005].

- Cycle elimination algorithms which collapse cycles in the constraint graph. *On-the-fly* algorithms use heuristics to detect cycles during the fixed point computation, whereas *offline* algorithms perform the detection and elimination once, before the fixed point computation [Fähndrich et al., 1998; Rountev and Chandra, 2000; Hardekopf and Lin, 2007].

- Abstract garbage collection which, like concrete garbage collection, removes unreachable objects from the (abstract) heap thereby reducing memory usage [Jagannathan et al., 1998; Might and Shivers, 2006].

- Work-list heuristics or strategies which improve precision and speed-up the fixed point computation [Chambers et al., 1996; Oh, 2009].

- Language specific optimizations, such as type filters, which exploit knowledge of the language to reduce spurious flow [Lhoták, 2002].

These techniques are often combined with general-purpose optimizations to squeeze out the last drop of performance:

- Caches to eliminate redundant computations.

- Efficient data structures, e.g. hash sets, hash maps and balanced trees, to index and store large data sets.

- Compact data structures, e.g. bitmaps, sparse arrays and binary decision diagrams (BDDs), to store large data sets [Berndl et al., 2003; Whaley and Lam, 2004].

- Immutable data structures, combined with functional programming techniques, to reduce memory usage and speed-up joins [Cousot et al., 2005, 2007].

Using these techniques present their own set of challenges: On one hand they are necessary to ensure acceptable performance and scalability. On the other hand they are a source of complexity. The techniques tend to interact with many analysis components and obscure the implementation: The elegant one-to-one corresponds between semantic rules and implementation is lost. Consequently, the correctness of the analysis may become in doubt. For example, how can we be sure that the transfer functions are actually monotone? Furthermore, these techniques tend to be rei-mplemented in every static analysis tool which is both wasteful and error-prone.

Based on these observations, we have the research challege:

> **Research Challenge**: The implementation of highly efficient, yet obviously correct, static analyses.

We propose to separate the analysis *specification* from the analysis *computation* inspired by similar approaches in SAT and SMT-solvers [De Moura and Bjørner, 2011]. The specification should closely follow the semantic rules. The fixpoint solver should be efficient and come with many of the above optimizations built-in. Importantly, the codebase for the specification and the solver should be completely independent.

The Datalog programming language has been proposed as a realization of the idea above. However, as we shall see, Datalog has several theoretical and practical limitations.

9.1 Datalog and its Limitations

Datalog is a declarative constraint-based programming language on relations [Ceri et al., 1989]. Datalog has similarities to the widely used structured query language (SQL) used in databases [Chamberlin and Boyce, 1974], but its expressive power is strictly greater since Datalog has recursive constraints.

Datalog has several interesting theoretical properties:

- All Datalog programs terminate.
- All Datalog programs have a least fixed point.
- All Datalog programs are solvable by semi-naïve bottom-up evaluation [Ceri et al., 1989].
- Any polynomial time algorithm can be implemented in Datalog [Papadimitriou, 1985].

Datalog has been successfully used in several high-profile points-to analyses the Java programming language [Lam et al., 2005; Hajiyev et al., 2006; Bravenboer and Smaragdakis, 2009; Smaragdakis et al., 2011; Smaragdakis and Bravenboer, 2011; Kastrinis and Smaragdakis, 2013; Smaragdakis et al., 2014]. However, Datalog has seen limited use beyond points-to analysis. And, as we shall see, its relational nature make many simply dataflow analyses akward or even impossible to express in Datalog.

Declarative Languages. What is a declarative language? Robert Harper states, in an interesting blog post, that there is no commonly agreed upon definition and provides several possible definitions [Harper, 2013]. My opinion is that declarative programming is about "the what, not the how" (one of Harper's suggestions). The "what" means that the programmer specifies *constraints* which the solution must satisfy. The "how", i.e. how to compute the result efficiently, is left up to implementation.

Limitations of Datalog

We illustrate the limitations of Datalog by attempting to implement three numeric dataflow analyses: a sign, constant propagation and interval analysis. In general-purpose languages these analyses are straightforward to implement.

Sign Analysis. A *sign analysis* determines whether a numeric expression is negative, zero or positive [Cousot and Cousot, 1977]. The sign lattice consists of five elements: `Neg`, `Zer`, and `Pos` combined with bottom $\bot$ and top $\top$ elements. The bottom element represents "absurdity", i.e. that the expression is non-numeric, whereas the top element represents the lack any information, i.e. the expression may be negative, zero or positive.

The sign lattice *cannot* be implemented directly in Datalog. We can, however, encode a similiar analysis using sets. We let the empty set $\emptyset$ correspond to the bottom element and represent each element `Neg`, `Zer`, and `Pos` by its singleton set. The top element is then the set $\{\texttt{Neg}, \texttt{Zer}, \texttt{Pos}\}$. The encoding is not perfect: The set $\{\texttt{Neg}, \texttt{Zer}\}$ has no corresponding element in the original sign lattice. We can implement monotone functions over these relations, e.g. abstract addition and multiplication, as explicitly tabulated Datalog relations. Using these schemes it is possible, although not exactly elegant nor efficient, to implement a sign analysis in Datalog.

Constant Propagation. A *constant propagation analysis* determines the value of any expression which is constant [Cousot and Cousot, 1977]. The constant propagation lattice consists of an infinite set of constants (e.g. integers) combined with bottom $\bot$ and top $\top$ elements. As before, the bottom element represents "absurdity", whereas the top element represents that the expression is non-constant.

The constant propagation lattice *cannot* be implemented directly in Datalog and is difficult to encode. A possible encoding would consider a finite, fixed set of input constants and then propagate sets of these constants. Unfortunately, this has two major downsides: First, the top element is very expensive to represent and compute with since it consists of the set of all constants. Second, it is impossible to implement many monotone functions, e.g. for abstract

addition and multiplication, since these functions have infinite domains and codomains and cannot be tabulated. Thus the constant propagation analysis cannot be implemented in Datalog.

Interval Analysis. An *interval analysis* determines the interval range, i.e. *lower* and *upper* bounds, of the value of a numeric expression [Cousot and Cousot, 1977]. The interval lattice consists of an infinite set of intervals combined with bottom $\bot$ and top $\top$ elements. Again, the bottom element represents "absurdity", whereas the top element represents the "unbounded" interval from negative infinity to positive infinity.

The interval lattice is hopeless to implement in Datalog. All the same difficulties of the constant propagation analysis applies equally well to the interval analysis. Furthermore, the elements of the interval lattice are *pairs* of numbers, but Datalog lacks compound data types.

As these examples demonstrate, implementation of common dataflow analyses in Datalog is difficult or impractical.

9.2 The FLIX Language

We propose to overcome these issues with the design and implementation of a new declarative programming language called FLIX. FLIX is strictly more expressive than Datalog, but aims to preserves the safety properties of Datalog, i.e. termination and the existence of a least fixed point. The key ideas are:

- A constraint language which is a superset of Datalog: From recursive constraints on relations to recursive constraints on lattice elements.
- A functional programming language for specification of lattices, e.g. the partial order and least upper bound, and monotone functions over lattice elements.
- A type system, the simply-typed lambda calculus with various extensions, to ensure type safety and termination of all functions.
- An (optional) verifier to ensure safety properties, e.g. monotonicity, which cannot be proved in the simply-typed lambda calculus.

FLIX, like Datalog, is by design Turing-*incomplete*. The simply-typed lambda calculus (STLC) ensures that all well-typed terms reduce to a value in a finite number of steps [Pierce, 2002; Harper, 2012]. Recursion is allowed at the constraint level, but the constraints are monotone over finite height lattices, and thus termination is ensured.

Analysis Frameworks. The literature is rich with static analysis frameworks [Martin, 1998; Vallée-Rai et al., 1999b; Lhoták and Hendren, 2003; Cuoq et al., 2009; Ramsey et al., 2010; Fink, 2014]. Why then design a language like FLIX? The answer is twofold: First, FLIX is much more general than any framework. FLIX aims to provide a set of programming constructs, but does make any assumptions about the analysis being implemented. Frameworks, on the other hand, tend to make assumptions about the underlying analysis. For example, a common assumption is the existence of a control-flow graph. Second, FLIX can verify that the analysis components behave as expected. In contrast, frameworks simply assume that the analysis components are correctly implemented, and otherwise behave unpredictably.

Example. Figure 9.1 shows a FLIX implementation of the parity domain. The parity domain determines whether a natural number is odd or even.

The program declares the `Parity` namespace in which all inner declarations belong. `Flix` namespaces allow reuse of a name across different namespaces. Namespaces are *not* modules. By design FLIX has no interfaces, data abstractions or information hiding. All names must be known at compile-time. This is necessary for FLIX to prove the safety properties.

Inside the namespace, the program declares the elements of the parity lattice which are `Odd`, `Even`, `Top` and `Bot`. The partial order and least upper bound are declared as functions named `leq` and `lub`. The lattice is assembled with the lattice declaration `lat` which associates the bottom element, partial order and least upper bound with the `Parity` type.

The abstract `sum` function is declared as taking two `Parity` elements and returning a single `Parity` element. As with the partial order and least upper bound, the implementation is made easy with pattern matching. The `@strict` annotation instructs the compiler to verify that the function is strict, i.e. that if an input to the function is bottom then the output is bottom. The `@monotone` annotation instruct the compiler to verify that the function is monotone, i.e. that if the inputs grow according to their partial order then the outputs grow according to their partial order.

At last, the program declares a lattice variable named `R` and a single fact. The lattice variable is an element of the parity lattice and grows monotonically during the fixed point computation. The fact specifies that `R` must be *at least*, according to the partial order, the result of applying the `sum` function to the two constant arguments `Odd` and `Even`. The sum of `Odd` and `Even` is `Top` and execution of the program produces `R` = `Top`. Obviously a real analysis would make use of multiple lattices, lattice variables and monotone functions.

```
namespace Parity {
  enum Parity {
                case Top,
      case Odd,            case Even,
                case Bot
  };

  def leq(e1: Parity, e2: Parity): Bool =
    match (e1, e2) with {
      case (Bot, _)        => true;
      case (Odd, Odd)      => true;
      case (Even, Even)    => true;
      case (_, Top)        => true;
      case _               => false;
    };

  def lub(e1: Parity, e2: Parity): Parity =
    match (e1, e2) with {
      case (Bot, _)        => e2;
      case (_, Bot)        => e1;
      case (Odd, Odd)      => Odd;
      case (Even, Even)    => Even;
      case _               => Top;
    };

  lat Parity {
    elms = Parity, bot = Bot, leq = leq, lub = lub
  };

  @strict @monotone
  def plus(e1: Parity, e2: Parity): Parity =
    match (e1, e2) with {
      case (_, Bot)        => Bot;
      case (Bot, _)        => Bot;
      case (Odd, Odd)      => Even;
      case (Odd, Even)     => Odd;
      case (Even, Odd)     => Odd;
      case (Even, Even)    => Even;
      case _               => Top;
    };

  var R: Parity;

  fact R sum(Odd, Even);
};
```

Figure 9.1: A FLIX implementation of the Parity domain.

9.3 Safety Properties

We briefly mentioned that several safety properties must be satisfied to ensure termination and the existence of a least fixed point for a FLIX program. Indeed, *any* static analysis in the abstract interpretation framework [Cousot and Cousot, 1977; Nielson et al., 1999] must satisfy these or similar properties. The strength of FLIX is that it attempts to verify these properties automatically, instead of leaving the burden on the analysis implementer, as is the case for many static analysis frameworks. Specifically, given a lattice $L = (\tau, \bot, \sqsubseteq, \sqcup)$, where τ is the type of the elements, FLIX checks:

- The bottom $\bot$ element is the least: $\forall x : \bot \sqsubseteq x$.
- The partial order is reflexive: $\forall x : x \sqsubseteq x$.
- The partial order is anti-symmetric: $\forall x, y : x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y$.
- The partial order is transitive: $\forall x, y, z : x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z$.

and that the partial order and least upper bound are consistent:

- $\forall x, y : x \sqsubseteq (x \sqcup y) \land y \sqsubseteq (x \sqcup y)$.
- $\forall x, y, z : x \sqsubseteq z \land y \sqsubseteq z \Rightarrow (x \sqcup y) \sqsubseteq z$.

Furthermore, FLIX checks that functions are strict and monotone:

- The function $f : \tau \to \tau$ is *strict* if $f(\bot) = \bot$.
- The function $f : \tau \to \tau$ is *monotone* if $\forall x, y : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.

Strictness is used to simplify parts of the solver and not always required. Monotonicity ensures that the computation terminates as the lattice variables always moves upwards in the lattice.

A final requirement is that the lattice has finite height. FLIX requires the analysis implementer to specify a height function, essentially a termination function, which is used to prove that the lattice has no infinite ascending chains. Specifically, the programmer must provide a function $h : \tau \to \mathit{Nat}$ such that:

$$\forall x, y : x \sqsubseteq y \Rightarrow h(x) > h(y)$$

FLIX then proves that the partial order satisfies the termination function.

The purpose of all these properties is to ensure that FLIX programs are well-behaved. Without these properties it would be impossible for the solver to *guarantee* that the program terminates with the least fixed point.

When we write "FLIX checks that ..." we should be careful and say more accurately that FLIX *attempts* to verify the property. In general, FLIX cannot prove that a function satisfies these property, even if the function is simply-typed. The reason is that we have enriched the simply-typed lambda calculus with Peano arithmetic, i.e. numbers with addition and multiplication, which is undecidable [Weisstein, 2015].

9.4 Verification

FLIX uses an combination of *partial evaluation* [Jones et al., 1993; Consel and Danvy, 1993] and *satisfiability module theory* (SMT) solvers [De Moura and Bjørner, 2008; Kroening and Strichman, 2008; De Moura and Bjørner, 2011] to automatically prove the properties listed in the previous section.

Assume, for the purpose of example, that FLIX must prove that a function $f : \tau \to \tau$ is monotone. A key insight is that this property can be represented as a boolean function inside FLIX:

$$f' = \lambda x.\, \lambda y.\, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

Replacing $a \Rightarrow b$ by its logical equivalent $\neg a \vee b$ we obtain:

$$f' = \lambda x.\, \lambda y.\, \neg\, (x \sqsubseteq y) \vee (f(x) \sqsubseteq f(y))$$

This function f' has type: $\tau \to \tau \to$ *Bool* inside FLIX. Now, to verify the original property, i.e. that the function f is monotone, it is sufficient to verify that the above function f' is true for all inputs!

If the type τ happens to be inhabited by finite number of values, as is the case for the `Parity` lattice, then we can simply enumerate all inputs to the function f', evaluate it and verify that it returns true. If the function f' returns false for some input then that is a counter-example proving the non-monotonicity of f. Providing counter-examples to the analysis implementer is very useful for finding and understanding bugs.

What happens if an input to f' is not enumerable? The constant propagation lattice, as an example, has finite height, but infinite width: It is impossible to enumerate all constants. In this case, we simply leave the input variable unbound (free). Then we partially evaluate the term. In some cases this can prove the property out-right, in other cases a residual term is left. Although the FLIX language has algebraic datatypes, tuples, pattern matching, etc. these are all eliminated by the partial evaluation. The reason is that the function f' has type $\tau \to \tau \to$ *Bool*, thus the residual term can only consist of a boolean expression (conjunction, disjunction, negation) over some un-interpreted operations (e.g. addition or multiplication). This expression is easy to convert into an SMT formula which can then be sent to an SMT solver. The SMT solver

will then output true, false or loop. If it outputs true, then the property holds, if false then there is a counter-example, and if it loops then it is unknown whether the property holds due to the undecidability of the underlying theory, i.e. Peano arithmetic. The elegance of this approach is that many properties can be proven within FLIX itself, and those which cannot are easy to transform into SMT queries or proof burdens for the programmer.

Summary. We have presented a declarative programming language, called FLIX, for the specification of fixed point algorithms. FLIX builds on and improves Datalog by adding support for lattices and monotone transfer functions over lattice elements. Experiments with FLIX show that it can be used to express several well-known dataflow analyses, including the IFDS [Reps et al., 1995] and IDE [Sagiv et al., 1996] algorithms and the strong-update points-to analysis [Lhoták and Chung, 2011]. Research on FLIX is on-going and the subject of my forthcoming post doc at the University of Waterloo.

10

Conclusions

"I hear you say "Why?" Always "Why?" You see things; and you say "Why?" But I dream things that never were; and I say "Why not?""

— George Bernard Shaw

Dynamic programming languages remain popular and widely used. These languages are praised for their flexibility and expressive power. Unfortunately, these characteristics limit the availability of tool-support for those languages. Specifically, the lack of static type systems imply that many bugs are not discovered until run-time and that the construction of developer tools, such as integrated development environments, is difficult.

This thesis has investigated the application of static analyses techniques to remedy these shortcomings. The use of static analysis, and in particular whole-program dataflow analysis, allow static reasoning about programs written in these languages without changing their nature or imposing unrealistic restrictions on the programmers.

In conclusion, this thesis contributes four results on static analysis of features found in dynamic programming languages:

- A new framework for static analysis of event-based JavaScript applications. The framework describes the semantics of a single-threaded event-based language. Based on the concrete semantics, the framework describes three analysis variants: A *baseline*, *event* and *listener* sensitive analyses. The baseline analysis does not track any orderings between events nor listeners. The event sensitive analysis accurately tracks emitted events, but is computationally expensive. The listener sensitive analysis offers a compelling alternative which tracks dependencies between listeners, but is faster than the event sensitive analysis. Experimental evaluation on real-world StackOverflow questions show that the listener sensitive analysis is able to detect and explain real bugs.

- A collection of efficient and compact abstract string domains for use in static analysis of dynamic field accesses. The work distinguishes itself based on three key features. First, a focus on the precision of abstract string equality to reduce the number of spuriously read or written fields. Second, a focus on the abstract domains as a "drop-in" replacement for traditional constant propagation. Third, efficient $\mathcal{O}(1)$ time operations for the string equality, concatenation and the least upper bound operations. Experimental results show that a combination of the character inclusion lattice and the string hash lattice is effective at reducing the number of spuriously accessed fields.

- A new technique called *use analysis* which enables static analysis of JavaScript applications dependent on libraries without analyzing the library code. Use analysis works by recovering information about *dead flow*, due to missing information about the library, by inspecting how objects are used by the application and then unifying this information with its knowledge of all objects in the heap. Experimental results show that use analysis improves call graph resolution and that in practice its unsoundness is limited.

- A framework for static analysis of JavaScript web applications which execute in the web browser. The framework consists of a simple model of the HTML page which ignores its tree structure, but preserves the "type-like" structure of its various elements, and a model of the execution of event listeners similiar to the baseline analysis, but while taking into account certain statically known properties, such as the execution of `onload` event listeners before any other listeners. Experimental results show that the simple model of HTML elements has sufficient precision for reasoning about the safety of field reads. However, other challenges related to JavaScript remain.

Furthermore, this thesis contributes two results for the design and implementation of static analyses:

- A new sparseness technique to improve the performance of whole-program dataflow analyses. The technique is a novel algorithm for on-the-fly construction of static single assignment-form for heap allocated memory. SSA-form enables a sparse representation of def-use edges which are used to propagate dataflow directly from definition sites to use sites across the control-flow graph. A key insight is the use of quad trees to efficient store and retrieve definition sites together with dominance information. Experimental results show that on-the-fly SSA construction is competetive with pre-analysis based approaches.

- A new programming language called FLIX for the declarative specification of static analyses. FLIX cleanly separates the analysis specification from the analysis computation thereby freeing the analysis designer of performance concerns. FLIX is inspired by the Datalog language, but has strictly greater expressive power. FLIX combines declarative rule-based programming with functional programming. The essence of the language is relational constraints with monotone functions over lattice elements. By design, FLIX is Turing-*incomplete* to ensure decideability or semi-decideability of a range of safety properties. The safety properties are used to ensure the existence of a least fixed point which is computable in a finite number of steps. Experiments show that several existing techniques, e.g. IFDS and IDE, can be expressed declaratively in FLIX. Work on FLIX is on-going.

The thesis statement claimed that static analysis techniques could be used to reason about programs written in dynamic languages. This thesis has shown four areas, described above, where static analysis techniques are capable of reasoning about a particular aspect or feature of dynamic programming languages. However, not all problems are solved. Scalability and soundness remain challenging aspects of static analysis and an on-going research area. Furthermore, there are features of dynamic programming languages which are yet to be explored in the context of static analysis.

This work is not the end of the road. It is only the beginning.

Part II

Papers

11

Static Analysis of Event-based JavaScript Applications

This is a draft of a manuscript by Magnus Madsen, Ondřej Lhoták and Frank Tip.

Abstract

Many JavaScript programs are written in an event-driven style. In particular, in server-side NodeJS applications, operations involving sockets, streams, and files are typically performed in an asynchronous manner, where the execution of listeners is triggered by events. Several types of programming errors are specific to such event-based programs (e.g., unhandled events, and listeners that are registered too late). We present the *event-based call graph*, a program representation that can be used to detect bugs related to event handling. We have designed and implemented three analyses for constructing event-based call graphs. Our results show that these analyses are capable of detecting problems reported on StackOverflow. Moreover, we show that the number of false positives reported by the analysis on a suite of small NodeJS applications is manageable.

11.1 Introduction

JavaScript has rapidly become one of the most popular programming languages[1] and is now being used in several areas beyond its original domain of client-side scripting. For example, NodeJS[2] is a popular platform for building server-side web applications written in JavaScript, and JavaScript is one of the primary languages used for application development in the Tizen Operating System[3] for mobile devices.

As a result of JavaScript's increasing popularity, there has been a growing demand for tools that assist programmers with tasks such as program understanding and maintenance [Schäfer et al., 2013b; Madsen et al., 2013], bug detection and localization [Artzi et al., 2012; Kashyap et al., 2013], refactoring [Feldthaus et al., 2011; Feldthaus and Møller, 2013], and detecting and preventing security vulnerabilities [Guarnieri and Livshits, 2009; Tripp et al., 2013; Møller and Schwarz, 2014].

Many such tools rely on static analysis to approximate a program's behavior. One program representation that is commonly used in static analysis is the call graph, which associates with each call site in a program the set of functions that may be invoked from that site. For example, a tool for finding security vulnerabilities might use a call graph to detect possible data flows from tainted inputs to security-sensitive operations, and a refactoring tool may rely on a call graph to determine how to inline a function call.

However, a traditional call graph reflects only the interprocedural flow of control due to function calls, and ignores the event-driven flow of control in many JavaScript applications. For example, interactive applications that access the HTML Document Object Model (DOM) typically do so by associating event listeners with DOM nodes that correspond to fragments of an HTML document in a browser. Similarly, server-side applications based on NodeJS are typically written in an event-based style which heavily relies on callbacks that are invoked when an asynchronously executed operation has completed.

Several new types of bugs may arise in event-based programs. For example, an application may emit events for which no listener has been registered yet, or an event listener may be unreachable code because an event name was misspelled or the event listener was registered on the wrong object. In Section 11.2, we discuss a few examples of such errors that were reported on StackOverflow, a popular discussion forum for programming-related problems.

Our goal in this paper is to detect errors in event-based JavaScript programs using static analysis. We observe that traditional call graphs are not suitable in this context, because they do not reflect the flow of control that

[1] See http://langpop.com/.
[2] See http://www.nodejs.org/.
[3] See https://www.tizen.org/.

is implied by the registration of event listeners and the emission of events. Therefore, we propose the *event-based call graph*, an extension of the traditional notion of a call graph with nodes and edges that reflect the flow of control due to event handling.

We show how event-based call graphs can be used as the basis for detecting several types of errors, and present a family of analyses for computing event-based call graphs with varying cost and precision. Specifically, we extend the λ_{JS} calculus [Guha et al., 2010] with constructs for event listener registration and event emission. Based on this extended calculus, we present an analysis framework and three instantiations: A *baseline*, an *event-sensitive* and a *listener-sensitive* analyses. The event- and listener sensitive analyses compute separate dataflow facts based on what events have been emitted and what listeners are registered, respectively. We implement these variants in a static analysis tool called RADAR.

We experimentally evaluate RADAR by applying it to buggy NodeJS programs from the StackOverflow website and to programs from the NodeJS documentation [Dahl, 2014] and the *Node.js in Action* book [Cantelon et al., 2014].

In summary, our paper makes the following contributions:

- We identify errors that may arise in event-based JavaScript programs.
- We extend the λ_{JS} calculus with constructs for event listener registration and event emission.
- We define the notion of an *event-based call graph*, which extends the traditional notion of a call graph with nodes and edges that reflect the flow of control due to events, and show how this graph can be used to detect errors.
- We demonstrate that event-based call graphs are useful for finding errors by applying them to several buggy NodeJS programs from the StackOverflow website.
- We present three analyses, with varying cost and precision, for constructing event-based call graphs. For convenience, we will refer to these as the *baseline*, *event-sensitive* and *listener-sensitive* analysis .

11.2 Motivating Examples

In this section, we examine a few examples of buggy NodeJS programs (taken from StackOverflow) that reflect actual problems experienced by NodeJS developers. The static analyses presented in this paper are capable of detecting these problems, as will be discussed in Section 11.6.

```
var fs = require('fs');
var rest = require('restler');

var restlerHtmlFile = function(url) {
  rest.get(url).on('complete', function(res) {
      fs.writeFileSync('file.html', res);
  });
};

if (require.main == module) {
  restlerHtmlFile('http://obscure-refuge-7370.herokuapp.com/');
  fs.readFileSync('file.html');
} else {
  exports.checkHtmlFile = checkHtmlFile;
}
```

Figure 11.1: A NodeJS program in which the programmer is incorrectly combining synchronous and asynchronous calls. See http://stackoverflow.com/questions/19081270/.

StackOverflow Question 19081270

Consider the program of Figure 11.1, which relies on the `restlr` library to facilitate interaction with HTTP servers. Lines 4–8, assign a function to variable `restlerHtmlFile`. The call `rest.get(Url)` within this function creates a GET request to obtain the contents of a URL. The call `.on('complete', ...)` on line 5 serves to write the page contents to `file.html` when the request completes. Then, on line 11, the function bound to `restlerHtmlFile` is invoked to read the contents of the URL http://obscure-refuge-7370.herokuapp.com/ into the file `file.html`, and on line 12, this file is read by calling `fs.readFileSync(' file.html')`. The programmer reports on StackOverflow that the program crashes with an error message "no such file or directory 'file.html' at Object.fs.openSync (fs.js:427:18)".

The problem here has to do with the fact that the function passed in the call `.on('complete', ...)` on line 5 is invoked *asynchronously*, when the GET request has completed. However, line 12 in which the generated file `file.html` is written executes immediately after line 11, without making sure that the file creation has completed.

There are various ways in which the code can be fixed. One solution, which is suggested on StackOverflow, is to move the call `fs.readFileSync('file-.html')` inside the definition of the asynchronous event listener, so that it will not execute before the writing of the file has completed.

The question at this point is how the programmer could have observed that the code is buggy. Here, the key issue is that the programmer implicitly assumed that the read-operation on line 12 will always execute after the

```
var writing = fs.createWriteStream('video.mp4');
var stream = ytdl('https://www.youtube.com/watch?v=jofNR_WkoCE',
    { filter: function(format) {
                return format.container === 'mp4';
            },
      quality: "lowest"
    });
stream.pipe(writing);
var completed_len = 0;
var total_len = 0;
writing.on('data', function(chunk) {
    console.log('received data!');
    completed_len += chunk.length;
});
writing.on('close', function () {
    console.log('close');
    res.send('completed!');
});
```

Figure 11.2: A NodeJS program in which the programmer registers a listener with the wrong stream. See `http://stackoverflow.com/questions/19167407/`.

write-operation on line 6. The static analyses that we present in this paper can determine that no such ordering exists in this case, and can be used in the context of a bug-finding tool to alert programmers early to this kind of problem.

StackOverflow Question 19167407

Figure 11.2 shows another problematic NodeJS code fragment taken from StackOverflow. Here, the programmer's goal is to write an application that reads an `mp4` stream from YouTube, and write its contents to a local file. To this end, the programmer relies on the `ytdl()` function provided by the YouTube downloader module for NodeJS.

The program first creates a stream where the contents are to be written, on line 1. Then, on lines 2–7 a stream is created for reading the contents from YouTube. Next, on line 8, the contents of the latter stream are piped into the former.

After initializing some counters on lines 9 and 10, two listeners are registered with the write-stream. First, on lines 11–14 a listener is associated with `data` events to update the log and counter. Second, on lines 15–18, a listener is associated with `close` events to write a message to the log, and create an appropriate response.

The programmer reports that "Weirdly enough, even though 'close' fires whenever the download is done, I'm not getting any logs from the 'data'

event. The video is written correctly.". The problem (correctly diagnosed on StackOverflow) is that `data` events are only associated with readable streams, and not with writable streams. On lines 11–14, the programmer inadvertently bound the event listener to the wrong stream.

Our static analysis tool can detect this type of "dead listener" problem automatically by discovering that the `data` event is never emitted on the writable stream, and as a consequence, report lines 11–14 as unreachable.

11.3 Language

The focus of this paper is to develop program representations and analyses that are useful for understanding the behavior of event-based JavaScript programs.

JavaScript is a complex language with features such as prototype-based inheritance, dynamic property access, implicit coercions and on-the-fly code evaluation with `eval`. Much recent research effort has been invested in studying these features [Jensen et al., 2011; Feldthaus et al., 2011; Meawad et al., 2012; Jensen et al., 2012; Madsen and Andreasen, 2014; Kashyap et al., 2014]. However, surprisingly little research has focused on the event-driven nature of control flow in JavaScript applications. We wish to provide a formalization of how events interact with JavaScript, but without having to deal with the complicated features mentioned above. Therefore, we extend a minimal JavaScript calculus [Guha et al., 2010] with constructs for event handling.

Design Choices

We highlight some important design choices that were made when formalizing the semantics of events and listeners. These choices reflect similiar choices made by the NodeJS developers and of JavaScript in general.

- Execution is single-threaded and non-preemptive. An event listener must run to completion before another event listener may begin execution.
- An object may have multiple event listeners registered for the same event. The event listeners are executed in registration order, if that event is emitted.
- If the execution of an event listener f for an event τ registers an event listener f' for the same event τ then f' is not executed until τ is emitted again.
- The event names are drawn from a known finite set.[4]

[4] In NodeJS event names are strings which could be computed using string operations. Yet, we have not seen a single program with this behaviour.

Syntax of λ_ϵ

Guha et al. [2010] describe a core calculus for JavaScript called λ_{JS}. The syntax of this language is shown in Figure 11.3. The reduction semantics are provided in their paper and are largely unchanged, except as noted below. The language λ_{JS} has primitive values, objects, functions and references. The primitive values are booleans, numbers, strings, `null` and `undefined`. Values are primitive values, heap locations, objects or functions. Objects are immutable maps from fields to values. Functions are lambda expressions. The expressions are standard except for two operations that manipulate the heap: The `ref` and `deref` expressions store respectively retrieve values to and from the heap. JavaScript features that are not in λ_{JS} are compilable from JavaScript into λ_{JS} as shown in [Guha et al., 2010]. Thus λ_{JS} provides a minimal calculus which is still sufficiently expressible to model all aspects of JavaScript (except the `eval` expression).

We turn λ_{JS} into an event-based language λ_ϵ by introducing *events* and *listeners* into the syntax and semantics. Specifically, we introduce three new terms:

- **Listen:** $e_1.\texttt{listen}(\tau, e_2)$: An expression that registers an *event listener* for when τ is *emitted* on the receiver object. The expression e_1 is the receiver object on which the listener is registered, the expression e_2 is the event listener (i.e., a function value), and τ is the *event name*.

- **Emit:** $e_1.\texttt{emit}(\tau, e_2)$: Emits (i.e., triggers) an event on an object, which results in *scheduling* all event listeners *registered* for that event. If multiple event listeners are registered for the same event then they are *scheduled* in registration order. The expression e_1 is the address of the object, τ is the event name and e_2 is the argument passed to the event listener(s).

- **Loop:** $\bullet$: An expression which represents the "event-loop", i.e. the situation when the call stack is empty and scheduled event listeners are extracted from the event queue and executed. We assume, by transformation if necessary, that the last expression of the "top-level" function is $\bullet$. Intuitively, the purpose of $\bullet$ is to explicitly represent when event listeners are allowed to execute.

We call the extended language λ_ϵ. The new syntax is shown in Figure 11.4.

Runtime of λ_ϵ

The runtime of λ_ϵ consists of (1) a heap σ, which is a partial map from addresses to values, (2) a map of registered event listeners ϑ, which is a map from addresses and event names to a list of event listeners, and (3) a queue of

$$
\begin{array}{rcll}
c \in \mathit{Cst} & = & \mathit{bool} \mid \mathit{num} \mid \mathit{str} \mid \texttt{null} \mid \texttt{undef} & \text{[constant]} \\
\\
v \in \mathit{Val} & = & c & \text{[literal]} \\
& \mid & a & \text{[address]} \\
& \mid & \{\mathit{str} : v \cdots\} & \text{[object]} \\
& \mid & \lambda\,(x \cdots)\ e & \text{[function]} \\
\\
e \in \mathit{Exp} & = & v & \text{[value]} \\
& \mid & x & \text{[variable]} \\
& \mid & e = e & \text{[assignment]} \\
& \mid & \texttt{let}\ (x = e)\ e & \text{[binding]} \\
& \mid & e\,(e \cdots) & \text{[call]} \\
& \mid & e.f & \text{[field load]} \\
& \mid & e.f = e & \text{[field store]} \\
& \mid & \texttt{ref}\ e & \text{[address of]} \\
& \mid & \texttt{deref}\ e & \text{[value at]} \\
\\
x \in \mathit{Var} & = & \text{is a finite set of variable names.} & \\
f \in \mathit{Fld} & = & \text{is a finite set of field names.} & \\
a \in \mathit{Addr} & = & \text{is an infinite set of memory addresses.} & \\
\lambda \in \mathit{Lam} & = & \text{is the set of all lambda expressions.} &
\end{array}
$$

Figure 11.3: Syntax of λ_{JS}.

$$
\begin{array}{rcll}
e \in \mathit{Exp} & = & \bullet & \text{[no-op]} \\
& \mid & e.\texttt{listen}\,(\tau, e) & \text{[attach listener]} \\
& \mid & e.\texttt{emit}\,(\tau, e) & \text{[emit event]} \\
\\
\tau \in \mathit{Event} & = & \text{is a finite set of event names.} &
\end{array}
$$

Figure 11.4: Syntax of λ_{ϵ}.

scheduled functions π, which is also a map from addresses and event names to an ordered list of functions calls. (The calls are not *evaluated* until they are executed.) The runtime environment of λ_{ϵ} is depicted in Figure 11.6.

The queue π must maintain a separate list of listeners for each address and event name because no total ordering exists on the event listener executions. However, for each object and event, the order of listener executions is defined,

$$
\begin{array}{rcl}
E & = & \Box \\
& | & E.\texttt{listen}\,(\tau, e) \mid v.\texttt{listen}\,(\tau, E) \\
& | & E.\texttt{emit}\,(\tau, e) \mid v.\texttt{emit}\,(\tau, E)
\end{array}
$$

Figure 11.5: Evaluation Contexts for λ_ϵ. The $\Box$ symbol represents the hole in the evaluation context.

$$
\begin{array}{rcl}
\sigma \in \mathit{Heap} & = & \mathit{Addr} \to \mathit{Val} \\
\vartheta \in \mathit{Listener} & = & \mathit{Addr} \times \mathit{Event} \to \mathit{Lam}^\star \\
\pi \in \mathit{Queue} & = & \mathit{Addr} \times \mathit{Event} \to (\mathit{Lam} \times \mathit{Val})^\star \\
s \in \mathit{State} & = & \mathit{Heap} \times \mathit{Listener} \times \mathit{Queue} \times \mathit{Exp}
\end{array}
$$

Figure 11.6: Runtime of λ_ϵ.

so a single global unordered set of pending listener executions would be imprecise.

A configuration is a 4-tuple $c = \langle \sigma, \vartheta, \pi, e \rangle$ consisting of the heap, the event listeners, the queue and the current expression.

Semantics of λ_ϵ

The semantics of λ_ϵ is defined by adding four new rules to those in [Guha et al., 2010]. As in [Guha et al., 2010], we use *evaluation contexts* [Felleisen et al., 2009] to specify the order in which sub-expressions are evaluated in expressions. An evaluation context is tree with a hole. The hole is denoted by $\Box$ and represents where in the expression evaluation should proceed. The evaluation contexts for the new constructs are shown in Figure 11.5.

The semantics for the new terms are in Figure 11.7 and explained below:

[E-CTX] : This rule uses the evaluation context to evaluate sub-terms and re-compose them. That is, if some term e_1 can evaluate to e_2 in one or more steps, and e_1 occurs in the hole according to the evaluation context, then we can plug the hole with e_2.

[E-LISTEN] $o.\texttt{listen}(\tau, f)$: Registers f as an event listener for event τ on the object pointed-to by o, if o is an address and f is a function value.

[E-EMIT] $o.\texttt{emit}(\tau, v)$: Adds all event listeners which are registered for event τ on the object pointed-to by o to the event queue. The v value is used as argument to each of the event listeners.

$$\frac{\langle \sigma, \vartheta, \pi, e\rangle \hookrightarrow^\star \langle \sigma', \vartheta', \pi', e'\rangle}{\langle \sigma, \vartheta, \pi, E[e]\rangle \hookrightarrow \langle \sigma', \vartheta', \pi', E[e']\rangle} \quad \text{[E-CTX]}$$

$$\frac{o \in \mathit{Addr} \qquad f = \lambda\,(x \cdots)\ e \qquad \vartheta' = \vartheta[(o, \tau) \mapsto f :: \vartheta(o, \tau)]}{\langle \sigma, \vartheta, \pi, o.\mathtt{listen}(\tau, f)\rangle \hookrightarrow \langle \sigma, \vartheta', \pi, \mathtt{undef}\rangle} \quad \text{[E-LISTEN]}$$

$$\frac{o \in \mathit{Addr} \qquad \vartheta(o, \tau) = \lambda_1 :: \cdots :: \lambda_n \qquad \pi' = \pi[o \mapsto (\lambda_1, v) :: \cdots :: (\lambda_n, v) :: \pi(v)]}{\langle \sigma, \vartheta, \pi, o.\text{emit}(\tau, v)\rangle \hookrightarrow \langle \sigma, \vartheta, \pi', \mathtt{undef}\rangle} \quad \text{[E-EMIT]}$$

$$\frac{o \in \mathit{Addr} \qquad \pi(o, \tau) = (\lambda_1, v_1) :: \cdots :: (\lambda_n, v_n) \qquad e = \lambda_n(v_n); \ldots ; \lambda_1(v_1) \qquad \pi' = \pi[(o, \tau) \mapsto \mathit{Nil}]}{\langle \sigma, \vartheta, \pi, \bullet\rangle \hookrightarrow \langle \sigma, \vartheta, \pi', e; \bullet\rangle} \quad \text{[E-LOOP]}$$

Figure 11.7: Semantics of λ_ϵ. Here the notation $x :: xs$ is cons and *Nil* denotes the empty list.

[E-LOOP] $\bullet$: Non-deterministically chooses an address o and an event τ and executes all event listeners scheduled for that object and event by moving them from the queue into the current expression as a sequence of statements. Notice that listeners are executed in registration order (since π is maintained in reverse-registration order).

Other Event Related Features

Asynchronous Callbacks. Asynchronous callbacks, as shown in Figure 11.8, can be expressed in terms of the primitives that we have defined for λ_ϵ. To express a function f that asynchronously executes another function g, we make f allocate a fresh object, register function g on that object as a listener for some event, and emit that event on the object:

```
function f(g) {
        var x = new Object(); // fresh object
        x.listen("e", g); // register g for event "e"
        x.emit("e"); // emit "e" on the object
}
```

Listener Unregistration. NodeJS supports unregistration of event listeners. However, this feature is used rarely because event listeners typically follow

```
1 var fs = require("fs");
2 fs.rename('a.txt', 'b.txt', function (err) {
3   fs.stat('b.txt', function (err, stats) {
4     console.log('stats: ' + stats);
5   });
6 });
```

Figure 11.8: An example of asynchronous callbacks. The call to `fs.rename` registers an asynchronous callback that is executed at some point in the future when the `rename` operation has completed. In the callback function the call to `fs.stat` registers a new callback which is provided with filesystem information about the renamed file.

the lifetime of an object by being registered immediately after the object is created and then never removed. In all code we have looked at, we have not yet encountered a situation where unregistering listeners was needed. If necessary, this could be supported by a straightforward extension to λ_ϵ semantics.

11.4 Beyond Call Graphs

A (traditional) call graph is a directed graph that connects *call sites* with *call targets* (i.e., function declarations). Call graphs are useful for debugging, refactoring, and many other applications. In languages with polymorphism and/or higher-order functions, the call graph is not immediately available from the source code, but must be statically approximated, e.g., using points-to analysis. Traditionally, a call graph helps the programmer answer two key questions:

- Who are the *callers* of a function? (i.e. "who calls me")
- Who are the *callees* of a function? (i.e. "who do I call?")

However, in languages with asynchronous callbacks and event listeners a traditional call graph provides incomplete information because it does not reflect precisely how events give rise to indirect calls.

In a traditional call graph, the callers of function λ are the functions that may invoke it. However, in event-based systems, control flow is also determined by: (i) functions that may *register* a function λ as a listener, and (ii) functions that may *emit* an event that causes λ to be scheduled for execution. In traditional call graphs, this correlation between events and listeners is lost, and functions that are invoked due to event-handling are typically modeled as "roots" of a call graph (i.e., call-backs from a runtime environment).

Event-Based Call Graphs

We overcome the limitations of traditional call graphs by introducing the *event-based call graph*, a directed graph whose nodes are expressions and lambdas from the source code, and which has four kinds of edges. A *call edge* connects a call expression to a function declaration λ:

$$\{e\,(e \cdots) \rightarrow \lambda \mid \text{if the call may invoke } \lambda\}$$

As stated earlier, the call edges answer the questions: (i) which functions directly invoke a function? and (ii) which functions are directly invoked by a function? A *listen edge* connects a listen expression to a function λ:

$$\{e_1.\texttt{listen}(\tau, e_2) \xrightarrow{\tau} \lambda \mid \begin{array}{l}\text{if the } \texttt{listen}\text{-expression} \\ \text{may register function } \lambda\end{array}\}$$

From such listen edges, the following questions can be answered: (i) Which functions does a given listen expression register as listeners for a given event? (ii) Which listen expressions may register a given function λ as a listener for an event? An *emit edge* connects an emit expression to a function declaration λ:

$$\{e.\texttt{emit}(\tau) \xrightarrow{\tau} \lambda \mid \text{if event } \tau \text{ may schedule } \lambda\}$$

From the emit edges in an event-based call graph, the following questions can be answered: (i) which functions does an emit expression schedule for execution by emitting the event τ? and (ii) which emit expressions may cause a given function to be scheduled for execution?

Finally, we introduce *may-happen-before* edges between expressions:

$$\{(e_1, e_2) \mid \text{if } e_1 \text{ may happen before } e_2.\}$$

where e_1 and e_2 are either `listen` or `emit` expressions. Here, e_1 may possibly execute before e_2 *only if there is an edge* (e_1, e_2). If, at the same time, there is the edge (e_2, e_1) then e_1 and e_2 may execute in any order. Crucially, if there is no (e_2, e_1) edge then e_1 *cannot be executed before* e_2.

We can use these edges to decide whether an event listener is executed. Specifically, in order for an event listener λ to be executed, three edges must be present:

1. A listen edge, for an event τ, from a `listen` expression to λ.
2. An emit edge, for an event τ, from an `emit` expression to λ.
3. A may-happen-before edge from the `listen` expression to the `emit` expression.

If any of these three edges are missing, for any listener λ, then the listener can never be executed.

```
1  let (x = {})
2    let (y = {})
3      let (f = λ0() x.listen(τ3, λ3 () ...))
4        x.listen(τ1, λ1() y.emit(τ2));
5        y.listen(τ2, λ2() f());
6        x.emit(τ1)
```

Figure 11.9: A small λ_ϵ program. The program contains three event listeners λ_1, λ_2 and λ_3 which are registered for the events τ_1, τ_2 and τ_3, respectively.

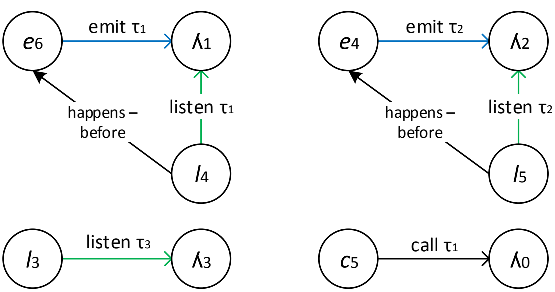


Figure 11.10: The event-based call graph for the λ_ϵ program in Figure 11.9. The call graph shows that λ_1 and λ_2 are executed due to the events τ_1 and τ_2, respectively. Furthermore, λ_0 is executed due to a regular function call, and λ_3 is never executed since the event τ_3 is never emitted.

Example. Figure 11.9 shows a small program with calls, listen and emit expressions. Figure 11.10 shows the event-based call graph for this program. It has a single *call* edge from the call site `f()` on line 5 to λ_0 on line 3. It has two *emit* edges: one from the emit on line 4 to λ_2 on line 5 and one from the emit on line 6 to λ_1 on line 4. It has three *listen* edges: one from line 3 to λ_3, one from line 4 to λ_1, and one from line 5 to λ_2.

The event-based call graph of Figure 11.10 also contains several may-happen-before edges that are implied by the order in which the various expressions are executed. For instance, the `listen` on line 4 *may-happen-before* the `emit` on line 6, since the lines 4, 5 and 6 are executed in sequence.

Bug Finding

The event-based call graph enables us to discover a range of interesting program behaviors and potential bugs:

- **Dead Listeners:** We can detect situations where an event listener is registered for an event τ on an object o, but the event τ is never emitted

on o. This can happen for several reasons: (a) the event listener could be registered on the wrong object, (b) the event listener might be registered for the wrong event, (c) the event might be emitted *before* the event listener is registered, and never emitted again.

- **Dead Emits:** We can detect situations where an event τ is emitted on an object, but that object has no event listener for τ. Such an event might be emitted by the application code or by a framework such as NodeJS. If the event is emitted by the application, then the absence of a listener would seem unintentional and potentially indicates the presence of a bug. If, on the other hand, the event is emitted by the framework, then whether the event must be handled depends on the semantics of the event.

- **Mismatched Synchronous/Asynchronous Calls:** In NodeJS, most I/O operations are asynchronous, but for some operations, synchronous variants are also provided. For instance, reading a file can be done asynchronously with `readFile` and synchronously with `readFileSync`. If a file is accessed both synchronously and asynchronously there is likely a race condition. We can detect such races by inspecting the may-happen-before relation.

- **Mixed Calls**: We can detect functions that are executed by both regular function calls and as event listeners. A function that is used as an event listener, but also called directly, likely indicates that the programmer forgot that the function is supposed to be called asynchronously.

- **Unreachable Functions**: We can detect functions that are unreachable, i.e., never executed directly or indirectly by the event system. Such functions are dead code and can be removed from the program safely.

We observe that many of these properties require flow-sensitivity to determine the presence (or rather the absence) of a bug. For instance, an event listener is dead if the `listen` statement happens after the `emit` statement.

Example. Figure 11.11 shows a buggy λ_ϵ program. Figure 11.12 shows the event-based call graph for the program of Figure 11.11. The graph reveals several bugs in the program: First, λ_1 has an incoming listen edge, but no emit edge, thus it is a *dead listener*. Second, the emit expression e_4 has no outgoing emit edge(s), thus it is a *dead emit*. Third, λ_2 has both incoming listen and emit edges, but the emit e_3 happens before the listen l_5, thus e_3 is a *dead emit* and l_5 is *dead listener*.

```
1  let (x = {})
2    x.listen(τ1, λ1 () ...);
3    x.emit(τ2);
4    x.emit(τ3);
5    x.listen(τ2, λ2 () ...)
```

Figure 11.11: A buggy λ_ϵ program.

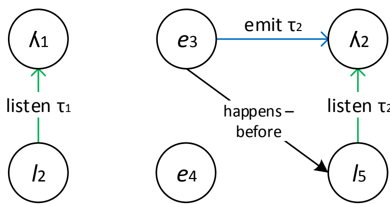


Figure 11.12: An event-based call graph for Figure 11.11.

11.5 Analysis Framework

Static Analysis. The exact call graph of a given program is uncomputable in general. We evaluate a static analysis framework for λ_ϵ that soundly over-approximates the exact event-based call graph. If a call, listener registration, or event emission occurs in some concrete execution, then the call graph must include it. However, the call graph may contain *spurious* behaviour that does not actually occur in any execution. An analysis is said to be more precise than another if its output contains less spurious behaviour. In the case of analyses that construct event-based call graphs, this means that an analysis is more precise than another if it produces a call graph with fewer edges. For example, two emit edges $\texttt{emit}_1 \hookrightarrow \lambda$ and $\texttt{emit}_2 \hookrightarrow \lambda$ in a generated call graph indicate that either $\texttt{emit}_1$ or $\texttt{emit}_2$ may cause λ to be scheduled. However, a call graph that contains only $\texttt{emit}_1 \hookrightarrow \lambda$ guarantees that only $\texttt{emit}_1$ causes λ to be scheduled.

Design Motivation. Based on our experience with NodeJS programs, we make three observations that guide the design of the analysis framework:

Observation 1: It is common for one event listener to register another. A very common pattern is to perform some asynchronous I/O operation first. Then, when that operation completes, another I/O operation is performed, and so on, until the entire computation is complete.

Observation 2: It is common for event names to overlap between different objects. In NodeJS the event names `connection`, `data` and `finished` occur in

$$
\begin{aligned}
\widehat{\sigma} \in \widehat{Heap} &= \widehat{Addr} \to \widehat{Val} \\
\widehat{\vartheta} \in \widehat{Listener} &= \widehat{Addr} \times Event \to \mathcal{P}(Lam) \\
\widehat{\pi} \in \widehat{Queue} &= \widehat{Addr} \times Event \to \mathcal{P}(Lam \times \widehat{Val}) \\
\widehat{s} \in \widehat{State} &= \widehat{Heap} \times \widehat{Listener} \times \widehat{Queue} \times Exp \\
\ell \in Lattice &= Ctx \times Loc \to \widehat{State} \\
c \in Ctx &= \text{is a set of contexts.} \\
l \in Loc &= \text{is a set of source code locations.}
\end{aligned}
$$

Figure 11.13: Abstract runtime of λ_ϵ.

many contexts.

Observation 3: It is uncommon for a single object to have multiple event listeners registered for the same event.

Abstract Semantics. The static analysis framework is based on a parameterized abstraction of the concrete semantics of λ_ϵ presented in Figure 4.2. The abstract semantics is shown in Figure 11.13. The abstractions of runtime addresses *Addr* and primitive values *Val* are parameters that can be instantiated with different abstract domains. Our implementation uses the allocation site abstraction to abstract memory addresses and the constant propagation lattice to abstract values. Further details are provided in Section 11.6.

The most interesting aspect of the design space is how to abstract the map of registered listeners and the event queue. Recall that the map of registered listeners maintains, for every object and event, the order in which the listeners were added. Similarly, the event queue map maintains, for every object and event, which listeners are scheduled for execution.

We choose a pragmatic abstraction that replaces both sequences by sets, forgetting the order in which event listeners were registered, or listeners are scheduled. However, we do maintain separation between different event listeners (and their associated event) and their execution.

Discussion. We sketch a few alternative abstractions and discuss why we ultimately decided on the current one. A more precise alternative would maintain the two sequences up to some bound, e.g., the order of the first k event listeners to be scheduled or executed. Such an abstraction would be more precise, since it can distinguish between the order in which event listeners are executed for the same object and event. On the other hand, it is less common in practice to have multiple listeners registered for the same event and object. A less precise abstraction could ignore on what specific object or event an event

```
let (x = {})
    x.listen(τ1, λ1 () x.emit(τ2));
    x.listen(τ2, λ2 () ...);
    x.emit(τ1)
```

Figure 11.14: A small λ_ϵ program which illustrates the increased precision due to *event sensitivity*.

```
let (x = {})
    x.listen(τ, λ1 ()
        x.listen(τ, λ2 () ...);
        x.emit(τ)
    );
    x.emit(τ)
```

Figure 11.15: A small λ_ϵ program which illustrates the increased precision due to *listener sensitivity*.

listener is registered. However, this would immediately lead to imprecision since different objects commonly share the same event names in NodeJS.

Context-Sensitivity Policies. In a NodeJS program, after the execution of initialization code, the runtime enters an event loop that repeatedly identifies events that have occurred, and schedules and executes corresponding listeners. To understand the temporal behavior of a NodeJS program, it is important to understand how the event loop behaves in different contexts. For example, suppose that in a given program, listener A registers listener B on an event. A simplistic understanding would be that the event loop can call both listeners at any time. To identify that A must execute first, the analysis must consider the event loop first in the context before B has been registered, and separately in the context after B has been registered. To evaluate the precision of different context abstractions, the analysis framework is parameterized by a context-sensitivity policy. Context selection is driven by emitted events and registered event listeners. We have evaluated the following 3 context-sensitivity policies:

1. Baseline Analysis. The *baseline analysis* ignores any dependencies between registered event listeners and emitted events. It collapses the event loop into a single program point where all dataflow is merged. In effect, this analysis assumes that listeners can run at any time, in an arbitrary order, and any number of times. The baseline analysis has only a single context $Ctx = \circ$. This analysis approach is inexpensive, but the precision benefits of flow sensitivity are lost.

2. Event-Sensitive Analysis An *event-sensitive* analysis separates dataflow based on which listeners have executed so far due to emitted events. This analysis captures the order in which event listeners, for different objects/events, are extracted from the (abstract) event queue $\widehat{\pi}$ and executed. In the event-sensitive analysis, a context consists of a set of object and event pairs:

$$c = \{(o, \tau) \mid \tau \text{ has been emitted on object } o\}$$

For example, the context $\{(\tau_1, o_1), (\tau_1, o_2), (\tau_2, o_2)\}$ indicates that event τ_1 was emitted on objects o_1 and o_2, that τ_2 was emitted on o_2, and that no other events were emitted. The *event sensitive* context is given by the power set:

$$Ctx = \mathcal{P}(\widehat{Addr} \times Event)$$

where the initial context is the empty set.

Example. Figure 11.14 shows a small λ_ϵ program that illustrates the increased precision due to *event sensitivity*. Here, two event listeners λ_1 and λ_2 are registered for events τ_1 and τ_2 on some object. The evaluation of λ_1 emits an event that causes λ_2 to be scheduled. Thus, all side-effects of λ_1 precede λ_2. In the baseline analysis, this correlation is lost but the event sensitive analysis preserves the correlation by returning to the event loop in the context: $\{(\tau_1, o_1)\}$.

3. Listener-Sensitive Analysis. A *listener-sensitive* analysis separates dataflow based on which listeners have been registered. In the listener-sensitive analysis, a context is a set of triples of an object o, an event τ, and a listener λ that is registered on that object and for that event:

$$c = \{(o, \tau, \lambda) \mid \lambda \text{ is registered on object } o \text{ for event } \tau\}$$

The *listener sensitive* context is given by the power set:

$$Ctx = \mathcal{P}(\widehat{Addr} \times Event \times Lam)$$

where the initial context is the empty set.

The strength of the listener sensitive analysis is its ability to track listeners that register other listeners as part of their execution. But in general the precision of the event- and listener-sensitive analyses is incomparable.

Example. Figure 11.15 shows an example demonstrating the benefits of listener-sensitive analysis. Here, two listeners for the same event τ, are registered, one following the other. The execution of λ_1 causes λ_2 to be registered

for the same event, and on the same object, as λ_1. The baseline and event-sensitive analyses cannot capture that λ_1's execution is known to precede λ_2's. This correlation, however, is captured by listener sensitivity: The event loop is unfolded into the two contexts: $\{(o_1, \tau, \lambda_1)\}$ and $\{(o_1, \tau, \lambda_1), (o_1, \tau, \lambda_2)\}$

11.6 Evaluation

In this section, we discuss a tool that implements the analyses described in Section 11.5, and report on experiments in which the tool is applied to small NodeJS programs.

Implementation

We have implemented a static analysis tool, RADAR, for NodeJS applications written in JavaScript. RADAR can detect errors such as the ones discussed in Section 11.2 and supports the full JavaScript language, including higher-order functions, prototype inheritance and dynamic property access. Similar to other work on static analysis for JavaScript, RADAR does not support implicit calls to user-defined `toString` functions, and the `with` and `eval` constructs.

The implementation is approximately 27,000 lines of Scala code, of which around 2,000 lines are related to the event system. The implementation has been used in previous work on dynamic field access [Madsen and Andreasen, 2014] and sparse dataflow analysis [Madsen et al., 2014]. All experiments were performed on a Intel Core i5-4300U 2.5GHz CPU with 2GB memory allocated for the JVM.

Analysis Details. In essence, our analysis is a flow-sensitive dataflow analysis. Flow-sensitivity means that the order of statements is respected, which is important for determining *may-happen-before* relationships between, e.g., `emit` and `listen` statements. The analysis incorporates a field-sensitive subset-based points-to analysis. Field-sensitivity means that the points-to analysis distinguishes between the values of fields with different names. The points-to analysis models the (potentially) unbounded heap using the allocation site abstraction, meaning that objects allocated at the same location in the source code are represented using an abstract summary object. The analysis constructs the event-based call graph on-the-fly due to both indirect calls and events. Specifically, the base object(s) of `emit` and `listen` statements are not known ahead-of-time, but gradually discovered as the points-to graph is resolved.

NodeJS Model. We model the NodeJS framework with JavaScript stubs that use the `listen` and `emit` constructs. This has three consequences: First, it demonstrates that our framework is sufficiently powerful to capture the

Module	Lines	Functions	`listen`	`emit`
Filesystem	508	73	31	31
Http	321	45	24	13
Network	328	39	31	36
Stream	111	12	2	19
Total	1,268	169	86	96

Table 11.1: The NodeJS model. The four columns show, for each module, the number of lines of code, functions, and listen and emit statements in the stub model.

semantics of NodeJS. Second, it makes the analysis simpler to implement since events in the application and the library can be treated uniformly. Third, it makes it easy to model other event-based libraries such as `socket.io` and `async.js`.

NodeJS itself consists of around thirty modules. Of these, we have modeled `filesystem`, `http`, `network` and `stream`, which make the most heavy use of events; Modeling other modules is straightforward. Table 11.1 shows some statistics about the models. For instance, the model of the `http` module has 45 functions with 24 `listen` and 13 `emit` statements. In total, the four modules define 169 functions containing 86 listen and 96 emit statements.

In our model of the `http` and `network` modules we make the simplifying (but unsound) assumption that only a small number of `net.socket` objects exist at the same time. This is necessary for the analysis to accurately track the state of each socket. We stress that the analysis framework is sound and the rest of the analysis aims for soundness.

Research Questions

We evaluate RADAR based on two research questions:

Q1: Is the tool capable of detecting event-related bugs?

Q2: How many false positives does the tool report?

Sections 11.6 and 11.6 below address these research questions in detail, for each of the *baseline*, *event-sensitive* and *listener-sensitive* analyses.

Q1: Finding Bugs

In order to determine whether RADAR can help the programmer find event-related bugs, we apply it to buggy program fragments obtained from Stack-Overflow, a popular question-and-answer forum for software developers. In

Program	Baseline		Event		Listener	
	Orig.	Fixed	Orig.	Fixed	Orig.	Fixed
StackOver. 11790224	✓	✗	✓	✓	✓	✓
StackOver. 13338350	✓	✗	✓	✓	✓	✓
StackOver. 16903844	✓	✓	✓	✓	✓	✓
StackOver. 17894000	✓	✓	✓	✓	✓	✓
StackOver. 18295923	✓	✓	✓	✓	✓	✓
StackOver. 19081270	✓	✗	✓	✓	✓	✓
StackOver. 19167407	✓	✓	✓	✓	✓	✓
StackOver. 19171045	✓	✓	✓	✓	✓	✓
StackOver. 19342910	✓	✓	✓	✓	✓	✓
StackOver. 23437008	✓	✓	✓	✓	✓	✓
StackOver. 25650189	✓	✗	✓	✓	✓	✓
StackOver. 26061335	✓	✗	✓	✓	✓	✓

Table 11.2: Twelve buggy program fragments from the StackOverflow. A ✓ in the `Orig.` column means that the analysis correctly reported the bug for the original program. A ✗ in the `Fixed` column means that the analysis still reported a (spurious) warning for the fixed (corrected) program.

each case, the programmer supplied a program fragment and a description of the intended behavior. We selected twelve questions based on the criteria that the program had to contain a single event-related bug. We evaluate each analysis by applying it to the program and observing:

1. Does the analysis report a warning? How useful is the warning for finding the bug?
2. Does the warning disappear once the program has been corrected? If the warning remains, due to analysis imprecision, then that may confuse the programmer.

Table 11.2 shows the results for each analysis. The most interesting program fragments are discussed further below:

- In Q17894000, entitled "event handling not working as expected nodejs", the programmer creates a socket, registers a listener for the `data` event and inside that listener registers another listener for the `close` event. The programmer then expects the listener for the `close` event to be executed. However, it is permissible for a socket to be closed before any data is

			Baseline			Event-Sensitive			Listener-Sensitive		
Program	Lines	Nodes	Listener	Mhb	Time	Listener	Mhb	Time	Listener	Mhb	Time
Filesystem	60	186	1	3	0.8s	0	2	0.9s	0	2	0.9s
Http	250	618	10	66	2.5s	–	–	–	0	41	3.0s
Network #1	200	546	8	61	2.2s	–	–	–	1	38	3.7s
Network #2	170	463	6	54	1.7s	0	33	22.1s	0	33	1.6s
Node in Action, ch. 3	330	668	8	49	2.5s	–	–	–	1	27	6.0s
Node in Action, ch. 4	390	873	10	81	3.5s	–	–	–	3	66	17.0s

Figure 11.16: Analysis results for 6 programs; 4 from [Dahl, 2014] and 2 from [Cantelon et al., 2014]. The `Lines` and `Nodes` columns show the number of source code lines and nodes in the control-flow graph for each program. The `Listener`, `Mhb` and `Time` columns show, for each analysis, the number of dead listeners reported, the number of may-happen-before relations and the total analysis time. The double dashed line, –, indicates that the analysis timed out after 60 seconds.

transmitted, and thus there is no guarantee that the `close` listener is ever registered. Each of the three analyses identifies the problem by warning that `close` may be emitted before the listener is registered. Moving the listener registration for the `close` event out of the listener for the `data` event fixes the problem, and none of the three analyses report any spurious warnings after the fix.

- In Q18295923, entitled "nodejs stdin readable event not triggered", the programmer registers a listener for the `readable` event on the standard-in `process.stdin` stream. The programmer then expects the listener to be executed when the user writes to standard-in. However, the `process.stdin` stream is by default in a *paused* mode where the programmer must call `resume()` before the `readable` events are emitted. All three analyses report a dead listener, and no spurious warning on the fixed program.

- In Q19081270, entitled "Why my fs.readFileSync does not work", the program writes a file asynchronously, but then immediately reads back the file synchronously, before the asynchronous operation has had a chance to complete. (NodeJS provides both asynchronous and synchronous variants of many operations, and it is easy for programmers to get confused and make mistakes such as this one). The analyses do not have any knowledge of the external world (e.g., the file system), so they cannot directly pinpoint the error. They can, however, report that the read always occurs before the write, which can help the programmer understand the problem. The recommended fix is to register an asynchronous listener that executes when the write completes. This solves the problem, but the baseline analysis still reports that the read may execute before the write, since it cannot accurately track the may-happen-before dependencies.

- In Q19167407, with the title "Data event not firing NodeJS", the programmer creates a readable stream and a writable stream, and pipes data from the readable stream to the writable stream. The programmer registers listeners for the `data` and `close` events on the writable stream. However, the listener for the `data` event is never executed. The problem is that only the readable stream emits `data` events, not the writable stream: The programmer has registered the listener on the wrong object. All three analyses report a dead listener and report no spurious warnings when applied to the fixed program.

- In Q19342910, entitled "when is the `connect` event in nodejs net module emitted?", the programmer creates a TCP server object passing in a listener function. Inside that function the programmer registers a listener

for the connect event. However, the inside listener is never executed. The problem is that function, which is passed during the construction of the TCP server object, is implicitly registered for the connect event. Thus, once the connect event is emitted, the outer function executes, which then registers a listener for the very same event. Since the connect event is only emitted once (per connection), the inner listener is never executed. All three analyses report the inner listener as dead and no spurious warnings on the fixed program.

We conclude that the analyses appear to be useful for finding event-related bugs in small NodeJS programs. However, the baseline analysis reports many spurious warnings.

Q2: Spurious Warnings

To answer this question, we apply each analysis to 6 programs; 4 from the online NodeJS API documentation and 2 from the book: *Node.js in Action*. The programs from the NodeJS documentation illustrate the filesystem, http and network modules. The *Node.js in Action* programs show how to perform a HTTP request and create a small HTTP server. Although these programs are small, their sizes are consistent with benchmarks used in recent literature on flow-sensitive, subset-based points-to analysis for JavaScript [Jensen et al., 2011; Madsen et al., 2014; Kashyap et al., 2014]. We report, for each analysis, the number of warnings for dead listeners, the number of may-happen-before relations and the analysis time. The results are shown in Table 11.16.

As an example, the “Filesystem” example is 60 lines of code (for the program and relevant parts of the NodeJS model). The baseline analysis reports one dead listener and three may-happen-before edges in 0.8 seconds, both the event-sensitive and listener-sensitive analyses report no dead listeners and two may-happen-before edges in 0.9 seconds.

The event-sensitive analysis times out on 4 benchmarks and runs slowly on another. An explanation for the poor performance, compared to the listener-sensitive analysis, is that it unnecessarily separates abstract states. E.g., if the event queue contains $\{\tau_1, \tau_2\}$, the event-sensitive analysis may separate dataflow for the states $\{\tau_1\}$, $\{\tau_2\}$ and $\{\tau_1, \tau_2\}$, even if the same listeners are present in each case. This can lead to an exponential blow-up in the number of states.

The listener-sensitive analysis offers a compelling alternative. Compared to the baseline analysis, the number of may-happen-before edges decreases from 314 to 207, showing that flow-sensitive precision is increased significantly, resulting in a reduction in the number of warnings from 43 to 8. Analysis

times increase by a factor from 1.0x to 5.0x. Thus, for a reasonable increase in analysis time, the number of spurious warnings is reduced significantly.

We examined the remaining spurious warnings reported by the listener-sensitive analysis, and found two causes:

- We model asynchronous callbacks by creating a fresh object, registering the callback on that object for some fixed event τ, and then emitting τ. This has the consequence that if the same function is used twice, e.g., `fs.readFile`, then the object becomes summarized and we cannot determine that the registration always happens before the emit. A possible solution is to use heap sensitivity to separate allocation sites depending on which event listeners are registered.

- We chose to abstract the *listener ordering* for event listeners registered on the *same object* and for the *same event* with a set. Thus, if λ_1 is registered for some event τ and λ_2 is later registered for the same event, we lose the ordering between λ_1 and λ_2. We found one case where this scenario cause a loss of precision.

In summary, the listener-sensitive analysis is preferable to the baseline- and event-sensitive analyses; the baseline analysis produces too many spurious warnings whereas the event-sensitive analysis has poor performance.

11.7 Related Work

To our knowledge, little prior work exists on whole-program *static analysis* of *event-based* JavaScript code. We discuss two categories of related work: static analysis of JavaScript, and dynamic analysis of event-based JavaScript code.

Static Analysis of JavaScript. Guarnieri and Livshits [2009] present GATEKEEPER, a tool for enforcing security policies for JavaScript widgets. GATEKEEPER used one of the first points-to analyses for JavaScript to discover dataflow disallowed by a security policy.

Jensen et al. [2011] present a dataflow analysis, TAJS, for browser-based JavaScript applications. The focus is on how to represent the DOM, but events are discussed briefly. Their analysis conservatively assumes that any browser event, e.g., `onclick`, may execute at any time and makes no attempt, other than for *onload*, to separate dataflow based on what listeners are registered. By contrast, our baseline analysis is more precise, as it only considers events that are actually emitted. More recently, TAJS was extended to handle certain cases of `eval` [Jensen et al., 2012] and to analyze the popular jQuery library [Andreasen and Møller, 2014].

Zheng et al. [2011] present a static analysis for detecting race conditions related to asynchronous AJAX requests. Specifically, a race condition exists when an event listener may read a global variable, at any time, and an asynchronous response listener may write to the same global variable. In relation to our work, Zheng et al. are not concerned about the *specific* order in which event listeners are executed, but the existence of any order that could potentially cause a race.

Dynamic Analysis of Event-Based JavaScript Programs. Artzi et al. [2011] present a framework for feedback-directed random testing of JavaScript applications. This framework keeps track of event-handler registrations and attempts to increase code coverage by generating sequences of events likely to execute unexplored code. Artzi's work aimed to find execution errors and situations where malformed HTML was created, and did not consider event-handling related errors.

Petrov et al. [2012] define a *happens-before* relationship on the various kinds of operations performed by web applications (HTML parsing, access to variables and DOM nodes, event-handler execution). A notion of logical memory locations is defined, to abstract both JavaScript heap locations and locations in a browser-specific native data structures. A *data race* is defined as a situation where two operations access the same logical memory locations, at least one of these accesses is a write, and no ordering exists between the operations. Petrov et al. implemented WebRacer, a dynamic detector for such races, and used WebRacer to find races in sites of Fortune 500 companies. Petrov's happens-before relation includes orderings between operations corresponding to `emit` and `listen` edges, and the event-dispatch races detected by WebRacer resemble our StackOverflow examples.

Raychev et al. [2013] observe that race detection techniques such as the one of Petrov et al. report an overwhelming number of races in cases where event handler execution is coordinated via shared variables. They reduce the number of false positives by introducing a notion of *race coverage* where a race a covers a race b if treating a as synchronization eliminates b as a race. Raychev et al. implemented their work in a tool called EventRacer, and showed that, by focusing the user's attention on uncovered races, the number of issues that need to be considered is reduced dramatically.

The position paper by Mutlu et al. [2014] is focused on *observable races* in web applications that have visually apparent symptoms (e.g., missing elements in a user-interface). To this end, Mutlu et al. employ various strategies to instrument an application so that delays are inserted at each XMLHttpRequest. By comparing the screenshots generated in each case, observable races are detected.

Hong et al. [2014] present WAVE, a framework for detecting various concurrency-related errors in client-side JavaScript code. Here, an execution is monitored to create an execution model. From this model, test cases are generated that permute the order of operations in the original execution. A problem is reported if a test raises an exception, does not terminate, or produces a result different from that of the original execution. Among the types of problems that WAVE detects are data races like the ones found by EventRacer [Raychev et al., 2013], and atomicity violations.

11.8 Conclusion

We have introduced the *event-based call graph* and shown that it is useful for finding various event-related bugs in event-based NodeJS applications. We have designed and implemented three analyses for computing event-based call graphs. Experiments show that our analyses are capable of detecting bugs related to event-handling on small buggy programs from StackOverflow and that the number of false positives appears to be managable.

12

String Analysis for Dynamic Field Access

By Magnus Madsen and Esben Andreasen published in proc. 23rd International Conference on Compiler Construction (CC '14).

Abstract

In JavaScript, and scripting languages in general, dynamic field access is a commonly used feature. Unfortunately, current static analysis tools either completely ignore dynamic field access or use overly conservative approximations that lead to poor precision and scalability. We present new string domains to reason about dynamic field access in a static analysis tool. A key feature of the domains is that the equal, concatenate and join operations take $\mathcal{O}(1)$ time.

Experimental evaluation on four common JavaScript libraries, including jQuery and Prototype, shows that traditional string domains are insufficient. For instance, the commonly used constant string domain can only ensure that at most 21% dynamic field accesses are without false positives. In contrast, our string domain $\mathcal{H}$ ensures no false positives for up to 90% of all dynamic field accesses. We demonstrate that a dataflow analysis equipped with the $\mathcal{H}$ domain gains significant precision resulting in an analysis speedup of more than 1.5x for 7 out of 10 benchmark programs.

12.1 Introduction

JavaScript is a notoriously difficult language for static analysis due to its many dynamic features, including a flexible object-model, prototype-based inheritance, dynamic property accesses[1], non-standard scope rules, coercions, and the `eval`-construct [Crockford, 2008; Maffeis et al., 2008; Guarnieri and Livshits, 2009; Chugh et al., 2009; Jensen et al., 2009, 2011].

This paper focuses on the problem of dynamic property accesses in points-too dataflow analysis of JavaScript, that is, reads or writes to objects where the property names are computed on-the-fly. This involves statements such as `v = o[p]` or `o[p] = v` where the value of `p` is not statically known. A simple sound approach is to treat the first statement as a read of *any* property of `o` and the second statement as a write to *all* properties of `o`. However, such an approach loses the benefits of field-sensitivity. And, as the following sections illustrate, it is too imprecise in practice. In JavaScript, string manipulations and dynamic property accesses are common, and to paraphrase an old mantra:

> *"One man's string is another man's heap location".*

Dynamic Reads. The JavaScript code below shows three different ways of accessing a property of an object `o`.

```
x = o.p;              // a static read of 'p'
x = o["p" + "q"];     // a dynamic read of 'pq'
x = o[c ? "p" : "q"]; // a dynamic read of 'p' or 'q'
```

Line 1 is straightforward to analyze. Line 2 can be handled using syntactic constant folding. However, if the concatenation involves variables or heap locations the syntactic approach is no longer viable, instead some kind of string analysis is required. Line 3 is even more nefarious for a static analysis. If the statement is analyzed using the constant string lattice – without context sensitivity or path sensitivity – the result will be $\top$ (corresponding to any property) and thus it is unknown which property is read from `o`. A sound analysis will then conservatively include *all* properties accessible on the `o` object in the result. However this includes all properties available in the prototype hierarchy of `o`! If `o` is a regular object and its prototype is `Object[[proto]]` then around 10 properties are involved, including functions such as `toString` and `__defineSetter__`. If the internal prototype object is `Array[[proto]]` then the problem is exacerbated by an *additional* 20 properties, including mutators such as `pop`, `push`, and `reverse`, leading to even more spurious flow.

[1] In JavaScript a field is called a property and reading/writing a field is called a property access. We will use this terminology for the remainder of the paper.

Dynamic Writes. The JavaScript code below shows three different ways to store a value into an object property.

```
o.p = function() {}
o.["p" + "q"] = function() {}
o.[c ? "p" : "q"] = function() {}
```

The first two statements can be handled like in the previous section. However, the third statement requires extra care. If it is not known to which property a value is written, then the analysis must conservatively write it to *all* properties of that object using a weak update, i.e. by joining the new value into the existing values. Thus, after the last statement, any property of object o can point to the function defined on line 3.

Dynamic Reads and Writes. Even more precision is lost when dynamic reads and writes are combined as shown below:

```
o[p][q] = function() {};
```

If neither p nor q are known by the analysis, e.g. if the constant string lattice is $\top$ for both, then p could potentially be the string "__proto__" and as a result o[p] could be the internal prototype object of o. If o is a regular object then this would be the Object[[proto]] object. Thus, the write will cause the function to be written to *all* properties of the Object[[proto]] object which is shared by *all* JavaScript objects. In Java, for instance, this would correspond to overriding all fields and methods of the java.lang.Object with a spurious function. To handle such scenarios, a better string abstraction is required, in particular, the abstraction of p and q should be able to rule out property names such as __proto__. Furthermore, should a loss of precision occur for p, then the abstraction of q should still limit the damage done to Object[[proto]] by writing to just a few of its properties.

Event Handlers. An additional challenge occurs for JavaScript web applications. In JavaScript, an event handler may be registered on a HTML object by writing to several special properties, e.g. onclick, ondblclick, onload and onsubmit. For instance, writing a function value to the onclick property registers that function as a callback which is executed whenever the user clicks the mouse on its corresponding HTML object.

A sound analysis must take such registrations into account. If a dynamic property write occurs, where a HTML object is the base object, and the analysis cannot rule out that the write occurs to one of these special properties, then it must conservatively assume that an event handler registration occurs. This can lead to spurious event handler registration and spurious dataflow.

Usage in Practice. According to a study of JavaScript behavior by Richards et al. [2010]: 8.3% of all property reads are dynamic and 10.3% of all property writes are dynamic (c.f. Section 5.2 in [Richards et al., 2010] and the associated web page[2]). Furthermore, as Table 12.5 shows, many popular JavaScript libraries contain several hundred dynamic property reads and writes.

Contributions In summary our paper makes the following contributions:

- We describe twelve different string abstractions – five previously known and seven new. We focus on abstractions which require $\mathcal{O}(1)$ space and support the equal, concatenate and join operations in $\mathcal{O}(1)$ time. We place a strong emphasis on the precision and performance of the equal operation.
- We experimentally evaluate each string abstraction on four common JavaScript libraries: jQuery, Prototype, MooTools and jQuery UI. We base our evaluation on concrete executions of each library thus providing an analysis independent upper bound on the precision of each string abstraction.
- We propose a precise and efficient string abstraction $\mathcal{H}$ for reasoning about dynamic property accesses. Experiments show that $\mathcal{H}$ has no spurious flow for up to 90% of all dynamic property accesses compared to at most 21% for the constant string abstraction.
- We equip a dataflow analysis with the proposed $\mathcal{H}$ string abstraction and show that it leads to a significant improvement in precision and performance. In particular, the analysis achieves a speedup of at least 1.5x for 7 out of 10 benchmark programs.

[2]http://dumbo.cs.purdue.edu/js/analysis-charts/events.html

12.2 Related Work

String Analysis. Costantini et al. [2011] presents an abstract interpretation-based framework for string analysis and instantiates the framework for four different abstract domains: a) The *character inclusion* domain, which tracks what characters *may* or *must* occur within the string, b) the *prefix/suffix* domain, which tracks the k first and last characters of the string, c) the *bricks* domain, where a brick $b = [\mathcal{P}(s)]_{min}^{max}$ represents all strings that can be generated by concatenating elements of $\mathcal{P}(s)$ between min and max times, and d) the *string graph* domain for which we refer the reader to [Costantini et al., 2011] for details. Costantini et al.'s work does not discuss string equality which is a key issue for our work. Another difference is that Costantini et al. focus on the theoretical aspects of the strings domains, whereas we provide an experimental evaluation of the precision and performance of the domains.

Christensen et al. [2003] presents the Java String Analyzer (JSA), a static analysis tool which approximates string expressions in a Java program by a regular language. The technique is based on translation from the control-flow graph into the def-use graph, which is then translated to a context-free grammar and finally widened into a regular language. JSA has found a wide variety of applications; including verification of generated SQL statements and validation of dynamically constructed HTML. In comparison to our work Christensen et al. focus on string analysis in general, whereas we focus on string analysis for reasoning about dynamic property accesses. Furthermore, we place a strong emphasis constant time and space bounds for our abstract domains compared to the potentially exponential time bound for the whole JSA analysis.

Zheng et al. [2013] present Z3-str a general purpose string solver based on the Microsoft Z3 SMT solver. The solver models strings as a primitive type together with booleans and integers. Its supported operations include concatenation, equality, sub-string and replace. Kiezun et al. [2009] present HAMPI a string solver based on constraints on regular languages and fixed-size context-free languages. In relation to our work, general purpose string solvers such as Z3-str and HAMPI, are heavy-weight. We aim to construct a light-weight string domain, which can be used in any points-to or dataflow analysis, to address the problem of dynamic property accesses.

JavaScript Analysis. Guarnieri and Livshits [2009] present GATEKEEPER, a tool for static enforcement of security policies for JavaScript programs. The authors present an Andersen-style [Andersen, 1994] inclusion-based, context-insensitive, points-to analysis for JavaScript. GATEKEEPER classifies whether JavaScript "widgets" are safe with respect to a security policy by inspecting information from the computed points-to sets and call graph. GATEKEEPER

cannot soundly reason about dynamic property accesses and thus must resort to runtime enforcement of the security policy for every dynamic read or write (c.f. Section 3.2.2, [Guarnieri and Livshits, 2009]).

Guarnieri et al. [2011] present ACTARUS, a static taint analysis for JavaScript. ACTARUS tracks information flow to ensure that data from an untrusted source cannot reach a high-integrity sink. The analysis, like the GATEKEEPER project, is based on inclusion-based points-to analysis. ACTARUS handles dynamic property accesses (called reflective property accesses in their paper) by keeping known string constants separated and creating new abstract objects when strings objects are concatenated (Section 3.3 in [Guarnieri et al., 2011]). Yet, abstraction must be introduced at some point, and it is not clear from the paper, how this is implemented in ACTARUS.

Jensen et al. [2009] present the Type Analysis for JavaScript (TAJS) tool based on inter-procedural dataflow analysis. The analysis aims for soundness and goes to great lengths to faithfully model the semantics of JavaScript. The string abstraction is based on the constant string lattice extended to track whether the string may or must be a number-string. In more recent work, Jensen et al. [2012] extends TAJS with the *Unevalizer*, a technique for analyzing certain invocations of `eval`. For this purpose, the string lattice is extended to track strings which are valid JavaScript identifiers or contain characters which are valid inside identifiers. Jensen et al. [2011] originally identified the problem of dynamic property writes to HTML objects.

Sridharan et al. [2012] present *correlation tracking*, a technique for identifying and tracking dynamic property reads and writes which are related. The purpose of their technique is to ensure that e.g. for-each-in loops which copy properties from one object $\texttt{o}_{src}$ to another $\texttt{o}_{dst}$ maintain the relation s.t. $\texttt{o}_{dst}[p] = \texttt{o}_{src}[p]$. Thus, preserving field-sensitive precision. We believe that correlation tracking is a step in the right direction for scaling points-to and dataflow analyses for large JavaScript libraries. However, we do not believe the problem to be fully solved: First, the authors acknowledge that some scalability challenges remain. Second, it is not clear from the paper what precision can be expected, i.e. for what client applications are the results sufficient? Third, not all dynamic property accesses are correlated and thus our techniques provide an orthogonal way to improve precision.

In summary, except for Sridharan et al., most work use very simple techniques for dealing with dynamic property accesses.

We give credit to Logozzo and Fähndrich [2008] for their work on the abstract pentagon domain, an efficient integer domain for reasoning about array indices, which inspired us to investigate light-weight string domains.

12.3 String Domains

In this section we present some existing and several new abstract string domains. We have marked the domains which we believe are new to the literature with the $\star$ symbol.

Assumptions. We assume, for the rest of the paper, an underlying points-to or dataflow analysis with a standard field-sensitive heap abstraction. It is our goal to design string lattices which can be used together with the analysis without increasing its running time.

String Operations JavaScript has around 15 built-in string operations. We consider the abstract equality ($\hat{=}$), abstract concatenation ($+$) and lattice join ($\sqcup$) operations central for reasoning precisely and efficiently about dynamic property accesses. The $\hat{=}$ operation is applied at every dynamic property access to decide which property names may be referenced. Thus, it must be both *precise* – to rule out many property names – and *efficient* since it will be evaluated often. Similarly, the $+$ and $\sqcup$ operations should be efficient, while maintaining as much knowledge about the underlying strings as possible. Additional string operations are discussed in Section 12.3. All domains described in the following have finite height, thus widening is not required to ensure termination.

Constant String

The *constant string* lattice $\mathcal{C}$ tracks a single concrete string. The lattice is elements are $\bot, \top$ and $s \in \mathit{Str}$ where $\bot$ and $\top$ are the bottom and top elements, respectively. The $\bot$ element represents no concrete strings, whereas $\top$ represents all possible concrete strings. The lattice supports the equal, concatenate and join operations in $\mathcal{O}(n)$ time in the length of the string. In practice most strings are short so we do not consider the linear complexity to be a problem. The constant string lattice is the standard solution used by much prior work (as discussed in Section 12.2) and is used as the baseline abstraction in Section 12.4.

String Set

The *string set* lattice $\mathcal{SS}$ is the powerset lattice ordered by subset-inclusion of a bounded number of concrete strings. The lattice elements are $\top$ and $\{s | s \in \mathcal{P}(\mathit{Str}) \land |s| \leq k\}$ where s is a set of up to k strings and $\top$ represents all possible concrete strings. The lattice supports the equal, concatenate and join operations in $\mathcal{O}(k^2 \times n)$ time, where k is the bound and n is length of the longest string.

```
def concat(A: Long, B: Long): Long = {
  var R: Long = Long.reverse(B);
  var C: Long = 0L;
  for (i <- 0 until b) {
      r = Long.rotateLeft(r, 1);
      if ((A & R) != 0L) {
        C |= (1 << i);
      }
  }
  return C;
}
```

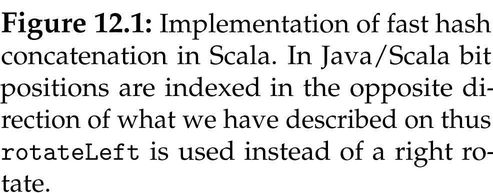


Figure 12.1: Implementation of fast hash concatenation in Scala. In Java/Scala bit positions are indexed in the opposite direction of what we have described on thus `rotateLeft` is used instead of a right rotate.

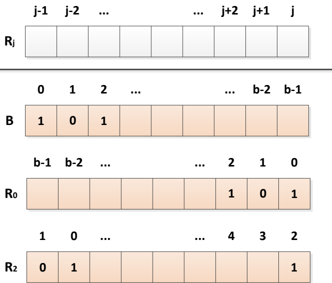


Figure 12.2: The top part of the figure shows the bitvector R_j, obtained by reversing and right-rotating B j times. The bottom part is an example where R_0 and R_2 are obtained from the bitvector B.

Length Interval

The *length interval* lattice $\mathcal{I}$ is the interval lattice on the string length. It tracks the minimum and maximum length of the concrete strings it represents. The length interval lattice can distinguish property names which are usually short, from data strings such as HTML code, image data or other serialized data. The interval representation is standard, with a bounded width k, and supports the equal, concatenate and join operations in $\mathcal{O}(1)$ time. Finally, we note that the length interval lattice can be useful for coercions from strings to booleans as it tells us whether the string may be the empty string, and thus can coerce to `false`.

Length Hash $\star$

The length interval lattice $\mathcal{I}$ loses much precision whenever strings of disparate length are joined. We propose to overcome this by introducing the length hash lattice $\mathcal{LH}$. The length hash lattice tracks *a set of string length hashes* instead of tracking the minimum and maximum string length. We take a universe of fixed size $U = \{0 \dots b\}$ and a hash function $h : S \rightarrow U$ s.t. each string length hashes to a particular bucket in the universe. The lattice is the powerset lattice of U ordered by subset-inclusion (i.e. $\bot$ is the empty set and represents no concrete strings). If we fix b at the word size of the target architecture we can efficiently implement $\mathcal{LH}$ as a bitset. The equal and join operations can then be implemented as bitwise operations in $\mathcal{O}(1)$ time.

Concatenation is more tricky. If we require the hash function h to be distributive, s.t. ($h(s_1 + s_2) = h(s_1) + h(s_2) \mod b$), then concatenation can be implemented precisely. Concatenation of the abstract strings $\hat{s}_1$ and $\hat{s}_2$ is

computed by summing all lengths in $\hat{s}_1$ with all lengths in $\hat{s}_2$ and taking the modulus. A naive implementation calculates these sums inside two nested loops. The complexity of this implementation is $\mathcal{O}(b^2)$ where b is the size of the universe. This is $\mathcal{O}(1)$ since b is a fixed constant, but in practice $b = 64$ and thus the naive implementation may require up to 4096 iterations.

A better solution achieves $\mathcal{O}(b)$ time by only iterating through the lengths of $\hat{s}_1$ and summing with the lengths of $\hat{s}_2$ *simultaneously* by using a few clever bit operations. Let A and B be the bitvectors representing $\hat{s}_1$ and $\hat{s}_2$ respectively. We observe that the k'th position in the resulting bitvector C depends on all $A[i]$ and $B[j]$ where $i + j \equiv k \mod b$.

We define R_j to be the bitvector obtained from B by first reversing it and then right rotating the result j positions. Thus, e.g. R_0 is the reverse of B and R_2 is the reverse of B right rotated two positions, as shown in Figure 12.2. We can now compute $C[k]$ by evaluating $A \wedge R_{k+1} \neq 0$, since $R_{k+1}[i] = B[(b-1-i)+(k+1) \mod b] = B[k-i \mod b] = B[j]$ and thus:

$$C[k] = (A \wedge R_{k+1} \neq 0) = \bigvee_{i=0}^{b-1} A[i] \wedge B[j]$$

which is equivalent to what is computed by the naive implementation. The code in Figure 12.1 implements this strategy. In a synthetic benchmark the above code resulted in a factor 70 speedup compared to the naive implementation. As an example, the abstraction of {`abc`, `abcdef`} is a bitset containing the elements 3 and 6. This bitset represents all strings of length $\{l | l = 3 + b * i \vee l = 6 + b * i, \forall i \geq 0\}$.

Prefix and Suffix Characters

The *prefix-suffix character* lattice $\mathcal{PS}$ tracks the first and last character symbol of the string. It is formed as the cartesian product of two constant character lattices; one for the prefix and one for the suffix. The lattice supports the equal, concatenation and join operations in $\mathcal{O}(1)$ time.

In jQuery HTML tags can be passed into to the $-function to construct new HTML elements. Inside the $-function, the following test is used to inspect whether an argument is an HTML tag:

```
var length = selector.length;
if (selector.charAt(0) === "<" &&
    selector.charAt(length - 1) === ">" &&
    length >= 3) {
```

The prefix-suffix character lattice can analyze code like the above by providing information about whether the first and last character *may* or *must not* be the < and > characters, respectively.

Character Inclusion

The *character inclusion* lattice $\mathcal{CI}$ tracks what character symbols *may* and *must* occur within a string. It is formed as the cartesian product of the four sub-lattices: c_{may}, c_{must}, e_{may} and e_{must}. The c_{may} and c_{must} lattices are powerset lattices of character symbols ordered by subset- and superset inclusion, respectively. The e_{may} and e_{must} boolean lattices tracks whether the concrete set of strings may or must include the empty string or a character symbol which is not representable by c_{may} or c_{must}. As an example, the empty string, and the strings `foo` and `moo` are represented as:

$$\mathcal{CI} = (c_{\text{may}} = \{\texttt{f}, \texttt{m}, \texttt{o}\}, c_{\text{must}} = \{\texttt{o}\}, \top_{e_{\text{may}}}, \bot_{e_{\text{must}}})$$

The equal operation of $\mathcal{CI}_1$ and $\mathcal{CI}_2$ is implemented as:

1. If $\mathcal{CI}_1$ or $\mathcal{CI}_2$ is $\bot_{\mathcal{CI}}$ then the result is $\bot_{\text{bool}}$, i.e. if one (or both) of the lattices represents the empty set of concrete strings then the results represents the empty set of concrete booleans (denoted by $\bot_{\text{bool}}$).

2. If $c^1_{\text{must}} \cap c^2_{\text{may}} = \emptyset$ or $c^2_{\text{must}} \cap c^1_{\text{may}} = \emptyset$ the result is `False`, i.e. if a character *must* be in $\mathcal{CI}_1$ but at the same time is *definitely not* present in $\mathcal{CI}_2$ the strings cannot be the same (and vice versa).

3. If $c^1_{\text{may}} \cap c^2_{\text{may}} = \emptyset$ and $e^1_{\text{may}} = e^2_{\text{may}} = \bot$ then the result is `False`, since no characters overlap between $\mathcal{CI}_1$ and $\mathcal{CI}_2$, and none of them are the empty string.

4. If $\mathcal{CI}_1$ must contain the empty string or an unrepresented character and $\mathcal{CI}_2$ definitely does not (or vice versa) the result is `False`, since either contains characters which the other does not.

5. Otherwise the result is $\top_{\text{bool}}$, i.e. the concrete set of `true` and `false`.

We implement the character inclusion lattice with two bitsets. The first tracks may-information and the second tracks must-information. In each bitset we use a bit to track whether the string may/must be the empty string or contain an unrepresentable character. The remaining bits are reserved for character symbols. If we have 64 bits this leaves space for 63 characters.

We represent character symbols in the ASCII range from 32 to 95, which includes the characters 0-9, A-Z, the special characters `!"#$%&'()*+,-./:;<=>?@` and space. Lowercase letters can be accommodated by converting them to uppercase, i.e. the character inclusion lattice is case-insensitive. In summary, the bitset-based character inclusion lattice supports the equal, concatenate and join operations in $\mathcal{O}(1)$ time.

Index Predicate ⋆

The *index predicate* lattice ($\mathcal{IP}$) tracks whether a boolean valued predicate $\rho(c)$ may or must hold for the character symbol c at index i of the string, where the index is bounded by a constant b. That is, the lattice only tracks the predicate for the first b characters. Most property names are short, and thus having incomplete information for long strings is unlikely to be a problem in practice. We can instantiate the lattice with predicates like the following:

- *Lowercase / Uppercase* – whether the character at index i may or must be a lowercase or uppercase letter. This is useful for property names that use camel casing, e.g. `hasOwnProperty`.
- *Underscore* – whether the character at index i may or must be an underscore. Like above, this is useful for "hidden" property names with underscores, e.g. `__defineGetter__`.
- *Digit* – whether the character at index i may or must be a digit. If all character symbols must be digits then the entire string represents a number.
- *Non-identifier Character* – whether the character at index i may or must be a non-identifier character. (A generalization of the `idPart` in *Unevalizer* Jensen et al. [2012])
- *Whitespace* – whether the character at index i may or must be white space (i.e. space, tabs or newline) which is useful for e.g. `split`.

The index predicate lattice is the cartesian product of two powerset lattices of indices $i_{\texttt{may}}$ and $i_{\texttt{must}}$ and the length interval lattice. The length interval lattice is used to handle concatenation precisely.

We implement the $i_{\texttt{may}}$ and $i_{\texttt{must}}$ lattices as bitsets for the first 64 string indices. The length interval lattice uses the standard representation. The join operation is straightforward to implement in $\mathcal{O}(1)$ time. The equal operation can be implemented similarly to the equal operation for the character inclusion lattice. The concatenate operation, however, requires more legwork. If the strings $s_1 = (i^1_{\texttt{may}}, i^1_{\texttt{must}})$ and $s_2 = (i^2_{\texttt{may}}, i^2_{\texttt{must}})$ are concatenated and the length of s_1 is not an interval, but a concrete number, then concatenation is simply a matter of merging the $i^1_{\texttt{may}}$ and $i^2_{\texttt{may}}$ bitsets using the concrete length of s_1 as an offset. On the other hand, if the length of the string s_1 is an interval then the $i^1_{\texttt{may}}$ and $i^2_{\texttt{may}}$ bitsets must be merged by all offsets in that interval. Similarily for the must bitsets, as shown in Figure 12.3.

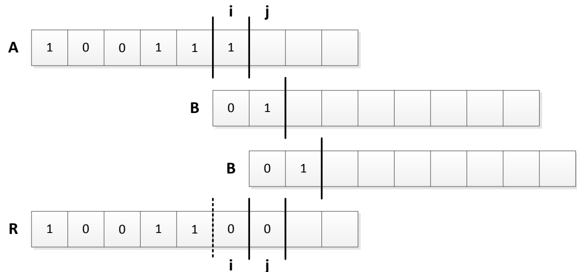


Figure 12.3: Concatenation of two index predicate lattices A and B for the $i^1_{\texttt{must}}$ and $i^2_{\texttt{must}}$ sets, respectively. Here the length of A is between $[5, 6]$. The example shows how the indices i and j are computed by bitwise-and.

Sliding Index Predicate ⋆

The *sliding index predicate* lattice $\mathcal{SIP}$ tracks a boolean valued predicate for pairs of consecutive characters. That is, the predicate is of the form $\rho : \texttt{Char} \times \texttt{Char} \rightarrow \texttt{Bool}$, where the two characters are adjacent inside the string. We can instantiate the lattice with predicates like the following:

- Gemination - whether two consecutive characters are the same. E.g. in the property names `__defineGetter__` and `__defineSetter__` there are three geminations, one for the preceding underscores, one for the double t's and one for the succeeding underscores.
- Inversions - whether two consecutive characters are inverted with respect to their lexicographical ordering. E.g. in the property name `valueOf` the characters `v` and `a` are inverted. If no characters may be inverted then the characters in the string must be sorted.

The sliding index predicate lattice is similar to the index predicate lattice. However, in addition to may- and must- bitsets and the length interval lattice, it must be equipped with the prefix-suffix lattice. This lattice is required for the concatenation operation: When s_1 and s_2 are concatenated the prefix-suffix is used to evaluate the predicate for the last character of s_1 and the first character of s_2 thus ensuring that knowledge of the predicate is preserved for all consecutive pairs of characters in the resulting string.

Prefix Suffix Inclusion ⋆

The *prefix-suffix inclusion* lattice $\mathcal{PSI}$ is inspired by the prefix-suffix and character inclusion lattices. It tracks the *set of characters* that the first and last character in the string *may* or *must* be. As for the character inclusion lattice, it tracks whether the string is the empty string or if the prefix/suffix may be a non-representable character symbol.

Its representation is based on no less than *four* bitsets: May-and must-bitsets for both the prefix and suffix character. Equal, concatenation and join is implemented as bitwise operations in $\mathcal{O}(1)$ time.

The prefix-suffix inclusion lattice can rule out equality of the concrete string `prototype` and the abstract string $\hat{s}$, if the first character of $\hat{s}$ is *definitely not* `p` or the last character of $\hat{s}$ is *definitely not* `e`.

String Hash ⋆

The *string hash* lattice $\mathcal{SH}$ lattice is inspired by the length hash lattice, but instead of hashing the string length, it hashes the string itself: It uses a hash function $h : S \to U$ which takes the sum of all character codes in the string and hashes it into a bucket (as described in Section 12.3). The strength of the string hash lattice is that it can keep separate strings for which the other lattices might lose all information. Consider the example:

$$\texttt{"foo"} \hat{=} (\texttt{"The"} \sqcup \texttt{"quick"} \sqcup \texttt{"brown"} \sqcup \texttt{"fox"})$$

Here, for instance, the length interval, the length hash and the character inclusion lattices lose information and cannot rule out that the strings may be equal. In contrast, the four strings hash to 33, 29, 40 and 13, respectively, and `"foo"` hash to 4, and thus the abstraction is able to rule out equality between the left and right side.

We implement the string hash lattice as a single bitset which supports equal, concatenate and join in $\mathcal{O}(1)$ time. Concatenation is implemented in the same way as the length hash lattice and requires the hash function to be distributive (see Section 12.3).

Number Strings ⋆

In JavaScript it is common for numbers to be coerced to strings. We introduce the number string lattice $\mathcal{N}$ to track JavaScript numbers encoded as strings. It is the powerset lattice of the elements ∞, $-\infty$, `NaN`, $\mathbb{N}$ and `Other` ordered by subset-inclusion:

$$\mathcal{N} = (\mathcal{P}\{\infty, -\infty, \texttt{NaN}, \mathbb{N}, \texttt{Other}\}, \subseteq)$$

Here ∞ represents the number "positive infinity" which coerced to a string yields `"Infinity"`, similarly $-\infty$ coerces to `"-Infinity"`, `NaN` represents "not-a-number" which coerces to `"NaN"` and $\mathbb{N}$ which represents any natural number which coerces to itself as a string.

The number string lattice is implemented as a bitset and supports equal, concatenate and join operations in $\mathcal{O}(1)$ time. With respect to concatenate, we take a pragmatic approach and let it return $\top$, i.e. all lattice elements.

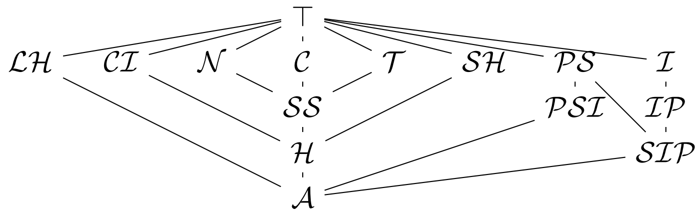


Figure 12.4: A diagram showing how the precision of the lattices relate to each other. As an example, the precision of the prefix-suffix lattice $\mathcal{PS}$ lattice is fully subsumed by the prefix-suffix inclusion lattice $\mathcal{PSI}$.

Type Strings ⋆

In JavaScript the `typeof` operator inspects the runtime type of a value and returns one of the string constants: `boolean`, `function`, `object`, `string` and `undefined`. The `typeof` operator is widely used in jQuery, for instance:

```
stop: function(type, clearQueue, gotoEnd) { // ...
    if (typeof type !== "string") { // ...
```

Here the behaviour of the `stop` function depends on the type of its first argument. We introduce the type string lattice $\mathcal{T}$ to explicitly track the five strings returned by `typeof`:

$$\mathcal{T} = (\mathcal{P}(\{\texttt{Bool}, \texttt{Func}, \texttt{Obj}, \texttt{Str}, \texttt{Undef}, \texttt{Other}\}), \subseteq)$$

The `Other` element, as for the number string lattice, represents all strings other than the type strings. We implement the lattice as a single bitset which supports the equal and join operations in $\mathcal{O}(1)$ time.

The Hybrid Lattice ⋆

We introduce the hybrid string lattice $\mathcal{H}$ as the cartesian product of the string set $\mathcal{SS}$ (Section 12.3), character inclusion $\mathcal{CI}$ (Section 12.3) and string hash $\mathcal{SH}$ (Section 12.3) lattices. The intuitive idea behind the lattice is to track a few concrete strings with full precision and then "fallback" to the character inclusion and string hash lattices when there are too many strings to track. As will be shown in Section 12.4, the hybrid lattice achieves almost the same precision as the combination of all presented lattices.

Lattice Relations

Figure 12.4 shows how the precision of the lattices relate to each other. As discussed in the previous section, the figure shows that the hybrid string lattice $\mathcal{H}$ is at least as precise as the string set $\mathcal{SS}$, character inclusion $\mathcal{CI}$ and string hash lattices $\mathcal{SH}$. We call the cartesian product of all lattices $\mathcal{A}$.

	$\mathcal{C}$	$\mathcal{SS}$	$\mathcal{I}$	$\mathcal{LH}$	$\mathcal{PS}$	$\mathcal{CI}$	$\mathcal{IP}$	$\mathcal{PSI}$	$\mathcal{SH}$	$\mathcal{N}$	$\mathcal{T}$	$\mathcal{H}$
New				✓			✓	✓	✓	✓	✓	✓
Structural	✓	✓			✓		✓	✓		✓	✓	✓
Subset		✓		✓		✓	✓	✓	✓	✓	✓	✓
Parametric						✓	✓	✓	✓			✓
Space	$\lvert s\rvert$	$k \times \lvert s\rvert$	2	1	2	2	4	5	1	1	1	$k \times \lvert s\rvert + 3$

Table 12.1: Overview of lattice characteristics.

Overview

We briefly summarize some characteristics of the presented lattices:

- **New:** We believe that the lattice is new to the literature.
- **Structural:** The lattice tracks the structure of the string. As an example, the prefix-suffix lattice $\mathcal{PS}$ tracks the first and last character of the string.
- **Subset:** The lattice subset or superset-based.
- **Parametric:** The lattice has different instantiations.For instance, the index predicate lattice $\mathcal{IP}$ can be instantiated with different predicates.
- **Space:** The space (memory) required to represent a single lattice element. In machine words, except for $\mathcal{C}$ and $\mathcal{SS}$ which must store the entire string(s).

Table 12.1 shows an overview of these characteristics.

Additional String Operations

We now describe some additional string operations which the lattices support.

charAt and charCodeAt. The *charAt*(i) and *charCodeAt*(i) string functions return the character or character code at position i inside the string.

- $\mathcal{PS}$– if the index is zero then the prefix-suffix lattice knows the precise result.
- $\mathcal{IP}$– the index predicate lattice can provide an upper bound on what character symbols may occur at index i. E.g. if the predicate is `isUpperCase` and it holds for index i, then the character must be in the set $[A - Z]$.
- $\mathcal{CI}$– the character inclusion lattice can provide an upper bound on what character symbols may occur at index i.

indexOf, lastIndexOf and search. The *indexOf*(s) and *lastIndexOf*(s) functions return the index of respectively the first and last occurrence of s in the string. If s is not contained in the string, the value -1 is returned.

- $\mathcal{CI}$– if the query string is a single character the character inclusion lattice can decide whether that character may or must occur within the string. It cannot give the precise index, but it can decide whether the -1 value should be part of the return value.
- $\mathcal{IP}$– if the query string is a single character and some property of that character is tracked by the index predicate lattice, then a set of indices can be returned. E.g. if the query string is an uppercase `A` and the index predicate lattice tracks uppercase letters, then the lattice can provide all indices where uppercase letters may occur.
- $\mathcal{I}$– the length interval lattice can provide a bound on the returned index.

substring. The *substring(b, e)* function returns the substring beginning at position *b* and ending immediately before position *e*.

- $\mathcal{I}$ & $\mathcal{LH}$– the length interval and length hash lattices simply restrict their intervals to the range $[b, e]$.
- $\mathcal{PS}$– if the extracted string is a prefix, i.e. if $b = 0$, then the prefix-suffix lattice can retain its first component.
- $\mathcal{CI}$– the character inclusion lattice can retain its *may-set* of character symbols, but its *must-set* must be replaced by $\top$.
- $\mathcal{IP}$ & $\mathcal{SIP}$– the index predicate and sliding index predicate lattices can retain all their information for the substring.

12.4 Evaluation

We have described the theoretical properties of the lattices and now turn to their practical application by considering the research questions:

- **Q1**: How precise are the lattices, independent of any particular analysis, for reasoning about strings used in dynamic property accesses?
- **Q2**: To what degree does a more precise string lattice, for dynamic property accesses, improve the overall precision and performance of a static analysis?

We now address each question in turn.

Library	Lines	Reads		Writes	
		Locations	Properties	Locations	Properties
jQuery-1.9.1	9,597	400	7.0	124	5.8
jQuery-1.8.1	9,301	377	12.0	102	8.3
jQuery-1.7.1	9,266	401	6.8	101	6.6
Prototype-1.7.0	7,036	226	9.8	43	14.7
MooTools-1.4.5	5,976	281	13.7	110	14.2
jQueryUI-1.8.24	11,377	265	8.1	75	7.4

Figure 12.5: The JavaScript libraries used for the dynamic analysis evaluation. Here the *locations* column indicates the number of syntactic occurrences of dynamic property accesses. The *properties* column indicates the average number of property names read/written by a dynamic property access expression.

Dynamic Analysis

We investigate Q1 by performing a dynamic analysis of strings and dynamic property accesses in four large JavaScript libraries. Inspired by Liang et al. [2010] the dynamic analysis is used to provide a (static-) analysis independent upper bound on the precision of each lattice. That is, the best precision each lattice can possibly provide for a set of concrete execution traces.

We instantiate the string set lattice with $k = 3$ (see Section 12.3), the length interval lattice with width $k = 20$ (see Section 12.3), the index predicate lattice with the uppercase predicate (see Section 12.3) and the remaining lattices are instantiated as described in their respective sections.

Benchmarks. We collect concrete execution traces for the four large JavaScript libraries shown in Table 12.5. The traces expose a total of 80,000 dynamic property accesses of which 60,000 are reads. We obtained the traces by loading twelve popular websites according to the Alexa rankings[3]. Since jQuery is prevalent, we include three different versions. The websites are automatically modified to use instrumented versions of the libraries which record information about every dynamic property access.

We explain Table 12.5 by example. The table shows that the jQuery-1.9.1 source code has 400 dynamic property read expressions and 124 write expressions. For the read expressions, an average of 7.0 properties are read by each expression, and an average of 5.8 properties are written by each expression.

[3]http://www.alexa.com/topsites

Concrete Traces. We instrument the source code to register the following for every dynamic property access `o[p]`:

$$\mathcal{T} = (\mathcal{R}, \mathcal{L}, \mathcal{E}, \mathcal{P}_o, \mathcal{P}_p) \text{ , where}$$

- $\mathcal{R}$ is a unique identifier for the concrete *run*.
- $\mathcal{L}$ is the physical *location* of the dynamic property access in the source.
- $\mathcal{E}$ is the *expression tree* corresponding to how the property name, which is being used for the dynamic property access, was created. An expression tree is a tree where the leaves are string constants and the internal nodes are string operations, which are equipped with their source code location.
- $\mathcal{P}_o$ is the set of properties available on the object o itself.
- $\mathcal{P}_p$ is the set of properties available on the prototype objects of o.

Here $\mathcal{R}$ and $\mathcal{L}$ is meta data about the concrete trace and $\mathcal{E}, \mathcal{P}_o, \mathcal{P}_p$ is information about the dynamic property access. As an example, the execution of the code: snippet on the left produces the trace on the right.

```
x = new Object();
x.a = 42;
x.b = 21;
z = x["p" + "q"];
```

$$\mathcal{T} = (0, \texttt{input.js:4}, \texttt{Toplevel}, \mathcal{E}, \mathcal{P}_o, \mathcal{P}_p)$$
$$\mathcal{E} = \texttt{Concat(input.js:4, "p", "q")}$$
$$\mathcal{P}_o = \{\texttt{a}, \texttt{b}\}$$
$$\mathcal{P}_p = \{..., \texttt{toString}, \texttt{valueOf}, ...\}$$

Here `Toplevel` is the name of the toplevel "function", $\mathcal{E}$ is the expression tree for the string concatenation of `"p"` and `"q"`, $\mathcal{P}_o$ contains `a` and `b` (i.e all properties of `o`) and $\mathcal{P}_p$ contains all properties of the `Object[[prototype]]` object.

Abstract Traces. We simulate the effects of abstraction by merging several concrete traces into a smaller set of abstract traces. We merge concrete traces which share the same location $\mathcal{L}$ to obtain a single abstract trace for that location. In particular, given two concrete traces $\mathcal{T}^1 = (\mathcal{R}^1, \mathcal{L}^1, \mathcal{E}^1, \mathcal{P}_o^1, \mathcal{P}_p^1)$ and $\mathcal{T}^2 = (\mathcal{R}^2, \mathcal{L}^2, \mathcal{E}^2, \mathcal{P}_o^2, \mathcal{P}_p^2)$ we define the abstract trace $\hat{\mathcal{T}} = (\mathcal{L}, \hat{\mathcal{E}}, \hat{\mathcal{P}}_o, \hat{\mathcal{P}}_p)$ where

$$\mathcal{L} = \mathcal{L}^1 = \mathcal{L}^2 \qquad \hat{\mathcal{E}} = \{\mathcal{T}^1_{\mathcal{E}}, \mathcal{T}^2_{\mathcal{E}}\} \qquad \hat{\mathcal{P}}_o = \mathcal{T}^1_{\mathcal{P}_o} \cup \mathcal{T}^2_{\mathcal{P}_o} \qquad \hat{\mathcal{P}}_p = \mathcal{T}^1_{\mathcal{P}_p} \cup \mathcal{T}^2_{\mathcal{P}_p}$$

The generalization to multiple traces is straightforward.

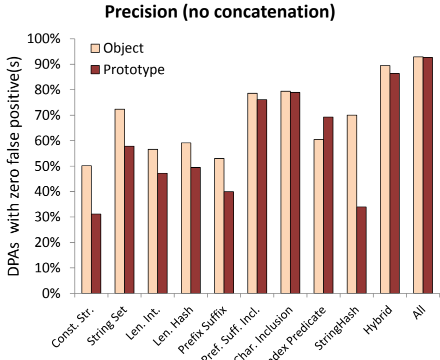
We now consider two scenarios. First, we evaluate the precision of the lattices on the abstract traces where the expression tree $\mathcal{E}$ is fully evaluated before any abstraction. That is, if an abstract trace has the two expressions trees $\mathcal{E}_1 = "a"$ and $\mathcal{E}_2 = "b" + "c" + "d"$ then we consider the abstraction $\alpha("a") \sqcup \alpha("bcd")$, i.e. concatenation occurs *before* abstraction. Second, we evaluate the precision when concatenation occurs *after* abstraction. For instance, we would evaluate $\alpha("a") \sqcup (\alpha("b") + \alpha("c") + \alpha("d"))$. Here α is the abstraction function which lifts a concrete string into the abstract domain.

Precision without Concatenation. Figure 12.6 shows the percentage of dynamic property access locations with *zero* false positives for each lattice. That is, a value of 100% implies that the lattice is complete for all dynamic property accesses. A value of 50% implies that half of all locations of dynamic property accesses have at least one false positive. The figure shows two bars for each lattice; the light bar represents locations with false positives involving properties in the base object, and the dark bar represents false positives involving properties in the prototype objects.

We observe that the constant string lattice ensures that at most 50% of all dynamic property accesses involving base object properties have zero false positives. If we consider prototype properties, the number drops to 31%. This means that for more than half of all dynamic property accesses the constant string lattice will cause spurious flow. For the string set lattice the percentages are not surprisingly higher at 72% and 58%, respectively. The character inclusion lattice achieves the highest precision with 79% and 78% of all dynamic property accesses having zero false positives. Remarkably, the prefix-suffix inclusion lattice achieves nearly the same precision, even though it only tracks information about the first and last characters in the string. The hybrid lattice achieves 89% and 86% which is only slightly lower than the all lattice (the product of all lattices). The number string and type string lattices achieve less than 25% of property accesses with zero false positives and are omitted from the graphs.

We attribute the difference in precision for object and prototype properties to the fact that most objects have fewer properties than their corresponding prototype object(s).

Precision with Concatenation. Figure 12.7 shows the percentage of dynamic property accesses with *zero* false positives for each lattice restricted to traces which involve at least one concatenation. This restriction reduces the number of concrete traces from 80,000 to around 8,000. Note that this implies that Figures 12.6 and 12.7 are not directly comparable.



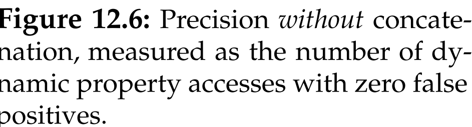


Figure 12.6: Precision *without* concatenation, measured as the number of dynamic property accesses with zero false positives.

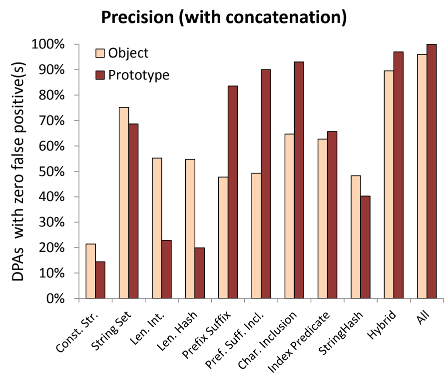


Figure 12.7: Precision *with* concatenation, measured as the number of dynamic property accesses with zero false positives.

We observe, when concatenation is involved, the constant string lattice is only able to achieve a zero false positive rate of 21% and 14% for object and prototype properties. Again, the character inclusion lattice achieves the best precision with 64% and 93% property accesses with zero false positives. The hybrid lattice achieves 90% and 97% which is only slightly less than all the lattices combined. The number string and type string lattices achieve less than 10% with zero false positives and are omitted from the graphs. We leave the evaluation of the sliding index predicate as future work for two reasons: First, initial experiments on the index predicate lattice showed that it has very poor performance for concatenation. Second, it was not clear to us what kind of predicate would be a good discriminator for property names.

We answer Q1 by concluding that the precision of the constant string lattice is worse than most other of the presented lattices. Furthermore, the hybrid string lattice $\mathcal{H}$ achieves almost the same precision as all the lattices combined.

Static Analysis

We investigate Q2 by comparing the precision and performance of a static analysis equipped with the constant string lattice $\mathcal{C}$ and the proposed hybrid string lattice $\mathcal{H}$.

Dataflow Analysis. We have implemented an inter-procedural, flow-sensitive and context-insensitive dataflow analysis for JavaScript in the style of Jensen et al. [2009]. The analysis can be instantiated with different string lattices without any changes to the rest of the abstraction.

Program	Lines	Nodes	Precision		Performance		
			PAs↓	PointsTo↓	Constant	Hybrid	Speedup
3d-cube.js	343	2,794	14%	10%	1.7s	1.0s	1.6x
3d-raytrace.js	443	2,874	57%	41%	38.5s	4.7s	8.2x
access-nbody.js	170	828	9%	7%	0.3s	0.2s	2.1x
astar.js	355	1,406	68%	58%	5.9s	0.3s	16.8x
crypto-md5.js	295	1,422	93%	93%	0.3s	0.2s	1.8x
garbochess.js	2,812	15,795	78%	77%	56.3s	24.3s	2.3x
javap.js	1,400	5,104	28%	27%	7.5s	7.3s	1.0x
richards.js	541	1,602	3%	2%	3.7s	3.3s	1.1x
simplex.js	450	2,056	73%	72%	0.6s	0.3s	2.0x
splay.js	398	1,016	2%	2%	0.4s	0.4s	1.0x

Table 12.2: Static Analysis results. Lines is the number of lines of source code. Nodes is the number of control-flow graph nodes. PAs↓ is the percentage of property reads with improved precision. PointsTo↓ is the average reduction in the size of points-to sets for *all* property reads.

Benchmarks. We evaluate the analysis on the programs shown in Table 12.2. The `3d-cube.js`, `3d-raytrace.js`, `access-nbody.js` and `crypto-md5` programs originate from the Mozilla SunSpider benchmark suite, `richards.js` and `splay.js` originate from the Google Octane benchmark suite and `astar.js`, `garbochess.js`, `javap.js` and `simplex.js` were collected from GitHub and various sources on the Internet. The table lists the benchmark name, number of lines of code and the number of control-flow graph nodes in the first three columns.

We use these benchmarks, instead of the libraries from the previous section, since we know of no analysis which is yet able to analyze such large and complex libraries.

Precision. We compare the precision of the string lattices in two ways. First, we compute for how many property read locations that the points-to sets are smaller. Second, we compute on average how much smaller the points-to sets are. We look at all property reads and not just dynamic property reads. The reason is that spurious flow in one dynamic property access may cause imprecision in a non-dynamic read. Thus, by looking at all reads we get a clearer picture of overall analysis precision.

The PAs↓ column in Table 12.2 shows the percentage of property reads where the use of the hybrid string lattice results in a smaller points-to set than the constant string lattice, that is, the percentage of reads where the hybrid

string lattice yields at least one less pointer than the constant string lattice. The results show that for 5 of the 10 programs at least 50% of all property reads have improved precision, and that all programs show some improvement. The PointsTo↓ column shows how much smaller on average the points-to sets are for all property reads. The results show that the hybrid lattice ensures significantly smaller sets and that for 5 of the 10 programs the reduction is more than 40%. Thus the hybrid lattice improves precision for many property accesses and is effective at reducing spurious flow compared to the constant string lattice.

Performance. The last three columns of Table 12.2 compare the analysis time with the two different lattices. The results show that for 7 of the 10 programs the analysis is more than 1.5x faster, and for 5 of the programs the analysis is more than 2.0x faster. We attribute this to the fact that the analysis is more precise with the hybrid lattice and propagates less spurious flow. In the case of `javap.js`, `richards.js` and `splay.js` there is no significant speedup. In case of `richards.js` and `splay.js` this can be explained by the fact that these two benchmarks gain little in terms of improved precision. The `javap.js` program appears to be an outlier which gains significantly improved precision, but no corresponding boost in performance. Naturally, the degree of speedup will vary from analysis to analysis. In particular, if the analysis is efficient at representing and propagating large point-to sets the performance improvement will likely be less pronounced.

We answer Q2 by concluding that the hybrid string lattice $\mathcal{H}$ is preferable to the commonly used constant string lattice $\mathcal{C}$. We have shown that the hybrid string lattice lattice leads to significantly improved precision and performance.

12.5 Conclusion

We have described twelve different string abstractions – five previously known and seven new – for reasoning about dynamic property accesses in static analysis of JavaScript. Experimental evaluation on four common and large JavaScript libraries, including jQuery, suggests that dynamic property accesses are prevalent and that the standard approach of tracking strings with the constant string lattice is insufficient.

We have presented the hybrid lattice $\mathcal{H}$ which supports the equal, concatenate and join operations in $\mathcal{O}(1)$ time. Experimental results on 10 JavaScript programs show that the hybrid string lattice leads to significantly improved precision and performance when used in a dataflow analysis.

13

Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries

By Magnus Madsen, Benjamin Livshits and Michael Fanning published in proc. 9th European Software Engineering Conference / International Symposium on Foundations of Software Engineering (ESEC/FSE '13).

Abstract

JavaScript is a language that is widely-used for both web-based and standalone applications such as those in the upcoming Windows 8 operating system. Analysis of JavaScript has long been known to be challenging due to its dynamic nature. On top of that, most JavaScript applications rely on large and complex libraries and frameworks, often written in a combination of JavaScript and native code such as C and C++. *Stubs* have been commonly employed as a partial specification mechanism to address the library problem; however, they are tedious to write, incomplete, and occasionally incorrect.

However, the manner in which library code is *used* within applications often sheds light on what library APIs *return* or *consume* as parameters. In this paper, we propose a technique which combines *pointer analysis* with *use analysis* to handle many challenges posed by large JavaScript libraries. Our approach enables a variety of applications, ranging from call graph discovery to auto-complete to supporting runtime optimizations. Our techniques have been implemented and empirically validated on a set of 25 Windows 8 JavaScript

applications, averaging 1,587 lines of code, demonstrating a combination of scalability and precision.

13.1 Introduction

While JavaScript is increasingly used for both web and server-side programming, it is a challenging language for static analysis due to its highly dynamic nature. Recently much attention has been directed at handling the peculiar features of JavaScript in pointer analysis [Chugh et al., 2009; Guarnieri and Livshits, 2009, 2010], data flow analysis [Jensen et al., 2009, 2011], and type systems [Thiemann, 2005b,a].

The majority of work thus far has largely ignored the fact that JavaScript programs usually execute in a rich execution environment. Indeed, Web applications run in a browser-based environment interacting with the page through the extensive HTML DOM API or sophisticated libraries such as jQuery. Similarly, NodeJS applications run inside an application server. Finally, JavaScript applications in the Windows 8 OS, which are targets of the analysis in this paper, call into the underlying OS through the Windows 8 runtime. In-depth static analysis of these application hinges on our understanding of libraries and frameworks.

Unfortunately, environment libraries such as the HTML DOM and the Windows 8 runtime lack a full JavaScript implementation, relying on underlying C or C++, which is outside of what a JavaScript static analyzers can reason about. Popular libraries, such as jQuery, have a JavaScript implementation, but are hard to reason about due to their heavy use of reflective JavaScript features, such as `eval`, computed properties, runtime addition of fields, etc.

The standard solution to overcome these problems is to write partial JavaScript implementations (also known as *stubs*) to partially model library API functionality. Unfortunately, writing stubs is tedious, time-consuming, and error-prone.

Use analysis. The key technical insight that motivates this paper is that observing *uses* of library functionality within application code can shed much light on the structure and functionality of (unavailable) library code. By way of analogy, observing the effect on surrounding planets and stars can provide a way to estimate the mass of an (invisible) black hole. This paper describes how to overcome the challenges described above using an inference approach that combines pointer analysis and *use analysis* to recover necessary information about the structure of objects returned from libraries and passed into callbacks declared within the application.

Motivating Example. We open with an example illustrating several of the challenges posed by libraries, and how our technique can overcome these challenges. The code below is representative of much DOM-manipulating code that we see in JavaScript applications.

```
var canvas = document.querySelector("#leftcol .logo");
var context = canvas.getContext("2d");
context.fillRect(20, 20, c.width / 2, c.height / 2);
context.strokeRect(0, 0, c.width, c. height);
```

In this example, the call to `querySelector` retrieves a `<canvas>` element represented at runtime by an `HTMLCanvasElement` object; the `context` variable points to a `CanvasRenderingContext2D` object at runtime. `context` is then used for drawing both a filled and stroked rectangle on the `canvas`. Since these objects and functions are implemented as part of the browser API and HTML DOM, no JavaScript implementation that accurately represents them is readily available. Furthermore, note that the return value of the `querySelector` call depends on the CSS expression provided as a string parameter, which is difficult to reason about without an accurate model of the underlying HTML page. There are two common approaches to statically analyze this code:

- Model `querySelector` as (unsoundly) returning a reference to `HTML-Element.prototype`. This approach suffers from a simple problem: `HTML-Element.prototype` does not define `getContext`, so this call will still be unresolved. This is the approach used for auto-completion suggestions in the Eclipse IDE.

- Model `querySelector` as returning *any* HTML element within the underlying page. While this approach correctly includes the canvas element, it suffers from returning elements on which `getContext` is undefined. While previous work [Jensen et al., 2011] has focused on tracking elements based on their ids and types, extending this to tracking CSS selector expressions is non-trivial.

Neither solution is really acceptable. In contrast, our analysis will use *pointer information* to resolve the call to `document.querySelector` and then apply *use analysis* to discover that the returned object must have at least three properties: `getContext`, `width`, and `height`, assuming the program runs correctly. Looking through the static heap approximation, only the `HTMLCanvas-Element` has all three properties. Assuming we have the whole program available for analysis, this must be the object returned by `querySelector`. From here on, pointer analysis can resolve the remaining calls to `fillRect` and `strokeRect`.

Applications of the Analysis

The idea of combining pointer analysis and use analysis turns out to be powerful and useful in a variety of settings:

- **Call graph discovery.** Knowledge about the call graph is useful for a range of analysis tools. Unlike C or Java, in JavaScript call graphs are surprisingly difficult to construct. Reasons for this include reflection, passing anonymous function closures around the program, and lack of static typing, combined with an ad-hoc namespace creation discipline.

- **API surface discovery.** Knowing the portion of an important library such as WinRT utilized by an application is useful for determining the application's attack perimeter. In aggregate, running this analysis against *many* applications can provide general API use statistics for a library helpful for maintaining it (e.g., by identifying APIs that are commonly used and which may therefore be undesirable to deprecate).

- **Capability analysis.** The Windows 8 application model involves a manifest that requires *capabilities* such as `camera_access` or `gps_location_access` to be statically declared. However, in practice, because developers tend to over-provision capabilities in the manifest [Guha et al., 2011a], static analysis is a useful tool for inferring capabilities that are actually needed [Grace et al., 2012].

- **Automatic stub creation.** Our analysis technique can create stubs from scratch or identify gaps in a given stub collection. When aggregated across a range of application, over time this leads to an enhanced library of stubs useful for analysis and documentation.

- **Auto-complete.** Auto-complete or code completion has traditionally been a challenge for JavaScript. Our analysis makes significant strides in the direction of better auto-complete, without resorting to code execution, as shown in Section 13.5.

Throughout this paper, our emphasis is on providing a practically useful analysis, favoring practical utility even if it occasionally means sacrificing soundness. Note that our analysis can be a useful building block for (sound) verification tools. We further explain the soundness trade-offs in Section 13.2.

Contributions. We make the following contributions:

- We propose a strategy for analyzing JavaScript code in the presence of large complex libraries, often implemented in other languages. As a key technical contribution, our analysis combines pointer analysis with a novel *use analysis* that captures how objects returned by and passed into libraries are used within application code, without analyzing library code.

- Our analysis is declarative, expressed as a collection of Datalog inference rules, allowing for easy maintenance, porting, and modification.

- Our techniques in this paper include *partial* and *full* iterative analysis, the former depending on the existence of stubs, the latter analyzing the application without any stubs or library specifications.

- Our analysis is useful for a variety of applications. Use analysis improves points-to results, thereby improving call graph resolution and enabling other important applications such as inferring structured object types, interactive auto-completion, and API use discovery.

- We experimentally evaluate our techniques based on a suite of 25 Windows 8 JavaScript applications, averaging 1,587 line of code, in combination with about 30,000 lines of partial stubs. When using our analysis for call graph resolution, a highly non-trivial task for JavaScript, the median percentage of resolved calls sites increases from 71.5% to 81.5% with partial inference.

- Our analysis is immediately effective in two practical settings. First, our analysis finds about twice as many WinRT API calls compared to a naïve pattern-based analysis (Section 13.5). Second, in our auto-completion case study (Section 13.5) we out-perform four major widely-used JavaScript IDEs in terms of the quality of auto-complete suggestions.

13.2 Analysis Challenges

Before we proceed further, we summarize the challenges faced by any static analysis tool when trying to analyze JavaScript applications that depend on libraries.

Whole Program Analysis

Whole program analysis in JavaScript has long been known to be problematic [Chugh et al., 2009; Guarnieri and Livshits, 2010]. Indeed, libraries such as the Browser API, the HTML DOM, NodeJS and the Windows 8 API are all implemented in native languages such as C and C++; these implementations are therefore often simply unavailable to static analysis. Since no JavaScript implementation exists, static analysis tool authors are often forced to create stubs. This, however, brings in the issues of stub completeness as well as development costs. Finally, JavaScript code frequently uses dynamic code loading, requiring static analysis at runtime [Guarnieri and Livshits, 2010], further complicating whole-program analysis.

Underlying Libraries & Frameworks

While analyzing code that relies on rich libraries has been recognized as a challenge for languages such as Java, JavaScript presents a set of unique issues.

Complexity. Even if the application code is well-behaved and amenable to analysis, complex JavaScript applications frequently use libraries such as jQuery and Prototype. While these are implemented in JavaScript, they present their own challenges because of extensive use of reflection such as `eval` or computed property names. Recent work has made some progress towards understanding and handling `eval` [Richards et al., 2011; Jensen et al., 2012], but these approaches are still fairly limited and do not fully handle all the challenges inherent to large applications.

Scale of libraries. Underlying libraries and frameworks are often very large. In the case of Windows 8 applications, they are around 30,000 lines of code, compared to 1,587 for applications on average. Requiring them to be analyzed every time an application is subjected to analysis results in excessively long running times for the static analyzer.

Tracking Interprocedural Flow

Points-to analysis effectively embeds an analysis of interprocedural data flow to model how data is copied across the program. However, properly modeling

interprocedural data flow is a formidable task.

Containers. The use of arrays, lists, maps, and other complex data structures frequently leads to conflated data flow in static analysis; an example of this is when analysis is not able to statically differentiate distinct indices of an array. This problem is exacerbated in JavaScript because of excessive use of the DOM, which can be addressed both directly and through tree pointer traversal. For instance `document.body` is a direct lookup, whereas `document.getElementsByName("body")[0]` is an indirect lookup. Such indirect lookups present special challenges for static analyses because they require explicit tracking of the association between lookup keys and their values. This problem quickly becomes unmanageable when CSS selector expressions are considered (e.g., the jQuery `$()` selector function), as this would require the static analysis to reason about the tree structure of the page.

Reflective calls. Another typical challenge of analyzing JavaScript code stems from reflective calls into application code being "invisible" [Livshits et al., 2005]. As a result, callbacks within the application invoked reflectively will have no actuals linked to their formal parameters, leading to variables within these callback functions having empty points-to sets.

Informal Analysis Assumptions

Here we informally state some analysis assumptions. Our analysis relies on property names to resolve values being either a) returned by library functions or b) passed as arguments to event handlers. Thus, for our analysis to operate properly, it is important that property names are static. In other words, we require that objects being passed to and from the library exhibit class-like behavior:

- Properties are not dynamically added or removed after the object has been fully initialized.
- The presence of a property does not rely on program control-flow, e.g. the program should not conditionally add one property in a `then` branch, while adding a different property in an `else` branch.
- Libraries should not overwrite properties of application objects or copy properties from one application object to another. However, the library is allowed to augment the global object with additional properties.
- Property names should not be computed dynamically (i.e. one should not use `o["p" + "q"]` instead of `o.pq`).

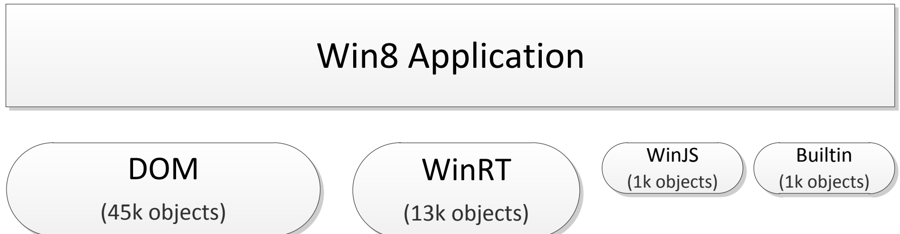


Figure 13.1: Structure of a typical Windows 8 JavaScript app.

Soundness

While soundness is a highly attractive goal for static analysis, it is not one we are pursuing in this paper, opting for practical utility on a range of applications that do not necessarily require soundness. JavaScript is a complex language with a number of dynamic features that are difficult or impossible to handle fully statically [Richards et al., 2011; Jensen et al., 2012] and hard-to-understand semantic features [Maffeis et al., 2008; Gardner et al., 2012].

We have considered several options before making the decision to forgo conservative analysis. One approach is defining language subsets to capture when analysis results are sound. While this is a well-known strategy, we believe that following this path leads to unsatisfactory results.

First, language restrictions are too strict for real programs: dealing with difficult-to-analyze JavaScript language constructs such as with and `eval` and intricacies of the execution environment such as the Function object, etc. While language restrictions have been used in the past [Guarnieri and Livshits, 2009], these limitations generally only apply to small programs (i.e. Google Widgets, which are mostly under 100 lines of code). When one is faced with bodies of code consisting of thousands of lines and containing complex libraries such as jQuery, most language restrictions are immediately violated.

Second, soundness assumptions are too difficult to understand and verify statically. For example, in Ali et al. [Ali and Lhoták, 2012] merely *explaining* the soundness assumptions requires three pages of text. It is not clear how to *statically* check these assumptions, either, not to mention doing so efficiently, bringing the practical utility of such an approach into question.

13.3 Overview

The composition of a Windows 8 (or Win8) JavaScript application is illustrated in Figure 13.1. These are frequently complex applications that are not built in isolation: in addition to resources such as images and HTML, Win8 applica-

Name	Lines	Functions	Alloc. sites	Fields
Builtin	225	161	1039	190
DOM	21881	12696	44947	1326
WinJS	404	346	1114	445
Windows 8 API	7213	2970	13989	3834
Total	29723	16173	61089	5795

Figure 13.2: Approximate stub sizes for widely-used libraries.

tions depend on a range of JavaScript libraries for communicating with the DOM, both using the built-in JavaScript DOM API and rich libraries such as jQuery and WinJS (an application framework and collection of supporting APIs used for Windows 8 HTML development), as well as the underlying Windows runtime. To provide a sense of scale, information about commonly used stub collections is shown in Figure 13.2.

Analysis Overview

The intuition for the work in this paper is that despite having incomplete information about this extensive library functionality, we can still discern much from observing how developers use library code. For example, if there is a call whose base is a variable obtained from a library, the variable must refer to a function for the call to succeed. Similarly, if there is a load whose base is a variable returned from a library call, the variable must refer to an object that has that property for the load to succeed.

Example. A summary of the connection between the concrete pointer analysis and use analysis described in this paper is graphically illustrated in Figure 13.3. In this example, function `process` invokes functions `mute` and `playSound`, depending on which button has been pressed. Both callees accept variable `a`, an alias of a library-defined `Windows.Media.Audio`, as a parameter. The arrows in the figure represent the flow of constraints.

Points-to analysis (downward arrows) flows facts from actuals to formals — functions receive information about the arguments passed into them, while the *use analysis* (upward arrows) works in the opposite direction, flowing *demands* on the shape of objects passed from formals to actuals.

Specifically, the points-to analysis flows variable `a`, defined in `process`, to formals `x` and `y`. Within functions `playSound` and `mute` we discover that these formal arguments must have functions `Volume` and `Mute` defined on them,

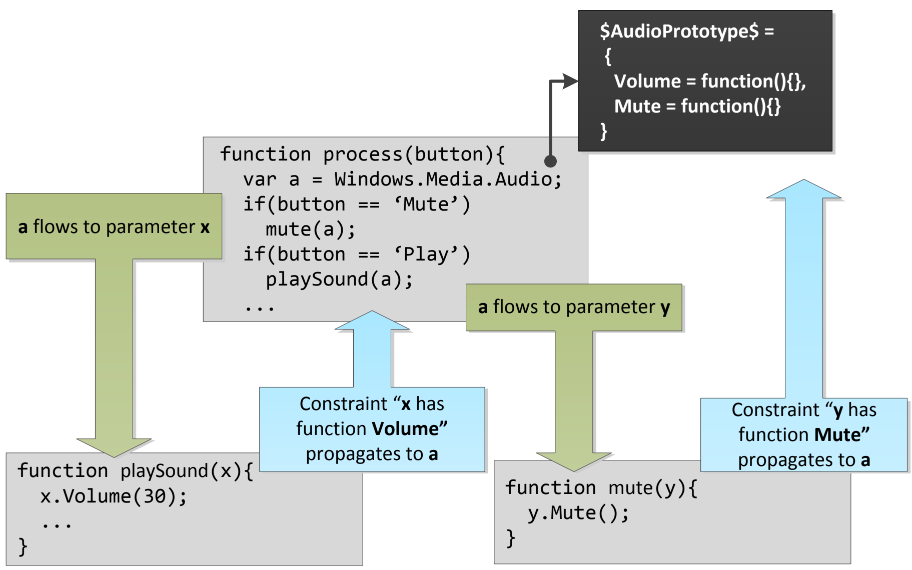


Figure 13.3: Points-to flowing facts downwards and use analysis flowing data upwards.

which flows back to the library object that variable `a` must point to. Its shape as a result must contain at least functions `Volume` and `Mute`.

Use analysis. The notion of use analysis above leads us to an inference technique, which comes in two flavors: *partial* and *full*.

- *Partial inference* assumes that stubs for libraries are available. Stubs are not required to be complete implementations, instead, function bodies are frequently completely omitted, leading to missing data flow. What is required is that all objects, functions and properties (JavaScript term for *fields*) exposed by the library are described in the stub. Partial inference solves the problem of missing flow between library and application code by linking together objects of matching shapes, a process we call *unification* (Section 13.4).

- *Full inference* is similar to partial inference, but goes further in that it does not depend on the existence of any stubs. Instead it attempts to infer library APIs based on uses found in the application. Paradoxically, full inference is often faster than partial inference, as it does not need to analyze large collections of stubs, which is also wasteful, as a typical application only requires a small portion of them.

In the rest of this section, we will build up the intuition for the analysis we formulate. Precise analysis details are found in Section 13.4.

Library stubs. Stubs are commonly used for static analysis in a variety of languages, starting from `libc` stubs for C programs, to complex and numerous stubs for JavaScript built-ins and DOM functionality.
Here is an example of stubs from the WinRT library. Note that stub functions are empty.

```
Windows.Storage.Stream.FileOutputStream =
  function() {};
Windows.Storage.Stream.FileOutputStream.prototype = {
  writeAsync = function() {},
  flushAsync = function() {},
  close = function() {}
}
```

This stub models the structure of the `FileOuputStream` object and its `prototype` object. It does not, however, capture the fact that `writeAsync` and `flushAsync` functions return an `AsyncResults` object. Use analysis can, however, discover this if we consider the following code:

```
var s = Windows.Storage.Stream;
var fs = new s.FileOutputStream(...)
fs.writeAsync(...).then(function() {
  ...
});
```

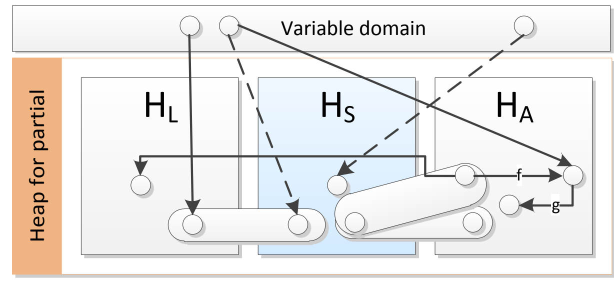
We can observe from this that `fs.writeAsync` should return an object whose `then` is a function. These facts allow us to unify the return result of `writeAsync` with the object, the `Promise [Proto]` prototype of the `Promise` object declared in the WinJS library.

Symbolic Locations and Unification

Abstract locations are typically used in program analyses such as a points-to analysis to approximate objects allocated in the program at runtime. We employ the allocation site abstraction as an approximation of runtime object allocation (denoted by domain H in our analysis formulation). In this paper we consider the partial and full inference scenarios.

It is useful to distinguish between abstract locations in the heap within the application (denoted as H_A) and those within libraries (denoted as H_L). Additionally, we maintain a set of *symbolic locations* H_S; these are necessary for reasoning about results returned by library calls. In general, both library and application abstract locations may be returned from such a call.

It is instructive to consider the connections between the variable V and heap H domains. Figure 13.4a shows a connection between variables and



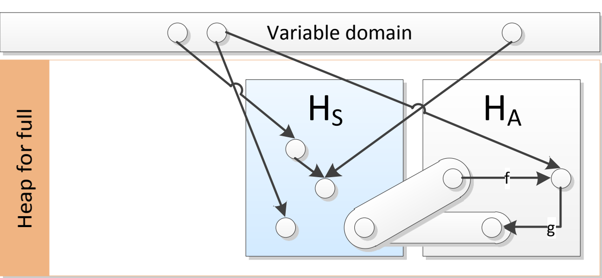


Figure 13.4: Partial (a, above) and full (b, below) heaps.

the heap $H = H_A \cup H_S \cup H_L$ in the context of partial inference. Figure 13.4b shows a similar connection between variables and the heap $H = H_A \cup H_S$ in the context of full inference, which lacks H_L. Variables within the V domain have points-to links to heap elements in H; H elements are connected with points-to links that have property names.

Since at runtime actual objects are either allocated within the application (H_A) or library code (H_L), we need to unify symbolic locations H_S with those in H_A and H_L.

Inference Algorithm

Because of missing interprocedural flow, a fundamental problem with building a practical and usable points-to analysis is that sometimes variables do not have any abstract locations that they may point to. Of course, with the exception of dead code or variables that only point to `null`/`undefined`, this is a static analysis artifact. In the presence of libraries, several distinct scenarios lead to

- **Dead Returns:** when a library function stub lacks a return value;

- **Dead Arguments:** when a callback within the application is passed into a library and the library stub fails to properly invoke the callback;
- **Dead Loads:** when the base object reference (receiver) has no points-to targets.

Strategy. Our overall strategy is to create symbolic locations for all the scenarios above. We implement an iterative algorithm. At each iteration, we run a points-to analysis pass and then proceed to collect dead arguments, returns, and loads, introducing symbol locations for each. We then perform a unification step, where symbolic locations are unified with abstract locations. A detailed description of this process is given in Section 13.4.

Iterative solution. An iterative process is required because we may discover new points-to targets in the process of unification. As the points-to relation grows, additional dead arguments, returns, or loads are generally discovered, requiring further iterations. Iteration is terminated when we can no longer find dead arguments, dead returns, or dead loads, and no more unification is possible. Note that the only algorithmic change for full analysis is the need to create symbolic locations for dead loads. We evaluate the iterative behavior experimentally in Section 13.5.

Unification strategies. Unification is the process of linking or matching symbolic locations with matching abstract locations. In Section 13.4, we explore three strategies: unify based on matching of a single property, all properties, and prototype-based unification.

13.4 Techniques

We base our technique on pointer analysis and use analysis. The pointer-analysis is a relatively straightforward flow- and context-insensitive subset-based analysis described in Guarnieri and Livshits [2009]. The analysis is field-sensitive, meaning that it distinguishes properties of different abstract objects. The call-graph is constructed on-the-fly because JavaScript has higher-order functions, and so the points-to and call graph relations are mutually dependent. The use analysis is based on unification of symbolic and abstract locations based on property names.

Pointer Analysis

The input program is represented as a set of facts in relations of fixed arity and type summarized in Figure 13.5 and described below. Relations use the fol-

NEWOBJ$(v_1 : V, h : H, v_2 : V)$	object instantiation
ASSIGN$(v_1 : V, v_2 : V)$	variable assignment
LOAD$(v_1 : V, v_2 : V, p : P)$	property load
STORE$(v_1 : V, p : P, v_2 : V)$	property store
FORMALARG$(f : H, z : Z, v : V)$	formal argument
FORMALRET$(f : H, v : V)$	formal return
ACTUALARG$(c : C, z : Z, v : V)$	actual argument
ACTUALRET$(c : C, v : V)$	actual return
CALLGRAPH$(c : C, f : H)$	indicates that f may be invoked by a call site c
POINTSTO$(v : V, h : H)$	indicates that v may point to h
HEAPPTSTO$(h_1 : H, p : P, h_2 : H)$	indicates that h_1's p property may point to h_2
PROTOTYPE$(h_1 : H, h_2 : H)$	indicates that h_1 may have h_2 in its internal prototype chain

Figure 13.5: Datalog relations used for program representation and pointer analysis.

APPALLOC$(h : H)$, APPVAR$(v : V)$	represents that the allocation site or variable originates from the application and not the library
SPECIALPROPERTY$(p : P)$	properties with special semantics or common properties, such as `prototype` or `length`
PROTOTYPEOBJ$(h : H)$	indicates that the object is used as a prototype object

Figure 13.6: Additional Datalog facts for use analysis.

lowing *domains*: heap-allocated objects and functions H, program variables V, call sites C, properties P, and integers Z.

The pointer analysis implementation is formulated declaratively using Datalog, as has been done in range of prior projects such as Whaley et al. [2005] for Java and Gatekeeper for JavaScript [Guarnieri and Livshits, 2009]. The JavaScript application is first normalized and then converted into a set of facts. These are combined with Datalog analysis rules resolved using the Microsoft Z3 fixpoint solver [De Moura and Bjørner, 2008].

$$
\begin{array}{lll}
\textsc{PointsTo}(v, h) & :- & \textsc{NewObj}(v, h, _). \\
\textsc{PointsTo}(v_1, h) & :- & \textsc{Assign}(v_1, v_2), \textsc{PointsTo}(v_2, h). \\
\textsc{PointsTo}(v_2, h_2) & :- & \textsc{Load}(v_2, v_1, p), \textsc{PointsTo}(v_1, h_1), \textsc{HeapPtsTo}(h_1, p, h_2). \\
 & & \\
\textsc{HeapPtsTo}(h_1, p, h_2) & :- & \textsc{Store}(v_1, p, v_2), \textsc{PointsTo}(v_1, h_2), \textsc{PointsTo}(v_2, h_2). \\
\textsc{HeapPtsTo}(h_1, p, h_3) & :- & \textsc{Prototype}(h_1, h_2), \textsc{HeapPtsTo}(h_2, p, h_3). \\
\textsc{Prototype}(h_1, h_2) & :- & \textsc{NewObj}(_, h_1, v), \textsc{PointsTo}(v, f), \textsc{HeapPtsTo}(f, \texttt{"prototype"}, h_3). \\
 & & \\
\textsc{CallGraph}(c, f) & :- & \textsc{ActualArg}(c, 0, v), \textsc{PointsTo}(v, f). \\
\textsc{Assign}(v_1, v_2) & :- & \textsc{CallGraph}(c, f), \textsc{FormalArg}(f, i, v_1), \textsc{ActualArg}(c, i, v_2), z > 0. \\
\textsc{Assign}(v_2, v_1) & :- & \textsc{CallGraph}(c, f), \textsc{FormalRet}(f, v_1), \textsc{ActualRet}(c, v_2).
\end{array}
$$

(a) Inference rules for an inclusion-based points-to analysis expressed in Datalog.

$$
\begin{array}{lll}
\textsc{ResolvedVariable}(v) & :- & \textsc{PointsTo}(v, _). \\
\textsc{PrototypeObj}(h) & :- & \textsc{Prototype}(_, h). \\
 & & \\
\textsc{DeadArgument}(f, i) & :- & \textsc{FormalArg}(f, i, v), \neg\textsc{ResolvedVariable}(v), \textsc{AppAlloc}(f), i > 1. \\
\textsc{DeadReturn}(c, v_2) & :- & \textsc{ActualArg}(c, 0, v_1), \textsc{PointsTo}(v_1, f), \textsc{ActualRet}(c, v_2), \\
 & & \neg\textsc{ResolvedVariable}(v_2), \neg\textsc{AppAlloc}(f). \\
 & & \\
\textsc{DeadLoad}(h, p) & :- & \textsc{Load}(v_1, v_2, p), \textsc{PointsTo}(v_2, h), \neg\textsc{HasProperty}(h, p), \textsc{AppVar}(v_1), \textsc{AppVar}(v_2). \\
\textsc{DeadLoad}(h_2, p) & :- & \textsc{Load}(v_1, v_2, p), \textsc{PointsTo}(v_2, h_1), \textsc{Prototype}(h_1, h_2), \\
 & & \neg\textsc{HasProperty}(h_2, p), \textsc{Symbolic}(h_2), \textsc{AppVar}(v_1), \textsc{AppVar}(v_2). \\
\textsc{DeadLoadDynamic}(v_1, h) & :- & \textsc{LoadDynamic}(v_1, v_2), \textsc{PointsTo}(v_2, h), \neg\textsc{ResolvedVariable}(v_1), \\
 & & \textsc{AppVar}(v_1), \textsc{AppVar}(v_2). \\
 & & \\
\textsc{DeadPrototype}(h_1) & :- & \textsc{NewObj}(_, h, v), \textsc{PointsTo}(v, f), \textsc{Symbolic}(f), \neg\textsc{HasSymbolicPrototype}(h). \\
 & & \\
\textsc{CandidateObject}(h_1, h_2) & :- & \textsc{DeadLoad}(h_1, p), \textsc{HasProperty}(h_2, p), \textsc{Symbolic}(h_1), \neg\textsc{Symbolic}(h_2), \\
 & & \neg\textsc{HasDynamicProps}(h_1), \neg\textsc{HasDynamicProps}(h_2), \neg\textsc{SpecialProperty}(p). \\
\textsc{CandidateProto}(h_1, h_2) & :- & \textsc{DeadLoad}(h_1, p), \textsc{HasProperty}(h_2, p), \textsc{Symbolic}(h_1), \neg\textsc{Symbolic}(h_2), \\
 & & \neg\textsc{HasDynamicProps}(h_1), \neg\textsc{HasDynamicProps}(h_2), \textsc{PrototypeObj}(h_2). \\
 & & \\
\textsc{NoLocalMatch}(h_1, h_2) & :- & \textsc{Prototype}(h_2, h_3), \\
 & & \forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_2, p), \\
 & & \forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_3, p), \\
 & & \textsc{CandidateProto}(h_1, h_2), \textsc{CandidateProto}(h_1, h_3), h_2 \neq h_3. \\
 & & \\
\textsc{UnifyProto}(h_1, h_2) & :- & \neg\textsc{NoLocalMatch}(h_1, h_2), \textsc{CandidateProto}(h_1, h_2). \\
 & & \forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_2, p). \\
\textsc{FoundPrototypeMatch}(h) & :- & \textsc{UnifyProto}(h, _). \\
\textsc{UnifyObject}(h_1, h_2) & :- & \textsc{CandidateObject}(h_1, h_2), \neg\textsc{FoundPrototypeMatch}(h_1) \\
 & & \forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \textsc{HasProperty}(h_2, p).
\end{array}
$$

(b) Use analysis inference.

Figure 13.7: Inference rules for both points-to and use analysis.

Rules for the Andersen-style inclusion-based [Andersen, 1994] points-to analysis are shown in Figure 13.7a.

- ASSIGN(v_1, v_2) represents an assignment from v_2 into v_1. We use ASSIGN to model JavaScript assignments, but also for linking actual argument values to formal arguments, and actual return values to formal returns.
- NEWOBJ(v_1, h, v_2) represents the creation of an object at allocation site h which is stored in variable v_1 and v_2 is a variable holding the constructor function. Each syntactic new-statement has its own allocation site. Furthermore each function declaration or expression has an associated allocation site for its prototype object. Finally, we have allocation sites for built-in objects such as the global object, the `Array` object (and its prototype), the `String` object (and its prototype) and so on.
- LOAD(v_1, v_2, p) represents a load from property p. In JavaScript this corresponds to expressions such as $v_1 = v_2.p$. Qualified expressions, such as `foo.bar.baz`, are broken into several atomic `Load` facts by introducing temporary variables. The LOADDYNAMIC(v_1, v_2) constraint represents a dynamic or computed property, i.e. a load where the property name is not known syntactically, which is helpful for modeling arrays.
- STORE(v_1, p, v_2) is similar to LOAD and represents JavaScript expressions such as $v_1.p = v_2$. Likewise STOREDYNAMIC(v_1, v_2) represents a dynamic store where the property name is not known.
- FORMALARG(f, i, v), ACTUALARG(c, i, v), FORMALRET(f, v) and ACTUALRET(c, v) are used for modeling argument passing and function returns. A FORMALARG(f, i, v) fact represents that the i'th argument of function f is read from v. Similarly, ACTUALARG(c, i, v) represents that the i'th actual argument at call site c is stored in v. If a function f flows to a call site c these facts are linked together using ASSIGN. The FORMALRET and ACTUALRET facts are used in a similar way.
- PROTOTYPE(h_1, h_2) states that the internal prototype of h_1 may be h_2. Note that the loads/stores to the external prototype property are handled using regular LOAD and STORE facts.

Extending with Partial Inference

We now describe how the basic pointer analysis can be extended with use analysis in the form of *partial inference*. In partial inference we assume the existence of stubs that describe all objects, functions and properties. Function implementations, as stated before, may be omitted. The purpose of partial inference is to recover missing *flow* due to missing implementations. Flow may be missing in three different places: arguments, return values, and loads.

DEADLOAD$(h : H, p : P)$ where h is an abstract location and p is a property name. Records that property p is accessed from h, but h lacks a p property. We capture this with the rule:

$$\begin{aligned}\text{DEADLOAD}(h,p) \quad :- \quad & \text{LOAD}(v_1, v_2, p),\\ & \text{POINTSTO}(v_2, h),\\ & \neg\text{HASPROPERTY}(h, p),\\ & \text{APPVAR}(v_1),\ \text{APPVAR}(v_2).\end{aligned}$$

Here the POINTSTO(v_2, h) constraint ensures that the base object is resolved. The two APPVAR constraints ensure that the load actually occurs in the application code and not the library code.

DEADARGUMENT$(f : H, i : Z)$ where f is a function and i is an argument index. Records that the i'th argument has no value. We capture this with the rule:

$$\begin{aligned}\text{DEADARGUMENT}(f,i) \quad :- \quad & \text{FORMALARG}(f, i, v),\\ & \neg\text{RESOLVEDVARIABLE}(v),\\ & \text{APPALLOC}(f),\ i > 1.\end{aligned}$$

Here the APPALLOC constraint ensures that the argument occurs in a function within the application code, and not in the library code; argument counting starts at 1.

DEADRETURN$(c : C, v : V)$ where c is a call site and v is the result value. Records that the return value for call site c has no value.

$$\begin{aligned}\text{DEADRETURN}(c, v_2) \quad :- \quad & \text{ACTUALARG}(i, 0, v_1),\\ & \text{POINTSTO}(v_1, f),\\ & \text{ACTUALRET}(i, v_2),\\ & \neg\text{RESOLVEDVARIABLE}(v_2),\\ & \neg\text{APPALLOC}(f).\end{aligned}$$

Here the POINTSTO(v_1, f) constraint ensures that the call site has call targets. The $\neg$APPALLOC(f) constraint ensures that the function called is not an application function, but either (a) a library function or (b) a symbolic location.

We use these relations to introduce symbolic locations into POINTSTO, HEAPPTSTO, and PROTOTYPE as shown in Figure 13.8. In particular for every dead load, dead argument and dead return we introduce a fresh symbolic location. We restrict the introduction of dead loads by requiring that the base

```
INFERENCE(constraints, facts, isFull)
1  relations = SOLVE-CONSTRAINTS(constraints, facts)
2  repeat
3       newFacts = MAKE-SYMBOLS(relations, isFull)
4       facts = facts ∪ newFacts
5       relations = SOLVE-CONSTRAINTS(constraints, facts)
6  until newFacts == ∅
```

```
MAKE-SYMBOLS(relations, isFull)
 1  facts = ∅
 2  for (h, p) ∈ relations.DEADLOAD : H×P
 3       if ¬SYMBOLIC(h) or isFull
 4            facts ∪ = new HEAPPTSTO(h, p, new H)
 5  for (f, i) ∈ relations.DEADARGUMENT : H×Z
 6       v = FORMALARG[f, i]
 7       facts ∪ = new POINTSTO(v, new H)
 8  for (c, v) ∈ relations.DEADRETURN : C×V
 9       facts ∪ = new POINTSTO(v, new H)
10  // Unification:
11  for h ∈ relations.DEADPROTOTYPE : H
12       facts ∪ = new PROTOTYPE(h, new H)
13  for (h1, h2) ∈ relations.UNIFYPROTO : H×H
14       facts ∪ = new PROTOTYPE(h1, h2)
15  for (h1, h2) ∈ relations.UNIFYOBJECT : H×H
16       for (h3, p, h1) ∈ relations.HEAPPTSTO : H×P×H
17            facts ∪ = new HEAPPTSTO(h3, p, h2)
18  return facts
```

Figure 13.8: Iterative inference algorithms.

object is not a symbolic object, unless we are operating in full inference mode. This means that every load must be unified with an abstract object, before we consider further unification for properties on that object. In full inference we have to drop this restriction, because not all objects are known to the analysis.

Unification

Unification is the process of linking or matching symbolic locations s with matching abstract locations l. The simplest form of unification is to do no unification at all. In this case no actual flow is recovered in the application. Below we explore unification strategies based on property names.

$\exists$ **shared properties.** The obvious choice here is to link objects which share at least one property. Unfortunately, with this strategy, most objects quickly

become linked. Especially problematic are properties with common names, such as `length` or `toString`, as all objects have these properties.

$\forall$ shared properties. We can improve upon this strategy by requiring that the linked object must have *all* properties of the symbolic object. This drastically cuts down the amount of unification, but because the shape of s is an over-approximation, requiring all properties to be present may link to too few objects, introducing unsoundness. It can also introduce imprecision: if we have s with function `trim()`, we will unify s to all string constants in the program.

The following rule

$$
\begin{aligned}
\textsc{CandidateObject}(h_1, h_2) \quad :- \quad & \textsc{DeadLoad}(h_1, p), \\
& \textsc{HasProperty}(h_2, p), \\
& \textsc{Symbolic}(h_1), \\
& \neg\textsc{Symbolic}(h_2), \\
& \neg\textsc{HasDynamicProps}(h_1), \\
& \neg\textsc{HasDynamicProps}(h_2), \\
& \neg\textsc{SpecialProperty}(p).
\end{aligned}
$$

expresses which symbolic and abstract locations h_1 and h_2 are *candidates* for unification. First, we require that the symbolic and abstract location share at least one property. Second, we require that neither the symbolic nor the abstract object have *dynamic* properties. Third, we disallow commonly-used properties, such as `prototype` and `length`, as evidence for unification.

The relation below captures when two locations h_1 and h_2 are unified:

$$
\begin{aligned}
\textsc{UnifyObject}(h_1, h_2) \quad :- \quad & \textsc{CandidateObject}(h_1, h_2), \\
& \forall p.\textsc{DeadLoad}(h_1, p) \Rightarrow \\
& \quad \textsc{HasProperty}(h_2, p).
\end{aligned}
$$

This states that h_1 and h_2 must be candidates for unification and that if a property p is accessed from h_1 then that property must be present on h_2. If h_1 and h_2 are unified then the HEAPPTSTO relation is extended such that any place where h_1 may occur h_2 may now also occur.

Prototype-based unification. Instead of attempting to unify with all possible abstract locations l, an often better strategy is to only unify with those that serve as prototype objects. Such objects are used in a two-step unification procedure: first, we see if all properties of a symbolic object can be satisfied by a prototype object, if so we unify them and stop the procedure. If not, we consider all non-prototype objects. We take the prototype hierarchy into consideration by unifying with the most precise prototype object.

The following example illustrates how this can improve precision:

```
var firstName     = "Lucky";
var lastName      = "Luke";
var favoriteHorse = "Jolly Jumper";
function compareIgnoreCase(s1, s2) {
  return s1.toLowerCase() < s2.toLowerCase();
}
```

Here we have three string constants and a comparator function. Assume that the comparator is passed into a library as a callback. In this case the pointer analysis does not know what the two arguments `s1` and `s2` may point to, but the use analysis knows that these arguments must have a `toLowerCase` property. The unification, described so far, would continue by linking the arguments to all abstract locations which have the `toLowerCase` property.

Unfortunately, all string constants have this property, so this over-approximation is overly imprecise. We obtain better unification by first considering prototype objects. In this case we discover that the `String [Proto]` object has the `toLowerCase` property. In prototype-based unification, we merely conclude that the prototype of `s1` and `s2` must be `String [Proto]`.

In the above discussion we did not precisely define what we consider to be prototype objects: we consider all objects which may flow to the `prototype` property of some object to be prototype objects. Furthermore built-in prototype objects, such as `Array [Proto]` and `String [Proto]`, are known to be prototype objects. This is captured by the PROTOTYPEOBJ rule.

Extending with Full Inference

As shown in the pseudo-code in Figure 13.8, we can extend the analysis to support full inference with a simple change. Recall, in full inference we do not assume the existence of any stubs, and the application is analyzed completely by itself. We implement this by dropping the restriction that symbolic locations are only introduced for non-symbolic locations. Instead we will allow a property of a symbolic location to point to another symbolic location.

Introducing these symbolic locations will resolve a load, and in doing so potentially resolve the base of another load. This is in turn may cause another dead load to appear for that base object. In this way the algorithm can be viewed as a frontier expansion along the known base objects. At each iteration the frontier is expanded by one level. This process cannot go on forever, as there is only a fixed number of loads, and thereby dead loads, and at each iteration at least one dead load is resolved.

Namespace Mechanisms

JavaScript has no built-in namespace mechanism and lacks features for proper encapsulation. Thus, many libraries provide functionality for emulating these

features. A common approach is to have a function which takes a string (the namespace) and an object literal, and then creates a namespace of the shape captured by the object literal:

```
WinJS.Namespace.define("GameManager", {
  scoreHelper: new ScoreHelper()
});
GameManager.scoreHelper.newScore(...);
```

This, of course, presents a challenge to static analysis. Luckily, based on the shape information, use analysis and unification can infer that the global variable `GameManager` points to the object literal passed into the `define` function. As a result, we are able to resolve the call to `newScore`.

While unification succeeds for the above example, some namespace mechanisms require more work:

```
var AssetManager = WinJS.Class.define(null, {
  playSound: function (sound, volume) { ... }
});
var assetManager = new AssetManager();
assertManager.playSound();
```

Here the namespace mechanism is used to create a constructor. The prototype of this function is set to the object literal passed into `define`. Of course, the analysis has no way of knowing this, since no implementation is available.

We remedy situations like the one described above by considering objects with *dead prototypes*. We introduce

$$\begin{aligned}\text{DEADPROTOTYPE}(h_1) \quad :- \quad & \text{NEWOBJ}(_, h, v),\\ & \text{POINTSTO}(v, f),\\ & \text{SYMBOLIC}(f),\\ & \neg\text{HASSYMBOLICPROTOTYPE}(h).\end{aligned}$$

to track objects that may have missing prototypes.

As shown in Figure 13.8, at each analysis iteration, for every dead prototype, we introduce a symbolic object into the prototype chain. This symbolic object is then subject to unification like all other symbolic objects. One issue remains: this symbolic prototype object is never considered in the definition of DEADLOAD and so is not subject to unification. We handle this by allowing

use analysis to propagate dead loads upwards in the prototype chain:

$$\begin{aligned}
\textsc{DeadLoad}(h_2, p) \quad :- \quad & \textsc{Load}(v_1, v_2, p), \\
& \textsc{PointsTo}(v_2, h_1), \\
& \textsc{Prototype}(h_1, h_2), \\
& \neg\textsc{HasProperty}(h_2, p), \\
& \textsc{Symbolic}(h_2), \\
& \textsc{AppVar}(v_1), \\
& \textsc{AppVar}(v_2).
\end{aligned}$$

Applied to the example above, the analysis will introduce a symbolic prototype object for the `new` statement. The (unresolved) access to `playSound` is then propagated up the prototype chain. Thus the access to `playSound` becomes a dead load on the symbolic prototype object. This in turn allows unification of the symbolic object and the object literal. Finally, the call to `playSound` is properly resolved using the newly unified object in the prototype chain.

Array Access and Dynamic Properties

We now show how our technique can be extended to deal with array accesses and computed (non-static) property names. We refer to such loads as *dynamic loads*. Unfortunately, reasoning about such loads precisely usually requires more expensive analysis features, such as flow sensitivity and analysis of symbolic expressions. As we do not wish to extend our analysis with these techniques, we opt for a light-weight approach. We extend the analysis to track dead dynamic loads:

- DEADLOADDYNAMIC(v, h) where v is a variable and h is an abstract location, states that there is load of an unknown property from h and the result of this load flows into v.

Unlike regular DEADLOADs we do not wish to introduce a symbolic object for every possible property of H, instead we introduce a single symbolic object and inject it directly into V. This creates a "disconnect" in the heap, but allows the use analysis to proceed.

Here is an example demonstrating this:

```
function copyAll(files, ext, toDir) {
  for(var i = 0; i < files.length; i++) {
    var f = files[i];
    if (f.FileType == ext) {
      f.CopyAsync(toDir);
    }
  }
}
```

The dynamic load occurs within expression `files[i]`. Our analysis does not track integers and conservatively believes that any property could be read from `files`. This in turn causes the use analysis to conclude that `files` may have all possible properties. Usually, this will cause unification to fail and never find any abstract locations to unify with.

We deal with this situation by introducing a *single* special symbolic location. This symbolic location is never unified with anything and so is left unresolved, but it does allow the use-analysis to continue for properties read from that symbolic object. In the above example, this special symbolic location is introduced for `files` and the use analysis proceeds to discover that `f` must have properties `FileType` and `CopyAsync`. This information is then used for unification of `f` with `StorageFile`.

13.5 Experimental Evaluation

This section talks about our experimental setup in Section 13.5. Call graph resolution is discussed in Section 13.5. Sections 13.5 and 13.5 presents case studies of WinRT API use inference and auto-completion. Finally, Section 13.5 talks about analysis performance.

Experimental Setup

We have implemented both the *partial* and *full* inference techniques described in this paper. Our tool is split into a front-end written in C# and a back-end which uses analysis rules encoded in Datalog, as shown in Section 13.4. The front-end parses JavaScript application files and library stubs and generates input facts from them. The back-end iteratively executes the Z3 Datalog engine [De Moura and Bjørner, 2008] to solve these constraints and generate new symbolic facts, as detailed in Section 13.4. All times are reported for a Windows 7 machine with a Xeon 64-bit 4-core CPU at 3.07 GHz with 6 GB of RAM.

We evaluate our tool on a set of 25 JavaScript applications obtained from the Windows 8 application store. To provide a sense of scale, Figure 13.9 shows line numbers and sizes of the abstract domains for these applications. It is important to note the disparity in application size compared to library stub size presented in Figure 13.2. In fact, the average application has 1,587 lines of code compared to almost 30,000 lines of library stubs, with similar discrepancies in terms of the number of allocation sites, variables, etc. Partial analysis takes these sizable stubs into account.

Lines	Functions	Alloc. sites	Call sites	Properties	Variables
245	11	128	113	231	470
345	74	606	345	298	1749
402	27	236	137	298	769
434	51	282	194	336	1007
488	53	369	216	303	1102
627	59	341	239	353	1230
647	36	634	175	477	1333
711	315	1806	827	670	5038
735	66	457	242	363	1567
807	70	467	287	354	1600
827	33	357	149	315	1370
843	63	532	268	390	1704
1010	138	945	614	451	3223
1079	84	989	722	396	2873
1088	64	716	266	446	2394
1106	119	793	424	413	2482
1856	137	991	563	490	3347
2141	209	2238	1354	428	6839
2351	192	1537	801	525	4412
2524	228	1712	1203	552	5321
3159	161	2335	799	641	7326
3189	244	2333	939	534	6297
3243	108	1654	740	515	4517
3638	305	2529	1153	537	7139
6169	506	3682	2994	725	12667
1587	134	1147	631	442	3511

Figure 13.9: Benchmarks, sorted by lines of code.

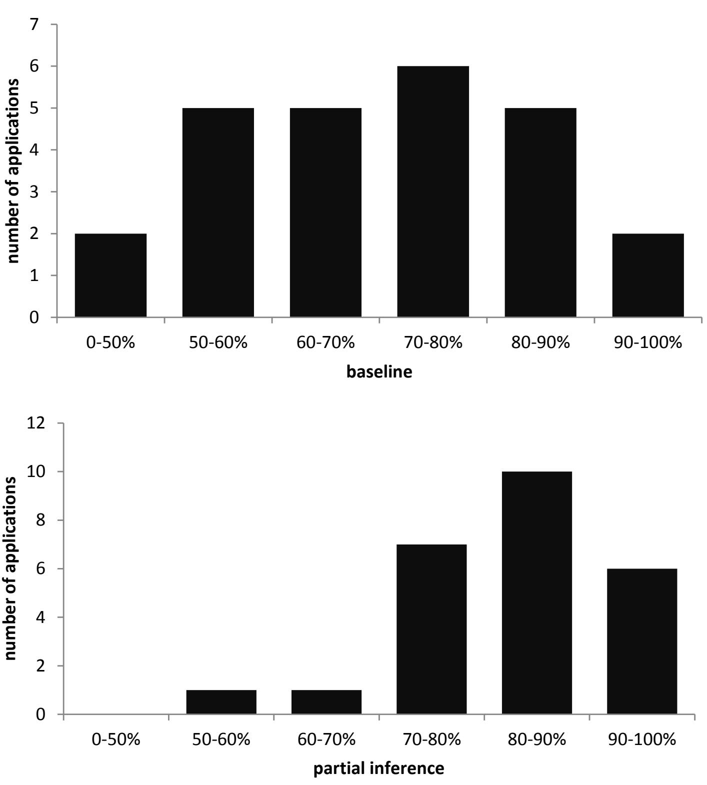


Figure 13.10: Percentage of resolved call sites for baseline (points-to without stubs) and partial inference.

Call Graph Resolution

We start by examining call graph resolution. As a baseline measurement we use the standard pointer analysis provided with stubs *without* use analysis. Figure 13.10 shows a histogram of resolved call sites for baseline and partial inference across our 25 applications. We see that the resolution for baseline is often poor, with many applications having less than 70% of call sites resolved. For partial inference, the situation is much improved with most applications having over 70% call sites resolved. This conclusively demonstrates that the unification approach is effective in recovering previously missing flow. Full inference, as expected, has 100% of call sites resolved.

Note that the leftmost two bars in Figure 13.10 for partial inference are outliers, corresponding to a single application each.

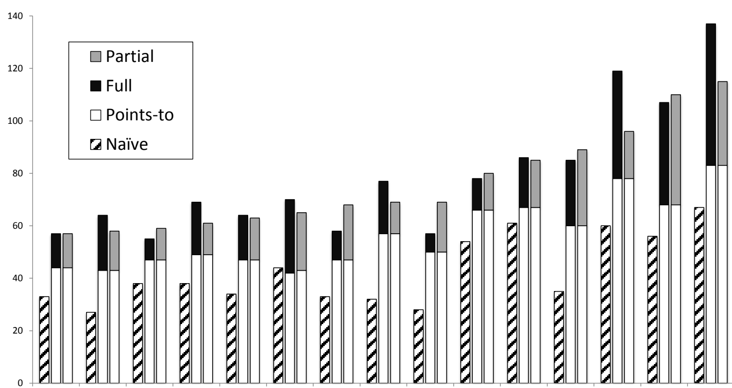


Figure 13.11: Resolving WinRT API calls.

Case study: WinRT API Resolution

Technique	APIs used
naïve analysis	684
points-to	800
points-to + partial	1,304
points-to + full	1,320

We have applied analysis techniques described in this paper to the task of resolving calls to WinRT API in Windows 8 JavaScript applications. WinRT functions serve as an interface to system calls within the Windows 8 OS. Consequently, this is a key analysis to perform, as it can be used to compute an application's actual (as compared to declared) capabilities. This information can identify applications that are over-privileged and which might therefore be vulnerable to injection attacks or otherwise provide a basis for authoring malware. The analysis can also be used for triaging applications for further validation and testing.

Figures in the text show aggregate statistics across all benchmarks, whereas Figure 13.11 represents the results of our analysis across 15 applications, those that are largest in our set. To provide a point of comparison, we implemented a naïve `grep`-like analysis by means of a JavaScript language parser which attempts to detect API use merely by extracting fully-qualified identifier names used for method calls, property loads, etc.

Techniques compared. As expected, we observe that the naïve analysis is very incomplete in terms of identifying WinRT usage.

The base points-to analysis provides a noticeable improvement in results but is still very incomplete as compared to the full and partial techniques. Partial and full analysis are generally comparable in terms of recall, with the

following differences:

- *All* analysis approaches are superior to naïve analysis.
- The number of API uses found by partial and full is roughly comparable.
- Results appearing only in full analysis often indicate missing information in the stubs.
- Partial analysis is effective at generating good results given fairly minimal observed in-application use when coupled with accurate stubs.
- As observed previously, common property names lead to analysis imprecision for partial analysis.

Examples. We want to highlight several examples that come from us further examining analysis results. Observing the following call

```
driveUtil.uploadFilesAsync(server.imagesFolderId).
          then( function (results) {...} ))
```

leads the analysis to correctly map `then` to the `WinJS.Promise.prototype.then` function. A load like:

```
var json = Windows.Storage.ApplicationData.current.
                           localSettings.values[key];
```

correctly resolves `localSettings` to an instance of `Windows.Storage.ApplicationDataContainer`. Partial analysis is able to match results without many observed uses. For instance, the call

```
x.getFolderAsync("backgrounds")
```

is correctly resolved to `getFolderAsync` on object `Windows.Storage.StorageFolder.prototype`.

Case Study: Auto-complete

We show how our technique improves auto-completion by comparing it to four popular JavaScript IDEs: Eclipse Indigo SR2 for JavaScript developers, IntelliJ IDEA 11, Visual Studio 2010, and Visual Studio 2012. Figure 13.12 shows five small JavaScript programs designed to demonstrate the power of our analysis. The symbol "␣" indicates the placement of the cursor when the user asks for auto-completion suggestions. For each IDE, we show whether it gives the correct suggestion (✓) and how many suggestions it presents (#); these tests have been designed to have a single correct completion.

We illustrate the benefits of both partial and full inference by considering two scenarios. For snippets 1–3, stubs for the HTML DOM and Browser

#	Category	Code	Eclipse ✓	Eclipse #	IntelliJ ✓	IntelliJ #	VS 2010 ✓	VS 2010 #	VS 2012 ✓	VS 2012 #
		Partial inference								

1	DOM Loop	`var c = document.getElementById("canvas");`
		`var ctx = c.getContext("2d");`
		`var h = c.height;`
		`var w = c.w␣`
			✗	0	✓	35	✗	26	✓	1

2	Callback	`var p = {firstName : "John", lastName : "Doe"};`
		`function compare(p1, p2) {`
		`  var c = p1.firstName < p2.firstName;`
		`  if(c != 0) return c;`
		`  return p1.last␣`
		`}`
			✗	0	✓	9	✗	7	✓*	k

3	Local Storage	`var p1 = {firstName: "John", lastName: "Doe"};`
		`localStorage.putItem("person", p1);`
		`var p2 = localStorage.getItem("person");`
		`document.writeln("Mr. " + p2.lastName +`
		`", " + p2.f␣);`
			✗	0	✓	50+	✗	7	✗	7

		Full inference								

4	Namespace	`WinJS.Namespace.define("Game.Audio",`
		`  play: function() {}, volume: function() {}`
		`);`
		`Game.Audio.volume(50);`
		`Game.Audio.p␣`
			✗	0	✓	50+	✗	1	✓*	k

5	Paths	`var d = new Windows.UI.Popups.MessageDialog();` `var m = new Windows.UI.␣`	✗	0	✗	250+	✗	7	✓*	k

Figure 13.12: Auto-complete comparison. $\star$ means that inference uses all identifiers in the program. "␣" marks the auto-complete point, the point where the developer presses Ctrl+Space or a similar key stroke to trigger auto-completion.

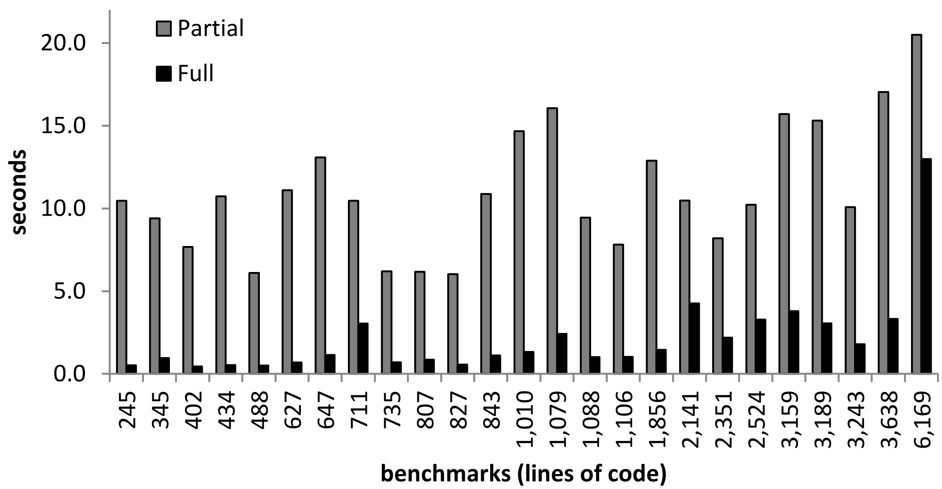


Figure 13.13: Running times, in seconds. Gray is partial inference. Black is full inference. Sorted by lines of code.

APIs are available, so we use partial inference. For Windows 8 application snippets 4–5, no stubs are available, so we use full inference. For all snippets, our technique is able to find the right suggestion *without* giving any spurious suggestions. We further believe our analysis to be incrementalizable, because of its iterative nature, allowing for fast, incremental auto-complete suggestion updates.

Analysis Running Time

Figure 13.13 shows the running times for partial and full inference. Both full and partial analysis running times are quite modest, with full usually finishing under 2–3 seconds on large applications. This is largely due to the fast Z3 Datalog engine. As detailed in our technical report, full inference requires approximately two to three times as many iterations as partial inference. This happens because the full inference algorithm has to discover the namespaces starting from the global object, whereas for partial inference namespaces are known from the stubs. Perhaps surprisingly, despite the extra iterations, full inference is approximately two to four times faster than partial inference.

Examining Result Precision and Soundness

We have manually inspected 20 call sites — twelve resolved, five polymorphic and three unresolved — in 10 randomly picked benchmark apps to estimate the precision and unsoundness of our analysis; this process took about two hours. Figure to the right provides results of our examination. Here *OK* is the number of call sites which are both sound and complete (i.e. their approximated call targets match the actual call targets), *Incomplete* is the number of call sites which are sound, but have some spurious targets (i.e. imprecision is present), *Unsound* is the number of call sites for which some call targets are missing (i.e. the set of targets is *too small*), *Unknown* is the number of call sites for which we were unable to determine whether it was sound or complete due to code complexity, *Stubs* is the number of call sites which were unresolved due to missing or faulty stubs.

App	**OK**	**Incomplete**	**Unsound**	**Unknown**	**Stubs**	**Total**
app1	16	1	2	0	1	20
app2	11	5	1	0	3	20
app3	12	5	0	0	3	20
app4	13	4	1	0	2	20
app5	13	4	0	1	2	20
app6	15	2	0	0	3	20
app7	20	0	0	0	0	20
app8	12	5	0	1	2	20
app9	12	5	0	0	3	20
app10	11	4	0	3	2	20
Total	135	35	4	5	21	200

Out of the 200 inspected call sites, we find 4 sites, which were unsoundly resolved, i.e. true call targets were missing. Of these, three were caused by JSON data being parsed and the fourth was caused by a type coercion, which is not handled by our implementation. Here is an example unsoundness related to JSON:

```
JSON.parse(str).x.split("~")
```

Remember that use analysis has no knowledge of the structure of JSON data. Thus, in the above code, use analysis can see that the returned object should have property `x`, however it cannot find any objects (application or library) with this property; thus it cannot unify the return value.

Furthermore, we found that for 35 call sites the analysis had some form of imprecision (i.e. more call targets than could actually be executed at runtime).

We found two reasons for this: (1) property names may be shared between different objects, and thus if use analysis discovers a property read like `x.slice` (and has no further information about `x`) it (soundly) concludes that `x` could be an array or a string as both has the `x.slice` function. We observed similar behavior for `addEventListener`, `querySelector`, and `appendChild`,

all frequently occurring names in HTML DOM; (2), we found several errors in existing stubs, for instance double-declarations like:

```
WinJS.UI.ListView = function() {};
WinJS.UI.ListView = {};
```

As an example of imprecision, consider the code below:

```
var encodedName = url.slice(url.lastIndexOf('/')+1);
```

Here `url` is an argument passed into a callback. In this case, use analysis knows that `url` must have property `slice`, however both arrays and strings have this property, so use analysis infers that `url` could be either an array or a string. In reality, `url` is a string.

13.6 Related Work

Pointer analysis and call graph construction. Declarative points-to analysis explored in this paper has long been subject of research [Whaley et al., 2005; Livshits and Lam, 2005; Bravenboer and Smaragdakis, 2009]. In this paper our focus is on call graph inference through points-to, which generally leads to more accurate results, compared to more traditional type-based techniques [Grove et al., 1997; Milanova et al., 2004].

Ali and Lhoták [2012] examine the problem of application-only call graph construction for the Java language. Their work relies on the *separate compilation assumption* which allows them to reason soundly about application code without analyzing library code, except for inspecting library types. While the spirit of their work is similar to ours, the separate compilation assumption does not apply to JavaScript, resulting in substantial differences between our techniques.

Static analysis of JavaScript. A project by Chugh et al. [2009] focuses on staged analysis of JavaScript and finding information flow violations in client-side code. Chugh et al. focus on information flow properties such as reading document cookies and URL redirects. A valuable feature of that work is its support for dynamically loaded and generated JavaScript in the context of what is generally thought of as whole-program analysis. The Gatekeeper project [Guarnieri and Livshits, 2009, 2010] proposes a points-to analysis together with a range of queries for security and reliability as well as support for incremental code loading.

Sridharan et al. [2012] presents a technique for tracking correlations between dynamically computed property names in JavaScript programs. Their technique allows them to reason precisely about properties that are copied from one object to another as is often the case in libraries such as jQuery. Their

technique only applies to libraries written in JavaScript, so stubs for the DOM and Windows APIs are still needed.

Type systems for JavaScript. Researchers have noticed that a more useful type system in JavaScript could prevent errors or safety violations. Since JavaScript does not have a rich type system to begin with, the work here is devising a correct type system for JavaScript and then building on the proposed type system. Soft typing [Cartwright and Fagan, 2004] might be one of the more logical first steps in a type system for JavaScript. Much like dynamic rewriters insert code that must be executed to ensure safety, soft typing must insert runtime checks to ensure type safety.

Several project focus on type systems for JavaScript [Anderson et al., 2005; Thiemann, 2005b]. These projects focus on a subset of JavaScript and provide sound type systems and semantics for their restricted subsets. As far as we can tell, none of these approaches have been applied to large bodies of code. In contrast, we use pointer analysis for reasoning about (large) JavaScript programs.

The Type Analysis for JavaScript (TAJS) project [Jensen et al., 2009] implements a data flow analysis that is object-sensitive and uses the recency abstraction. The authors extend the analysis with a model of the HTML DOM and browser APIs, including a complete model of the HTML elements and event handlers [Jensen et al., 2011].

13.7 Conclusions

This paper presents an approach that combines traditional pointer analysis and a novel use analysis to analyze large and complex JavaScript applications. We experimentally evaluate our techniques based on a suite of 25 Windows 8 JavaScript applications, averaging 1,587 lines of code, in combination with about 30,000 lines of stubs each. The median percentage of resolved calls sites goes from 71.5% to 81.5% with partial inference. The median running time for our analysis is 10.5 seconds. In our auto-completion case study we out-perform four major widely-used JavaScript IDEs in terms of the quality of auto-complete suggestions.

14

Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications

By Simon Holm Jensen, Magnus Madsen and Anders Møller published in proc. 8th European Software Engineering Conference / International Symposium on Foundations of Software Engineering (ESEC/FSE '11).

Abstract

Developers of JavaScript web applications have little tool support for catching errors early in development. In comparison, an abundance of tools exist for statically typed languages, including sophisticated integrated development environments and specialized static analyses. Transferring such technologies to the domain of JavaScript web applications is challenging. In this paper, we discuss the challenges, which include the dynamic aspects of JavaScript and the complex interactions between JavaScript, HTML, and the browser. From this, we present the first static analysis that is capable of reasoning about the flow of control and data in modern JavaScript applications that interact with the HTML DOM and browser API.

One application of such a static analysis is to detect type-related and dataflow-related programming errors. We report on experiments with a range of modern web applications, including Chrome Experiments and IE Test Drive applications, to measure the precision and performance of the technique. The experiments indicate that the analysis is able to show absence of errors related to missing object properties and to identify dead and unreachable code. By measuring the precision of the types inferred for object properties,

the analysis is precise enough to show that most expressions have unique types. By also producing precise call graphs, the analysis additionally shows that most invocations in the programs are monomorphic. We furthermore study the usefulness of the analysis to detect spelling errors in the code. Despite the encouraging results, not all problems are solved and some of the experiments indicate a potential for improvement, which allows us to identify central remaining challenges and outline directions for future work.

14.1 Introduction

A JavaScript web application is in essence an HTML page with JavaScript code and other resources, such as CSS stylesheets and image files. Program execution is driven by events in the user's browser: the page is initially loaded, the user interacts with the mouse and keyboard, timeouts occur, AJAX response messages are received from the server, etc. The event handler code reacts by modifying the program state and the HTML page via its DOM (Document Object Model) and by interacting with the browser API, for example to register new event handlers. Compared to other software platforms, the state of the art in development of such web applications is rather primitive, which makes it difficult to write and maintain robust applications. Statically typed languages, such as Java and C#, have long benefited from advanced IDEs and static analysis techniques with rich capabilities of locating likely programming errors during development. Examples of such tools include Eclipse, Visual Studio, and FindBugs. In contrast, existing tool support for JavaScript web application development is mostly limited to syntax highlighting and primitive code completion in IDEs, such as Eclipse, and NetBeans, often combined with record/play testing frameworks, such as Selenium, Watir, and Sahi.

The goal of our research is to develop static program analysis techniques that can detect—or show absence of—potential programming errors in JavaScript web applications. We focus on general errors that can be detected without the use of application-specific code annotations. Examples of such errors are (1) dead or unreachable code, which often indicates unintended behavior, (2) calls to built-in functions with a wrong number of arguments or with arguments of unexpected types, and (3) uses of the special JavaScript value `undefined` (which appears when attempting to read a missing object property) at dereferences or at function calls. The existence of the `undefined` value and implicit type coercions in the language means that even minor spelling errors, for example in a property name, often has surprising consequences at runtime. With statically typed languages, the type systems provide a strong foundation for detecting such errors. In contrast, because of the dynamic nature of JavaScript web application code, our analysis must be capable of reasoning about the flow of control and data throughout the applications.

We strive to make the analysis *sound*, meaning that all control flow and dataflow that is possible in the program being analyzed is captured by the analysis such that guarantees can be made about absence of errors. Also, it must be sufficiently *precise* and *fast* such that the user is not overwhelmed with spurious warnings and that the analysis can be integrated into the development cycle.

As an example, Figure 14.1 shows excerpts from a modern JavaScript web application. If one wants to detect or show absence of errors of the kinds

discussed above, a static analysis must reason about the subtle flow of control and data between the JavaScript code, the HTML code, and the browser event system, as explained in the figure text.

TAJS is a program analysis tool for JavaScript [Jensen et al., 2009, 2010]. To this point, TAJS has been developed to faithfully model the JavaScript language and the core library as specified in the ECMAScript standard [ECMA, 2015]. Most real JavaScript programs, however, exist in the context of an HTML page and operate in browsers where they access the HTML DOM and the browser API, which causes considerable challenges to the analysis of the flow of control and data [Richards et al., 2010]. We now take the step of extending TAJS to also model these aspects of JavaScript web applications.

In summary, the contributions of this paper are the following:

- We discuss the key challenges (Section 14.2) and suggest an approach toward modeling the JavaScript web application platform in static analysis (Section 14.4). In particular, this involves considerations about modeling the HTML pages and the event system.
- We show how the TAJS analysis (Section 14.2) can be extended to accommodate for the HTML DOM and the browser API. As result, we obtain the first static analysis tool that is capable of reasoning about the flow of control and data in JavaScript web applications.
- Through experimental evaluation we demonstrate that our model is sufficient to show absence of errors and to detect dead and unreachable code. In addition, we evaluate the precision of the types and call graphs inferred by the analysis (Section 14.5). We identify strengths and weaknesses of the approaches we have taken and suggest directions for future work (Section 14.7).

Several program analysis tools and techniques for JavaScript have been developed [Thiemann, 2005b; Anderson et al., 2005; Jang and Choe, 2009; Guha et al., 2009; Jensen et al., 2009; Chugh et al., 2009; Guarnieri and Livshits, 2009; Logozzo and Venter, 2010; Guarnieri and Livshits, 2010; Guha et al., 2011b; Fink, 2014], however, none of them provide a detailed model of the HTML DOM and the browser API, although all JavaScript web applications utilize those mechanisms. We describe connections to related work in Section 14.6.

Example. The code in Figure 14.1 is an excerpt from the Google Chrome Experiment *js touch* (where `//...` indicates omitted code). It displays a 3D model of an iPhone and allows the user to interact with it by moving the mouse. The application is written in pure JavaScript and uses the new HTML5 `canvas` object.

```
<html>
<head>
<script type=''text/javascript''>
window.P3D = {
  texture: null,
  g: null
};

P3D.clear = function(f, w, h) {
  var g = this.g;
  g.beginPath();
  g.fillStyle = f;
  g.fillRect(0, 0, w, h);
}

function TouchApp() {
  var _this = this;

  this.canvas = document.getElementById("cv");
  P3D.g = this.canvas.getContext("2d");
  //...

  this.mViewport = {};
  this.mViewport.w = 480;
  this.mViewport.h = 300;
  //...

  var tex = new Image();
  this.ipod.texture = tex;
  tex.onload = function(){ _this.start(); };
  tex.src = "20090319144649.png";
  //...
}

TouchApp.prototype = {
  start: function() {
    //...
    this.onInterval();
  },

  onInterval: function() {
    //...
    P3D.clear("#000",
              this.mViewport.w,
              this.mViewport.h);
    //...
    setTimeout(function(){
      _this.onInterval();
    }, 20);
  }
  //...
}
//...
</script>
</head>
  <body onload="void( new TouchApp() );">
    <canvas id="cv" width="480" height="300"/>
    //...
  </body>
</html>
```

Figure 14.1: Excerpts from the Google Chrome Experiment *JS Touch*.

Obviously, many things could go wrong when programming such an application. Three examples of correctness properties that the programmer may consider are: (1) Is the parameter g on line 11 always an object with a beginPath function? If not, a runtime error will occur when that line is executed. (2) In the call to the function fillRect on line 13, are the arguments always numeric? If not, the function call will not have the desired effect. (3) Is the function P3D.clear on line 9 reachable in some execution? If not, presumably there is an error in the control flow.

To catch such errors – or to show their absence, a static analysis must know about the flow of control and data in the program. In brief, the browser first loads the HTML page and executes the top-level JavaScript code and load event handlers. It then executes other event handlers for user input, timeouts, and other events that occur.

In this example application, the code on line 56 in the onload attribute of the body element creates a new TouchApp object and invokes its constructor function defined on line 16. This function looks up the JavaScript DOM object representing the canvas element on line 19 and then stores a reference to its associated CanvasRenderingContext2D in the g property of the P3D object on line 20. Note that P3D is a globally available object. Next, on line 28, the constructor function creates a new Image object, sets its load event handler to the start function and finally sets its src property. The browser loads the requested image and fires the load handler. The start function, defined on line 36, does some work and then invokes the onInterval function. This function, defined on line 41, calls P3D.clear with appropriate arguments taken from the this.mViewport object. Finally, using a call to setTimeout, it registers itself to be invoked by the browser 20ms later.

By automating this kind of reasoning, a static analysis can detect likely errors in the application code. Analyzing a complex JavaScript program, such as this one, requires a precise model of the JavaScript language, the HTML DOM, and the browser API. For this application, our analysis tool is capable of showing in 9 seconds among many other properties that (1) the variable g does always hold an object with a beginPath function, (2) the fillRect function is always called with numeric arguments, and (3) the function P3D.clear is likely to be reachable. In addition, the analysis reports that 98.9% of all property access operations are guaranteed free from TypeError exceptions caused by dereferencing undefined or null and that all calls to browser API functions are given arguments of meaningful types. More statistics for the unabridged experiment is in Section 14.5.

14.2 Challenges

We begin with a brief tour of the technologies involved and explain the central challenges that exist when developing static analyses for JavaScript web applications. Experienced JavaScript programmers who are used to reasoning "manually" about the behavior of their programs will recognize the issues brought forth here.

The JavaScript Language

The first obstacle we face is the JavaScript language itself. JavaScript has higher-order functions and closures, exceptions, extensive type coercion rules, and a flexible object model where methods and fields can be added or change types and inheritance relations can be modified during execution. As shown by Richards et al. [2010], commonly made assumptions in the research literature about JavaScript programs are often violated by the code actually being written by programmers, and JavaScript is described as "a harsh terrain for static analysis".

Implementations largely follow the ECMAScript standard [ECMA, 2015], however, there are subtle deviations. One such example is that many browsers for performance reasons do not implement the specified behavior of deleting properties of the `arguments` object (as in `delete arguments[0]`). Another example is that many browsers for security reasons do not correctly invoke the currently defined `Object` function when constructing objects from literals (as in `x={}`). Other peculiar JavaScript features and incompatibility issues are discussed in the paper on JavaScript semantics by Maffeis et al. [2008]. One choice we must make is whether to model the standard or one or more of the existing implementations. We return to this issue in Section 14.3.

On top of the language, ECMAScript contains a standard library consisting of 161 functions and other objects that all need to be modeled somehow by any tool that analyzes JavaScript web applications. Of particular interest is the `eval` function and its variant `Function` that allow dynamic construction of program code from text strings. Reasoning statically about the behavior of such code obviously requires knowledge about which strings may appear. Even so, studies of how these constructs are used in practice indicate that many cases are amenable to static analysis [Kromann-Larsen and Simonsen, 2007; Richards et al., 2010, 2011].

For now, we focus on the 3rd edition of ECMAScript (ECMA-262), which is currently the most widely used version. Supporting the more recent 5th edition requires the analysis to also reason about getters and setters, sealed and frozen objects, stronger reflection capabilities, and the so-called strict mode semantics, in addition to a range of new standard library functions.

The HTML DOM and Browser API

The browser environment gives rise to additional challenges. The JavaScript representation of HTML documents, CSS properties, and the event system is specified by the W3C DOM standards[1]. The HTML5 specification is currently being developed by the WHATWG group[2]. Together, these specifications contribute additional hundreds of functions and other objects to the program state. It is well known to all web application programmers that browsers do not adhere to these standards. Browsers provide nonstandard functionality, and many standard features are not supported[3]. In particular the event systems differ between browsers. Another problem is that no standard exists for the `window` object that acts as the global JavaScript object. Incompatibilities in the underlying JavaScript interpreters mostly involve subtle corner cases in the language, as discussed above, and often go unnoticed by the programmers. In contrast, incompatibilities in the browser environments are a major concern. When developing a program analysis, we need to choose which of these variations to model.

A typical workaround is seen in the following function `addEvent` from the Google Chrome Experiment *Tetris*[4].

```
function addEvent(el, event, handler) {
  if (el.addEventListener)
    el.addEventListener(event, handler, false);
  else if (el.attachEvent)
    el.attachEvent("on" + event, handler);
}
```

Reasoning statically about the behavior of such code requires not only modeling of different browsers but also flow sensitivity (i.e. taking statement order into account) and even path sensitivity (i.e. considering the branch conditions) to see that the calls to `addEventListener` and `attachEvent` do not cause `TypeError` exceptions.

Since all execution is driven by events, the analysis must also model the event system, which includes the dynamic registration and removal of different kinds of event handlers, as in the `addEvent` function above, the event bubbling and capturing mechanism, and the event object properties that depend on the specific kind of event. The event handlers work as callbacks, which often leads to fragmented code with unclear flow of control that the static analysis must resolve. A small example is seen in the Chrome Experiment *Aquarium*[5] (abbreviated for presentation):

[1] http://www.w3.org/DOM/
[2] http://www.whatwg.org/
[3] http://www.quirksmode.org/
[4] http://www.chromeexperiments.com/detail/domtris/
[5] http://www.chromeexperiments.com/detail/aquarium/

```
function mmouse(event) {
  mousex=event.pageX;
  mousey=event.pageY;
}
function work() {
  var dx=mousex-pesti[x].x;
  var dy=mousey-pesti[x].y;
  //...
}
setInterval(work,10);
```

The function `mmouse`, which is elsewhere registered as an event handler, stores information about the event in two global variables, `mousex` and `mousey`, that are read in another event handler, `work`. Unless these two variables are properly initialized, `dx` and `dy` will get the special value `NaN` if the `work` function happens to be triggered before `mmouse`, which will likely result in an error later in the execution.

For event handlers defined as HTML attributes, the HTML document structure interferes with the execution scope chains that are used when resolving variables. If an event handler defined literally as an attribute in an HTML element is triggered, the scope chain includes all the DOM objects that make up the path from the HTML element to the root of the document. This means that dataflow in the JavaScript code in general cannot be analyzed separately from the HTML code. The following example illustrates this mechanism:

```
<script type=''text/javascript''>
  var src = "foo.png";
</script>
<img src="bar.png" onclick="alert(src)"/>
```

The value of `src` inside the `onclick` event handler is that of the `src` attribute of the `img` element, not `foo.png` as one might have expected.

Many properties in the ECMAScript native objects have special attributes, such as *ReadOnly*, which also must be accounted for unless sacrificing either soundness or precision. Likewise, many DOM objects behave differently from ordinary objects. As an example, a new `form` element is created with `document.createElement("form")`, not with `new HTMLFormElement` although all `form` elements inherit from `HTMLFormElement.prototype`.

Besides the extent and the variations of browser environments, other concerns when developing a static analysis tool relate to the prevalence of nontrivial built-in setters, that is, assignment operations that involve complex conversions or other side-effects. For example, writing to the `onclick` property of an HTML element object causes a string to be treated as event handler code. Another example is the use of *value correspondence* where HTML element attributes are represented in multiple JavaScript objects. For instance, the `src`

attribute value of an `img` element appears both directly as a property of the `img` element object and indirectly as a property of an object that can be reached via the `attributes` property of the `img` element object. These are essentially aliases (although the former is always an absolute URL even when the latter is a relative URL), and modifications to one also affect the other, much like the connection between ordinary JavaScript function parameters and the `arguments` object. Consider also the `window.location` property, which holds a `Location` object. Assigning a new URL string to this property causes the browser to go to that URL after the current event handler and various unload handlers have been executed. As yet another example, writing a string to the (also nonstandard but widely used) `innerHTML` property of an element object causes the string to be parsed as HTML and converted to a DOM object structure, which then replaces the element contents.

A related issue is the *element lookup* mechanism, which provides support for `getElementById` and related functions. If an element with an `id` attribute is inserted into the HTML document, it is automatically added to the browser's element ID table for quick lookup. Similarly, `documents.images` automatically contains references to all images in the current HTML document.

Application Development Practice

Further complications are introduced by common application development practice. Although JavaScript is an interpreted language (perhaps with JIT compilation, transparently to the programmer) in practice it makes sense to distinguish between "source code" and "executable code". The reason is that JavaScript web application code is often subjected to *minification* (and sometimes also *obfuscation*) to reduce the code size and thereby make the applications load faster. A related trick is *lazy loading* where the applications are divided into parts that are loaded incrementally using AJAX or dynamically constructed `script` elements.

An example of lazy loading using a dynamically created `script` tag occurs in the Google Analytics[6] tool for collecting visitor statistics:

```
<script type="text/javascript">
  (function() {
    var ga = document.createElement('script');
    ga.type = 'text/javascript';
    ga.async = true;
    ga.src =
      ('https:' == document.location.protocol ?
       'https://ssl' :
       'http://www') + '.google-analytics.com/ga.js';
```

[6] http://www.google.com/analytics/

```
    var s = document.getElementsByTagName('script')[0];
    s.parentNode.insertBefore(ga, s);
  })();
</script>
```

Since our aim is to develop an analysis tool that can help the programmers catch errors during development, we choose to focus on the source code stage, as the programmers see the application before these techniques are applied. This means that we in many cases sidestep the issue of analyzing dynamically generated code. It also means, however, that the analysis tool we develop is not designed to be used for all the JavaScript web application code that is immediately available on public web sites, such as Gmail or Office Web Apps.

Many applications build on libraries that alleviate browser incompatibility problems, provide class-like abstractions and advanced GUI widgets and effects, and simplify common tasks, such as navigation in the HTML DOM structures and AJAX communication. This includes general libraries, for example jQuery, MooTools, and Prototype, but also a myriad of more specialized libraries, such as plugins for jQuery. From a static analysis point of view, libraries such as these in many cases make it difficult to track flow of control and data. By providing their own abstractions on top of event handling and DOM objects, a high degree of context sensitivity and detailed modeling of heap structures may be required by the analysis. An example of a challenging library construct is the `$` function in jQuery, which has very different behavior depending on whether it is passed a function, an HTML string, a CSS string, or a DOM element.

14.3 The TAJS Analyzer

We base the current work on the TAJS analysis tool that is described in previous publications [Jensen et al., 2009, 2010]. TAJS is a whole-program flow analysis that supports the full JavaScript language as defined in the ECMA-262 specification [ECMA, 2015], including the entire standard library except `eval`. The analysis is designed to be sound (although working with a real-world language and having no standardized formal semantics of the language nor of the HTML DOM and browser API, soundness is not formally proven). To this point, we do not consider the deviations from the ECMAScript standard that are discussed in Section 14.2, the reason being that these deviations are mostly corner cases that are irrelevant to most applications we have studied. If the need should arise, for all the deviations we are aware of, it is only a matter of making minor adjustments to the analysis tool.

TAJS is based on the classic monotone framework [Kam and Ullman, 1977] using a highly specialized analysis lattice structure. The lattice is based on

constant propagation for all the possible primitive types of JavaScript values. In addition, the lattice includes call graph information, allowing on-the-fly construction of the call graph to handle higher-order functions. It also contains a model of the heap based on allocation site abstraction extended with recency abstraction [Balakrishnan and Reps, 2006].

The analysis is object sensitive, meaning that it distinguishes between calling contexts with different values of `this`. It is also flow sensitive, meaning that it distinguishes between different program points (maintaining separate abstract states for different program points), and it has a simple form of path sensitivity to distinguish between different branches of conditionals.

On top of this, lazy propagation is used to ensure that only relevant parts of the abstract states are propagated, which improves both performance and precision [Jensen et al., 2010].

Altogether, this foundation largely addresses the challenges that are directly related to the ECMAScript language specification.

14.4 Modeling the HTML DOM and Browser API

We now present our approach to extending the analysis to accommodate for the HTML DOM and the browser API.

Regarding the multitude of APIs supported by different browsers that exist, we choose to model the parts that we believe is most widely used: the DOM Core, DOM HTML, and DOM Events modules of the W3C recommendations (Level 2, plus selected parts of Level 3), the essential parts of `window`[7] and related nonstandard objects, and the `canvas` and related objects from WHATWG's HTML5 (as of January 2011). The latter allows us to test the analysis on web applications that exploit cutting edge functionality supported by the newest browsers.

In total, the extensions comprise around 250 abstract objects with 500 properties and 200 transfer functions. To give an impression of the complexity, Figure 14.2 shows a small part of the object hierarchy of the initial abstract state. Each node represents an abstract object with its associated properties and functions, and the edges represent internal prototype links. The symbols `@` and `*` in the names indicate whether the abstract objects represent single or multiple concrete objects.

HTML Objects

The HTML page and resources linked to from the page define not only the program code but also the initial state for the execution, including the HTML

[7] `https://developer.mozilla.org/en/DOM/window`

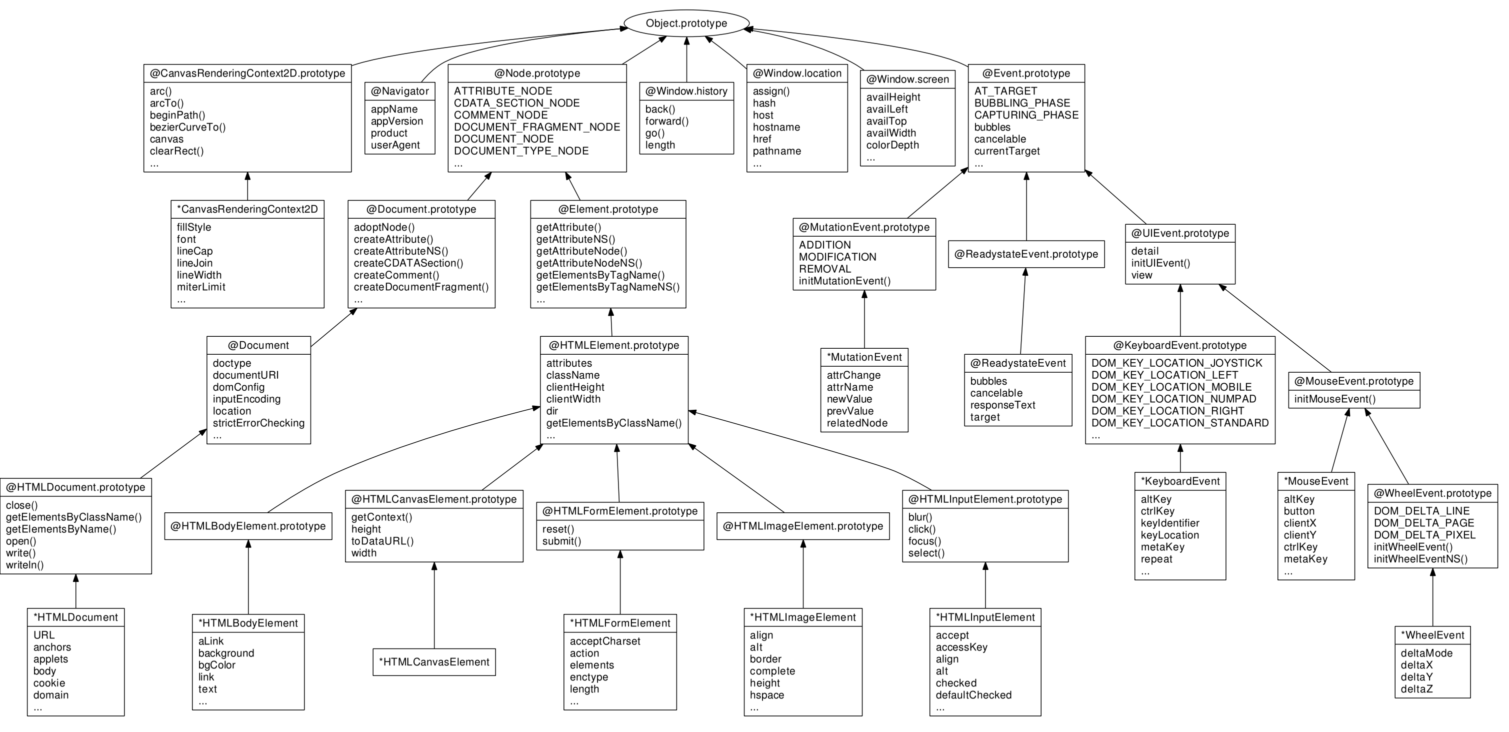


Figure 14.2: An excerpt from the HTML object hierarchy.

document object structure, element lookup tables, and event handlers.

At runtime, each HTML element gives rise to a range of JavaScript objects, and new HTML elements can be created dynamically. We need a bounded representation to ensure that the program analysis terminates (technically, the analysis lattice must have finite height), thus abstraction is necessary. A simple approach is to represent all HTML objects as one abstract object. This is essentially what is done in other program analyses [Guarnieri and Livshits, 2009; Guha et al., 2009] that perform a less detailed analysis than what we aim for. To preserve the inheritance relationships between the DOM objects, we choose an abstraction where all constructor objects and prototype objects are kept separate and that distinguishes between HTML elements of different kinds but where multiple elements of the same kind are merged. As an example, the `HTMLInputElement` abstract object (see Figure 14.2) models all HTML `input` elements. It has properties such as `accessKey` and `checked`, which in the analysis have types `String` and `Boolean`, respectively. The abstract object inherits from `HTMLInputElement.prototype`. This object contains common functionality, such as the `focus` function, shared by all `HTMLInputElement` objects. Looking further up the prototype chain we find `HTMLElement.prototype`, `Element.prototype` and finally `Node.prototype`, which define shared functionality of increasingly general character. Other types of HTML elements, such as `form` or `canvas` elements are similarly modeled by separate abstract objects. This approach respects the inheritance relationships and it smoothly handles programs that dynamically modify the central DOM objects, for example by adding new methods to the prototype objects.

To model the element lookup mechanism (see Section 14.2), we extend TAJS's notion of abstract states with appropriate maps, e.g. from element IDs to sets of abstract objects. The initial abstract state is populated with the IDs that occur in the HTML page. If the HTML page contains an `input` element with an attribute `id="foo"` then the ID map in the abstract state maps `foo` to the `HTMLInputElement` abstract object. These maps are updated during the dataflow analysis if new `id` attributes are inserted into the page. As result, `getElementById` and related functions are modeled soundly and with reasonable precision.

Events

As discussed in Section 14.2, the analysis must be extended to model dynamic registration, triggering, and removal of event handlers. This can be done with various levels of precision. We describe our choices in the following and evaluate the resulting system in Section 14.5.

First, we extend TAJS's abstract states again, this time with a collection of set of references to abstract objects that model the event handler function

objects. To distinguish between different kinds of events and event objects, we maintain one such set for each of the following categories of events: *load*, *mouse*, *keyboard*, *timeout*, *ajax*, and *other*. Object references are added to these sets either statically, due to presence of event attributes (`onload`, `onclick`, etc.) in the HTML page, or dynamically when encountering calls to `addEventListener` or assignments to event attributes during the analysis. This means that the abstract states always contain an upper approximation of which event handlers exist. Note that we choose to abstract away the information about where in the HTML DOM tree the event handlers are registered (i.e. the `currentTarget` of the events). This allows us to ignore event bubbling and capturing. Similarly, we ignore removal of event handlers (`removeEventListener`). These choices may of course affect precision, but analysis soundness is preserved.

Next, we need to model how events are triggered. A JavaScript web application is executed by first running the top-level code and then, until the page is unloaded, running event handlers as reaction to events. Each event handler is executed until completion, without being interrupted when new events occur.

In TAJS, JavaScript program code is represented by flow graphs, which are graphs where nodes correspond to primitive instructions and edges correspond to control flow (see [Jensen et al., 2009]). We have considered different approaches to incorporating the event handler execution loop after the top-level code in the flow graph:

- As a single loop where all event handlers in the current abstract state are executed non-deterministically. This is a simple and sound approach, but it does not maintain the order of execution of the individual event handlers.
- Using a state machine to model the currently registered event handlers. This is a considerably more complex approach, but it can in principle more precisely keep track of the possible order of execution of the event handlers.

Through preliminary experiments we have found for the correctness properties that we focus on, the execution order of event handlers is often not crucial for the analysis precision. However, we found that it is important to model the fact that *load* handlers are executed before the other kinds of event handlers. For this reason, we model the execution of event handlers as shown in Figure 14.3. (To simplify the illustration we here ignore flow of runtime exceptions.) The flow graph for the top-level JavaScript is extended to include two non-deterministic event loops, first one for the *load* event handlers and then one for the other kinds.

If only a single *load* handler is registered (and it is not subsequently removed) then we know that it is definitely executed once, and thus we can effectively remove the dashed edges. This increases precision because otherwise all state initialized by *load* handlers would be modeled as maybe absent.

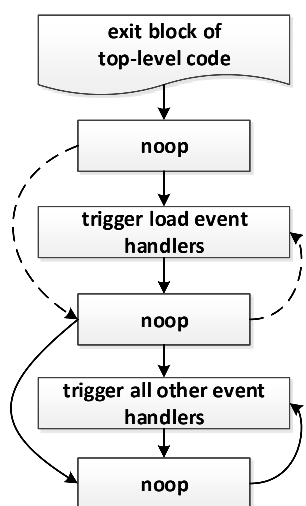


Figure 14.3: Modeling events in the flow graphs.

When triggering event handlers, we exploit the fact that the abstract states distinguish between the different event categories listed above. This allows us to model the event objects appropriately, for example using the abstract object `KeyboardEvent` (see Figure 14.2) to model keyboard event objects. Moreover, the analysis abstraction used in TAJS already has a fine-grained model of scope chains, so it is relatively easy to incorporate the HTML element objects to take the issues regarding scope chains (see Section 14.2) into account.

Special Object Properties

As discussed in Section 14.2, writes to certain object properties, such as `onclick`, `src`, and `innerHTML`, have special side-effects. The TAJS analysis infrastructure conveniently supports specialized transfer functions for such operations. This allows us to trigger the necessary modifications of the abstract state when property write operations occur for certain combinations of abstract objects and property names. With this, we can easily handle code such as the following that dynamically constructs an `img` element and sets the `id` and `onclick` properties, which affects not only the `img` object itself but also the element ID

lookup map and the event handler set:

```
var i = document.createElement("img");
f.id = "myImage";
f.onclick = function {...}
```

With this approach, the abstractions made elsewhere in the analysis can in principle lead to a cascade of spurious warnings. If the analysis detects a property write operation that involves one of the relevant objects but where the property name is unknown due to abstraction, a fully sound analysis would be required to trigger all the possible specialized transfer functions, which could cause a considerable loss of analysis precision. Instead, if this situation occurs, we choose to sacrifice soundness such that the analysis simply emits a general warning and skips the modeling of the special side-effects for that particular property write operation. In our experiments (see Section 14.5), this occurs 0 times, indicating that the analysis is generally precise enough to avoid the problem.

Dynamically Generated Code

We extend TAJS to support certain common cases involving `eval` and the related functions `Function`, `setTimeout`, and `setInterval`. Programmers who are not familiar with higher-order functions often simulate them by using strings instead, such as in this example from the program *Fractal Landscape*[8]:

```
animInterval = setInterval("animatedDraw()", 100);
```

This code works because the function `setInterval` supports being called with a string that will get evaluated in the global scope at the specified intervals. To accommodate for this, TAJS recognizes the syntax of a string consisting of a simple function call. The analysis transfer function for `setInterval` collects not only function objects but also such strings that represent event handler functions. When modeling the triggering of event handlers, the latter functions are then looked up in the global scope.

An often used application of `eval` is to parse JSON data received using AJAX. JSON data describes simple JavaScript object structures that cannot contain functions. TAJS can be configured to assume that string values that are read from AJAX connections contain only JSON data. We model this with the special dataflow value JSONString. If this abstract value is passed to `eval`, the analysis knows that no side-effects can happen, so the result can be modeled using an abstract value consisting of a generic abstract object and unknown primitive values.

[8] `http://10k.aneventapart.com/Entry/60`

14.5 Evaluation

We have extended the pre-existing TAJS analysis tool according to Section 14.4. The tool is implemented in Java and uses the JavaScript parser from the Mozilla Rhino project[9]. The new extensions amount to 7,500 lines of code on top of the existing 21,000 lines (excluding Rhino). Separately, the analysis is integrated into the Eclipse IDE as a plug-in that allows the programmer to view various aspects of the analysis results, as demonstrated in Figure 14.4.

Research Questions

With the implementation, we consider the following research questions regarding the quality of the analysis:

Q1 We wish to study the ability of the tool to detect programming errors of the kinds discussed in Section 14.1. Given that we do not expect many errors in the benchmark programs that presumable are thoroughly tested already, one way to study the analysis precision is to ask: To what extent can the analysis show the absence of errors in real programs? Since the analysis is designed to be sound (however see Section 14.4), absence of a warning from the tool can be interpreted as absence of an error in the program being analyzed.

[9] http://www.mozilla.org/rhino

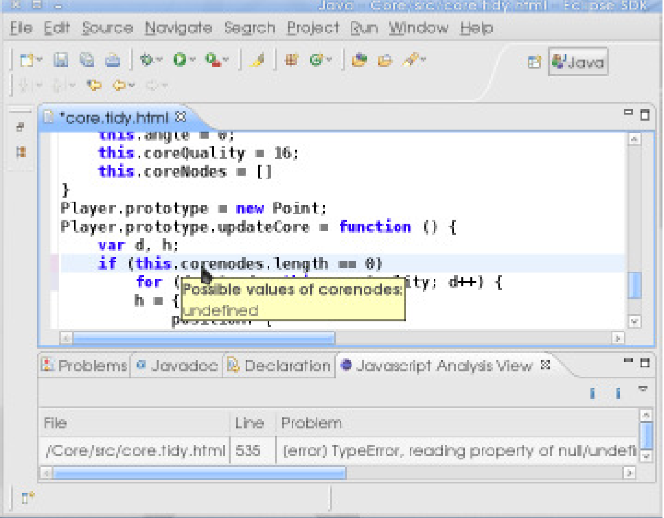


Figure 14.4: The TAJS analysis plug-in for Eclipse, reporting a programming error and highlighting the type inferred for the selected expression.

Q2 For programs with errors (again, of the kinds discussed in Section 14.1), can the analysis help the programmer find the errors? Specifically, are the warning messages produced by the tool useful toward leading the programmer to the source of the errors?

Q3 Having a good approximation of the call graph of a program is a foundation for other potential applications, such as program comprehension or optimization. This leads to the question: How precise is the call graph inferred by the analysis?

Q4 Similarly to the previous question, how precise are the inferred types?

Q5 Does the analysis succeed in identifying dead or unreachable code? In some situations, dead or unreachable code is unintended by the programmer and hence indicates errors. The ability of the analysis tool to detect such code can in principle also be used to reduce application code size before deployment.

Benchmark Programs

Our benchmark programs are drawn from three different sources: *Chrome Experiments*[10], *Internet Explorer 9 Test Drive*[11] and the *10K Apart Challenge*[12].

Chrome Experiments consist of JavaScript web applications that demonstrate the JavaScript features of the Chrome browser. Despite the name, the majority of these applications can be executed in any modern browser. Most of the applications use the new HTML5 `canvas` element to create graphics in various ways including games and simulations. Internet Explorer 9 Test Drive is a collection of applications written to test and demonstrate features of the newest version of the Internet Explorer browser. We exclude applications that contain no or very little JavaScript code or rely on Flash or other browser plug-ins. The 10K Apart Challenge collection consists of JavaScript web applications that are less than 10KB in size including code and markup.

The programmers of some of the 10K Apart Challenge applications have applied `eval` creatively to reduce the code size in ways that we believe are not representative of ordinary JavaScript web applications. For this reason, we disregard applications that syntactically use `eval` in other ways than those covered in Section 14.4. Moreover, analyzing applications that involve large libraries, such as jQuery, MooTools, and Prototype, is particularly challenging for the reasons discussed in Section 14.2. At present, we limit our level of ambition to applications that do not depend on such libraries. The applications

[10] http://www.chromeexperiments.com/
[11] http://ie.microsoft.com/testdrive/
[12] http://10k.aneventapart.com/

we thereby exclude can form an interesting basis for future work on static analysis in relation to `eval` or libraries.

The resulting collection of 53 JavaScript web applications is listed in Table 14.1 and available at http://www.brics.dk/TAJS/dom-benchmarks. In the table, the columns LOC, BB, and Time show the number of lines of code (pretty-printed and including HTML), the number of basic blocks of JavaScript code, and the analysis time (running on a 2.53Ghz Mac OS X computer with 4GB of memory). Dynamically generated code of the kind discussed in Section 14.4 appears in 17% of the applications. All the applications involve HTML and the event system, so none of them could be analyzed with TAJS before the new extensions described in this paper.

Experiments and Results

We address each research question in turn with experiments and evaluation.

For Q1, we focus on the following kinds of likely errors:

- Invoking a non-function value as a function.
- Accessing a property of the special values `undefined` or `null`.
- Reading an absent object property using the fixed-property notation (we here ignore operations that use the notation for dynamically computed property names).

The first two cause `TypeError` exceptions; the third yields the value `undefined`. Technically, these situations are not necessarily errors, but they are rarely intended by the programmer. One exception is that absent properties may appear in browser feature detection code, in which case the analysis can help ensuring that the code works for the browser being modeled.

For each error category we measure the percentage of flow graph nodes for which TAJS decides not to issue a warning of the particular kind. The results are shown in the three columns labelled CF, PA and FPU in Table 14.1, corresponding to the three kinds of likely errors. We see that TAJS is able to show absence of these particular kinds of errors for most of the program code, in many cases more than 90% of the places in the code where the errors could potentially occur. There are a few outliers that get lower results: Both *Tetris* and *Minesweeper* rely on multi-dimensional arrays for most of their state, which leads to imprecision in property reads. Complex object models, such as in the *Raytracer* benchmark, are also the cause of some imprecision.

As we do not expect our benchmarks to contain any of the error conditions listed above, we answer Q2 by introducing errors into the benchmark programs at random. We simulate spelling errors made by the programmer

	LOC	BB	CF	PA	FPU	UF	DC	MC	ATS	Time
3D Demo	1205	1770	99.2	97.9	98.9	125/58	7	100.0%	1.1	8.0s
Another World	1477	1437	100.0	99.3	98.3	45/0	0	100.0%	1.3	20.7s
Apophis	1140	1319	100.0	80.4	80.4	58/0	0	100.0%	1.1	16.3s
Aquarium	166	151	93.7	87.6	72.8	9/0	0	100.0%	1.3	3.2s
Bing-Bong	1148	1176	100.0	87.9	92.5	66/0	2	100.0%	1.1	17.9s
Blob	596	748	100.0	95.6	97.4	37/2	19	100.0%	1.0	6.4s
Bomomo	2905	3885	80.6	96.3	61.2	170/8	10	100.0%	1.3	57.1s
Breathing Galaxies	101	101	94.7	100.0	91.3	5/0	0	100.0%	1.0	1.3s
Browser Ball	434	771	99.0	97.7	98.1	32/11	0	100.0%	1.0	4.2s
Burn Canvas	180	207	100.0	97.7	100.0	12/0	0	100.0%	1.1	0.9s
Catch It	207	200	97.2	86.0	98.6	11/0	0	100.0%	1.1	3.3s
Core	566	611	100.0	98.7	98.4	23/1	10	100.0%	1.0	5.6s
JS Touch	1452	762	100.0	98.9	98.1	48/8	9	100.0%	1.1	5.8s
Kaleidoscope	249	334	98.9	88.6	82.1	14/1	3	100.0%	1.1	6.1s
Keylight	731	791	99.4	96.1	98.7	37/0	24	100.0%	1.0	7.4s
Liquid Particles	253	205	100.0	98.5	100.0	11/4	2	100.0%	1.0	1.8s
Magnetic	415	339	100.0	95.5	100.0	19/0	1	100.0%	1.0	4.1s
Orange Tunnel	102	133	100.0	80.3	100.0	7/1	0	100.0%	1.1	2.6s
Plane Deformations	552	514	100.0	100.0	95.1	17/0	5	100.0%	1.5	1.5s
Plasma	204	228	100.0	100.0	100.0	9/0	2	100.0%	1.1	1.6s
Raytracer	1380	1515	87.2	93.7	55.5	78/24	33	90.1%	1.3	20.6s
Starfield	231	393	98.7	79.0	87.6	21/6	2	100.0%	1.2	2.9s
Tetris	827	803	95.1	79.6	58.8	39/4	2	100.0%	1.8	9.7s
Trail	212	166	100.0	98.0	98.2	10/0	0	100.0%	1.0	12s
Voronoi	525	1066	100.0	78.8	99.7	70/7	10	99.5%	1.1	10.5s
Water Type	309	266	100.0	95.0	97.2	14/0	0	100.0%	1.1	1.9s
Asteroid Belt	319	707	100.0	94.6	97.0	27/5	30	100.0%	1.1	3.1s
Browser Flip	507	324	100.0	88.5	97.6	10/0	1	100.0%	1.1	3.2s
FishIE	336	717	99.4	96.0	95.5	19/2	30	100.0%	1.0	3.3s
Flying Images	589	497	100.0	97.5	91.8	33/0	0	100.0%	1.0	3.9s
Mr. Potato Gun	817	1015	98.7	97.6	95.0	31/1	12	100.0%	1.1	7.8s
10k World	439	930	100.0	86.9	91.4	47/2	3	100.0%	1.1	15.1s
3D Maker	427	773	100.0	67.3	70.5	29/3	0	100.0%	1.2	10.3s
Attractor	445	696	97.0	92.3	91.2	34/0	1	100.0%	1.3	5.8s
Defend Yourself	517	601	94.7	78.6	90.1	31/0	0	100.0%	1.1	7.9s
Earth Night Lights	129	245	100.0	100.0	100.0	14/0	0	100.0%	1.0	1.1s
Filterrific	697	995	96.5	86.7	72.3	55/0	4	99.0%	1.2	29.8s
Flatwar	444	685	99.2	97.4	93.6	19/1	0	100.0%	1.1	6.9s
Floating Bubbles	381	693	100.0	89.9	99.7	39/6	23	100.0%	1.1	6.4s
Fractal Landscape	171	162	100.0	100.0	97.7	7/0	0	100.0%	1.0	0.8s
Gravity	231	258	98.7	87.3	90.9	9/0	0	100.0%	1.0	5.2s
Heatmap	255	350	95.1	93.6	87.3	30/1	2	97.3%	1.1	3.1s
Last Man Standing	300	570	100.0	95.9	100.0	33/1	2	100.0%	1.1	4.2s
Lines	459	931	97.3	88.5	93.9	22/6	2	100.0%	1.2	4.7s
Minesweeper	175	358	100.0	81.4	68.5	15/0	3	100.0%	1.3	4.7s
NBody	479	450	99.1	68.7	43.6	15/0	0	100.0%	1.6	50.8s
RGB Color Wheel	455	700	97.7	82.7	85.0	38/0	2	100.0%	1.1	5.6s
Sinuous	349	488	100.0	96.3	98.5	23/0	10	100.0%	1.0	5.5s
Snowpar	338	519	100.0	88.6	88.6	31/0	0	100.0%	1.2	3.2s
Stairs to Heaven	210	422	100.0	94.5	100.0	25/8	1	100.0%	1.0	2.5s
Sudoku	316	612	96.2	81.0	60.4	33/0	0	100.0%	1.3	12.1s
TicTacToe	304	590	100.0	74.0	100.0	19/0	0	100.0%	1.2	7.4s
Zmeyko	344	601	100.0	96.7	96.3	33/1	0	100.0%	1.0	7.0s

Table 14.1: Benchmark results for Chrome Experiments, IE Test Drive and 10K Apart Challenge applications. The columns from left to right are: lines of code (LOC), number of basic blocks (BB), percentage of call site operations shown to invoke a function value (CF), property read operations where the base object is shown to be non-`null` and non-`undefined` (PA), fixed-property read operations not resulting in `undefined` (FPU), number of functions in total / number of functions shown to be definitely unreachable (UF), number of dead code operations (DC), percentage of call sites that are shown to be monomorphic (MC), average type size for all property read operations (ATS), and analysis time (Time).

by picking a random read or write property operation that uses the fixed-property notation (i.e. the `.` operator) and replacing the property name with a different one. For each benchmark, we run the analysis repeatedly and manually inspect whether each spelling error results in a warning by the analysis tool and how "useful" this warning is. We measure usefulness by two criteria: the source location of the warning that is issued should be close to where the error is inserted, and the warning should be prominent, i.e., appear near the top in the list of analysis messages.

This process has been carried out for a random subset of our benchmark programs. All show a common pattern: Spelling errors at read operations are reliably detected with a warning that appears at the top of the list of analysis messages. Not surprisingly, spelling errors introduced at write operations have more diverse consequences, as any warning will only occur when the program later attempts to read the property that was affected. Furthermore, errors introduced in connection to side-effects that are not modeled by TAJS, such as the DOM property `style`, are often not detected.

We present the results for the *Mr. Potato Gun* benchmark as a representative example. We analyzed it 50 times with a different spelling error introduced each time. In 84% of the cases the error resulted in one or more warnings. Of the errors introduced, 7 were in write operations and 43 in read operations. Only one of the write operation errors was detected, resulting in the warning `ReferenceError, reading absent property: (computed name)`, which is a high-priority warning that is issued for the location where the program tries to read the property that was misspelled. For the read operations, each error was reported as a warning such as `ReferenceError, reading absent property: AQ` issued for the exact source location of the error.

These experiments indicate that the information obtained by the analysis can be useful for detecting spelling errors in the program code, but a more thorough investigation is necessary to give a solid answer to Q2.

For Q3 we wish to evaluate the precision of the computed call graph. This is measured by calculating the ratio of call sites with a single invocation target compared to the total number of call sites in the program. If this ratio is one then every call site is monomorphic, i.e. it has a single invocation target. If a call site has a non-function value as a potential invocation target this is not included in the number of targets, since such a value would always result in a runtime error. This measure can be seen in the MC column. In Table 14.1 we see that despite the fact that JavaScript supports both the prototype lookup mechanism and higher-order functions, the analysis is able to show for 49 of the 53 of the benchmark programs that all call sites have a single invocation target, which gives testimony to the high precision of the analysis.

For Q4 we wish to measure the precision of the computed types. The

analysis tracks values of the following types: *boolean*, *number*, *string*, *object* (including null and function values) and the special type *undefined*. This means that an object property could potentially hold values of up to five different types. We measure this aspect of the accuracy of the analysis by calculating the average number of different types for all property read operations in the given program (excluding operations that the analysis finds to be unreachable). If this number is 1 then every read operation results in values of a unique type on all possible executions. The ATS column in Table 14.1 shows the resulting numbers. Despite the fact that the types of object properties may change dynamically in JavaScript, we note that the analysis is precise enough to show that the average number of different types for each property read operation in these benchmarks is quite close to 1. Of the 26,870 property read operations that appear in the benchmarks, the analysis finds that at most 4,019 can have multiple types.

For the last research question, Q5, we measure both unreachable code and dead code. Unreachable code consists of operations (i.e. flow graph nodes) that are never executed, and dead code is defined to be reachable assignments to properties that are never read. Write operations to special DOM properties, such as `onload`, may have side-effects, so even if there are no corresponding read operations in the program we do not count them as dead code.

The column labelled UF in Table 14.1 contains the total number of function in the program and how many of them are determined by TAJS to be unreachable. Some of the benchmarks use third-party libraries that are inlined directly in the source code, which explains the large number of unreachable functions in some benchmarks, such as *3D Demo* and *Raytracer*. All code that is found to be unreachable can safely be removed (unless the analysis detects the special situation discussed in Section 14.4), which would significantly reduce code size in some cases. Most current minifiers either unsoundly remove all functions not referenced syntactically in the code or simply do not remove any functions at all. With static analysis, guaranteed behavior preserving minification becomes possible.

The column labelled DC lists the number of dead code operations in each program. We see that the analysis is capable of locating many instances of dead code. Most of the dead code being detected appears to be code left from earlier revisions of the programs. For example, in the *Keylight* benchmark, a flag named `mouseIsDown` is set in all event handlers but it is never read.

The main threat to validity of our conclusions is that our benchmarks may not be representative for typical JavaScript web applications. For the reasons described in Section 14.5 we have excluded applications that rely on large libraries or on complex dynamically generated code. We will focus our attention on these two remaining challenges in future work. Nevertheless, the

benchmarks we consider are written by many different programmers, they exhibit a large variety of the functionality supported by the HTML DOM and the browser API, and our experiments show that the program analysis is able to infer many nontrivial properties about their behavior.

14.6 Related Work

Previous work on static analysis of JavaScript code has focused on the language itself, and often for restricted subsets of the language. To the best of our knowledge, the work reported on in this paper is the first that also models the nontrivial connections between the HTML page and the program code in JavaScript web applications.

One of the first attempts at developing static analysis for JavaScript was done by Anderson et al. [2005] who developed a type system and inference algorithm for modeling definite presence and potential absence of object properties in a small subset of JavaScript. The abstract domain used in TAJS subsumes such information. Other early work include the type system of Thiemann [2005b]. It has a soundness proof but no implementation. Although not tied to JavaScript in particular, Thiemann [2005a] has also designed a type system for catching errors related to manipulation of DOM structures, in particular to ensure that no loops occur.

More recently, Jang and Choe [2009] have presented a points-to analysis for a restricted subset of JavaScript based on set constraints. The points-to results are used for optimizations that inline property accesses. In comparison, our analysis yields points-to information as part of the result and supports more features of the language.

The Gatekeeper project by Guarnieri and Livshits [2009, 2010] includes an Andersen-style points-to analysis for JavaScript. The results of the analysis are used for verifying custom security policies expressed in datalog. The analysis uses a mock-up of the DOM API written in JavaScript and essentially ignores the HTML constituents.

Perhaps most closely related to our work is that of Guha et al. [2009] who use a k-CFA analysis to extract a model of the client behavior in an AJAX application as seen from the server. Their paper briefly discusses some of the challenges that relate to events, dynamically generated code, and libraries, but the focus of the paper is on the application for building intrusion-preventing proxies. In comparison, our analysis has a more precise treatment of dataflow and event handlers in connection to the DOM.

Recent work by Guha et al. [2011b] considers a combination of a type system and a flow analysis to reason about uses of the `typeof` operator in

JavaScript code with type annotations. The `typeof` operator appears in 11 of our 53 benchmarks, and TAJS models it with a special transfer function.

Chugh et al. [2009] use staged information flow analysis to protect against dynamic loading of malicious code. The analysis identifies fields that can flow into dynamically loaded code and creates runtime monitors to ensure that they are not accessed from untrusted code. The analysis uses a coarse abstraction of the HTML page and the browser API, without considering the challenges we describe in Section 14.2.

The RATA analysis of Logozzo and Venter [2010] uses light-weight abstract interpretation to specialize the general JavaScript number type to integer and floating point types for optimization purposes. Making this distinction in the abstract domain used in TAJS would be a straightforward task.

One way to guide the design of an analysis is to survey the practical use of the language. In one such survey by Richards et al. [2011] it is shown that many of the dynamic features of JavaScript are not widely used in practice. The study shows that the majority of method invocations in JavaScript are monomorphic. Our experimental results confirm this observation, but using practically sound static analysis instead of runtime measurements. In later work, the use of `eval` is studied [Richards et al., 2011]. The authors show that the categories of `eval` that are now supported by TAJS, i.e. JSON data and simple function calls, are often used. It is also shown that `eval` is used for lazy loading and as artifacts of generated code, which, as discussed in Section 14.2, is outside the scope of TAJS.

14.7 Conclusion

We have presented the first static analysis that is capable of reasoning precisely about the control flow and dataflow in JavaScript applications that run in a browser environment. The analysis has been implemented as an extension of TAJS and models both the DOM model of the HTML page and browser API. This includes the HTML element object hierarchy and the event-driven execution model. In the process we have identified the key areas where modeling the browser is important for precision and challenging for static analysis.

Our experimental evaluation of the performance of the analysis indicates that (1) the analysis is able to show absence of common programming errors in the benchmark programs, (2) the analysis can help detecting potential errors, such as misspelled property names, (3) the computed call graphs are precise as most call sites are shown to be monomorphic, (4) the computed types are precise as many expressions are shown to have unique types, and (5) the analysis is able to identify dead code and unreachable functions. Such information can give a foundation for providing better tool support for JavaScript web

application developers.

Interesting challenges remain. First, more work is required for investigating the more complicated uses of dynamically generated code. Second, better techniques are needed to handle commonly used libraries. Third, the techniques presented here can be adapted to model other JavaScript environments, such as desktop widgets or browser extensions.

15

Sparse Dataflow Analysis with Pointers and Reachability

By Magnus Madsen and Anders Møller published in proc. 21st International Static Analysis Symposium (SAS '14).

Abstract

Many static analyzers exploit sparseness techniques to reduce the amount of information being propagated and stored during analysis. Although several variations are described in the literature, no existing technique is suitable for analyzing JavaScript code. In this paper, we point out the need for a sparse analysis framework that supports pointers and reachability. We present such a framework, which uses static single assignment form for heap addresses and computes def-use information on-the-fly. We also show that essential information about dominating definitions can be maintained efficiently using quadtrees. The framework is presented as a systematic modification of a traditional dataflow analysis algorithm.

Our experimental results demonstrate the effectiveness of the technique for a suite of JavaScript programs. By also comparing the performance with an idealized staged approach that computes pointer information with a pre-analysis, we show that the cost of computing def-use information on-the-fly is remarkably small.

15.1 Introduction

Previous work on dataflow analysis has demonstrated that sparse analysis is a powerful technique for improving performance of many kinds of static analysis without sacrificing precision [Reif and Lewis, 1977; Wegman and Zadeck, 1991; Tok et al., 2006; Hardekopf and Lin, 2009, 2011; Oh et al., 2012], compared to more basic dataflow analysis frameworks [Kildall, 1973; Kam and Ullman, 1977]. The key idea in sparse analysis is that dataflow should be propagated directly from definitions to uses in the program code, unlike "dense" analysis that propagates dataflow along the control-flow. A potential advantage of sparse analysis is that it propagates and stores only relevant information, not entire abstract states. Another advantage is that transfer functions need only be recomputed when their dependencies change.

While developing analysis tools for JavaScript we have found that the existing approaches described in the literature for building sparse analyses do not apply to the language features and common programming patterns that appear in JavaScript code. Specifically, context-sensitive branch pruning (a variant of unreachable code elimination by Wegman and Zadeck [1991]) is an important analysis technique, as explained below, for handling the use of function overloading, which in JavaScript is programmed using reflection. Moreover, the common wisdom from analysis of e.g. Java code that context-insensitive analysis is usually faster than context-sensitive analysis [Smaragdakis et al., 2011] apparently does not apply to JavaScript code, which, as discussed below, makes it difficult to design useful staged sparse analyses for this language.

```
function f(b, x, y) {
  var r;
  if (b.p) {
    r = x.a;
  } else {
    r = y.a;
  }
  return r;
}
```

As a motivating example, consider the JavaScript function on the right that exhibits a form of overloading. Here, the branch condition `b.p` decides whether the `f` function should have one behavior or another (in real-world JavaScript code, complex function overloading is mimicked using various kinds of reflection in branch conditions, but the pattern is the same). It is often the case that the branch condition is *determinate* relative to the call context [Schäfer et al., 2013a]. That is, in one call context, `b.p` is known to be true, and in another call context, it is known to be false. In a context-sensitive dense analysis, this is no problem for the precision: `f` is simply analyzed in two contexts, corresponding to the two cases, such that the analysis logically clones `f` and analyzes it twice. When dataflow reaches the `if` statement, the analysis can then discover that one branch is dead and only propagate dataflow along the live branch.

To reason precisely about such program code, for example, with the purpose of computing call graphs or information about types of expressions, a static analysis must account for *reachability*, i.e. whether branches are live or dead in the individual contexts. At the same time it must handle heap allocated storage, as objects are pervasive in JavaScript. Moreover, even apparently simple operations in JavaScript, such as reading an object property, are complex procedures that involve e.g. type coercion and traversal of dynamically constructed prototype chains. This makes it beneficial to design analysis techniques that support complex transfer functions, for example, all computable monotone functions [Kam and Ullman, 1977]. It is well known how to accomplish all this using dense analysis (see e.g. the TAJS analysis [Jensen et al., 2009]). Our goal is to take the step to sparse analysis, without sacrificing precision compared to the original dense version.

One way to build sparse analyses for programs with pointers is to use a staged approach where a pre-analysis computes a sound approximation of the memory addresses that are defined or used at each operation in the code, and then establish def-use edges that the main analysis can use for sparse flow-sensitive dataflow propagation [Hardekopf and Lin, 2011; Oh et al., 2012]. Unfortunately, this does not work in our setting, unless the pre-analysis is as precise as the main analysis (and in that case, there would be no need for the main analysis, obviously): If the pre-analysis is flow-insensitive, for example, it would establish def-use edges from both `x.a` and `y.a` to `r` in our example, which would destroy the precision of the main analysis. Note that one of the main results of Oh et al. [2012] is that, in their setting, approximations in the pre-analysis may lead to less sparseness but it will never affect the precision of the main analysis (due to their use of data dependence instead of def-use chains). However, for a language like JavaScript where most operations may throw exceptions, their algorithm largely degrades to a dense analysis if reachability is involved. In another line of work, Tok et al. [2006] and Chase et al. [1990] compute def-use edges on-the-fly rather than using a pre-analysis, but also without taking reachability into account. Conversely, the sparse conditional constant analysis by Wegman and Zadeck [1991] handles reachability, but not pointers. In summary, no existing technique satisfies the needs for making sparse analysis for languages like JavaScript.

Our contributions are as follows:

- We present the first algorithm for sparse dataflow analyses that supports pointers, reachability, and arbitrary monotone transfer functions, while preserving the precision of the corresponding dense analysis, and without requiring a pre-analysis to compute def-use information.

- We describe experimental results, based on a dataflow analysis for JavaScript, that show a considerable performance improvement when

using sparse analysis compared to a traditional dense approach, which demonstrates that it is possible to perform efficient sparse analysis in a setting that involves pointers and reachability.

- We show experimentally that the overhead of computing dominating definitions on-the-fly is small, which makes our approach preferable to staged approaches that compute that information with a pre-analysis.
- We demonstrate that quadtrees are a suitable data structure for maintaining essential information about dominating definitions in sparse analysis.

We explain the technique as a framework where we can switch from dense to sparse analysis, without affecting the abstract domains or transfer functions.

15.2 A Basic Analysis Framework

Our starting point is a variant of the classical monotone framework for flow-sensitive dataflow analysis Kam and Ullman [1977] where programs are represented as control-flow graphs with abstract states associated with the entry and exit program points of each node. For simplicity this presentation focuses on intraprocedural analysis, although our implementation supports interprocedural analysis as discussed in Section 15.4.

We assume that we are given a control-flow graph where each node represents a statement $s \in S$, together with a set of abstract memory addresses $a \in A$, a lattice of abstract values $v \in V$, and a transfer function T_s for each statement s. The transfer functions are assumed to be expressed using the following three primitive operations:

- $\textsc{Read}(s \in S, a \in A) : V$. Returns the value v at the address a at the program point immediately before the statement s.
- $\textsc{Write}(v \in V, s \in S, a \in A)$. Writes the value v to the address a at the program point immediately after the statement s. An invocation of $\textsc{Write}(v, s, a)$ models a strong update [Chase et al., 1990]. A weak update is achieved with $\textsc{Write}(v \sqcup \textsc{Read}(s, a), s, a)$.
- $\textsc{Continue}(s_{src} \in S, s_{dst} \in S)$. Indicates that the transfer function $T_{s_{src}}$ has completed and that s_{dst} is a possible successor, in other words that s_{dst} is reachable from s_{src}. (For example, this allows the transfer function for an `if` statement to selectively propagate dataflow to one of its branches.)

As conventional, the framework applies the transfer functions using an iterative worklist algorithm, starting from a designated program entry statement, until the global fixpoint is reached. The ordering of the worklist W is left unspecified, so the analysis implementor may freely choose any. We assume the lattice V has finite height; for simplicity we ignore widening.

In the case of JavaScript, most transfer functions are complex operations that involve multiple READ and WRITE operations. For example, the transfer function for a simple assignment `x = y.p` in general requires traversal of scope chains and prototype chains. This can be accomplished as shown in previous work on the TAJS analysis [Jensen et al., 2009]. Although that analysis uses more elaborate abstract domains, it can in principle all be expressed within the present framework.

A traditional dense propagation strategy Kam and Ullman [1977] maintains an entire abstract state at each program point as a map from addresses to values:

$I : S \times A \to V$ is the map of *incoming states*

$O : S \times A \to V$ is the map of *outgoing states*

Reading from an address is then simply a matter of looking up its value in the abstract state in I, and writing similarly updates O. (In practice, analysis implementations often maintain only O, since the information in I can be inferred when needed; we include both maps explicitly to simplify the presentation in Section 15.3.) Continuing from s_{src} to s_{dst} is handled by joining the entire outgoing state at s_{src} into the incoming state at s_{dst}. To initiate the analysis, I and O return the bottom element $\bot$ of V for every statement and address, except that we assume an entry statement s_{entry} with a no-op transfer function (that just calls CONTINUE) and where $I(s_{entry}, a)$ and $O(s_{entry}, a)$ both describe the initial abstract state for every address a. The initial worklist is then $W = \{s_{entry}\}$.

More formally, reading the value of an address $a \in A$ at statement $s \in S$ is implemented simply by looking up the value in the incoming state:

READ$(s \in S, a \in A) : V$

1 **return** $I(s, a)$

Similarly, writing a value $v \in V$ to the address $a \in A$ at statement $s \in S$ is implemented by writing to the outgoing state:

WRITE$(v \in V, s \in S, a \in A)$

1 $O(s, a) := v$

Propagation of dataflow from statement $s_{src} \in S$ to $s_{dst} \in S$ is implemented by joining all the values from the outgoing state of s_{src} into the incoming state of s_{dst}. If a value is changed then s_{dst} is added to the worklist. Reachability is implicitly supported since an unreachable statement has every value set to the bottom element $\bot$, whereas we assume that every reachable statement will have at least one value set to non-bottom, and so propagation from a reachable statement to an unreachable statement will always cause the unreachable statement to be added to the worklist:

CONTINUE($s_{src} \in S, s_{dst} \in S$)

1 **for each** $a \in A$
2   **let** $v = O(s_{src}, a)$
3   **let** $v' = I(s_{dst}, a)$
4   **if** $v \not\sqsubseteq v'$
5         $I(s_{dst}, a) := v \sqcup v'$
6         $W := W \cup \{s_{dst}\}$

The main fixpoint computation is implemented by the SOLVE procedure. It maintains a global worklist W of pending statements and iteratively extracts a statement and evaluates its transfer function, which may cause new statements to be added to the worklist. The fixpoint is found when the worklist is empty:

SOLVE($E : A \to V$), where E is the entry state

1 $I(s, a) := O(s, a) := \bot$ for all $s \in S$, $a \in A$
2 $I(s_{entry}, a) := O(s_{entry}, a) := E(a)$ for all $a \in A$
3 $W := \{s_{entry}\}$
4 **while** $W \neq \emptyset$
5   **let** s = DEQUEUE(W)
6     $O(s, a) := I(s, a)$ for all $a \in A$
7   apply the transfer function T_s

15.3 Sparse Analysis

We now show how the basic analysis framework from the preceding section can be changed into our sparse analysis technique. As a first step, we modify the definitions of the incoming and outgoing states to become partial maps, $I : S \times A \hookrightarrow V$ and $O : S \times A \hookrightarrow V$, since we now want to maintain values only for the statements and addresses that are involved in READ or WRITE operations, respectively. Next, we add four new components that are all built incrementally during the fixpoint computation:

$R \subseteq S \times S$ is the set of *reachable edges*

$P : S \times A \hookrightarrow V$ specifies the placement and values of ϕ-*nodes*

$DU \subseteq S \times A \times S$ is the set of *def-use edges*

$F : S \times S \rightarrow \mathcal{P}(A)$ is the map of *frontier addresses*

The R component now explicitly tracks the set of reachable edges in the control-flow graph: if CONTINUE(s_{src}, s_{dst}) has been invoked, then $(s_{src}, s_{dst}) \in R$. As in previous sparse analysis techniques, we use SSA (static single assignment form) to ensure that each use site has a unique associated definition site [Cytron et al., 1991; Wegman and Zadeck, 1991; Hardekopf and Lin, 2009]. When $P(s, a)$ is defined with some value v, the statement s plays the role of a ϕ-node for address a, where v is then the merged value from the incoming dataflow. As effect we obtain SSA for all addresses, not only for local variables. Each triple $(s_1, a, s_2) \in DU$ represents a def-use edge, where s_1 is a definition site or a ϕ-node and s_2 is a use site or a ϕ-node for a.

Since the analysis discovers definition sites and use sites incrementally, the set of def-use edges changes during the analysis. The F map supports this construction of def-use edges whenever frontier edge becomes reachable, as explained later in this section.

The SOLVE procedure is unmodified, except that line 6 is omitted in the sparse analysis version. The remainder of this section explains the modifications of the READ, WRITE, and CONTINUE procedures.

Notation and terminology. We view maps as mutable dictionaries. For example, if $f : A \rightarrow B$ is a map, then $f(x) := v$ denotes the update of f such that subsequently $f(x) = v$. If $f : X \hookrightarrow Y$ is a partial map, then $f_\star$ denotes the domain of f, i.e. the subset of X where f is defined. We assume the reader is familiar with the concepts of *SSA*, *dominance frontiers*, and *dominator trees* from e.g. Cytron et al. [1991]. Specifically, a statement s_2 is in the *dominance frontier* of a statement s_0 if s_0 dominates some predecessor s_1 of s_2 in the control-flow

graph, but s_0 does not dominate s_2. The edge (s_1, s_2) is then called a *frontier edge* of s_0. We say that a statement s is a *ϕ-node* (resp. *definition site* or *use site*) for an address a if $(s, a) \in P_\star$ (resp. $(s, a) \in O_\star$ or $(s, a) \in I_\star$). Only merge points in the control-flow graph can be used as ϕ-nodes. For simplicity, we assume that the statements at merge points are no-ops, such that they cannot be definition sites or use sites (thus, $P_\star$ and $I_\star \cup O_\star$ are disjoint).

The following key invariants are maintained by the READ, WRITE, and CONTINUE operations in the sparse analysis framework:

[*flow*] If $(s_1, a, s_2) \in DU$ for some statements s_1, s_2 and some address a, then

- either $O(s_1, a)$ or $P(s_1, a)$ is defined with some value v,
- either $I(s_2, a)$ or $P(s_2, a)$ is defined with some value v', and
- the value of a at s_1 has been propagated to s_2, i.e. $v \sqsubseteq v'$.

[*def-use*] If (and only if) a statement s_1 is a definition site or ϕ-node for some address a, i.e. $(s_1, a) \in O_\star \cup P_\star$, s_k is a use site or ϕ-node for a, i.e. $(s_k, a) \in I_\star \cup P_\star$, such that s_1 dominates s_k and there is a path $s_1, s_2, \ldots, s_k$ where each step is reachable, i.e. $(s_i, s_{i+1}) \in R$ for all i, and moreover, there is no definition site or ϕ-node for a between s_1 and s_k, i.e. $(s_i, a) \notin I_\star \cup P_\star$ for all $i = 2, 3, \ldots, k-1$, then there exists a def-use edge $(s_1, a, s_k) \in DU$.

[*phi-use*] If s is a ϕ-node for a, i.e. $(s, a) \in P_\star$, then for every reachable incoming control-flow graph edge $(s_1, s) \in R$ there is a def-use edge $(s_2, a, s) \in DU$ where s_2 is the nearest dominator of s_1 and s_2 is a definition site or ϕ-node for a.

[*phi*] If a statement s_0 is a definition site or ϕ-node for some address a, i.e. $(s_0, a) \in O_\star \cup P_\star$, then for every frontier edge (s_1, s_2) of s_0 that is reachable, i.e. $(s_1, s_2) \in R$, the statement s_2 is a ϕ-node for a, i.e. $(s_2, a) \in P_\star$.

[*frontier*] If $a \in F(s_1, s_2)$ for some statements s_1, s_2 and some address a, then (s_1, s_2) is a frontier edge of a dominator s_0 of s_1 that defines a, i.e. $(s_0, a) \in O_\star \cup P_\star$.

Intuitively, the [*flow*] invariant ensures that dataflow has always been propagated along the existing def-use edges; [*def-use*] expresses the main requirements for construction of def-use edges, in particular that def-use edges respect reachability and dominance of definitions; [*phi-use*] ensures that def-use edges to ϕ-nodes also exist for all reachable incoming edges; [*phi*] ensures that ϕ-nodes are created along reachable dominance frontiers; and [*frontier*] expresses that F records which addresses are relevant for contructing def-use edges whenever a frontier edge becomes reachable.

Reading Values

The READ(s, a) operation retrieves the requested value from the incoming state if s is already known to be a use site for a. If a new use is discovered, the appropriate def-use edge must be introduced and the value propagated to s:

```
READ(s ∈ S, a ∈ A) : V
1  if (s, a) ∉ I⋆
2      I(s, a) := ⊥
3      let s1 = FINDDEF(s′, a) where s′ is the immediate dominator of s
4      DU := DU ∪ {(s1, a, s)}
5      PROPAGATE(s1, a, s)
6  return I(s, a)
```

The FINDDEF procedure searches up the dominator tree to find the nearest definition site or ϕ-node for a:

```
FINDDEF(s ∈ S, a ∈ A) : S
1  if (s, a) ∈ O⋆ ∪ P⋆
2      return s
3  else
4      return FINDDEF(s′, a) where s′ is the immediate dominator of s
```

We show in Section 15.3 how FINDDEF can be implemented more efficiently than this pseudo-code suggests. Also note that by initializing $I(s_{entry}, a)$ and $O(s_{entry}, a)$ for every address a according to the initial abstract state when the analysis starts, READ and FINDDEF are well-defined because s_{entry} is the root of the dominator tree.

The PROPAGATE procedure, which is also used by the WRITE operation later, propagates a single value from a definition site or ϕ-node to a use site or ϕ-node, in order to satisfy the [*flow*] invariant. If the destination is a use site and its incoming state changes, then that statement is added to the worklist W. If the destination is a ϕ-node then propagation is invoked recursively for all its outgoing def-use edges:

```
PROPAGATE(s_src ∈ S, a ∈ A, s_dst ∈ S)
 1  if (s_src, a) ∈ O⋆
 2      let v = O(s_src, a)
 3  else // must have (s_src, a) ∈ P⋆
 4      let v = P(s_src, a)
 5  if (s_dst, a) ∈ I⋆
 6      let v_old = I(s_dst, a)
 7      if v ⋢ v_old
 8          I(s_dst, a) := v ⊔ v_old
 9          W := W ∪ {s_dst}
10  else // must have (s_dst, a) ∈ P⋆
11      let v_old = P(s_dst, a)
12      if v ⋢ v_old
13          P(s_dst, a) := v ⊔ v_old
14          for each s where (s_dst, a, s) ∈ DU
15              PROPAGATE(s_dst, a, s)
```

Notice that recursive calls to PROPAGATE can only happen along chains of def-use edges between ϕ-nodes, which are placed only at merge points, so the recursion is bounded by the block nesting depth of the program being analyzed.

Writing Values

The WRITE operation writes the given value to the outgoing state. If a new definition site is discovered, the set of def-use edges must be updated. Moreover, the written value is propagated along the outgoing def-use edges:

```
WRITE(v ∈ V, s ∈ S, a ∈ A)
1  if (s, a) ∉ O⋆
2      UPDATE(s, a)
3      O(s, a) := v
4      FORWARD(s, a)
5  else
6      O(s, a) := v
7  for each s_dst where (s, a, s_dst) ∈ DU
8      PROPAGATE(s, a, s_dst)
```

Whenever a new definition site is discovered in WRITE (line 1), def-use edges that bypass the new definition site and have the same address must be updated (line 2) and ϕ-nodes must be introduced at the iterated dominance frontiers along with associated def-use edges (line 4).

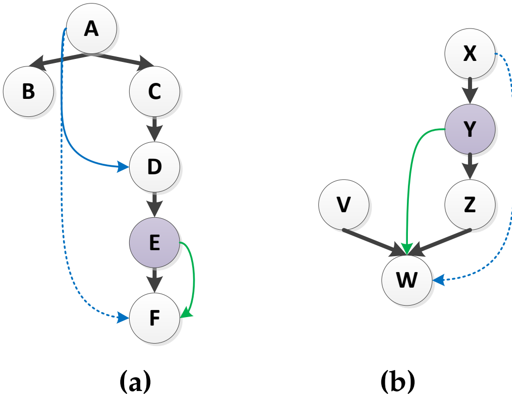


Figure 15.1: Two control-flow graph fragments (with thick edges representing control-flow) that illustrate the UPDATE procedure. **(a)** The statement A defines some address that is used at both statements D and F (corresponding to the def-use edges A $\rightarrow$ D and A $\rightarrow$ F). At some point a definition is discovered at E. The dominating definition at E is A. Its use at D is *not* dominated by E and thus not affected by the new definition at E. The use at F, however, is dominated by E, so the def-use edge A $\rightarrow$ F is replaced by E $\rightarrow$ F. **(b)** The statement X defines some address that is used by the ϕ-node at W (corresponding to the def-use edge X $\rightarrow$ W). If a new definition is discovered at Y then X $\rightarrow$ W must be replaced by Y $\rightarrow$ W since X dominates Y.

UPDATE($s \in S, a \in A$)

1 **let** $s_1 = \text{FINDDEF}(s, a)$
2 **for each** s_2 **where** $(s_1, a, s_2) \in DU$
3   **if** s strictly dominates s_2
4         $DU := (DU \setminus \{(s_1, a, s_2)\}) \cup \{(s, a, s_2)\}$
5 **for each** $(s_3, s_4) \in S \times S$ that is a frontier edge of s
6   **let** $s_0 = \text{FINDDEF}(s_3, a)$
7   **if** s_0 strictly dominates s
8   **if** $(s_0, a, s_4) \in DU$
9             $DU := (DU \setminus \{(s_0, a, s_4)\}) \cup \{(s, a, s_4)\}$

The UPDATE procedure updates the def-use edges that bypass the new definition. The first part (lines 1–4) handles the def-use edges that end at a statement dominated by the new definition, to restore the [*def-use*] invariant as illustrated in Figure 15.1(a); the second part (lines 5–9) handles the def-use edges that end at a dominance frontier node of the new definition, corresponding to [*phi-use*] as illustrated in Figure 15.1(b).

Note that DU does not always grow monotonically, since UPDATE both adds and removes edges. Termination is still ensured: a def-use edge (s_1, a, s_2) is only removed if a new definition site s_d is discovered such that s_d dominates s_2. Definitions are never removed, so the edge (s_1, a, s_2) can never be re-added, and only a finite number of def-use edges can be created. All other components

in the sparse framework are monotonically increasing during the fixpoint computation.

The purpose of the FORWARD procedure is to introduce ϕ-nodes at the iterated dominance frontiers, together with def-use edges for the corresponding reachable frontier edges, and maintain the [*frontier*] invariant:

```
FORWARD(s ∈ S, a ∈ A)
1  for each (s₁, s₂) ∈ S × S that is a frontier edge of s
2      if a ∉ F(s₁, s₂)
3          F(s₁, s₂) := F(s₁, s₂) ∪ {a}
4          if (s₁, s₂) ∈ R
5              MAKEPHI(s₂, a)
```

Although a statement typically has a single frontier edge, it is possible to have multiple, for example, in connection to statements that may throw exceptions. Line 3 in FORWARD adds a to the frontier addresses of the frontier edge (s_1, s_2), which indicates that a has been defined by a statement that dominates s_1. If that edge is already known to be reachable, we may need to add a new ϕ-node at the frontier, which is handled by MAKEPHI as explained next.

```
MAKEPHI(s ∈ S, a ∈ A)
1  if (s, a) ∉ P⋆
2      UPDATE(s, a)
3      P(s, a) := ⊥
4  for each s₁ ∈ S where s₁ is a predecessor of s in the control-flow graph
5      if (s₁, s) ∈ R
6          let s₂ = FINDDEF(s₁, a)
7          DU := DU ∪ {(s₂, a, s)}
8          PROPAGATE(s₂, a, s)
9  FORWARD(s, a)
```

The MAKEPHI procedure is only invoked with s being a dominance frontier node. If s is not already a ϕ-node for a (line 1), we mark it as one (line 3). However, since a ϕ-node has a similar effect as a definition site, UPDATE is called first to update the def-use edges, c.f. line 2 in WRITE. A ϕ-node also has a similar effect as a use site, although generally with multiple incoming def-use edges. For this reason, we make sure a def-use edge exists for every reachable income edge (lines 4–8), c.f. lines 3–5 in READ. Finally, the process is continued recursively for the iterated dominance frontiers (line 9).

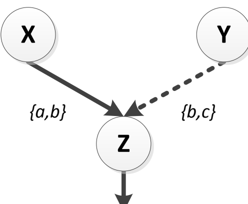


Figure 15.2: A control-flow graph fragment, where the frontier edges X $\rightarrow$ Z and Y $\rightarrow$ Z hold the addresses $\{a, b\}$ and $\{b, c\}$, respectively. The edge X $\rightarrow$ Z is already reachable, and ϕ-nodes for $\{a, b\}$ at Z have already been introduced. Now CONTINUE is invoked for the edge Y $\rightarrow$ Z. CONTINUE ensures that appropriate def-uses edge to Z are introduced for all the addresses a, b, and c, and for all the incoming edges to Z.

Propagating Reachability

As the propagation of dataflow values is performed along def-use edges by PROPAGATE, the primary role of CONTINUE is to propagate reachability:

CONTINUE$(s_{src} \in S, s_{dst} \in S)$
1 **if** $(s_{src}, s_{dst}) \notin R$
2 $\quad R := R \cup \{(s_{src}, s_{dst})\}$
3 $\quad W := W \cup \{s_{dst}\}$
4 $\quad$ **for each** $a \in A$ **where** $a \in F(s_{src}, s_{dst}) \;\vee\; (s_{dst}, a) \in P_\star$
5 $\quad\quad$ MAKEPHI(s_{dst}, a)

If the given control-flow graph edge (s_{src}, s_{dst}) is not already reachable, we mark it as reachable (line 2) and add s_{dst} to the worklist. However, this may trigger calls to MAKEPHI in two situations, corresponding to the two cases in line 4: The condition $a \in F(s_{src}, s_{dst})$ signals that (s_{src}, s_{dst}) is a frontier edge of a statement that defines a, so we must ensure that s_{dst} is a ϕ-node for a and therefore call MAKEPHI. The condition $(s_{dst}, a) \in P_\star$ captures the case where s_{dst} is already a ϕ-node for a, but now there is a new reachable incoming edge, which is handled by lines 4–8 in MAKEPHI as illustrated in Figure 15.2. Notice that we carefully ensure in FORWARD, MAKEPHI, and CONTINUE that no dataflow is propagated across control-flow edges, in particular frontier edges, until they are known to be reachable.

Proposition 1. *The sparse framework has same analysis precision as the basic framework. Specifically, if $O(s, a) = v$ for some $s \in S$, $a \in A$, and $v \in V$ after analyzing a given program with the sparse framework, then we also have $O(s, a) = v$ when analyzing the program with the basic framework.*

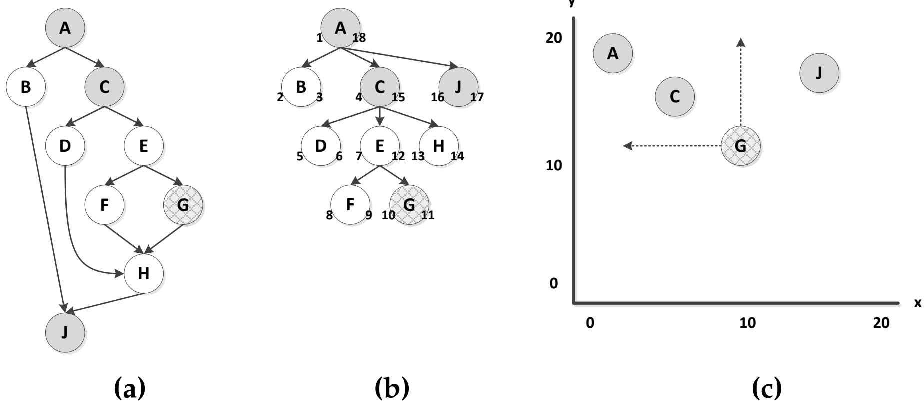


Figure 15.3: (a) A control-flow graph fragment where the statements A, C and J are definition sites for some address. The statement G is a use site of the same address. **(b)** The dominator tree for the control-flow graph with the definition sites and use site marked. **(c)** The 2d points associated with the timestamps of the dominator tree.

A Data Structure for Finding Dominating Definitions

During the fixpoint computation new definition sites and use sites are discovered incrementally by the READ and WRITE operations. A key challenge is how to ensure that the FINDDEF operation is able to quickly find the nearest dominating definition for any statement in the control-flow graph.

The naive version of FINDDEF from Section 15.3 is easy to implement, as also suggested by Chase et al. [1990]. It is, however, impractical because each invocation requires a traversal along a spine of the dominator tree, in the worst case from the given statement all the way to the root. As an example, consider a straight-line program consisting of k statements in sequence. Invoking FINDDEF at the last statement may then require traversal of all k statements to find the nearest dominating definition.

A better approach is to maintain dominator information separately for each address and only for the nodes that are known to be definition sites or ϕ-nodes. The idea is to equip each statement in the control-flow graph with two numbers, x and y, as shown in Figure 15.3. The numbers are obtained through a depth-first traversal of the dominator tree, such that the first number x is the discovery time and the second number y is the exit time. Using these numbers we can determine dominance between nodes: if p dominates q, then p must have been discovered before q, i.e. $x_p < x_q$, and all children of q must have been visited before exiting p, i.e. $y_q < y_p$. As an example, in Figure 15.3, statement E dominates F since $x_{\mathsf{E}} < x_{\mathsf{F}}$ and $y_{\mathsf{F}} < y_{\mathsf{E}}$.

A key observation is that to find the *nearest* dominating definition of a node q we need to find a node p where $x_p < x_q$ and $y_q < y_p$. Of all nodes that

satisfy these conditions we wish to find the one with maximum x value. For example, in Figure 15.3 the nearest dominating definition at G is C, which satisfies these properties.

One approach is to store the definitions in a resizable array and perform a linear scan to find the nearest dominating definition, as in Staiger-Stöhr [2013]. Unfortunately, this requires $\mathcal{O}(d)$ time, where d is the number of definitions. Another approach is to define an ordering such that finding all dominating definitions takes $\mathcal{O}(\log d)$ time and then scan through these to find the nearest dominating definition, as in Tok et al. [2006]. However, the scanning may still require $\mathcal{O}(d)$ time.

Our solution works as follows. If we interpret the number pair (x_q, y_q) of a node q as a point in a two dimensional space, then finding the nearest dominating definition p is equivalent to finding the point (x_p, y_p) in the rectangle $[0, x_q] \times [y_q, \infty]$ with the maximum x_p. This is a well-known problem in the computational geometry literature; one data structure solving this problem is the quadtree de Berg et al. [1997]. Finding the nearest dominating definition then takes $\mathcal{O}(\sqrt{n})$ time where n is the number of control-flow graph nodes, and a new node can be inserted in $\mathcal{O}(\log n)$ time. Quadtrees are simple to implement and have a low constant-factor overhead. Our experimental comparison (see Section 15.4) confirms that quadtrees lead to a faster implementation than the naive version of FINDDEF and Staiger-Stöhr's approach. In principle one could combine the techniques and get $\mathcal{O}(\min(\sqrt{n}, d))$, but in practice using the quadtrees alone seems to work well.

To summarize, we pre-compute the two numbers for every statement in the control-flow graph and then maintain a quadtree y_a for each $a \in A$ containing every statement $s \in S$ where $(s, a) \in O_\star \cup P_\star$. The FINDDEF procedure from Section 15.3 is then replaced by a search in y_a.

15.4 Implementation and Evaluation

Implementation. We have implemented a dataflow analysis for JavaScript, configurable for both the traditional dense propagation (Section 15.2) and the sparse analysis with on-the-fly SSA construction (Section 15.3). The dataflow lattices and transfer functions are designed in the style of the TAJS analysis by Jensen et al. [2009], structured such that all transfer functions are expressed using the READ, WRITE and CONTINUE operations. For the quadtrees, we use a variant called *compressed quadtrees* [Har-Peled, 2011]. Interprocedural dataflow is handled by straightforward generalizations of the algorithms from the preceding sections. Call graphs are built on-the-fly, similar to TAJS. For the interprocedural sparse analysis, ϕ-nodes are made at function entries and at no-op statements that are placed after call sites where dataflow may

merge from different functions. Searching for dominating definitions may then span multiple functions backward via the call edges, and similarly, definitions inside a function are propagated forward along dominance frontiers of the call sites. The full implementation is approximately 20,000 lines of Scala code, whereof the core that corresponds to Section 15.2 and 15.3 constitutes less than 1,000 lines.

Benchmarks. Our experiments are based on the collection of JavaScript programs shown in Table 15.1. (Our current implementation does not contains models of the browser API and HTML DOM, so we settle for stand-alone JavaScript programs.) The collection contains programs from the Mozilla SunSpider and Google Octane benchmark suites, plus a few additional programs found on the web. All experiments are performed on an Intel Core 2 Duo 2.5 GHz PC. The analysis implementation and all benchmarks are available online.

Research Questions We consider the following three research questions:

Q1: Is our sparse analysis technique more efficient than the basic analysis framework? The literature shows that sparse analysis is usually highly effective, but since none of the existing techniques are applicable to our setting, which involves both pointers and reachability, we cannot know *a priori* whether our sparse analysis has similar advantages.

Q2: How does the performance of our sparse analysis algorithm compare to staged analysis techniques? As argued in Section 15.1, performing sparse analysis on the basis of imprecise reachability information would affect not only the degree of sparseness but also the precision of the main analysis, and we want our sparse analysis to be as precise as with the basic framework. On the other hand, our algorithm could potentially be simplified without affecting analysis precision by using a pre-analysis to compute definition sites and use sites for SSA construction, instead of performing it all on-the-fly. For this reason, it is interesting to measure the overhead of computing that information in our on-the-fly sparse analysis framework.

Q3: Are quadtrees a suitable choice in practice, compared to other techniques for maintaining information about reaching definitions? In Section 15.3 we argued that quadtrees have a good theoretical complexity, however, this needs to be supported empirically.

Program			*Total time*		*SSA overhead*			
Name	*Lines*	*Nodes*	*Basic*	*Sparse*	%	*Quadtree*	*Naive*	*Array*
`deltablue.js`	885	3303	*timeout*	35780	43%	15223	23347	21927
`richards.js`	541	1655	*timeout*	703	31%	216	391	405
`splay.js`	398	1058	79844	705	28%	198	268	273
`3d-cube.js`	343	2875	*timeout*	1974	24%	482	991	899
`3d-raytrace.js`	443	3000	*timeout*	2723	30%	812	1686	2954
`access-nbody.js`	170	847	63864	488	39%	189	262	218
`crypto-aes.js`	426	2581	*timeout*	713	25%	177	451	486
`crypto-md5.js`	295	1508	30561	2091	3%	53	111	117
`garbochess.js`	2812	16146	*timeout*	5764	26%	1501	3138	4362
`simplex.js`	450	2121	*timeout*	465	28%	128	282	235
`jpg.js`	889	5146	*timeout*	2621	21%	538	1035	992
`javap.js`	1430	5561	*timeout*	1693	33%	559	1479	3037

Table 15.1: Experimental results. *Lines* shows the number of source code lines and *Nodes* shows the number of control-flow graph statements for each program, to indicate their sizes. *Basic* and *Sparse* are the total analysis time for the basic and sparse frameworks, respectively. *SSA overhead* shows the time spent on SSA construction during the sparse analysis, using three different implementations for maintaining dominating definitions. All times are shown in milliseconds, with *timeout* representing a timeout of 90 seconds.

To answer Q1 we instantiate the analysis with both configurations (using quadtrees for the sparse analysis). The columns *Basic* and *Sparse* in Table 15.1 show the corresponding analysis times. The numbers show that our sparse analysis is in most cases more than an order of magnitude faster than the basic framework. As result, we have demonstrated that it is possible to perform efficient sparse analysis in a setting that involves pointers and reachability.

We address Q2 by assuming an *ideal* pre-analysis that computes the definition sites and use sites and from this constructs the SSA form, with the full precision of the on-the-fly sparse analysis. Designing a realistic pre-analysis involves a trade-off: it has to be fast (at least, faster than the original dense analysis), and it has to be reasonably precise (since imprecision can lead to less sparseness in the main analysis). With such a pre-analysis, we can perform sparse analysis and still account for reachability – reminiscent of the sparse conditional constant analysis by Wegman and Zadeck [1991]. The *SSA overhead* columns (% and *Quadtree*) in Table 15.1 show how much time is spent by our sparse analysis inside the operations FINDDEF, UPDATE, FORWARD, and MAKEPHI (excluding PROPAGATE), relative to the entire sparse analysis

and in milliseconds, when using the quadtree implementation of FINDDEF. This constitutes work that in principle could be omitted if using a pre-analysis. We observe that between 3% and 43% of the analysis time is spent in these parts. In other words, the best imaginable pre-analysis will only be able to achieve a speedup of less than 1.7x (for `deltablue.js`) and on average less than 1.4x. Moreover, by using our on-the-fly approach, the analysis developer is relieved of the burden of designing and implementing a fast and precise pre-analysis.

Regarding Q3, the columns *Quadtree*, *Naive*, and *Array* show the time for SSA construction with different implementations of the data structure used for finding dominating definitions. The *Quadtree* column corresponds to our quadtree-based implementation described in Section 15.3, *Naive* corresponds to FINDDEF from Section 15.3, and *Array* follows the approach of Staiger-Stöhr [2013], as discussed in Section 15.3. In all cases the quadtree implementation is the fastest and typically outperforms the alternatives by a factor of 1.4x to 5.4x.

15.5 Related Work

The basic ideas in sparse analysis originate from Reif and Lewis [1977] who suggested the use of *global value graphs*, for example for efficient constant propagation analysis. The concept of SSA form is attributed to Rosen et al. [1988]. Cytron et al. [1991] introduced the concept of dominance frontiers as an effective mechanism for placing ϕ-nodes. As mentioned in the introduction, the *sparse conditional constant* analysis by Wegman and Zadeck [1991] builds on top of this work and takes reachability into account during the analysis by tracking which def-use edges represent executable flow. The notion of *dependence flow graphs* by Johnson and Pingali [1993] is a variant of SSA that incorporates branch conditions and thereby supports subsequent dataflow analysis with reachability. Common to this line of work is that heap objects and pointers are not supported.

For programming languages with pointers, most work on sparse analysis has focused on pointer analysis, not dataflow analysis in general. The *semi-sparse* pointer analysis by Hardekopf and Lin [2009] uses SSA and sparse analysis for top-level variables that are not accessed via pointers, whereas address-taken variables and heap allocated data are treated using standard flow-sensitive analysis without sparseness.

Other techniques handle pointers typically by staging the analysis using a pre-analysis to approximate possible definition sites and use sites [Chow et al., 1996; Hardekopf and Lin, 2011; Oh et al., 2012]. However, as discussed previously, that approach cannot support reachability without sacrificing analysis

precision or sparseness. By computing definition sites and use sites on-the-fly, we avoid that problem.

The analysis by Chase et al. [1990] handles pointers and performs sparse analysis on the basis of ϕ-nodes that are computed on-the-fly, however, it does not account for reachability. The analysis framework by Tok et al. [2006] is based on similar ideas. The algorithms used in those analyses for finding dominating definitions are discussed in Section 15.3. A related analysis framework has been presented by Staiger-Stöhr [2013].

Numerous other program analysis techniques have been designed to prevent various kinds of redundancy in the dataflow propagation. Of particular relevance is the *lazy propagation* technique by Jensen et al. [2010] that restricts dataflow at call sites that is not needed by the function being called. When use sites are incrementally discovered, the relevant values are recovered by a backward traversal of the call graph, which is reminiscent of the search for nearest dominating definitions in our sparse analysis. We conjecture that our sparse analysis may be more efficient than lazy propagation; however, lazy propagation is known to work smoothly together with recency abstraction [Balakrishnan and Reps, 2006; Jensen et al., 2009], which is a useful technique for boosting analysis precision, and it is an open problem whether sparse analysis and recency abstraction can also be combined effectively.

In summary, the present work can be understood as a generalization and combination of ideas from on-the-fly SSA construction [Chase et al., 1990; Tok et al., 2006; Staiger-Stöhr, 2013] while taking reachability into account [Wegman and Zadeck, 1991; Johnson and Pingali, 1993]. Furthermore, we propose a more efficient data structure, based on insights from computational geometry [de Berg et al., 1997], for managing dominating definitions, compared to the existing techniques [Tok et al., 2006; Staiger-Stöhr, 2013].

15.6 Conclusion

We conclude that it is possible to perform efficient sparse dataflow analysis in a setting that requires reasoning about pointers and reachability. Our experimental evaluation shows not only that the sparse analysis is significantly faster than the dense counterpart, but also that the overhead of on-the-fly SSA construction is small, which makes the approach a promising alternative to staged analyses. Moreover, we have demonstrated that quadtrees are suitable for maintaining information about dominating definitions.

Acknowledgements. The authors thank Casper Kejlberg-Rasmussen, Jesper Sindal Nielsen, and Ondřej Lhoták for inspiring discussions about data structures and dataflow analysis.

16

A Framework for Declarative Specification of Static Analyses

This is a draft of a manuscript by Magnus Madsen and Ondřej Lhoták.

Abstract

We present FLIX, a language for declaratively specifying and solving least fixed point problems, particularly static program analyses. FLIX is more expressive than Datalog because it allows arbitrary user-defined lattices, and because general-purpose functions can be called from declarative rules. This enables the rules to trigger computations that are too complicated to be expressed purely declaratively. It also enables the rules to directly access data structures constructed in a general purpose language. A verifier ensures that despite the added generality, correctness properties such as computation of the least solution and termination can still be guaranteed. The utility and generality of the language is evaluated on several well-known static program analyses.

16.1 Introduction

Least fixed point problems are ubiquitous in mathematics and computer science, significantly in programming languages, and particularly in program analysis. Given a function F on a semi-lattice, the goal is to find the least x for which $F(x) = x$. At the lowest and most general level, a program is a function F that instructs a machine how to change its overall state at each computation step. A static analysis computes an abstract state $\hat{x}$ that overapproximates all possible concrete states that a program can reach. Every sound approximation must satisfy $\hat{F}(\hat{x}) \sqsubseteq \hat{x}$, where $\hat{F}$ is an abstraction of the concrete transformation function F, since if a state in $\hat{x}$ can be reached by a computation, then so can a state in $\hat{F}(\hat{x})$. The least $\hat{x}$ satisfying this inequality also satisfies the fixed-point property $\hat{F}(\hat{x}) = \hat{x}$. The most common technique for solving such least fixed point problems is by starting from the least element $\perp$ of the semi-lattice and iteratively applying $\hat{F}$ (or simpler functions derived from $\hat{F}$) until the fixed point is reached [Cousot and Cousot, 1977; Kam and Ullman, 1977].

Because real programming languages are complex and real program analyzers must consider their complicated details, analyzers are often implemented in expressive general-purpose programming languages such as C++ or Java. The many mutual dependencies are typically implemented using a combination of recursion and complicated arrangements of worklists that are needed to break the otherwise infinite recursive loops imposed by the fixed point problem. The decision of how to structure the worklists is global, so these large program analyzers become difficult to restructure and modify. It also becomes difficult to understand precisely the analysis problem that the implementation is actually solving, and to assure oneself that the implementation is correct. Moreover, the complexity of the dependencies between inter-related sub-analyses often leads analysis builders to sacrifice analysis precision. For example, some interprocedural analysis frameworks use a call graph preconstructed with conservative assumptions to compute dataflow information that would enable a more precise call graph. As another example, some semi-sparse analyses construct an SSA form making worst-case assumptions, and later compute pointer information that would enable a more precise SSA form.

To overcome these difficulties, some analysis designers have turned to Datalog [Ceri et al., 1989; Bravenboer and Smaragdakis, 2009; Whaley and Lam, 2004]. A Datalog program is a set of declarative constraints over a set of relational variables, and its solution is the set of smallest relations that satisfy those constraints. Thus, the result of a Datalog program is the least fixed point of a function defined by the constraints on the lattice of relations. An important benefit of a Datalog program is its modularity: because the constraints are declarative, individual sub-analyses can be easily composed

by taking the union of their constraints, and the Datalog solver takes care of the mutual dependencies automatically. Thus, it is easy to understand an analysis by understanding its components individually. Correctness of each component implies correctness of the overall analysis.

However, Datalog has important limitations that restrict its applicability to program analysis:

- **Lattices:** Datalog programs are limited to the lattice of relations, powersets of tuples of atomic values, but common analyses operate over a wide variety of other lattices. Some simple lattices can be embedded in powersets but at a very high computational cost, and more interesting lattices cannot be encoded at all. For example, we can embed the constant propagation lattice over a finite domain in the following way: $\perp$ is represented by the empty set, each constant is represented by a singleton set, and $\top$ is represented by any set that contains a specially designated $\top$ element. We then add a constraint that adds the $\top$ element to every set of two or more elements. However, this $\top$ constraint cannot prevent the Datalog program from processing the original non-singleton, non-$\top$ sets. We get the worst of both worlds: the precision is the same as with the constant propagation lattice, but the computational cost is the same as with the much more expensive arbitrary-sets-of-constants lattice. Moreover, when the domain of the constants is infinite, such as the integers, the lattice cannot be simulated at all, because the Datalog program would never terminate.

- **Functions:** Datalog programs are limited to the relational constraints. Even if we could represent a constant propagation lattice of integers, we would not be able to define basic computations on its elements such as abstract addition. When writing a full program analyzer, one needs many algorithms that are cumbersome or impossible to express in the limited language of constraints. For example, the code to read a program from disk, scan it, parse it, and build an intermediate representation is much more convenient to write in a general-purpose language than in Datalog. Even parts of the actual static analysis that model complicated aspects of the semantics of a programming language are more conveniently expressed in general-purpose code than in constraints. Thus, Datalog-based analysis systems have large preprocessors written in general-purpose languages that pre-analyze the programs and compute relations suitable as input to a Datalog program: Doop Bravenboer and Smaragdakis [2009] uses Soot Vallée-Rai et al. [1999a], and bddbddb Whaley and Lam [2004] uses joeq [Whaley, 2003] and other

preprocessors. Functions that are infeasible to compute in Datalog must be precomputed and tabulated in the preprocessor.

- **Interoperability:** A related limitation is that Datalog programs are self-contained and do not interact deeply with systems written in general-purpose languages. To use Datalog for a fixed-point computation, one must first serialize input data structures and functions into relations, then execute the Datalog program, and then deserialize the resulting relations into general-purpose data structures. The overhead is not only in implementation complexity: the computational cost of inputting and outputting relations can even exceed the cost of the Datalog computation itself.

To overcome these limitations, we propose FLIX, a declarative language for fixed-point problems. FLIX allows arbitrary lattices that may be needed for a particular analysis problem. FLIX also interacts deeply with a general-purpose language by allowing declarative rules to call general-purpose functions. These functions can be used to perform complex computations that would not be feasible to express declaratively, and to access data structures from the general-purpose language. Thus, one can easily instantiate FLIX programs whenever a fixed point computation is needed in a general-purpose program.

An important benefit of Datalog that we wish to preserve in FLIX is an assurance of correctness. A Datalog program is guaranteed to compute the least solution of the constraints, and to terminate (in polynomial time). Such properties are difficult to automatically ensure for a general-purpose program. However, FLIX takes advantage of the mostly-declarative structure of a fixed-point problem to break these high-level properties into a set of simple proof obligations. As long as the general-purpose functions are monotone and the user-defined semi-lattices actually satisfy the semi-lattice properties, FLIX is guaranteed to compute the least solution of the rules. Similarly, as long as the semi-lattice is of finite height and the general-purpose functions are terminating, FLIX is guaranteed to terminate. For many analyses, FLIX can prove most of these proof obligations automatically using either an internal prover or an external SMT solver. This verification process can also produce counterexamples that are very useful for debugging. The remaining proof obligations are given to the programmer to verify. Even if only a subset of them are verified automatically, this reduces the proof effort required of the programmer.

In summary, the paper makes the following contributions:

- We design FLIX, a language to declaratively specify least fixed point problems. Like Datalog, FLIX is declarative in spirit, but more expres-

sive because it allows user-defined lattices and calls to general-purpose functions.

- We show how the overall correctness of a FLIX program can be decomposed into simple proof obligations, many of which can be proven automatically.
- We validate the expressive power of the language by implementing several well-known program analyses.

Language Design

A FLIX program is a set of declarative rules that specify constraints on a set of lattice-valued variables. The program is intended as input to two independent components. The fixed point solver takes the program and an initial set of input facts and computes the least solution of the rules. The verifier also takes the program and tries to prove basic properties that together ensure its correctness and termination. The rest of this section presents the language design, Section 16.2 discusses the verifier, and Section 16.3 discusses the fixed point solver.

Program Structure. The syntax of FLIX is shown in Figure 16.1. A program consists of lattice declarations (Section 16.1), global variable declarations, function declarations (Section 16.1), and rules (Section 16.1).

The basic constants in FLIX are integers, strings, boolean constants, and a unit constant.

Values include the constants, sets of values, tuples of values, maps from values to values, and tagged values, where the tag can discriminate the cases of a tagged union. A map associates values to a finite set of keys. When the range type of a map is a lattice, the map returns a default value of $\bot$ for every key that does not have an explicit value in the map.

The types correspond directly to these forms of values. We keep separate the type of the map lattice $[\tau_1 \mapsto \tau_2]$ from function types $\tau_1 \to \tau_2$. Function types are used only in the function language to be discussed in Section 16.1, and are not allowed anywhere in the lattice variables or rules.

The tagged union type is used to define custom lattices. A value of type $\mathrm{Tag}_1\ \tau_1, \ldots, \mathrm{Tag}_\mathrm{n}\ \tau_n$ must be a tagged value $\mathrm{Tag}_\mathrm{i}\ v$, where v is of the type τ_i corresponding to Tag_i. A union with a single tag creates new, distinct types from existing types: for example, the types $\mathrm{Varname}$ `String` and $\mathrm{ProcName}$ `String` define two distinct types of string values that cannot be inadvertently mixed.

Each FLIX program defines a set of global variables and specifies a type for each of them. The goal of evaluating a FLIX program is to find a valuation of the global variables that respects their types and satisfies all of the rules.

$\mathcal{P}$	$::=$		*Program*
		$\overline{\langle \tau, F_{\sqsubseteq}, F_{\sqcup}, F_{\bot} \rangle}$	*Lattice declarations*
		$\overline{A : \tau}$	*Variable declarations*
		$\overline{F := t : \tau}$	*Function declarations*
		$\overline{C}$	*Rules*
k	$::=$	*Bool* | *Int* | *Str*	*Constant*
v	$::=$	k	*Value*
	$\mid$	$\{\overline{v}\}$	*Set*
	$\mid$	$\langle\overline{v}\rangle$	*Tuple*
	$\mid$	$[\overline{v_k \mapsto v_v}]$	*Map*
	$\mid$	Tag v	*Tagged value*
Tag		is a tag name	
τ	$::=$	`Bool` | `Int` | `String` | `Unit`	*Type*
	$\mid$	$\{\tau\}$	*Set type*
	$\mid$	$\langle\overline{\tau}\rangle$	*Tuple type*
	$\mid$	$[\tau_1 \mapsto \tau_2]$	*Map type*
	$\mid$	$\tau_1 \to \tau_2$	*Function type*
	$\mid$	$\overline{\text{Tag}\ \tau}$	*Tagged union type*

Figure 16.1: Program Syntax.

C	$::=$	$P \Longleftarrow \overline{P}$	*Rule*
P	$::=$	$p \sqsubseteq x \mid l = \mathrm{F}(p)$	*Predicate*
p	$::=$	$k \mid l \mid \{\overline{p}\} \mid \langle\overline{p}\rangle \mid [p_k \mapsto p_v] \mid \text{Tag}\ p$	*Pattern*
X	$::=$	$A \mid l$	*Variable*
F		is a function name	
l		is a variable local to a rule or function	
A		is a global lattice variable	

Figure 16.2: Rule Syntax.

Rules

Rules are the main component of a FLIX program, since they specify the constraints on the desired solution. Their syntax is defined in Figure 16.2. A rule consists of a single *head* predicate and a sequence of *body* predicates. Informally, the rule states that the predicates in the body imply the predicate

in the head. The head predicate of every rule is required to be of the form $p \sqsubseteq x$, where A must be a global variable.

Variable Binding. In general, a predicate can be viewed in two ways: as a function that can be evaluated to return a boolean, or as a rule on the values of variables occurring within it. In this section, we formalize these two roles of a predicate in FLIX.

Definition 1. *The* variables *of a pattern p are all variables that occur within it. The pattern p* constrains *all variables that occur within p outside of any tagged sub-pattern.*

For example, the pattern $\langle l_1, l_2, \text{Tag}\ \langle l_2, l_3 \rangle\rangle$ *contains* the variables l_1, l_2, and l_3, and *constrains* l_1 and l_2.

Intuitively, to evaluate a pattern, all of its *variables* must have values. A pattern assigns values to all of its *constrained* variables. We will discuss later why variables occurring only within a tagged sub-pattern are not constrained.

We now extend these definitions to predicates:

Definition 2. *The* variables *of a predicate are all variables that occur within it. A predicate of the form $p \sqsubseteq x$* constrains *the variables that p constrains. A predicate of the form $l = F\,(p)$* constrains *only variable l.*

For every rule $P \impliedby P_1 \wedge \ldots \wedge P_n$ and for all i, every local variable l of P_i must be constrained by either P_i itself or by some earlier predicate P_j with $j \sqsubseteq i$. In addition, every local variable l of the head predicate P must be constrained by at least one of the body predicates. This ensures that every variable has a value before it is used. This is the first part of the *binding requirement*, formalized in Figure 16.3. Given an unordered set of predicates in the body of a rule, FLIX can automatically find an ordering that satisfies this requirement, if one exists. Therefore, the author of a FLIX program may list the predicates in any order. If no such ordering exists, then FLIX rejects the program with an error message.

$$\begin{aligned} \forall j.\text{variables}(P_j) &\subseteq \cup_{i \sqsubseteq j}\text{constrains}(P_i) \\ \text{variables}(P) &\subseteq \cup_i \text{constrains}(P_i) \\ \forall i \sqsubseteq j.\text{freezes}(P_i) \cap \text{constrains}(P_j) &= \emptyset \end{aligned}$$

Figure 16.3: The binding requirement.

Every predicate that constrains a variable affects the value assigned to the variable. The value of a variable may constrain other variables in other

predicates. FLIX must ensure that after a variable has been used to constrain others, it is not constrained further itself. For example, consider a rule with the body $l \sqsubseteq A_{12} \wedge l' \sqsubseteq l \wedge l \sqsubseteq A_{23}$, where A_{12} has value $\{1, 2\}$ and A_{23} has value $\{2, 3\}$. For this body, l would be constrained to $\{1, 2\}$ after the first predicate, l' would also be constrained to $\{1, 2\}$ by the second predicate, but the third predicate would further constrain l to $\{2\}$. The resulting valuation would not satisfy the second predicate, since $\{1, 2\} \not\sqsubseteq \{2\}$, so this kind of rule must be disallowed. The problem is that l was used to constrain l' in the second predicate, and later re-constrained in the third predicate. This is disallowed by the second part of the *binding requirement* in Figure 16.3. Any variable that is used to constrain other variables in a predicate P_i becomes *frozen* and cannot be further constrained in a later predicate P_j. It remains to formally define what it means for a predicate to *freeze* a variable:

Definition 3. *A predicate of the form* $p \sqsubseteq X$ freezes X *if* X *is a local variable. A predicate of the form* $l = \mathrm{F}\,(p)$ freezes *all of the variables occurring in the argument pattern* p.

Type Checking. The type checking rules for patterns in FLIX are straightforward because the syntax of types follows directly the syntax of patterns, and because the type system does not include subtyping. We omit the rules themselves for lack of space. The rules for predicates are natural: in $p \sqsubseteq X$, X and p are required to have the same type, and in $l = \mathrm{F}\,(p)$, the type of F determines the required type of p and the type of l.

Type checking and inference are done top-down on patterns. Recall that the types of global variables and functions are explicitly specified in a FLIX program, and the type of a local variable is inferred in the first predicate that constrains it. When type-checking a rule of the form $p \sqsubseteq X$, the required type of p is known because it is required to be the same as the type of X. Similarly, when type-checking a rule of the form $l = \mathrm{F}\,(p)$, the type of p is required to be the declared parameter type of F. The pattern p is type-checked top-down by recursively opening the structure of both the required type and of the pattern simultaneously. As each predicate is type-checked, all local variables that it constrains are assigned types to be used in type-checking later predicates.

Rule Evaluation

To solve a FLIX program $\mathcal{P}$ with rules $C_1, \ldots, C_n$, we need to find the least valuation σ_0 of the global variables that satisfies all of the rules. To do this, we define an evaluation function $[\![C_i]\!]$ that transforms a valuation according to a rule C_i. A valuation σ_0 is a fixed point of $[\![C_i]\!]$ if and only if σ_0 satisfies C_i. The goal is then to find the least fixed point of all of the functions $[\![C_i]\!]$. This is can

be done by chaotic iteration [Cousot and Cousot, 1992]. We begin by setting σ_0 to the bottom valuation (which assigns bottom to every global variable), and successively replace it with $[\![C_i]\!](\sigma_0)$ until we reach a σ_0 that is a fixed point for all of the $[\![C_i]\!]$.

In general, evaluating a rule $p \sqsubseteq A \impliedby P_1 \wedge \ldots \wedge P_n$ consists of finding the set of valuations that satisfy the body $P_1 \wedge \ldots \wedge P_n$, applying each valuation σ to p, and updating A so that the head predicate $\sigma p \sqsubseteq A$ is satisfied. In the following subsections, we first discuss valuations, then the overall procedure for computing the set of all satisfying valuations of the body, and finally the details of extending a valuation in response to a specific body predicate.

Valuations. Generally, a valuation is a map from variables to values. A valuation can be partial in that it is not defined for all of variables that occur in a rule. There may be exponentially or infinitely many valuations that satisfy a given rule body. For example, if the body contains the constraint $l \sqsubseteq A$, where the type of A is the constant propagation lattice and the current value of A is $\top$, then the satisfying valuations are those that assign to l the value $\top$, $\bot$, or any integer.

For efficiency and computability, we define a valuation in FLIX to assign to each variable either an exact value, written $v!$, or a downward-closed value, written $v{\downarrow}$. If $\sigma(l) = v{\downarrow}$, then the FLIX valuation σ implicitly represents the set of all valuations that assign to l any value v' such that $v' \sqsubseteq v$.

It turns out that it is possible to determine statically, based on the syntactic structure of a rule, whether a given variable will be bound to a downward-closed or exact value (or not bound at all) at each point in the evaluation of a rule. Therefore, our implementation of valuations does not actually dynamically keep track of whether the value v bound to a variable means $v{\downarrow}$ or $v!$, but obtains this information statically.

Rule Evaluation. We now define the semantics of evaluating a rule. The goal of evaluating a rule $p \sqsubseteq A \impliedby P_1 \wedge \ldots \wedge P_n$ given a valuation σ_0 of the global variables is to compute the least value v such that $\sigma_0(A) \sqsubseteq v$, and such that $\sigma_n p \sqsubseteq v$ for every valuation σ_n that satisfies body of the rule. After v has been computed, the current valuation of global variables is updated by setting A to v. If the body of the rule does not refer to A, the rule is guaranteed to be satisfied under the new valuation. We define the meaning of a rule as:

Definition 4.

$$[\![p \sqsubseteq A \impliedby P_1 \wedge \ldots \wedge P_n]\!](\sigma_0) = \sigma_0[A \mapsto v']$$
$$\textit{where } v' = \sigma_0(A) \sqcup \bigsqcup_{\sigma_n \in [\![P_1 \wedge \ldots \wedge P_n(\sigma_0)]\!]} \sigma_n p$$

In the above definition, $[\![P_1 \wedge \ldots \wedge P_n]\!](\sigma_0)$ is a function that gives the set of all valuations σ_n that:

1. agree with σ_0 on the global variables,
2. are defined for all variables in $P_1, \ldots, P_n$, and
3. satisfy the body $P_1 \wedge \ldots \wedge P_n$.

We call this the *valuation requirement*. Given an initial σ_0, the value of the function is computed by considering each predicate one at a time, and successively computing $S_0 = [\![]\!](\sigma_0)$, $S_1 = [\![P_1]\!](\sigma_0)$, $S_2 = [\![P_1 \wedge P_2]\!](\sigma_0)$, and so on. By construction, each S_i is the set of all valuations that satisfy the valuation requirement for $P_1 \wedge \ldots \wedge P_i$. At each step, we iterate over all of the valuations $\sigma_{i-1} \in S_{i-1}$, *extend* each to the set of all valuations that satisfy the valuation requirement for $P_1 \wedge \ldots \wedge P_i$, and compute the union of the resulting sets of valuations as the set S_i. The details of extending a valuation σ_{i-1} to respect an additional predicate P_i are presented in the next section.

For the correctness of the fixed-point iteration algorithm, the evaluation function for a rule C must be monotone:

Proposition 2. *For every rule C, $[\![C]\!]$ is a monotone function: $\sigma_0 \sqsubseteq \sigma'_0 \implies [\![C]\!](\sigma_0) \sqsubseteq [\![C]\!]\sigma'_0$.*

Proof sketch. We prove the proposition by proving it individually for each term in the join in the definition of $[\![C]\!]$. It is immediate from the premise that $\sigma_0(A) \sqsubseteq \sigma'_0(A)$. We define a partial order on sets of valuations as follows: $S \prec S'$ if $\forall \sigma \in S. \exists \sigma' \in S'. \sigma \sqsubseteq \sigma'$. Let $S_n = [\![P_1 \wedge \ldots \wedge P_n]\!](\sigma_0)$ and $S'_n = [\![P_1 \wedge \ldots \wedge P_n]\!](\sigma'_0)$. We prove that $S_n \prec S'_n$ by induction on n. The base case when $n = 0$ is immediate from the definition of S_0. For the inductive case, assume that $S_{i-1} \prec S'_{i-1}$, so for every $\sigma_{i-1} \in S_{i-1}$, there is a $\sigma'_{i-1} \in S'_{i-1}$ such that $\sigma_{i-1} \sqsubseteq \sigma'_{i-1}$. The lemma that we will prove at the end of the next section ensures that $S_i \prec S'_i$. □

Valuation Extension. Given a valuation σ and a predicate P_i, the process of *extension* computes the set of all valuations that satisfy P_i, and still satisfy all predicates that were satisfied by σ.

The extension process is defined in Figure 16.4 in a Scala-like for-comprehension syntax. The rows define extension for the syntactic forms of a pattern p_0. For a given value v, the middle column defines the extension of σ to the set of all valuations σ' that satisfy $\sigma' p_0 \sqsubseteq v$, and the right-most column defines the extension satisfying $\sigma' p_0 = v$. When the predicate P_i is of the form $p \sqsubseteq X$, we compute the extension according to the middle column with the requirement $p \sqsubseteq \sigma(X)$. When the predicate P_i is of the form $l = \mathrm{F}\,(p)$, we evaluate $\mathrm{F}\,(\sigma p)$ to

<table>
<tr><td>p_0</td><td>σ.extend($p_0 \sqsubseteq v$)</td><td>σ.extend($p_0 = v$)</td></tr>
<tr><td>k</td><td>if($k \sqsubseteq v$) Set(σ) else Set()</td><td>if(k == v) Set(σ) else Set()</td></tr>
<tr><td>l</td><td>if(!σ.isDefinedAt(l)) Set(σ + (l -> $v{\downarrow}$)
else if(σ(l) == $v'{\downarrow}$) Set(σ + (l -> $(v \sqcap v'){\downarrow}$))
else if(σ(l) == $v'!$ && $v' \sqsubseteq v$) Set(σ + (l -> $v'!$)
else Set()</td><td>if(!σ.isDefinedAt(l)
|| (σ(l) == $v'{\downarrow}$ && $v \sqsubseteq v'$)
|| (σ(l) == $v'!$ && v == v'))
Set(σ + (l -> $v!$))
else Set()</td></tr>
<tr><td>Tag p</td><td>Set(σ)</td><td>Set(σ)</td></tr>
<tr><td>$\{p\}$</td><td>for{elem <- v
σ' <- σ.extend($p =$ elem)
} yield σ'</td><td>if(v.size != 1) Set() else
for{elem <- v
σ' <- σ.extend($p =$ elem)
} yield σ'</td></tr>
<tr><td>$\langle p_1, p_2 \rangle$</td><td>for{σ' <- σ.extend($p_1 \sqsubseteq v._1$)
σ'' <- σ'.extend($p_2 \sqsubseteq v._2$)
} yield σ''</td><td>for{σ' <- σ.extend($p_1 = v._1$)
σ'' <- σ'.extend($p_2 = v._2$)
} yield σ''</td></tr>
<tr><td>$[p_k \mapsto p_v]$</td><td>for{key <- v.keys
σ' <- σ.extend($p_k =$ key)
σ'' <- σ'.extend($p_v \sqsubseteq v$(key))
} yield σ''</td><td>if(v.keys.size != 1) Set() else
for{key <- v.keys
σ' <- σ.extend($p_k =$ key)
σ'' <- σ'.extend($p_v = v$(key))
} yield σ''</td></tr>
</table>

Figure 16.4: Definition of *extension* of an existing valuation to account for an additional predicate.

yield a value v, and compute the extension according to the right-most column with the requirement $l = v$.

We now explain each of the rows of Figure 16.4.

Constants. When p_0 is a constant k, the valuation cannot change the truth value of $k \sqsubseteq v$ or $k = v$. We only keep the valuation in a singleton set if the condition is satisfied, or return the empty set of valuations if it isn't.

Variables. When p_0 is a variable l, intuitively we would like to update the valuation of l to $v{\downarrow}$ or $v!$ to satisfy the condition $l \sqsubseteq v$ or $l = v$, respectively. We do exactly this if σ does not yet have an existing value for l. Otherwise, we must intersect the new value given to l to satisfy P_i with the old value already in σ, so that the resulting value continues to satisfy the preceding predicates $P_1, \ldots, P_{i-1}$. When both the new value v and he old value v' are downward-closed, their intersection is their meet, also downward-closed: $(v \sqcap v'){\downarrow}$. When one of the values is downward-closed but the other is exact, their intersection is the exact value if it is in the downward closure; otherwise their intersection is empty, so we return the empty set of valuations. Similarly, when both v and v' are exact, we return the existing valuation if $v = v$, or the empty set of valuations if $v! = v'$.

Tagged Values. When p_0 is a tagged pattern $\mathrm{Tag}\ p$, we just return the existing valuation σ. We do not attempt to bind values to the variables in the sub-pattern p. Since the $\sqsubseteq$ relation for the tagged type is defined only programmatically, we have no way of enumerating the values of p that would satisfy the condition; sometimes, there are even infinitely many. Instead, we leave the variables in p unconstrained. Later, as a last step after we have computed the extension of σ ignoring all tagged patterns in P_i, we filter the extended valuations and keep only those that satisfy the tagged patterns in P_i. Thus, the final set of valuations contains only those that satisfy all of P_i, including tagged patterns inside P_i.

Sets. When p_0 is a set pattern $\{p\}$, we iterate through all of the elements of the set v and successively bind p to each of them in a set of extended valuations. The binding must be exact ($p =$ `elem`), since binding p to a different element that is less than `elem` would not satisfy the condition. When the condition is exact ($\{p\} = v$), we additionally check that v is a singleton set. For clarity of presentation, Figure 16.4 only considers the case when the pattern p_0 is a singleton set pattern $\{p\}$, but it is straightforward to extend the semantics for set patterns with multiple elements ($\{p_1, \ldots, p_n\}$), and our implementation does so. When a set pattern contains multiple elements, we require the extension to

assign them distinct values. For example, the predicate $\{l, l'\} \sqsubseteq X$ holds only when X contains at least two elements, and constrains l and l' to all pairs of distinct elements selected from X.

Tuples. When p_0 is a tuple pattern $\langle p_1, p_2 \rangle$, we distribute the condition $\langle p_1, p_2 \rangle$ over the components of the tuple and enforce the corresponding rules $p_1 \sqsubseteq v_1$ and $p_2 \sqsubseteq v_2$. The figure only shows the case of a pair, but our implementation naturally extends the technique to tuples of all sizes.

Maps. When p_0 is a map pattern $[p_k \mapsto p_v]$, we enumerate the keys of v and successively assign each of them to p_k in different valuations. Like for the set pattern, this assignment must be exact ($p_k = \texttt{key}$). Then, for each key, we compute the extension respecting $p_v \sqsubseteq v(\texttt{key})$. This extension uses $\sqsubseteq$ because $[v_k \mapsto v_v] \sqsubseteq [v_k \mapsto v'_v]$ whenever $v_v \sqsubseteq v'_v$.

After constructing the new valuations in the first step, we check whether they satisfy the condition $p_0 \sqsubseteq v$ and discard those that do not. When p_0 does not contain any tagged sub-patterns, all valuations satisfy the condition by construction. For each sub-pattern of p_0 of the form $\mathrm{Tag}\ p$, we instantiate the variables in p using each of the valuations σ that we have computed, and evaluate $\mathrm{Tag}\ \sigma p \sqsubseteq v'$, where v' is the subterm of v corresponding syntactically to the sub-pattern $\mathrm{Tag}\ p$ of p_0. Whenever the condition evaluates to false, we discard the valuation σ. It is guaranteed that σ is defined for all variables occurring in p, since they are also variables occurring in P_i, and the binding requirement ensures that all variables in P_i are constrained by some P_j with $j \sqsubseteq i$.

We complete our presentation of the extension process with the lemma that was used in the previous section:

Lemma 1. *If $\sigma_{i-1} \sqsubseteq \sigma'_{i-1}$, and S_i, S'_i are the extensions of $\sigma_{i-1}, \sigma'_{i-1}$, respectively, for predicate P_i, then $S_i \prec S'_i$, where $\prec$ is defined as in the proof of Proposition 2.*

Proof sketch. When P_i is of the form $p \sqsubseteq X$, we let $v = \sigma_{i-1}(X)$ and $v' = \sigma'_{i-1}(X)$, so $v \sqsubseteq v'$. We prove the result by induction on the structure of p following the definitions of `extend` in Figure 16.4. Most of the cases are trivial. The case for $p = l$ uses the premise that $v \sqsubseteq v'$ to ensure that the resulting valuations satisfy the required ordering. When P_i is of the form $l = \mathrm{F}(p)$, monotonicity of F and of the pattern constructors in p ensures that values assigned to l respect the required ordering, so again $S_i \prec S'_i$. □

Functions

Much of the expressive power of FLIX comes from functions. Functions, in contrast to the relations of Datalog or the lattices of FLIX, are represented

$$
\begin{array}{rcl}
e \in \mathit{Exp} & ::= & \mathit{Bool} \mid \mathit{Int} \mid \mathit{Str} \mid \odot\ e \mid e_1 \otimes e_2 \\
& \mid & \{\overline{e}\} \mid \langle \overline{e} \rangle \mid [\overline{e_k \mapsto e_v}] \mid \mathrm{Tag}\ e \\
& \mid & l \mid \lambda\ l : \tau\ \ e \mid e_1\,(e_2) \\
& \mid & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
& \mid & e\ \texttt{match}\ (\overline{p, e}) \\
\odot & = & \text{is a set of unary operators.} \\
\otimes & = & \text{is a set of binary operators.}
\end{array}
$$

Figure 16.5: Expressions in FLIX.

as code that is executed by the solver when needed. Functions are used to specify the operations that define custom lattices ($\sqsubseteq$, $\sqcup$), and can be used in the declarative rules to perform monotone operations over lattice elements (e.g. the abstract $\widehat{\texttt{sum}}$ function).

There are two options for specifying a function, with associated tradeoffs. First, FLIX defines a function implementation language based on the simply-typed lambda calculus (STLC). Second, a FLIX program may call external functions written in a general-purpose language. The first option guarantees that every function terminates, and the proof obligations that ensure correctness of a FLIX program can be proven automatically in most cases. The second option enables the programmer to express more general functions, and to interface with data structures constructed in a general-purpose host language, but if a correctness proof of the FLIX program is desired, then the programmer must prove the relevant proof obligations manually for those functions.

The syntax of function definitions is in Figure 16.5. Functions are pure in that every term is an expression and has no side-effects. The syntax includes expressions for constructing the various values, and unary and binary operators for performing computations on values. Values are destructed using pattern matching, which also acts a control-structure together with the if-then-else expression. The language also has lambda abstractions and applications. All functions take a single parameter, but they may use currying to implement multiple parameters. As a form of syntactic sugar, we allow a rule to provide multiple arguments to a function; in this case, the curried function is successively applied to each of the arguments, so that all of the curried parameters are bound.

Lattices

In FLIX, a lattice is a 4-tuple $\langle \tau, \bot, \sqsubseteq, \sqcup \rangle$, where τ is the element type, $\bot : \tau$ is the bottom element, $\sqsubseteq : \tau \to \tau \to$ *Bool* implements the less-than relation, and $\sqcup : \tau \to \tau \to \tau$ implements the least upper bound computation. Two additional components are optional: a meet operator $\sqcap : \tau \to \tau \to \tau$ and a normalization function $\boxdot : \tau \to \tau$, which will be explained in the next subsection. The FLIX solver statically detects whether any of the rules in a program may need a meet operator for a given lattice at runtime, and issues a static error if a meet operator is not specified in that case. In addition to satisfying the above types, the lattice must satisfy a set of properties that will be defined in Section 16.2.

Built-in & User-specified Lattices. The set, map and product lattices are automatically inferred for every type. For every set type $\{\tau\}$ used in a FLIX program, the solver automatically declares the power-set lattice: $\langle \{\tau\}, \emptyset, \subseteq, \cup, \cap, \lambda v.v \rangle$. For every tuple type whose components are all lattices, the solver automatically declares the component-wise lattice.

For every map type $[\tau \mapsto \tau']$ where τ' is a lattice the solver automatically declares the map lattice:

$$\langle [\tau \mapsto \tau'], \bot, \lambda v.\lambda v'.\ \forall k.\ v(k) \sqsubseteq v'(k),$$
$$\lambda v.\lambda v'.\ \lambda k.v(k) \sqcup v'(k), \lambda v.\lambda v'.\ \lambda k.v(k) \sqcap v'(k), \lambda v.v \rangle$$

The programmer may also explicitly declare a lattice for any tagged type. The automatically inferred lattices can be overridden by wrapping the desired type in a tag and defining a lattice for the tagged type.

Preorders and Partial Orders

Sometimes, a lattice is defined over a set that is inconvenient to represent exactly with a data structure, and the programmer chooses a slightly more general data representation instead. For example, the interval semi-lattice is defined on the set $\{\bot\} \cup \{\langle l, u \rangle : l \sqsubseteq u\}$. It is convenient to represent an element of this lattice by a pair $\langle l, u \rangle$, but this representation admits many pairs that are not elements of the lattice, such as $\langle 1, 0 \rangle$. It is also convenient to declare that all such inverted intervals should be a representation of $\bot$, and this can be done by defining the less-than relation to satisfy $\{l, u\} \sqsubseteq \{l', u'\}$ whenever $u < l$. However, the resulting order is now a preorder rather than a partial order, because it is not anti-symmetric. Note that a preorder that is not a partial order can never be a semi-lattice, because no upper bound can ever be least for two elements for which $v \sqsubseteq v' \sqsubseteq v$. However, every preorder induces a minimal partial order on the equivalence classes of elements that

satisfy $v \sqsubseteq v' \sqsubseteq v$. In the case of the interval representation, this induced partial order is exactly desired on: it unifies all of the inverted intervals as one element (equivalence class) that is less than every element, and it leaves the valid intervals unchanged.

FLIX allows a lattice to be defined in this way: the programmer defines $\sqsubseteq$ as a preorder, and the lattice is considered to be defined on the minimal partial order induced by that preorder. The programmer must provide a normalization function $\boxdot$ that maps each element to a unique representative of its equivalence class. For example, the $\boxdot$ function for the interval lattice could map every inverted interval to $\langle 1, 0 \rangle$. FLIX applies the normalization function to every lattice element that it computes, so that it always represents an equivalence class internally by its unique representative. The verifier checks that the normalization function is consistent with the defined $\sqsubseteq$ relation, and that the order after normalization satisfies all of the partial order and lattice properties. Another example in which this feature is useful is in Section 16.4.

Syntactic Sugar

The core language is small and easy to reason about. We add shorthands to make it easier to write and read programs.
The $\in$ operator expresses set membership:

$$p \in A \;\rightsquigarrow\; \{p\} \sqsubseteq A$$

Programs can look up elements in maps:

$$p_v \sqsubseteq A(p_k) \;\rightsquigarrow\; [p_k \mapsto p_v] \sqsubseteq A$$

When the key is a tuple, the tuple brackets can be omitted:

$$p_v \sqsubseteq A(\overline{p_k}) \;\rightsquigarrow\; p_v \sqsubseteq A(\langle \overline{p_k} \rangle)$$

In the core language, the value returned from a function must be bound to a local variable. When a function call appears in a pattern or in a predicate of the form $p \sqsubseteq \mathrm{F}\,(p')$, we automatically introduce a fresh variable:

$$p \sqsubseteq \mathrm{F}\left(p'\right) \;\rightsquigarrow\; l = \mathrm{F}\left(p'\right) \;\wedge\; p \sqsubseteq l$$

A function that returns a boolean can be used as a predicate:

$$\mathrm{F}\,(p) \;\rightsquigarrow\; \text{true} \sqsubseteq \mathrm{F}\,(p)$$

Every instance of the wildcard variable _ (underscore) translates into a fresh variable name.
The surface language allows the definition of names that represent types, so that complicated types (especially tagged unions) do not have to be repeated every time they are used.

16.2 Verification

Before executing a FLIX program, the solver checks statically the binding requirement and the type rules defined in the previous section. However, we desire a stronger correctness guarantee like Datalog's: a FLIX program should compute the least solution of the rules, and it should eventually terminate. In this section, we define a set of simple properties and declare a FLIX program *correct* if it is known to satisfy all of them. A correct program satisfies the higher-level property of computing the least solution in finite time.

Lattice & Function Properties

A user-defined semi-lattice must respect the properties that define a semi-lattice: it must be a partial order with a least element, the definition of $\sqsubseteq$ must be consistent with the definition of $\sqcup$, and if its domain is infinite, then a termination function is required to prove its bounded height. Together, these properties ensure the existence of a least fixed point that can be computed in a finite number of steps [Cousot and Cousot, 1992].

Partial Order. The $\sqsubseteq$ function must define partial order:

$$\forall x.\, x \sqsubseteq x \quad \text{(Reflexivity)}$$

$$\forall x, y.\, x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y \quad \text{(Anti-Symmetry)}$$

$$\forall x, y, z.\, x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z \quad \text{(Transitivity)}$$

and the bottom element $\bot$ must be the least element:

$$\forall x.\, \bot \sqsubseteq x \quad \text{(Least-Element)}$$

Least Upper Bound. The least-upper-bound $\sqcup$ must be consistent with the partial order $\sqsubseteq$:

$$\forall x, y.\, x \sqsubseteq (x \sqcup y) \land y \sqsubseteq (x \sqcup y) \quad \text{(Lub-1)}$$

$$\forall x, y, z.\, x \sqsubseteq z \land y \sqsubseteq z \Rightarrow (x \sqcup y) \sqsubseteq z \quad \text{(Lub-2)}$$

Greatest Lower Bound. The greatest-lower-bound $\sqcap$, if specified, must be consistent with the partial order $\sqsubseteq$:

$$\forall x, y.\, (x \sqcap y) \sqsubseteq x \land (x \sqcap y) \sqsubseteq y \quad \text{(Glb-1)}$$

$$\forall x, y, z.\, z \sqsubseteq x \land z \sqsubseteq y \Rightarrow z \sqsubseteq (x \sqcap y) \quad \text{(Glb-2)}$$

Lattice Height. The termination function h must be total, strictly decreasing and always non-negative:

$$\forall x, \exists y.\, y = h(x) \quad \text{(TOTAL)}$$
$$\forall x, y.\, x \sqsubseteq y \wedge x \neq y \Rightarrow h(y) < h(x) \quad \text{(DECREASING)}$$
$$\forall x.\, h(x) > 0 \quad \text{(NON-NEGATIVE)}$$

Preorders. If the lattice is defined as a preorder, the normalization function $\boxdot$ must ensure anti-symmetry:

$$\forall x, y.\, \boxdot(x) \sqsubseteq \boxdot(y) \wedge \boxdot(y) \sqsubseteq \boxdot(x) \Rightarrow \boxdot(x) = \boxdot(y). \quad (\boxdot\text{-ANTI-SYMMETRY})$$

Monotone Functions. Functions defined over lattice elements must be total and monotone:

$$\forall x.\, \exists y.\, y = f(x) \quad (\lambda\text{-TOTAL})$$
$$\forall x, y.\, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \quad (\lambda\text{-MONOTONE})$$

Automated Verification

The above list of proof obligations is helpful even if a programmer proves them by hand, because the obligations are simple and should be easy to prove, yet together, they guarantee a high-level correctness result. In many cases, FLIX can prove most of the properties automatically, further reducing the manual effort required to ensure correctness.

Encoding the proof obligations directly in the language of an SMT solver would be difficult because they contain quantification, and because FLIX has features such as tuples, algebraic data types and pattern matching which are not directly supported by most off-the-shelf SMT solvers.

Instead, we try to verify the properties inside our system, and only when necessary do we generate verification conditions over basic domains to be discharged by an arbitrary SMT solver. We observe that we can verify a property such as TRANSITIVITY by constructing a function:

$$f = \lambda x, \lambda y, \lambda z.(x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z)$$

and then verifying that f returns true for all inputs. We can encode all other properties in a similar way. If every parameter type τ is inhabited by a finite number of values, then we can prove the property by exhaustively evaluating the function for all inputs. Since the function is defined in the STLC, the evaluation is guaranteed to terminate. We can use this strategy to prove the properties for any finite lattice.

We can use a variation of the technique when a parameter type τ is inhabited by an infinite number of values (e.g. the *Int* type). We replace every parameter of infinite type by a symbolic argument and partially evaluate the function. Partial evaluation either proves the property, or reduces the function to a simpler one in which tuples, algebraic data types and pattern matching have been eliminated. The remaining function consists of only if-then-else and integer expressions, and can therefore be easily verified by an SMT solver or manually.

Whenever a proof obligation does not hold, the verifier outputs an error message with a counter-example that is helpful for debugging.

16.3 Solver

The least fixed point of a FLIX program can be computed by *bottom-up* evaluation. When a rule is evaluated, the global variable at the head of the rule is updated as defined in Section 16.1. A *naïve* bottom-up strategy would evaluate the rules in an arbitrary order. Whenever the value of a global variable changes, all rules must be reevaluated at some point in the future. The least fixed point is reached when the evaluation of every rule does not change any of the global variables. A more efficient strategy, *semi-naïve evaluation*, considers the dependencies between rules. If a global variable A is updated with a new value, then only rules that reference A in their body need to be reevaluated. The solver implements this strategy with a worklist of rules needing to be reevaluated. Whenever a global variable is updated, rules that reference it in their body are added to the worklist. The fixed point is known to be reached when the worklist is empty.

Datalog solvers often implement a performance improvement called *difference propagation*. When a set of elements is added to an existing set, only the new elements need to be considered by dependent rules. In FLIX, difference propagation is only sound when the predicates in a rule are distributive in the variables that they depend on: $\sigma P \vee \sigma' P \Leftrightarrow (\sigma \sqcup \sigma')P$. Some predicates, such a membership of a single element in a set, are distributive, but others are not. For example, the predicate $\{a, b\} \sqsubseteq A$ requires that the set A has at least two elements. If we add one element at a time to A, then the predicate would never hold for any of the difference sets, even though it holds for A as soon as two elements have been added to it. The FLIX solver implements difference propagation, but enables it only when the predicates have been designated as distributive. The correctness of distributivity declarations can be checked by the verifier.

We have implemented a prototype toolchain for FLIX in Scala. The implementation is approximately 5,000 lines of code, including parsing, type-

$$
\begin{array}{rcl}
a \in \mathrm{Pt}(p) & \Leftarrow & \langle p, a\rangle \in \text{AddrOf} \\
a \in \mathrm{Pt}(p) & \Leftarrow & \langle p, q\rangle \in \text{Copy} \;\wedge\; a \in \mathrm{Pt}(q) \\
\mathrm{Cst}\;\; b \sqsubseteq \mathrm{SUa}(l, a) & \Leftarrow & \langle l, p, q\rangle \in \text{Store} \;\wedge\; a \in \mathrm{Pt}(p) \;\wedge\; b \in \mathrm{Pt}(q) \\
b \in \mathrm{PtH}(a) & \Leftarrow & \langle _, p, q\rangle \in \text{Store} \;\wedge\; a \in \mathrm{Pt}(p) \;\wedge\; b \in \mathrm{Pt}(q) \\
b \in \mathrm{Pt}(p) & \Leftarrow & \langle l, p, q\rangle \in \text{Load} \;\wedge\; a \in \mathrm{Pt}(q) \;\wedge\; b \in \mathrm{PtSU}(l, a) \\
t \sqsubseteq \mathrm{SUb}(l_2, a) & \Leftarrow & \langle l_1, l_2\rangle \in \text{CFG} \;\wedge\; t \sqsubseteq \mathrm{SUa}(l_1, a) \\
t \sqsubseteq \mathrm{SUa}(l, a) & \Leftarrow & t \sqsubseteq \mathrm{SUb}(l, a) \;\wedge\; a \in \mathrm{Preserve}(l) \\
b \in \mathrm{PtSU}(l, a) & \Leftarrow & b \in \mathrm{PtH}(a) \;\wedge\; \mathrm{Cst}\; b \sqsubseteq \mathrm{SUb}(l, a) \\
a \in \text{AllAddrs} & \Leftarrow & \langle _, a\rangle \in \text{AddrOf} \\
a \in \mathrm{Preserve}(l) & \Leftarrow & \langle l, _, _\rangle \in \text{Load} \;\wedge\; \langle _, a\rangle \in \text{AddrOf} \\
a \in \mathrm{Preserve}(l) & \Leftarrow & \langle l, p, q\rangle \in \text{Store} \;\wedge\; b \in \mathrm{Pt}(p) \;\wedge\; \{a, b\} \in \text{AllAddrs}
\end{array}
$$

Figure 16.6: FLIX implementation of the Strong Update analysis [Lhoták and Chung, 2011]. SUa and SUb denote the Strong Update information after and before label l. Preserve is the complement of the Kill set.

checking, verification and the semi-naïve evaluation. The use of Scala, and as a consequence the Java Virtual Machine, means that the implementation can call any (external) function written in any language that targets the JVM. The overall design of FLIX is independent of Scala and the JVM, and could be integrated into any other general-purpose language. To solve the integer constraints generated by the verification phase, the implementation uses the Z3 SMT solver from Microsoft [De Moura and Bjørner, 2008].

16.4 Case Studies

We have used FLIX to implement several static analyses. We report on our experience in this section.

The Strong Update Analysis

The Strong Update analysis is a points-to analysis for C programs that propagates singleton points-to sets flow sensitively and larger sets flow insensitively. The paper that presents the analysis first presents it as a set of constraints (Figure 7 in that paper) and later as an imperative algorithm (Figure 9 in that paper), and evaluates a C++ implementation of the imperative algorithm [Lhoták and Chung, 2011]. The FLIX implementation shown in Figure 16.6 follows the constraint specification of the analysis directly: there is a one-to-one correspondence between the FLIX rules and the constraints from Figure 7 of the Strong

Update paper. There are only two differences. The paper defines the PtSU relation using a case split for which it is non-trivial to show monotonicity, while the FLIX program defines it with a simple rule that is clearly monotone. The last three lines of the FLIX program define the Preserve relation, while the paper defines the complementary Kill relation in an earlier figure.

The FLIX implementation does not require the use of any functions, but it does require a custom lattice. The type of the SUa and SUb variables is a lattice similar to a constant propagation lattice, but with abstract objects as constants instead of integers.

We tested the implementation on examples intended to illustrate the specifics of the Strong Update analysis. For this FLIX program, the verifier was able to prove all of the required properties, so we have a guarantee that the program terminates and computes the least solution satisfying the rules, which directly mirror the constraints that specify the analysis in the paper.

IFDS

IFDS is a framework that can be instantiated to solve a large class of interprocedural context-sensitive dataflow analyses, the interprocedural finite distributive subset analyses [Reps et al., 1995]. The paper that defines the framework presents it as a one-page algorithm containing many worklist updates and implicit quantifications. Anecdotally, many people find the algorithm difficult to understand, and checking its correctness requires a long proof.

Figure 16.7 specifies declaratively the desired properties of an IFDS solution. It is also a set of FLIX rules that can be executed to compute the solution. The rules compute a set of *path edges*, each leading from a data point d_1 at the start of a procedure to a data point d_3 at an instruction m within the procedure, and a set of *summary edges* that summarize the transfer function of each call to a procedure. To implement a specific analysis that is an instance of the IFDS framework, one needs to additionally implement the transfer functions E#Intra (intraprocedural), E#CS (call-site-to-start-node), and E#ER (end-node-to-return-site), which we have left as function calls in the rules.

The IFDS framework does not require any custom lattices. It does require the ability to call functions that implement the transfer functions of a given analysis; for most analyses, it would be very cumbersome to tabulate those transfer functions fully in input lattices.

We implemented the transfer functions for the possibly-uninitialized example analysis from the IFDS paper, and confirmed that its output matches the paper.

$$
\begin{aligned}
\langle d_1, \langle m, d_3 \rangle \rangle \in \text{PathE} &\Longleftarrow \langle d_1, \langle n, d_2 \rangle \rangle \in \text{PathE} \;\wedge\; \langle n, m \rangle \in \text{CFG} \;\wedge\; d_3 \in \text{E\#Intra}\,(n, d_2) \\
\langle d_3, \langle s, d_3 \rangle \rangle \in \text{PathE} &\Longleftarrow \langle d_1, \langle c, d_2 \rangle \rangle \in \text{PathE} \;\wedge\; \langle c, t \rangle \in \text{CG} \;\wedge\; d_3 \in \text{E\#CS}\,(\langle c, d_2 \rangle, t) \;\wedge\; s = \text{Start}\,(t) \\
d_5 \in \text{SummE}(c, d_4) &\Longleftarrow \langle _, \langle c, d_4 \rangle \rangle \in \text{PathE} \;\wedge\; \langle c, t \rangle \in \text{CG} \;\wedge\; d_1 \in \text{E\#CS}\,(\langle c, d_4 \rangle, t) \\
&\qquad \wedge\; e = \text{End}\,(t) \;\wedge\; \langle d_1, \langle e, d_2 \rangle \rangle \in \text{PathE} \;\wedge\; d_5 \in \text{E\#ER}\,(t, d_2, c) \\
\langle d_1, \langle m, d_3 \rangle \rangle \in \text{PathE} &\Longleftarrow \langle d_1, \langle n, d_2 \rangle \rangle \in \text{PathE} \;\wedge\; d_3 \in \text{SummE}(n, d_2) \;\wedge\; \langle n, m \rangle \in \text{CFG}
\end{aligned}
$$

Figure 16.7: FLIX implementation of the IFDS analysis framework Reps et al. [1995].

IDE

IDE is a more general framework that can be instantiated to solve a larger class of interprocedural context-sensitive dataflow analyses, the interprocedural distributive environment analyses [Sagiv et al., 1996]. The original presentation of IDE as an imperative algorithm requires two pages. Conceptually, the IDE framework is a direct extension if the IFDS framework, but this is not clear from the worklist-based algorithmic specifications. The IDE framework computes the same edges as IFDS, but each edge is decorated with a representation of a so-called micro-function. This correspondence between IFDS and IDE is immediately clear from the declarative specification of IDE shown in Figure 16.8. The rules mirror those of the IFDS implementation (with path edges and summary edges renamed to jump functions and summary functions respectively), except that each rule has additional handling of the micro-functions on the edges. The first, third, and fourth rule call a FLIX function Comp to compute the composition of micro-functions.

To instantiate the IDE framework with a specific program analysis, one must not only implement the transfer functions E#Intra, E#CS, and E#ER, but also specify two custom lattices: the value lattice V that is the domain and range of each micro-function, and the micro-function lattice F that efficiently represents certain functions from $V \to V$. Thus, the FLIX IDE implementation requires both functions and custom lattices.

The IDE paper uses a running example of a linear constant propagation analysis in which V is the constant propagation lattice. The elements of the micro-function lattice F are $\lambda l.\bot$ and functions of the form $\lambda l.(a \times l + b) \sqcup c$, where a and b are integers and c is an element of the constant propagation lattice. Figure 16.9 shows the FLIX implementation of $\sqcup$ for the micro-function lattice. The functions that implement $\sqsubseteq$ and micro-function composition have a similar structure. The natural representation of F as a triple $\langle a, b, c\rangle$ gives rise to a preorder, because when $c = \top$, $\lambda l.(a \times l + b) \sqcup c$ is equivalent to $\lambda l.\top$ regardless of the values of a and b. The normalization function is necessary to map all of these equivalent functions to a canonical representation $\langle 0, 0, \top\rangle$ in order to satisfy the partial order property.

We confirmed that the FLIX implementation of the linear constant propagation IDE example outputs the same results as given in the IDE paper.

Interval Analysis

An interval analysis determines, for each expression, an interval that contains every possible value of the expression. Each interval is either $\bot$ or a pair $[l, u]$ with $l \sqsubseteq u$. We can use all of the inverted intervals (for which $l > u$) to

$$
\begin{aligned}
f \sqsubseteq \text{JumpF}(d_1, \langle m, d_3 \rangle) &\Longleftarrow f_{12} \sqsubseteq \text{JumpF}(d_1, \langle n, d_2 \rangle) \ \wedge \ \langle n, m \rangle \in \text{CFG} \ \wedge \ [d_3 \mapsto f_{23}] \sqsubseteq \text{E\#Intra}\,(n, d_2) \ \wedge \ f = \text{Comp}\,(f_{12}, f_{23}) \\
id \sqsubseteq \text{JumpF}(d_3, \langle s, d_3 \rangle) &\Longleftarrow f_1 \sqsubseteq \text{JumpF}(d_1, \langle c, d_2 \rangle) \ \wedge \ \langle c, t \rangle \in \text{CG} \ \wedge \ [d_3 \mapsto f_2] \sqsubseteq \text{E\#CS}\,(\langle c, d_2 \rangle, t) \ \wedge \ s = \text{Start}\,(t) \\
&\quad \wedge \text{NotBot}\,(f_1) \ \wedge \ \text{NotBot}\,(f_2) \ \wedge \ id = \text{Id}\,() \\
[d_5 \mapsto cser] \sqsubseteq \text{SummF}(c, d_4) &\Longleftarrow f_1 \sqsubseteq \text{JumpF}(_, \langle c, d_4 \rangle) \ \wedge \ \langle c, t \rangle \in \text{CG} \ \wedge \ [d_1 \mapsto cs] \sqsubseteq \text{E\#CS}\,(\langle c, d_4 \rangle, t) \\
&\quad \wedge e = \text{End}\,(t) \ \wedge \ se \sqsubseteq \text{JumpF}(d_1, \langle e, d_2 \rangle) \ \wedge \ [d_5 \mapsto er] \sqsubseteq \text{E\#ER}\,(t, d_2, c) \\
&\quad \wedge cse = \text{Comp}\,(cs, se) \ \wedge \ cser = \text{Comp}\,(cse, er) \ \wedge \ \text{NotBot}\,(f_1) \\
f \sqsubseteq \text{JumpF}(d_1, \langle m, d_3 \rangle) &\Longleftarrow f_{12} \sqsubseteq \text{JumpF}(d_1, \langle n, d_2 \rangle) \ \wedge \ [d_3 \mapsto f_{23}] \sqsubseteq \text{SummF}(n, d_2) \ \wedge \ \langle n, m \rangle \in \text{CFG} \ \wedge \ f = \text{Comp}\,(f_{12}, f_{23})
\end{aligned}
$$

Figure 16.8: FLIX implementation of the IDE analysis framework [Sagiv et al., 1996].

$$
\begin{array}{l}
\lambda t_1 : \text{Transform } \ \lambda t_2 : \text{Transform } \ \langle t_1, t_2 \rangle \ \texttt{match}(\\
\langle \text{BotTrans}, _ \rangle, t_2, \qquad\qquad\qquad\qquad \langle _, \text{BotTrans} \rangle, t_1, \\
\langle \text{Trans } \langle a_1, b_1, c_1 \rangle, \text{Trans } \langle a_2, b_2, c_2 \rangle \rangle, \texttt{if } \ a_1 = a_2 \\
\ \texttt{then if } \ b_1 = b_2 \ \texttt{then } \ c_1 \sqcup c_2 \ \texttt{else } \ \text{Trans } \langle 1, 0, \top \rangle \\
\ \texttt{else if } \ (b_1 - b_2)\%(a_2 - a_1) \neq 0 \ \texttt{then } \ \text{Trans } \langle 1, 0, \top \rangle \\
\quad \texttt{else } \ \text{Trans } \langle a_1, b_2, \text{Cst } a_1 * (b_1 - b_2)/(a_2 - a_1) + b_1 \sqcup c_1 \sqcup c_2 \rangle)
\end{array}
$$

Figure 16.9: Microfunction join operation for the example IDE analysis [Sagiv et al., 1996].

represent $\bot$ by defining a preorder as follows:

$$\langle l, u \rangle \sqsubseteq \langle l', u' \rangle = u < l \vee (l' \sqsubseteq l \wedge u \sqsubseteq u')$$

We then define a normalization function to select a unique representative for the inverted intervals:

$$\boxdot(\langle l, u \rangle) = \texttt{if } \ l \sqsubseteq u \ \texttt{ then } \ \langle l, u \rangle \ \texttt{ else } \ \langle 1, 0 \rangle$$

Shortest Path Algorithm

It is possible to express a shortest path algorithm similar to Dijkstra's in FLIX. The input is a weighted connected graph $G = (V, E)$. Each edge $s \xrightarrow{d} t$ is labelled with the distance d from s to t. The maximum possible distance m between any two vertices is the sum of all distances. We define the following lattice over integers: $\langle \texttt{Int}, m, \geq, \mathit{min} \rangle$. The join of two distances is the shorter one, and the initial distance $\bot$ is m. The program to find shortest paths consists of two rules:

$$
\begin{array}{r}
0 \sqsubseteq \mathit{Dist}(x). \qquad\qquad\qquad\qquad\qquad\qquad \\
d + d' \sqsubseteq \mathit{Dist}(t) \Leftarrow \langle s, d, t \rangle \in \mathit{Edge} \wedge d' \sqsubseteq \mathit{Dist}(s).
\end{array}
$$

Here, x is the distinct source vertex, and $+$ is a monotone function under the order $\geq$.

16.5 Related Work

Static Analysis Frameworks. Many static analysis frameworks have been proposed over the years. The Program Analysis Generator PAG generates C code for static analyzers from high level descriptions [Martin, 1998]. PAG takes as input declarative specifications of lattices and transfer functions. The TJ Watson WALA library is a static analysis library written in Java [Fink, 2014]. WALA includes implementations of many common static analyses such as points-to analysis, class hierarchy analysis and the IFDS algorithm [Reps et al., 1995]. Soot is a Java bytecode analysis framework that has seen wide use in compilation and as a fronted for other static analyses [Vallée-Rai et al., 1999a]. Hoopl is a generic dataflow analysis framework written in Haskell [Ramsey et al., 2010]. These analysis frameworks are based around specific intermediate representations of a programs, the control-flow graph, and dataflow analysis. FLIX, on the other hand, is applicable to least fixed point problems in general. Furthermore, unlike these frameworks, FLIX attempts to statically verify the properties required of the analysis specification, such as the lattice properties for lattice specifications, and monotonicity for transfer functions.

Datalog. Datalog has it roots in the database community as a general purpose query language, but has found a niche in static analysis. Ceri et al. [1989] gives a comprehensive introduction and survey of Datalog. Huang et al. [2011] provides a more concise introduction, and argues that Datalog is experiencing a re-emergence in use.

Datalog was used to implement CodeQuest, a tool for querying numerous aspects of the source code of a program [Hajiyev et al., 2006]. Binary-decision-diagrams (BDDs) have been used to implement efficient points-to analyses specified as relations, both using custom relational language and Datalog [Lhoták and Hendren, 2003; Whaley and Lam, 2004]. The Doop framework is a precise and scalable points-to analysis for Java specified in Datalog [Bravenboer and Smaragdakis, 2009].

Various extensions of Datalog have been proposed. `Relationlog` is a typed extension of Datalog with sets and tuples [Liu, 1998]. A variant of Datalog has been extended with arithmetic constraints [Fribourg and Peixoto, 1994]. To our knowledge, FLIX is the first Datalog-like language with lattices and functions on them.

Alternation-free Least Fixed Point (ALFP) logic is an extension of Datalog-style Horn clauses with universal and existential quantification, stratified negation, and disjunction in rule bodies [Nielson et al., 2004]. Despite these expressive extensions, ALFP programs can still be solved efficiently. The extensions of ALFP are orthogonal to those of FLIX: rather than quantification,

FLIX extends relations to user-defined lattices, and enables use of functions in rules.

Set Constraint Systems. Set constraint systems [Aiken, 1999] are systems of subset constraints of the form $X \subseteq Y$, in which both X and Y are expressions over sets using union, intersection, and negation, but also construction and destruction (projection) of named tuples. Due to the possibility of expressions on both sides of the $\subseteq$ relation, set constraints are very expressive, and it would be possible to encode the rules expressed in FLIX as set constraints. However, the expressiveness of set constraints is a double-edged sword: practical solving algorithms are limited to specific subsets of set constraints. FLIX is an example of such a practical subset. Moreover, FLIX sharply separates the specification of lattices from the fixed point rules over lattice variables, which makes it easier to understand the meaning of a FLIX program and its solution than if the encodings were tangled together in a system that is more general than needed.

16.6 Conclusion

We have presented FLIX, a language for declaratively expressing and solving least fixed point problems, particularly static program analyses. Like Datalog, FLIX is declarative, but allows the use of arbitrary lattices and general-purpose functions. A verifier for simple proof obligations guarantees the same high-level correctness properties that are ensured by Datalog. We have demonstrated the expressivity of FLIX by implementing several popular program analyses.

Bibliography

Abramson, D. and Pike, L. (2011). When Formal Systems Kill: Computer Ethics and Formal Methods. *APA Newsletter on Philosophy and Computers*.

Adobe (2015). Adobe. `http://www.adobe.com/devnet/acrobat/javascript.html`.

Aho, A. V. (2003). *Compilers: Principles, Techniques and Tools*.

Aiken, A. (1999). Introduction to Set Constraint-based Program Analysis. *Science of Computer Programming*.

Ali, K. and Lhoták, O. (2012). Application-only Call Graph Construction. In *European Conference On Object-Oriented Programming (ECOOP)*.

Ali, K. and Lhoták, O. (2013). Averroes: Whole-program Analysis Without the Whole Program. In *European Conference On Object-Oriented Programming (ECOOP)*.

Andersen, L. O. (1994). *Program Analysis and Specialization for The C Programming Language*. PhD thesis, University of Copenhagen (DIKU).

Anderson, C., Giannini, P., and Drossopoulou, S. (2005). Towards Type Inference for JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*.

Andreasen, E. and Møller, A. (2014). Determinacy in Static Analysis for jQuery. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Artzi, S., Dolby, J., Jensen, S. H., Møller, A., and Tip, F. (2011). A Framework for Automated Testing of JavaScript Web Applications. In *International Conference on Software Engineering (ICSE)*.

Artzi, S., Dolby, J., Tip, F., and Pistoia, M. (2012). Fault Localization for Dynamic Web Applications. In *IEEE Transactions on Software Engineering*.

Balakrishnan, G., Reps, T., Melski, D., and Teitelbaum, T. (2008). WYSINWYX: What You See Is Not What You Execute. In *Verified software: Theories, Tools, Experiments*.

Balakrishnan, G. and Reps, T. W. (2006). Recency-Abstraction for Heap-Allocated Storage. In *International Static Analysis Symposium (SAS)*.

Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. (1985). Magic Sets and Other Strange Ways to Implement Logic Programs. In *Symposium on Principles of Database Systems*.

Berndl, M., Lhoták, O., Qian, F., Hendren, L., and Umanee, N. (2003). Points-to Analysis Using BDDs. In *Conference on Programming Language Design and Implementation (PLDI)*.

Bodík, R., Gupta, R., and Sarkar, V. (2000). ABCD: Eliminating Array Bounds Checks on Demand. In *Conference on Programming Language Design and Implementation (PLDI)*.

Bravenboer, M. and Smaragdakis, Y. (2009). Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Briggs, P., Cooper, K. D., Harvey, T. J., Simpson, L. T., et al. (1998). Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software Practice and Experience*.

Cantelon, M., Harter, M., Holowaychuk, T., and Rajlich, N. (2014). *Node.js in Action*.

Cartwright, R. and Fagan, M. (2004). Soft Typing. *Conference on Programming Language Design and Implementation (PLDI)*.

Ceri, S., Gottlob, G., and Tanca, L. (1989). What You Always Wanted to Know About Datalog (and Never Dared to Ask). *Knowledge and Data Engineering*.

Chamberlin, D. D. and Boyce, R. F. (1974). SEQUEL: A Structured English Query Language. In *Workshop on Data Description, Access and Control*.

Chambers, C., Dean, J., and Grove, D. (1996). Frameworks For Intra-and Interprocedural Dataflow Analysis. Technical report, University of Washington.

Chase, D. R., Wegman, M., and Zadeck, F. K. (1990). Analysis of Pointers and Structures. In *Conference on Programming Language Design and Implementation (PLDI)*.

Choi, J.-D., Cytron, R., and Ferrante, J. (1991). Automatic Construction of Sparse Data Flow Evaluation Graphs. In *Symposium on Principles of Programming Languages (POPL)*.

Chow, F., Chan, S., Liu, S.-M., Lo, R., and Streich, M. (1996). Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In *International Conference on Compiler Construction (CC)*.

Christensen, A. S., Møller, A., and Schwartzbach, M. I. (2003). Precise Analysis of String Expressions. In *International Static Analysis Symposium (SAS)*.

Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. (2009). Staged Information Flow for JavaScript. In *Conference on Programming Language Design and Implementation (PLDI)*.

Consel, C. and Danvy, O. (1993). Tutorial Notes on Partial Evaluation. In *Symposium on Principles of Programming Languages (POPL)*.

Cooper, K. D., Harvey, T. J., and Kennedy, K. (2001). A Simple, Fast Dominance Algorithm. *Software Practice and Experience*.

Costantini, G., Ferrara, P., and Cortesi, A. (2011). Static Analysis of String Values. In *Formal Methods and Software Engineering*.

Cousot, P. and Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symposium on Principles of Programming Languages (POPL)*.

Cousot, P. and Cousot, R. (1992). Abstract Interpretation Frameworks. *Journal of Logic and Computation*.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Min, A., Monniaux, D., and Rival, X. (2005). The ASTREE Analyzer. In *Programming Languages and Systems*.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Min, A., Monniaux, D., and Rival, X. (2007). Combination of Abstractions in the ASTREE Static Analyzer. In *Advances in Computer Science*.

Cowan, C., Beattie, S., Johansen, J., and Wagle, P. (2003). Pointguard TM: Protecting Pointers from Buffer Overflow Vulnerabilities. In *USENIX Security Symposium*.

Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. (1998). StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*.

Crockford, D. (2008). *JavaScript: The Good Parts*.

Cuoq, P., Signoles, J., Baudin, P., Bonichon, R., Canet, G., Correnson, L., Monate, B., Prevosto, V., and Puccetti, A. (2009). Experience Report: OCaml for an Industrial-strength Static Analysis Framework. In *International Conference on Functional Programming (ICFP)*.

Curtsinger, C., Livshits, B., Zorn, B., and Seifert, C. (2011). Zozzle: Low-overhead Mostly Static JavaScript Malware Detection. In *USENIX Security Symposium*.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1989). An Efficient Method of Computing Static Single Assignment Form. In *Symposium on Principles of Programming Languages (POPL)*.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *Transactions on Programming Languages and Systems (TOPLAS)*.

Dahl, R. (2014). Node.js online documentation.

de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. (1997). *Computational Geometry: Algorithms and Applications*.

De Moura, L. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TAPAS)*.

De Moura, L. and Bjørner, N. (2011). Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*.

Duesterwald, E., Gupta, R., and Soffa, M. L. (1995). Demand-driven Computation of Interprocedural Data Flow. In *Symposium on Principles of Programming Languages (POPL)*.

ECMA (2015). ECMAScript Language Specification, 3rd edition. ECMA-262.

Fähndrich, M., Foster, J. S., Su, Z., and Aiken, A. (1998). Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Conference on Programming Language Design and Implementation (PLDI)*.

Feldthaus, A., Millstein, T., Møller, A., Schäfer, M., and Tip, F. (2011). Tool-supported Refactoring for JavaScript. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Feldthaus, A. and Møller, A. (2013). Semi-Automatic Rename Refactoring for JavaScript. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Felleisen, M., Findler, R. B., and Flatt, M. (2009). *Semantics Engineering with PLT Redex*.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The Program Dependence Graph and Its Use in Optimization. *Transactions on Programming Languages and Systems (TOPLAS)*.

Fink, S. J. (2014). T.J. Watson Libraries for Analysis. `http://wala.sourceforge.net/`.

Fribourg, L. and Peixoto, M. (1994). Bottom-up Evaluation of Datalog Programs with Arithmetic Constraints. In *International Conference on Automated Deduction (CADE)*.

Gardner, P., Maffeis, S., and Smith, G. D. (2012). Towards A Program Logic for JavaScript. In *Symposium on the Principles of Programming Languages (POPL)*.

Gopan, D. and Reps, T. (2007). Low-level Library Analysis and Summarization. In *Computer Aided Verification (CAV)*.

Grace, M., Zhou, Y., Wang, Z., and Jiang, X. (2012). Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Network and Distributed System Security Symposium (NDSS)*.

Grove, D., DeFouw, G., Dean, J., and Chambers, C. (1997). Call Graph Construction in Object-oriented Languages. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Guarnieri, S. and Livshits, B. (2010). Gulfstream: Staged Static Analysis for Streaming JavaScript Applications. In *USENIX Conference on Web Application Development (WebApps)*.

Guarnieri, S. and Livshits, V. B. (2009). Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*.

Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., and Berg, R. (2011). Saving the World Wide Web from Vulnerable JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA)*.

Guha, A., Fredrikson, M., Livshits, B., and Swamy, N. (2011a). Verified Security for Browser Extensions. In *IEEE Symposium on Security and Privacy*.

Guha, A., Krishnamurthi, S., and Jim, T. (2009). Using Static Analysis for Ajax Intrusion Detection. In *International Conference on World Wide Web (WWW)*.

Guha, A., Saftoiu, C., and Krishnamurthi, S. (2010). The Essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*.

Guha, A., Saftoiu, C., and Krishnamurthi, S. (2011b). Typing Local Control and State Using Flow Analysis. In *European Symposium on Programming (ESOP)*.

Hajiyev, E., Verbaere, M., and de Moor, O. (2006). codeQuest: Scalable Source Code Queries with Datalog. In *European Conference on Object-Oriented Programming (ECOOP)*.

Har-Peled, S. (2011). *Geometric Approximation Algorithms*.

Hardekopf, B. and Lin, C. (2007). The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. In *Conference on Programming Language Design and Implementation (PLDI)*.

Hardekopf, B. and Lin, C. (2009). Semi-sparse Flow-sensitive Pointer Analysis. In *Principles of Programming Languages (POPL)*.

Hardekopf, B. and Lin, C. (2011). Flow-sensitive Pointer Analysis for Millions of Lines of Code. In *International Symposium on Code Generation and Optimization*.

Harper, R. (2012). *Practical Foundations for Programming Languages*.

Harper, R. (2013). What If Anything Is A Declarative Language? `https://existentialtype.wordpress.com/2013/07/18/what-if-anything-is-a-declarative-language/`.

Hind, M. (2001). Pointer Analysis: Haven't We Solved This Problem Yet? In *Workshop on Program Analysis for Software Tools and Engineering*.

Hong, S., Park, Y., and Kim, M. (2014). Detecting Concurrency Errors in Client-Side JavaScript Web Applications. In *International Conference on Software Testing, Verification and Validation (ICST)*.

Horwitz, S., Reps, T., and Sagiv, M. (1995). Demand Interprocedural Dataflow Analysis. In *Symposium on Foundations of Software Engineering (FSE)*.

Huang, S. S., Green, T. J., and Loo, B. T. (2011). Datalog and Emerging Applications: An Interactive Tutorial. In *International Conference on Management of Data*.

Jagannathan, S., Thiemann, P., Weeks, S., and Wright, A. (1998). Single and Loving It: Must-alias Analysis for Higher-order Languages. In *Symposium on Principles of Programming Languages (POPL)*.

Jang, D. and Choe, K.-M. (2009). Points-to Analysis for JavaScript. In *Annual Symposium on Applied Computing (SAC)*.

Jensen, S. H., Jonsson, P. A., and Møller, A. (2012). Remedying the Eval That Men Do. In *International Symposium on Software Testing and Analysis (ISSTA)*.

Jensen, S. H., Madsen, M., and Møller, A. (2011). Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Jensen, S. H., Møller, A., and Thiemann, P. (2009). Type Analysis for JavaScript. In *International Static Analysis Symposium (SAS)*.

Jensen, S. H., Møller, A., and Thiemann, P. (2010). Interprocedural Analysis with Lazy Propagation. In *International Static Analysis Symposium (SAS)*.

Johnson, R. and Pingali, K. (1993). Dependence-based Program Analysis. In *Conference on Programming Language Design and Implementation (PLDI)*.

Jones, N. D., Gomard, C. K., and Sestoft, P. (1993). *Partial Evaluation and Automatic Program Generation*.

Kam, J. B. and Ullman, J. D. (1977). Monotone Data Flow Analysis Frameworks. *Acta Informatica*.

Kashyap, V., Dewey, K., Kuefner, E., Wagner, J., Gibbons, K., Sarracino, J., Wiedermann, B., and Hardekopf, B. (2014). JSAI: A Static Analysis Platform for JavaScript. In *International Symposium on Foundations of Software Engineering (FSE)*.

Kashyap, V., Sarracino, J., Wagner, J., Wiedermann, B., and Hardekopf, B. (2013). Type Refinement for Static Analysis of JavaScript. In *Symposium on Dynamic Languages*.

Kastrinis, G. and Smaragdakis, Y. (2013). Efficient and Effective Handling of Exceptions in Java Points-to Analysis. In *International Conference on Compiler Construction (CC)*.

Khedker, U. P., Mycroft, A., and Rawat, P. S. (2012). Liveness-based Pointer Analysis. In *Static Analysis Symposium (SAS)*.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP)*.

Kiezun, A., Ganesh, V., Guo, P. J., Hooimeijer, P., and Ernst, M. D. (2009). HAMPI: A Solver for String Constraints. In *International Symposium on Software Testing and Analysis (ISSTA)*.

Kildall, G. A. (1973). A Unified Approach to Global Program Optimization. In *Symposium on Principles of Programming Languages (POPL)*.

Knobe, K. and Sarkar, V. (1998). Array SSA Form and Its Use in Parallelization. In *Symposium on Principles of Programming Languages (POPL)*.

Kowshik, S., Dhurjati, D., and Adve, V. (2002). Ensuring Code Safety Without Runtime Checks for Real-time Control Systems. In *International Conference on Compilers, Architecture, and Synthesis (CASES)*.

Kroening, D. and Strichman, O. (2008). *Decision Procedures*.

Kromann-Larsen, R. and Simonsen, R. (2007). Statisk Analyse af JavaScript: Indledende Arbejde. Master's thesis, Department of Computer Science, Aarhus University.

Lam, M. S., Whaley, J., Livshits, V. B., Martin, M. C., Avots, D., Carbin, M., and Unkel, C. (2005). Context-sensitive Program Analysis As Database Queries. In *Symposium on Principles of Database Systems*.

Lhoták, O. (2002). Spark: A Flexible Points-to Analysis Framework for Java.

Lhoták, O. and Chung, K.-C. A. (2011). Points-to Analysis With Efficient Strong Updates. In *Symposium on Principles of Programming Languages (POPL)*.

Lhoták, O. and Hendren, L. (2003). Scaling Java Points-to Analysis Using Spark. In *International Conference on Compiler Construction (CC)*.

Liang, P., Tripp, O., Naik, M., and Sagiv, M. (2010). A Dynamic Evaluation of The Precision of Static Heap Abstractions. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Liu, M. (1998). RelationLog: A Typed Extension to Datalog with Sets and Tuples. *The Journal of Logic Programming*.

Livshits, B. and Lam, M. S. (2005). Finding Security Errors in Java Programs With Static Analysis. In *USENIX Security Symposium*.

Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B.-Y. E., Guyer, S. Z., Khedker, U. P., Møller, A., and Vardoulakis, D. (2015). In Defense of Soundiness: A Manifesto. *Communications of the ACM*.

Livshits, B., Whaley, J., and Lam, M. S. (2005). Reflection Analysis for Java. In *Programming Languages and Systems*.

Logozzo, F. and Fähndrich, M. (2008). Pentagons: A Weakly Relational Abstract Domain for The Efficient Validation of Array Accesses. In *Symposium on Applied Computing*.

Logozzo, F. and Venter, H. (2010). RATA: Rapid Atomic Type Analysis by Abstract Interpretation - Application to JavaScript Optimization. In *International Conference on Compiler Construction (CC)*.

Lua (2015). Lua Specification. http://www.lua.org/manual/.

Madsen, M., , and Møller, A. (2014). Sparse Dataflow Analysis With Pointers and Reachability. In *International Static Analysis Symposium (SAS)*.

Madsen, M. and Andreasen, E. (2014). String Analysis for Dynamic Field Access. In *International Conference on Compiler Construction (CC)*.

Madsen, M., Livshits, B., and Fanning, M. (2013). Practical Static Analysis of JavaScript Applications in The Presence of Frameworks and Libraries. In *European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Maffeis, S., Mitchell, J. C., and Taly, A. (2008). An Operational Semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems (APLAS)*.

Martin, F. (1998). PAG An Efficient Program Analyzer Generator. *International Journal on Software Tools for Technology Transfer*.

Martin, J. C. (1991). *Introduction to Languages and the Theory of Computation*.

Mauborgne, L. and Rival, X. (2005). Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *Programming Languages and Systems*.

Meawad, F., Richards, G., Morandat, F., and Vitek, J. (2012). Eval Begone!: Semi-automated Removal of Eval From JavaScript Programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Might, M. and Shivers, O. (2006). Improving Flow Analyses via cfa: Abstract Garbage Collection and Counting. In *International Conference on Functional Programming (ICFP)*.

Milanova, A., Rountev, A., and Ryder, B. G. (2004). Precise and Efficient Call Graph Construction for Programs with Function Pointers. *Journal of Automated Software Engineering*.

Møller, A. and Schwarz, M. (2014). Automated Detection of Client-State Manipulation Vulnerabilities. *Transactions on Software Engineering and Methodology*.

MongoDB (2015). MongoDB Website. http://www.mongodb.org/.

Mozilla (2015). Ajax. `https://developer.mozilla.org/en/docs/AJAX`.

Mutlu, E., Tasiran, S., and Livshits, B. (2014). I Know It When I See It: Observable Races in JavaScript Applications. In *Workshop on Dynamic Languages and Applications*.

Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of Program Analysis*.

Nielson, F., Nielson, H. R., Sun, H., Buchholtz, M., Hansen, R. R., Pilegaard, H., and Seidl, H. (2004). The Succinct Solver Suite. In *Tools and Algorithms for the Construction and Analysis of Systems (TAPAS)*.

Nielson, H. R. and Nielson, F. (1992). *Semantics with Applications*.

NodeJS (2015). NodeJS Website. `http://www.nodejs.org/`.

Oh, H. (2009). Large Spurious Cycle in Global Static Analyses and Its Algorithmic Mitigation. In *Programming Languages and Systems*.

Oh, H., Heo, K., Lee, W., Lee, W., and Yi, K. (2012). Design and Implementation of Sparse Global Analyses for C-like Languages. In *Conference on Programming Language Design and Implementation (PLDI)*.

One, A. (1996). Smashing The Stack for Fun and Profit. *Phrack Magazine*.

Papadimitriou, C. H. (1985). A Note the Expressive Power of Prolog. *Bulletin of the EATCS*.

Petrov, B., Vechev, M., Sridharan, M., and Dolby, J. (2012). Race Detection for Web Applications. In *Conference on Programming Language Design and Implementation (PLDI)*.

Pierce, B. C. (2002). *Types and Programming Languages*.

Pnueli, M. (1981). Two Approaches to Interprocedural Data Flow Analysis. *Program Flow Analysis: Theory and Applications*.

Qian, F., Hendren, L., and Verbrugge, C. (2002). A Comprehensive Approach to Array Bounds Check Elimination for Java. In *Internal Conference on Compiler Construction (CC)*.

Ramalingam, G. (2002). On Sparse Evaluation Representations. *Theoretical Computer Science*.

Ramsey, N., Dias, J. a., and Peyton Jones, S. (2010). Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation. In *Haskell Symposium*.

Raychev, V., Vechev, M., and Sridharan, M. (2013). Effective Race Detection for Event-driven Programs. In *European Conference on Object-oriented Programming (ECOOP)*.

Reif, J. H. and Lewis, H. R. (1977). Symbolic Evaluation and The Global Value Graph. In *Symposium on Principles of Programming Languages (POPL)*.

Reps, T., Horwitz, S., and Sagiv, M. (1995). Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Symposium on Principles of programming languages (POPL)*.

Richards, G., Hammer, C., Burg, B., and Vitek, J. (2011). The Eval That Men Do – A Large-scale Study of The Use of Eval in JavaScript Applications. In *European Conference on Object-Oriented Programming, (ECOOP)*.

Richards, G., Lebresne, S., Burg, B., and Vitek, J. (2010). An Analysis of The Dynamic Behavior of JavaScript Programs. In *Programming Language Design and Implementation, (PLDI)*.

Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1988). Global Value Numbers and Redundant Computations. In *Symposium on Principles of Programming Languages (POPL)*.

Rountev, A. and Chandra, S. (2000). Off-line Variable Substitution for Scaling Points-to Analysis. In *Conference on Programming Language Design and Implementation (PLDI)*.

Sagiv, M., Reps, T., and Horwitz, S. (1996). Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theoretical Computer Science*.

Sarkar, V. and Knobe, K. (1998). Enabling Sparse Constant Propagation of Array Elements via Array SSA Form. In *Static Analysis Symposium (SAS)*.

Schäfer, M., Sridharan, M., Dolby, J., and Tip, F. (2013a). Dynamic Determinacy Analysis. *Conference on Programming Language Design and Implementation (PLDI)*.

Schäfer, M., Sridharan, M., Dolby, J., and Tip, F. (2013b). Effective Smart Completion for JavaScript. Technical report, IBM Research.

Schildt, H. (2010). *C# 4.0: The Complete Reference*.

Smaragdakis, Y. and Bravenboer, M. (2011). Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded*.

Smaragdakis, Y., Bravenboer, M., and Lhoták, O. (2011). Pick Your Contexts Well: Understanding Object-sensitivity. In *Symposium on Principles of Programming Languages (POPL)*.

Smaragdakis, Y., Kastrinis, G., and Balatsouras, G. (2014). Introspective Analysis: Context-sensitivity, Across The Board. In *Conference on Programming Language Design and Implementation (PLDI)*.

Sridharan, M. and Bodík, R. (2006). Refinement-based Context-sensitive Points-to Analysis for Java. In *Conference on Programming Language Design and Implementation (PLDI)*.

Sridharan, M., Dolby, J., Chandra, S., Schäfer, M., and Tip, F. (2012). Correlation Tracking for Points-to Analysis of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*.

Staiger-Stöhr, S. (2013). Practical Integrated Analysis of Pointers, Dataflow and Control Flow. *Transactions on Programming Languages and Systems (TOPLAS)*.

Steensgaard, B. (1996). Points-to Analysis in Almost Linear Time. In *Symposium on Principles of Programming Languages (POPL)*.

Thiemann, P. (2005a). A Type Safe DOM API. In *Workshop on Database Programming Languages*.

Thiemann, P. (2005b). Towards a Type System for Analyzing JavaScript Programs. In *European Symposium on Programming (ESOP)*.

Tiuryn, J. (1990). Type Inference Problems: A Survey. In *Mathematical Foundations of Computer Science*.

Tizen (2015). Tizen Website. http://www.tizen.org/.

Tok, T. B., Guyer, S. Z., and Lin, C. (2006). Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in The Presence of Pointers. In *International Conference on Compiler Construction (CC)*.

Tripp, O., Pistoia, M., Cousot, P., Cousot, R., and Guarnieri, S. (2013). Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *Conference on Fundamental Approaches to Software Engineering (FASE)*.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (1999a). Soot – A Java Bytecode Optimization Framework. In *Conference of the Centre for Advanced Studies on Collaborative Research*.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (1999b). Soot - A Java Bytecode Optimization Framework. In *Conference of the Centre for Advanced Studies on Collaborative Research*.

W3C (2004). Document Object Model (DOM) Level 3 Core Specification. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/.

W3C (2012). The WebSocket API. http://www.w3.org/TR/websockets/.

Wegman, M. N. and Zadeck, F. K. (1991). Constant Propagation with Conditional Branches. *Transactions on Programming Languages and Systems (TOPLAS)*.

Weisstein, E. W. (2015). Peano Arithmetic. http://mathworld.wolfram.com/PeanoArithmetic.html.

Wells, J. B. (1999). Typability and Type Checking in System F are Equivalent and Undecidable. *Annals of Pure and Applied Logic*.

Whaley, J. (2003). Joeq: A Virtual Machine and Compiler Infrastructure. In *Workshop on Interpreters, Virtual Machines and Emulators*.

Whaley, J., Avots, D., Carbin, M., and Lam, M. S. (2005). Using Datalog and Binary Decision Diagrams for Program Analysis. In *Asian Symposium on Programming Languages and Systems (APLAS)*.

Whaley, J. and Lam, M. S. (2004). Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Conference on Programming Language Design and Implementation (PLDI)*.

Winther, J. (2011). Guarded Type Promotion: Eliminating Redundant Casts in Java. In *Workshop on Formal Techniques for Java-Like Programs*.

Würthinger, T., Wimmer, C., and Mössenböck, H. (2007). Array Bounds Check Elimination for The Java HotSpot Client Compiler. In *Symposium on Principles and Practice of Programming in Java*.

Xie, Y. and Aiken, A. (2005). Scalable Error Detection Using Boolean Satisfiability. In *Symposium on Principles of Programming Languages (POPL)*.

YouTube (2015). Lawnmower Dreams - A Lawnmower Learns To Dream Big. A Lesson In Life We Can All Learn.
https://www.youtube.com/watch?v=9Yrt9qkBQ2Q.

Zheng, Y., Bao, T., and Zhang, X. (2011). Statically Locating Web Application Bugs Caused by Asynchronous Calls. In *International Conference on World Wide Web (WWW)*.

Zheng, Y., Zhang, X., and Ganesh, V. (2013). Z3-str: A Z3-based String Solver for Web Application Analysis. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.