Verifying Information Flow and Metaprogramming in Dynamically Typed Languages



Martin Mariusz Lester

Merton College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Hilary 2015

Acknowledgements

While this thesis is my own work, it is also a joint effort in the sense that it was produced with the support of many people.

On the academic side, I am grateful to my supervisor Luke Ong for his guidance throughout the project and for his helpful suggestions about where best to focus my efforts. It was also at his suggestion that I began to investigate information flow for JavaScript. I must also acknowledge Max Schäfer, not only for his contributions as a co-author (which are described elsewhere), but also for his energy and enthusiasm during the early stages of my research.

I am also indebted to the inhabitants of office 347 for the sense of camaraderie we developed. Of particular note are Robin, Steven and Chris, who introduced me to the ICFP Programming Contest, which has provided a number of fruitful diversions over the years. Also deserving of a mention is Markus, who inspired us with his sardonic wit and philosophical observations about academia, the United Kingdom and life in general.

Finally, I must thank my parents and my wife Cressida for their practical and emotional support, particularly towards the end of the process.

Abstract

Title: Verifying Information Flow and Metaprogramming in Dynamically Typed Languages
Author: Martin Mariusz Lester, Merton College
Submitted for: Doctor of Philosophy in Computer Science, Hilary 2015

The ubiquity of JavaScript in Web applications means that its analysis has become an important security problem. This thesis develops techniques for analysing information flow in JavaScript programs and verifying the absence of undesirable flows (for example, of sensitive data to untrusted third parties).

JavaScript presents a unique combination of challenges not usually addressed by information flow analyses: its semantics are quaint and poorly specified, making formal reasoning largely infeasible; it is dynamically typed, which precludes the use of the most advanced existing analyses; and its **eval** construct, which executes a string as program code, allows arbitrary behaviour to result from data manipulated within the program.

The thesis focuses on the last of these problems. It considers an idealised subset of JavaScript and augments it with *staged metaprogramming*, a formalism that captures the construction, execution and manipulation of code templates. The resulting language is called SLamJS. An information flow analysis for SLamJS is developed and its correctness is proved. This builds on the existing analysis 0CFA, but adds support for metaprogramming (paying particular attention to the difficult behaviour of variable capture) and information flow.

In order to demonstrate the applicability of this analysis to JavaScript, where **eval**-using programs operate on code strings formed through concatenation, rather than splicing of templates, an algorithm to transform an **eval**-using program into one that uses staged metaprogramming is also developed. The transformation repurposes an existing lexer/-parser (for example, one produced by the popular tools `lex` and `yacc`) to parse fragments of program code out-of-order.

The transformed program is then a suitable target for the analysis. This multi-step approach is vital to allow subdivision of the complex overall problem into smaller, more manageable parts.

Contents

1 Introduction **1**

2 Background **5**

2.1 Overview 5

2.2 Automated Verification 6

2.2.1 The Limitations of Software Testing 6

2.2.2 The Limitations of Social or Infrastructural Trust 6

2.2.3 The Difficulties of Automated Software Verification 7

2.3 Information Flow 9

2.3.1 Formalisations of Information Security 9

2.3.2 Explicit and Implicit Flows 10

2.3.3 Termination and Covert Channels 10

2.3.4 Quantitative Information Flow 11

2.3.5 Classical Noninterference 11

2.3.6 Observational Nondeterminism 16

2.4 JavaScript and other Dynamic Languages 16

2.4.1 Versions 17

2.4.2 Features 17

2.4.3 Semantics 19

2.4.4 Security Problems in JavaScript 20

2.5 Metaprogramming 22

2.5.1 Styles of Metaprogramming 22

2.5.2 A Metaprogramming Example 23

2.5.3 Abuses of Metaprogramming 25

2.5.4 Semantics of Metaprogramming 26

3 Previous Work **31**
3.1 Overview . . . 31
3.2 Verifying or Enforcing Information Flow Properties . . . 31
3.2.1 Instrumentation and Monitoring . . . 31
3.2.2 Isolation via Safe Subsets . . . 36
3.2.3 Type Systems . . . 37
3.2.4 Temporal Logic Model-Checking . . . 41
3.2.5 Hybrid Approaches . . . 42
3.3 Verifying JavaScript . . . 43
3.3.1 Dynamic Type Systems . . . 43
3.3.2 Control Flow Analysis . . . 44
3.4 Verifying Metaprogramming . . . 45
3.4.1 Verifying Staged Metaprogramming . . . 45
3.4.2 Verifying Eval . . . 46
3.4.3 String Analysis of Code . . . 46

4 Outline of Research **49**
4.1 Objectives . . . 49
4.2 Staged Metaprogramming . . . 51
4.2.1 Constructs of Staged Metaprogramming . . . 51
4.2.2 Expressivity of Metaprogramming . . . 51
4.3 The Language SLamJS . . . 54
4.3.1 Syntax . . . 54
4.3.2 Semantics . . . 55
4.3.3 Eval in SLamJS . . . 58

5 Verifying Information Flow with Staged Metaprogramming **61**
5.1 Overview . . . 61
5.2 Augmented Semantics of SLamJS . . . 62
5.3 Erasure and Stability . . . 64
5.4 0CFA for SLamJS . . . 65
5.4.1 Simple Analysis . . . 66
5.4.2 Improved Analysis . . . 70
5.5 Information Flow for SLamJS . . . 73
5.6 Information Flow Soundness . . . 76
5.7 Implementation and Examples . . . 78

6 The Boxing Algorithm: Transforming Eval to Staged Metaprogramming 83
6.1 Overview . . . 83
6.2 Key Ideas . . . 84
6.2.1 Prerequisites . . . 84
6.2.2 Building a Sequence of Program Approximations . . . 86
6.2.3 Parsing Expressions Out-Of-Order . . . 87
6.2.4 Constraint Solution and Resolution . . . 90
6.2.5 Combining Concatenations . . . 91
6.3 Cycle Description . . . 91
6.3.1 Termination . . . 95
6.3.2 Complexity . . . 95
6.3.3 Interaction of Concerns . . . 96
6.4 Worked Examples . . . 97
6.4.1 Handling String Concatenation . . . 97
6.4.2 Handling Nested Evals . . . 102
6.5 Soundness . . . 106
6.6 Implementation and Examples . . . 110

7 Conclusions 113
7.1 Summary of Results . . . 113
7.2 Comparison with Previous Work . . . 113
7.2.1 Dynamic Analyses . . . 113
7.2.2 Static Analysis of Staged Metaprogramming . . . 114
7.2.3 Analysing Eval . . . 115
7.3 Future Work . . . 115
7.3.1 Improving Precision . . . 116
7.3.2 Handling More Features . . . 116
7.3.3 Infrastructural Issues . . . 117
7.3.4 Investigating the Boxing Algorithm . . . 118
7.4 Conclusion . . . 119

A Coq Formalisations of Proofs 121
A.1 Library Functions — lib.v . . . 121
A.2 Language Definition — lambdajs.v . . . 123
A.3 Simple 0CFA and Information Flow Analysis for SLamJS — simple_cfa.v . 129
A.4 Improved 0CFA and Information Flow Analysis for SLamJS — cfa.v 136

Bibliography **151**

List of Figures

2.1 Noninterference for a simple imperative language 13
2.2 Example programs in a simple imperative language 14
2.3 Evaluation of programs satisfying and violating classical noninterference . 14
2.4 Leaking information with concurrency 15
2.5 An example of an information leak in JavaScript 22

3.1 A graphical comparison of areas covered by previous work 32
3.2 A type system for statically verifying classical noninterference 38

4.1 Syntax of SLamJS 54
4.2 Evaluation contexts 56
4.3 Environment propagation rules 56
4.4 Proper reduction rules 57
4.5 Adding **eval** to SLamJS 58

5.1 Semantic rules for lifts 62
5.2 Definition of $\lfloor e \rfloor_M$, the M-erasure of e 64
5.3 Abstract domains (simple analysis) 66
5.4 Rules for the 0CFA acceptability judgement (simple analysis) 67
5.5 The approximation judgement $\Gamma, \varrho \models v \approx t$ 68
5.6 The abstract value operation $\lceil k \rceil$ 68
5.7 Abstract domains (improved analysis) 70
5.8 Operations on abstract contexts 70
5.9 Rules for the 0CFA acceptability judgement (improved analysis) 71
5.10 Rules for generating information flow constraints 74
5.11 Information flow constraints for Example 5 76

6.1 Control flow in the Boxing Algorithm 84
6.2 Data flow in the Boxing Algorithm 85
6.3 Conditions on B, U, X and S in a cycle of the Boxing Algorithm 93

6.4 Boxing Algorithm concatenation example — cycle 1 input 98
6.5 Boxing Algorithm concatenation example — cycle 1 output 99
6.6 Boxing Algorithm concatenation example — cycle 2 input 100
6.7 Boxing Algorithm nested example — cycle 1 input 103
6.8 Boxing Algorithm nested example — cycle 1 output 103
6.9 Boxing Algorithm nested example — cycle 2 input 104
6.10 Boxing Algorithm nested example — cycle 2 output 105
6.11 Boxing Algorithm nested example — cycle 3 input 106

Chapter 1

Introduction

The scripting language JavaScript is ubiquitous in Web applications. JavaScript programs routinely handle sensitive or personal data, such as passwords, contact details and credit card numbers. We rely on them not only to function correctly, but also to be trustworthy: we do not want programs to share our information without our knowledge. Unfortunately, there are many obstacles to achieving this goal.

As this is a problem in information security, it is inherently adversarial. While some information leaks are accidental, malicious programmers will write and distribute software that deliberately leaks confidential data and does so in an obscure manner that hides its true purpose. Consequently, we cannot rely upon external assurances about the behaviour of a program; nor can we form a reliable judgement about the behaviour of a program from running it on a few test cases. Furthermore, the rate at which new JavaScript programs are created (often partially mechanically by Web application frameworks) [RLBV10] means that we cannot realistically submit programs to a trusted human expert for analysis.

What we need are automated methods to verify (with a degree of certainty similar to a mathematical proof) that a program cannot, under any circumstances, leak confidential information; I call this verifying information flow. Such methods are the focus of this thesis.

Automated verification of JavaScript programs often seems adversarial in another sense: the language and the programs written in it sometimes seem to be deliberately designed to make formal reasoning as difficult as possible. In particular: the language has a confusing natural-language specification [Int99], but no official formal semantics; its behaviour is full of difficult corner cases that sometimes differ between implementations [MMT08]; it is dynamically typed, which precludes the use of many existing powerful analysis techniques; code is often written by low-skilled programmers, who abuse language features out of ignorance.

Among the difficult features of JavaScript is the **eval** construct, which takes a string and executes it as if it was program code. As this allows essentially arbitrary behaviour that is dependent on the value of a string, its analysis might at first seem to be utterly hopeless. Indeed, many JavaScript analyses ignore **eval** on the basis that it is rarely used.

However, a study shows this claim to be untrue [RHBV11]: while most coding guidelines for JavaScript recommend against using **eval**, it is nonetheless widely used by a majority of popular websites, and it is used in non-trivial ways. For example, ignorant programmers sometimes use **eval** and string concatenation as a way of implementing higher-order functions, even though JavaScript supports them directly.

Returning to the observation that information flow verification is a security problem and hence adversarial, we cannot reasonably proscribe the use of certain features in programs we wish to analyse. For this reason, and because it has previously received little attention, I focus on handling **eval**. Handling all the intricacies of JavaScript, even without **eval**, would be an enormous undertaking, so I develop my analyses in the context of an idealised JavaScript-like language called SLamJS.

The **eval** construct is a form of metaprogramming, in the sense that it allows the construction, manipulation and execution of program code as if it is an ordinary data value (in this case, encoded as a string). It is the unstructured nature of that data value that makes **eval** particularly troublesome. In contrast, the formalism of staged metaprogramming [KYC06], which captures the creation, splicing together and execution of code templates, has similar expressive power, but is more amenable to analysis. SLamJS is based around adding these *Staged* constructs to a cut-down version of *Lambda JS* (λ_{JS}), an existing core calculus for JavaScript [GSK10].

The main results of this thesis are: an algorithm for transforming a program using string-based **eval** into an equivalent SLamJS program that uses staged metaprogramming; and an automated static analysis for information flow in SLamJS. There is a mechanised soundness proof (formalised using the theorem prover Coq) for the information flow analysis.

The structure of the remainder of this thesis is as follows. In Chapter 2, I review a range of relevant background material. I begin with a discussion of some of the motivations for and key concepts in automated program verification; this is followed by a look at the various formalisations of information flow and hence information flow security. The chapter concludes with a discussion of some of the difficulties with verifying JavaScript programs, paying special attention to the difficulties inherent in dealing with **eval** and similar metaprogramming constructs in this and other languages. Having identified the areas of interest, in Chapter 3, I survey previous work on verification in each of these areas.

Next, in Chapter 4, I outline the scope of my research and introduce the language SLamJS; I discuss to what extent its metaprogramming constructs models JavaScript's **eval** and hence how its analysis is applicable to that of JavaScript. In Chapter 5, I describe the information flow analysis for SLamJS, followed in Chapter 6 by the transformation from **eval** into staged metaprogramming. (The analysis is described before the transformation because the latter makes use of the former.) Finally, I conclude with an evaluation of the efficacy of both procedures, a summary of my results and some possible future research directions in Chapter 7.

The information flow analysis for SLamJS was first published at CSF 2013 [LOS13] and is joint work with Luke Ong and Max Schäfer. A high-level discussion of my motivation and approach, including the first work on how to transform **eval** to staged metaprogramming, was first published at PLAS 2013 [Les13]. A revised version of the CSF 2013 paper, including a more complete formulation of that transformation, has been accepted for inclusion in a special edition of JCS, which collects the best papers from CSF.

Supporting material, which includes mechanisations of the key results in the theorem prover Coq and an implementation of the analysis and transformation in OCaml, is available online [Les15].

Chapter 2

Background

2.1 Overview

Computer systems routinely handle information with varying levels of *confidentiality*. How can we be sure that one part of a system will not leak secret information to other parts of the system, or to other systems, while still allowing the system to process and handle this information usefully? This is the problem that motivates the study of *information flow security*. The language-based approach to information flow security focuses on *verifying* by analysis of the program source code that programs do not leak confidential information. I give a brief introduction to automated verification in Section 2.2.

An important consideration here is exactly what it means for a program to leak information. Most formalisations of this notion are based around the following intuition: suppose the *input and output channels* of a program can be partitioned into low-confidentiality and high-confidentiality channels; then the low-confidentiality outputs of the program should depend only on the low-confidentiality inputs. In Section 2.3 I discuss this idea and some variations on it.

An *information flow analysis* determines which values in a program can influence which parts of the result of the program. Using an information flow analysis, we can, for instance, prove that program inputs that are deemed high security do not influence low security outputs.

Early work on information flow focused mainly on applications in a military or government setting, where there might be strict rules about security clearance and classification of documents. More recently, there has been increased interest in information security (and hence its analysis) for Web applications.

JavaScript (a variant of ECMAScript) is a language used to embed programs within webpages that can be viewed as essentially functional in character. Following the Web 2.0 movement in web design, there has been a massive increase in the quantity of JavaScript

used in popular websites. Often, programs from different, possibly untrusted authors are combined into a single webpage. It would be useful to verify that these programs do not leak the potentially confidential information entered by users. I conclude the chapter with an overview of JavaScript and its features in Section 2.4, giving special attention to its **eval** construct and other forms of metaprogramming in Section 2.5.

2.2 Automated Verification

Automated verification is a process that allows us to develop trust in a program's behaviour. To see why I focus on this, rather than some other process, let us first consider the limitations of other methods by which we might come to trust a program. We shall then look at a few of the general difficulties faced by any automated verification technique and the methods used to overcome them.

2.2.1 The Limitations of Software Testing

It is widely accepted within the software engineering community (and even to some degree by the general public) that testing is a vital part of software development. We can often expose bugs in a program by running it in situations similar to those in which it is intended to be used. Indeed, this is probably the most widely-used and successful technique for ensuring the quality of software.

Unfortunately, it has a fundamental limitation: a program may be used in an effectively infinite range of situations (subject to limitations on computer memory). For example, in a password-checking program, every different possible password entered corresponds to a different run of the program. We can only ever hope to check finitely many such runs, so an infinite range of the behaviours of the program remain untested.

In a physical system, such as a railway bridge, this may not be a problem; physical systems tend to vary in a continuous fashion, so it may be reasonable to assume that a finite sample of use cases within the specified range of operating conditions is sufficient to expose any faults. However, this assumption does not hold in discrete systems such as computer programs.

2.2.2 The Limitations of Social or Infrastructural Trust

Another reason to trust a program is that someone, whose judgement we trust, says that it should be trusted. This approach is common on the Web, where websites and applets use cryptographic certificates to demonstrate, for example: that a program was written (or at

least distributed) by a named person or organisation; that the program's code has not been tampered with; and that the distributing entity is itself trusted by some other (perhaps more intrinsically trustworthy) organisation.

There are some practical weaknesses to this approach. For example, website certificates are often incorrectly configured, so users develop a habit of ignoring security warnings, even if they are genuine.

However, the fundamental problem here is that the process still needs to be bootstrapped somehow. We may think it is fine to trust a program because it is endorsed by a reputable company, but what basis does the company have for making that claim? If the company did indeed write the software, then perhaps its authors can be confident that it is not maliciously harmful. If they have tested it, perhaps they can have a high degree of confidence that it is not unintentionally harmful, but that does not preclude the possibility. Furthermore, if the reputable company outsourced some of the development to a disreputable company, it is possible that the program contains malicious code disguised as benign code.

In summary, while certification may be part of the infrastructure that allows us to distribute programs safely, it cannot be the basis of our trust of a program.

2.2.3 The Difficulties of Automated Software Verification

What we would like to do is produce a mathematical proof or other formal argument to demonstrate that a program is free from errors or malicious behaviours in all possible situations. As computer programs are complex systems and humans are liable to make errors in their reasoning, we would also like this proof to be produced automatically by a computer.

The Halting Problem. Unfortunately, this is impossible in the general case. One of the fundamental results in theoretical computer science, namely Alan Turing's Halting Theorem, tells us that it is impossible to produce an automated, general-purpose computer program that, given another program, will tell us whether that program ever finishes. More generally, Rice's Theorem tells us that it is impossible to produce an automated, general-purpose computer program that, given another program, will tell us whether that program has a particular interesting property. An example of such a property might be "this program has no malicious behaviours" or "this program cannot leak sensitive data". However, while it is impossible in the general case, it is possible in many specific cases: we can write automated computer programs that will handle large classes of programs and a wide range of interesting properties. This is the problem considered by the automated software verification community.

Finite abstraction of infinite systems. The key theoretical problem is that, as discussed when considering the limitations of testing, computer programs may have infinitely many distinct runs, but we can only ever directly consider finitely many of them. However, it is usually the case that many of these runs will be very similar. We can often partition the runs of a program into finitely many distinct classes (some containing infinitely many runs). Then, if the same argument applies to every run in one class, we can reason finitely about all the runs, even though there are infinitely many.

For example, we can view all the states of a computer program as nodes in an infinite directed graph G_C, with edges indicating transitions between states. A distinguished start state q_0 indicates the initial state of the program. A path in the graph from this start state then corresponds to a run of the program. A certain set of states E_C might correspond to errors (for example, dividing by zero). We might wish to verify that the program can never encounter one of these error states; this corresponds to checking that there is no path $q_0 \rightarrow^* q_e$ for any $q_e \in E_C$. We can find out what states are immediately accessible from any state, as this corresponds to simulating a single step of execution of the program, but as G_C is infinite, we cannot check the property directly. However, it may be possible to construct an *abstraction* consisting of a finite graph G_A with a graph homomorphism $\alpha : V(G_C) \rightarrow V(G_A)$ and to compute E_A, the image of E_C under α. Then, if there is no path $\alpha(q_0) \rightarrow^* q_{e'}$ for any $q_{e'} \in E_A$, the desired property holds in G_C.

The difficult part is choosing the correct α. For example, for an imperative program, we could choose $V(G_A)$ to be the line numbers of the program and $\alpha(q)$ to be the currently executing line of the program in state q. However, this would abstract away any information about the content of variables in the program. Consequently, we could probably only verify an absence of division by zero errors if the program never did any division at all! In order to be useful, the abstraction that relates the infinite program to its finite model must be sufficiently precise or "fine".

The State Explosion Problem. There is also an important practical problem to address in automated verification: the State Explosion Problem. The range of possible states a program can occupy grows exponentially with the amount of data it stores. If a program stores just a few hundred bits of data, it could easily have more states than there are atoms in the universe. Clearly, even if we could store one program state in computer memory using a single atom, there is no way we can ever explicitly represent all the states of even a small program with finite storage usage. Consequently, it is important not only that we can reason about a program's infinite behaviours with a finite model, but also that it should

be a small model. In order to be useful, the abstraction must be sufficiently imprecise or "coarse".

The practicality of automated verification. Historically, verification of computer programs has been time-consuming and expensive, as it has needed a high degree of human involvement and high-powered computers. Consequently, it has been seen as the preserve of aviation, space flight and control systems, where the high cost of failure justified the expense. However, advances in the last twenty years in both the theoretical tools and the underlying computer hardware mean that it is now becoming practical for automated verification to be used on applications software.

There is a fundamental assumption underlying this line of argument, namely that programs and their behaviours are mathematical objects and hence can be reasoned about mathematically. Researchers in the field of programming language semantics attempt to develop mathematical models of computer programs and their behaviour. When we say that we will construct a mathematical proof about the behaviour of a program, what we are really doing is constructing a proof about the corresponding mathematical model.

We might reasonably question whether the model is entirely faithful to the original program. In order to use this technique, we have to assume that the semantic model is an accurate representation of the programming language, which might otherwise only be described in a specification written using natural language or by a reference implementation written in a poorly-specified programming language. We must also make assumptions about the correctness of the compiler, the operating system, the underlying computer hardware and so on.

This may seem like a lot of unfounded assumptions; indeed, researchers in some areas (such as certified compilers) seek to minimise the assumptions necessary. Yet it is reasonable to make such assumptions, provided we are aware of them: in practice, it is preferable to be able to make a strong statement about the correctness of a program under certain reasonable assumptions, than to be unable to say much at all.

2.3 Information Flow

2.3.1 Formalisations of Information Security

When trying to formalise information security, we usually assume that data has an associated confidentiality level. There may be multiple such levels with an ordering between them; this is historically known as *multi-level security* (MLS), although the idea is now so widely accepted that it is rarely referenced explicitly.

For brevity, I refer to high-confidentiality channels or data as simply high channels or high data; likewise for low-confidentiality channels or data. This does not mean that there are only two levels of confidentiality, but rather that the key distinction between levels is whether one is higher (more confidential) than the other. As the aspect of security I consider is confidentiality, I use "confidentiality" and "security" somewhat interchangeably.

Most formalisations of information security properties are based around the intuition that the low output channels of a program should depend only on the low input channels; Goguen and Meseguer introduced this idea, known as *noninterference* [GM82], as a way of specifying security policies, including enforcement of information flow security. Noninterference and information flow security became almost synonymous, although Pottier and Conchon are careful to emphasise the distinction between the two [PC00]. However, there are some subtleties here are in what it means for an output to depend on an input and in what counts as a channel.

2.3.2 Explicit and Implicit Flows

If data from a high input is sent without modification to a low output, then there is an *explicit flow* of information from the high level to the low level. Similarly, if data from a high input is processed somehow and then sent to a low output, then there is the risk of an explicit flow. It is possible that the processing is such that there is no actual leak. For example, if a high integer input is multiplied by 0, the result gives no information about the input, so it could be output safely to a low channel. However, such situations are generally difficult to reason about mechanically.

More interestingly, if a program branches on the value of a high input, then within that branch writes to a low output, there is an *implicit flow* of information from high to low: some information about the value of the high input can be inferred from the presence or absence of the low output. This distinction between direct (explicit) flows and indirect (implicit) flows was first considered by Denning [Den76].

2.3.3 Termination and Covert Channels

With regard to what counts as an input or output channel, in the context of imperative programs, variables that are visible within the program and persist outside it can be viewed as channels. Whether the program terminates or not can also be viewed as a channel in some sense. Many notions of information security ignore this because termination analysis is hard and the channel can only reveal a single bit on each execution of the program. However, caution is needed: a potentially non-terminating program might be run multiple

times or run in parallel with another program that can observe where it diverges. There may also be various *covert channels*, such as the memory or processor usage of a program; these are difficult to reason about and are often of low bandwidth or difficult for an attacker to monitor, so they are usually ignored.

2.3.4 Quantitative Information Flow

To complicate matters further, it is sometimes desirable for a limited amount of confidential information to be revealed for a program to be useful: a password-checking program should depend on users' passwords, which are confidential, but it should not reveal them; a system utility should be able to reveal how much free disk space is available, but this may depend on the size of confidential files. These are problems of *declassification* (revealing confidential information) and *aggregation* (determining how much confidential information is revealed by the result of a computation) [SM03b]. They may be handled to some extent by *quantitative* treatments of information flow, but for simplicity I do not consider them further.

2.3.5 Classical Noninterference

I now outline formally what noninterference means for a simple language. Consider a simple imperative language with global store; the syntax of commands and expressions is given in Figure 2.1. Variables $v \in V$ in this language range over the integers $\mathbb{Z}$; thus a global store $s \in S$ can be encoded as a function $s : V \to \mathbb{Z}$.

Expressions e are built from variables, constants and the usual arithmetic and comparison operators; a false comparison evaluates to 0 and a true comparison evaluates to 1. Expressions have no side effects. In the condition of an **if** or **while** statement, an expression is false if it evaluates to 0 and true otherwise.

Within these constraints, the semantics of the language is standard. The semantics of a program c is denoted by the function $[\![c]\!]$, which maps the initial store s to the store s' at termination or (in the case of non-termination) $\perp$.

In general, following the example of Denning [Den76], one might consider a partial order or lattice $(L, \sqsubseteq, \sqcup, \sqcap)$ of *security labels*, with each label representing a different level of confidentiality. However, for many examples it is more instructive to consider the two-element lattice with $L = \{\mathsf{l}, \mathsf{h}\}$ and $\mathsf{l} \sqsubseteq \mathsf{h}$; l and h represent low and high levels of confidentiality respectively.

Each variable in the store has an associated confidentiality level, given by a security label. A *permissions* function $P : V \to L$ maps each variable to its associated label. The

idea is that a variable cannot store data of confidentiality below its label: a high variable can store a low value, but a low variable cannot store a high value.

The *low-view operator* $\downarrow_l^P$ removes from a store all variables that are invisible at level l under permissions P. Two stores s_1 and s_2 are *equal at level* l, written $s_1 =_l^P s_2$, if they match at all locations visible at level l.

In this setting, the values in the store at the start of program execution are the inputs and the values in the store at the end of program execution are the outputs. An input or output is high/low if it is stored in a variable that is high/low. Thus a program satisfies noninterference if, when run on initial stores equal at any level l, the resulting stores are *similar* at level l.

The similarity determines the form of noninterference: a program satisfies *termination-sensitive noninterference* if the two final stores are equal at level l or the program fails to terminate under both stores; for a program to satisfy *termination-insensitive noninterference*, it is also permissible for only one of the stores to lead to non-termination. Thus any program that satisfies the former also satisfies the latter.

To illustrate some of the subtleties present even in this most simple of settings, some example programs are shown in Figure 2.2. These examples are intended to be considered under the permissions mapping $\{\mathtt{l} \mapsto \mathsf{l}, \mathtt{h} \mapsto \mathsf{h}\}$. Figure 2.3 shows the results of evaluating them with 4 different initial stores, arranged as 2 pairs with equal low views; it also shows whether the programs satisfy termination-insensitive noninterference (TI), termination-sensitive noninterference (TS) or neither.

Programs 1 and 2 show an *explicit flow* of information from h to l and from l to h respectively; thus only 2 satisfies noninterference. Program 3 has an explicit flow (from h to l), but it occurs only when $\mathtt{l} = 0$; this shows that it is necessary to consider all possible executions of a program to verify information flow security. Program 4 has no explicit flow, as there are no assignments of expressions using h; yet it still leaks information via an *implicit flow*, as there is an assignment to l within a branch on h. Program 5 demonstrates the difference between termination-sensitive and termination-insensitive noninterference: it does not leak via a variable, as there are no assignments, but whether the program terminates or not depends on h. Program 6 satisfies termination-sensitive noninterference, as it can never terminate. But it is just an extension of program 4, which has an implicit flow; hence *noninterference is not a safety property* [TA05]. In a similar vein, program 7 also satisfies termination-sensitive noninterference, despite being an extension of program 4; however, unlike program 6, program 7 always terminates.

Syntax of Commands and Expressions

$$\begin{array}{lcl} e & ::= & n \mid v \mid e_1 \oplus e_2 \\ c & ::= & \textbf{skip} \mid v := e \mid c_1; c_2 \mid \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } e \textbf{ do } c \end{array}$$

Semantics

$v\ (\in V)$	a variable
$s : V \to \mathbb{Z}\ (\in S)$	a store
$s^?\ (\in S \uplus \{\bot\})$	a store or nontermination
$[\![c]\!] : S \to S \uplus \{\bot\}$	denotation of command c

Security Labels

$(L, \sqsubseteq, \sqcup, \sqcap)$	a lattice of security labels
$\mathsf{l} \sqsubseteq \mathsf{h}$	the simplest interesting such order
$P : V \to L$	a permissions mapping on variables

Low Views

Low view of a store

$$s \downarrow_l^P (v) \stackrel{def}{=} \begin{cases} s(v) & \text{if } P(v) \sqsubseteq l \\ \text{undefined} & \text{otherwise} \end{cases}$$

Equality of low views

$$s_1 =_l^P s_2 \stackrel{\text{def}}{\Leftrightarrow} s_1 \downarrow_l^P = s_2 \downarrow_l^P$$

Noninterference

Noninterference for commands

$$\forall s_1, s_2 \in S. \forall l \in L. s_1 =_l s_2 \Rightarrow [\![c]\!] s_1 \approx_l^P [\![c]\!] s_2$$

Termination-sensitive noninterference

$$s_1^? \approx_l^P s_2^? \stackrel{def}{=} \begin{cases} s_1^? =_l^P s_2^? & \text{if } s_1^?, s_2^? \in S \\ s_1^? = \bot \wedge s_2^? = \bot & \text{otherwise} \end{cases}$$

Termination-insensitive noninterference

$$s_1^? \approx_l^P s_2^? \stackrel{def}{=} \begin{cases} s_1^? =_l^P s_2^? & \text{if } s_1^?, s_2^? \in S \\ \mathit{true} & \text{otherwise} \end{cases}$$

Figure 2.1: Noninterference for a simple imperative language

Example Programs

1	$\mathtt{l} := \mathtt{h}$
2	$\mathtt{h} := \mathtt{l}$
3	**if** $\mathtt{l}$ **then** $\mathtt{l} := \mathtt{h}$ **else skip**
4	**if** $\mathtt{h}$ **then** $\mathtt{l} := 1$ **else** $\mathtt{l} := 0$
5	**if** $\mathtt{h}$ **then skip else** (**while** 1 **do skip**)
6	**if** $\mathtt{h}$ **then** ($\mathtt{l} := 1$; **while** 1 **do skip**) **else** ($\mathtt{l} := 0$; **while** 1 **do skip**)
7	**if** $\mathtt{h}$ **then** ($\mathtt{l} := 1$; $\mathtt{l} := 0$) **else** $\mathtt{l} := 0$

Figure 2.2: Example programs in a simple imperative language

Evaluation of Examples

Program c	$(s'(\mathtt{l}), s'(\mathtt{h}))$ for $s' = [\![c]\!]s$ with: $(s(\mathtt{l}), s(\mathtt{h}))$						security property
	$(0,0)$	$\approx$	$(0,1)$	$(1,0)$	$\approx$	$(1,1)$	
1	$(0,0)$		$(1,1)$	$(0,0)$		$(1,1)$	neither
2	$(0,0)$	$\approx$	$(0,0)$	$(1,1)$	$\approx$	$(1,1)$	TS
3	$(0,0)$	$\approx$	$(0,1)$	$(0,0)$		$(1,1)$	neither
4	$(0,0)$		$(1,1)$	$(0,0)$		$(1,1)$	neither
5	$\bot$	$\sim$	$(0,1)$	$\bot$	$\sim$	$(1,1)$	TI
6	$\bot$	$\approx$	$\bot$	$\bot$	$\approx$	$\bot$	TS
7	$(0,0)$	$\approx$	$(0,1)$	$(0,0)$	$\approx$	$(0,1)$	TS

Figure 2.3: Evaluation of programs satisfying and violating classical noninterference

Variations of Observational Determinism

(|| denotes parallel composition of programs.)

$$\mathtt{l} := 0 \parallel \mathtt{l} := 1 \parallel \mathtt{l} := \mathtt{h}$$

When this program is run with $\mathtt{h} = 0$ or $\mathtt{h} = 1$ under a scheduler that gives equal priority to all three parallel processes, the final value of $\mathtt{l}$ is more likely than not to be $\mathtt{h}$. However, the possible traces of the value of $\mathtt{l}$ are independent of the value of $\mathtt{h}$.

$$\begin{array}{l} \mathtt{l}_1 := 0; \mathtt{l}_2 := 0; \\ \textbf{while}\ \mathtt{h}\ \textbf{do}\ (\mathtt{h} := \mathtt{h} - 1; \mathtt{l}_1 := \mathtt{l}_1 + 1); \\ \mathtt{l}_2 := 1 \end{array}$$

Here, an observer who can see both low variables can wait until $\mathtt{l}_2$ is 1, then read $\mathtt{l}_1$ in order to discover the value of $\mathtt{h}$, provided it is non-negative. However, the trace of $\mathtt{l}_1$ is always a prefix of $0, 1, \ldots, \mathtt{h}$ (up to stuttering) and the trace of $\mathtt{l}_2$ is always stuttering-equivalent to $0, 1$. So Zdancewic's definition of observational determinism permits this program.

$$\begin{array}{l} \mathtt{l}_1 := 0; \mathtt{l}_2 := 0; \\ \textbf{if}\ \mathtt{h}\ \textbf{then}\ (\mathtt{l}_1 := 1; \mathtt{l}_2 := 1)\ \textbf{else}\ (\mathtt{l}_2 := 1; \mathtt{l}_1 := 1) \end{array}$$

The low traces of $(\mathtt{l}_1, \mathtt{l}_2)$ here are stuttering-equivalent to $(0,0), (1,0), (1,1)$ if $\mathtt{h}$ is non-zero and $(0,0), (0,1), (1,0)$ if $\mathtt{h}$ is 0. An observer could determine this by watching the order of updates. Yet Huisman and others' notion of observational determinism permits this program.

Figure 2.4: Leaking information with concurrency

2.3.6 Observational Nondeterminism

Programs with *concurrency* present new and subtle ways for information to be leaked. In the case where the concurrency is modelled as nondeterminism, the way in which the nondeterminism is resolved (by a scheduler) may lead to information being leaked via a *timing flow* [RWW96]. Observational determinism is a popular way of extending noninterference to a concurrent setting. Following the intuition that, in a concurrent system, the contents of memory are output constantly, rather than only at program termination, observational determinism requires that initial states with similar low views lead to *traces* with similar low views. A trace here is simply the sequence of states in a run of a program. The aim is that observational determinism enforces information flow security regardless of the behaviour of the scheduler.

In a nondeterministic system, a single input state may lead to several output traces; indeed, this is what usually makes concurrency interesting. Observational determinism does not require program *confluence*, but it does require that updates *observable* at low-security locations appear to be *deterministic* (hence its name).

As evidence of the subtlety of concurrent information flow, observe that several authors have suggested different forms of similarity in their formalisations of observational determinism; see Figure 2.4 for some examples. The formulation of Zdancewic and Myers requires that when the traces of a single low variable are projected from two program traces, they should be similar up to *stuttering* and *prefixing* [ZM03]. However, they consider only race-free programs, in which programs cannot synchronise via shared memory accesses. Huisman and others object to this because of the potential for information to be leaked via *synchronisation* between the traces for different variables, which is possible even without concurrency; they suggest removing the allowance for prefixing and requiring that variable traces be similar up to stuttering only [HWS06]. Finally, Terauchi observes that this still allows information to be leaked by observing the *ordering* of updates; he suggests allowing similarity up to stuttering and prefixing, but over the traces projected to the values of all low variables simultaneously, not individually [Ter08].

2.4 JavaScript and other Dynamic Languages

JavaScript is a scripting language that is mainly used to add client-side interactivity to webpages: JavaScript programs can be embedded within HTML webpages; these programs are run by a web browser when a certain event occurs, such as the user clicking on a button; the programs can change the content of the webpage. JavaScript programs are used: to reduce the latency of an action (as it can be performed on the client, rather than via communication

with the web server); to allow partial updates to a webpage; and to shift the overhead of a computation from the web server to the web browser. Asynchronous JavaScript And XML (AJAX) is one of the key components of the Web 2.0 design movement.

2.4.1 Versions

ECMAScript [Int99] is a standardised version of JavaScript described in the standard ECMA-262; JavaScript is commonly regarded as a dialect of ECMAScript, even though it predates the language specification. Each web browser has its own version of JavaScript; these versions differ in terms of what extra features they provide and their APIs for interacting with webpages and the browser. There are also some semantic differences in their adherence to the ECMAScript specification and their resolution of ambiguities within it [MMT08].

Most current JavaScript versions are based on ECMAScript Edition 3, which was released in December 1999 and corresponds to JavaScript versions 1.5–1.8. The following version, ECMAScript Edition 5, was released in December 2009. It adds various features that were already present in some versions of JavaScript, as well as a "strict" mode that requires programs to pass various static checks.

There is a variant called the ECMAScript Compact Profile, which is designed for use in embedded devices [Int01]; it is used for scripting in the HD-DVD format. This version does not require support for **eval** or **with**. It is interesting to note that the standards body considers these features to be sufficiently troublesome to merit the specification of a reduced language that excludes them.

2.4.2 Features

JavaScript has an unusual combination of language features, some of which make it difficult to reason about information flow security [GSK10].

First-class functions Functions are first-class values: they can be passed as arguments and stored in variables. For this reason, JavaScript can be considered to be a functional language in spirit, but with a syntax and features more typical of an imperative language. Objects subsume functions, in the sense that a function can be emulated by an object with a method that behaves as the function, and this object can be passed around like a first-class function. Indeed, in JavaScript, functions are implemented as objects.

Objects JavaScript supports objects, which behave much like associative arrays. Object members can be created, updated and deleted at run-time. Members can be selected using an expression computed at run-time, which makes alias analysis hard. *Inheritance* is implemented through *prototyping*: objects have a hidden member `__proto__` that may specify a prototype object; any attempt to look up a member that is absent from an object cascades to a lookup on the prototype object.

Exceptions Errors generate exceptions, or they can be generated explicitly using the **throw** statement. When an exception is thrown within a **try** block, execution transfers to the following **catch** block. Exceptions complicate analysis of program control flow. However, in their absence, programmers tend to abuse more primitive control features in ways that make analysis even harder.

Dynamic typing Variables in JavaScript do not have a fixed type: a variable may store values of different types during program execution. The "type" of an object is determined by its prototype fields. Furthermore, there is *implicit conversion* between numbers and strings. This makes it tricky to determine statically what the effect of an operation will be, and what type conversions (if any) will be invoked.

Scope objects Variable scoping in JavaScript depends on a hierarchy objects: the value of a scope object field determines to which object a particular name refers. These scope objects can themselves be manipulated: the **with** construct allows any object to be added to the bottom of the hierarchy of scope objects, while **this** is aliased to the object at the top of the hierarchy during execution of a function application.

Run-time code generation The **eval** construct allows a string containing the source code of a JavaScript program to be parsed and executed at run-time. In the presence of this feature, static analysis of most kinds quickly becomes undecidable.

JavaScript is often called a *dynamic language*. There is no strict definition of what constitutes a dynamic language. It is usually taken to mean one with dynamic features, where dynamic features are those that allow a program to do at run-time what might in another language only be possible at compile-time [Tra09]. Hence JavaScript's object systems and dynamic typing are dynamic features, as in many languages both the type hierarchy and the types of variables are fixed at compile-time. Similarly, run-time code generation is a dynamic feature because in a typical compiled language, source code must be generated before compilation.

Another possible definition of a "dynamic" language is one in which it is quick and easy to modify existing programs or develop them incrementally. For example, in a dynamically typed language, type changes in one part of a program do not need to be propagated throughout the entire program. The resulting program may not work as a whole, but in incremental development, one often only cares about the behaviour of a small part of a program at any one time.

The environment in which JavaScript programs typically execute is also noteworthy. As mentioned above, multiple JavaScript programs may be combined on a single webpage and are executed by the browser in response to events such as the user clicking a button. There is no concurrent execution: once an event handler is started, no other JavaScript programs may run until it has completed. JavaScript itself includes no facilities for I/O, but the browser provides a library of pre-defined functions through which programs may interact with the webpage or even send and receive messages over the network. Thus to some extent, a webpage containing JavaScript programs can be modelled as a reactive system [BPS+09]: input and output actions encode both interactions with elements on the webpage (which can be regarded as messages sent between the webpage and the user) and interactions with other computers.

2.4.3 Semantics

No ECMAScript specification includes a formal semantics. ECMAScript Edition 4 [TT07] was planned to be specified through a *reference implementation* in SML, which does have a formal semantics. However, this project was abandoned.

Some papers claim to offer a semantics for some core subset of JavaScript, via an applied lambda-calculus with some of the features of JavaScript and some superficial resemblance [YCIS07]. Given the current popularity of JavaScript, it must be tempting for researchers to throw their work onto the Web 2.0 bandwagon in an attempt to attract attention.

A complete *operational semantics* of ECMAScript Edition 3 exists [MMT08]. It follows the style of the specification and is correspondingly lengthy, but crucially it handles the whole core language; it has been used to prove properties of the language and for verification work on real JavaScript code [MT09].

The authors comment on the risks of equating JavaScript implementations found in browsers with their operational semantics based on ECMA-262: there are some ambiguities in the specification that are resolved in a particular way in the semantics, which may differ from the way chosen in implementations; some implementations unambiguously deviate

from the specification in certain areas; JavaScript implementations are permitted to include extra features, which are obviously not covered by the specification.

Bodin and others [BCF+14] have extended this line of work to produce: a translation of the ECMAScript Edition 5 specification into Coq (JavaCert); a reference interpreter in OCaml (JavaRef); and a mechanised proof that the interpreter implements the specification. Their work has exposed ambiguities and inconsistencies in the natural language specification. JavaCert could justifiably be used as a formal semantics for JavaScript in mechanised proofs, but as with the earlier work on ECMAScript Edition 3, its size would make it difficult to use when proving non-trivial properties of the language or program analyses.

A paper on JavaScript semantics by Guha and others [GSK10] applies the idea of a JavaScript subset to the goal of reasoning formally about the whole language. It defines not only a *core language* that is small enough to be comprehensible, but also a syntactic translation from real JavaScript (excluding **eval**) into this core language. It seems to present a promising basis for semantically-based JavaScript verification. Some caution is still needed, as the claim of the correctness of the translation is based on empirical evaluation against test suites, rather than an exhaustive examination of the specification.

2.4.4 Security Problems in JavaScript

When JavaScript code runs in a web browser, its access to the computer on which it runs is far more limited than that of an ordinary application. Such *client-side* JavaScript code can interact with the computer only through the restrictive API provided by the browser, which does not usually allow access to the filesystem or many other resources (although this is changing; for example, HTML 5 has a draft API for accessing files [RS13]). Consequently, many of the malicious behaviours that would usually be the focus of a security analysis of computer programs, such as the possibility of the program secretly overwriting executables of other programs, are not applicable to JavaScript in a browser. (They are still applicable when JavaScript runs on a web server, as has recently been popularised by the Node.js framework [**?**].) However, client-side JavaScript code still poses several security problems. Malicious behaviours of JavaScript can roughly be split into four categories: causing a nuisance, collaborating with other attack mechanisms, impersonating the user of the browser and leaking information.

Causing a nuisance. JavaScript code can cause a nuisance by wasting resources such as memory and processor time, making the computer on which it runs slow and unresponsive; this can often be the result of a bug in the code, rather than malicious intent. To counteract

this, many modern browsers will stop execution of a script that runs for a long time. Another source of annoyance is JavaScript code that causes a visual distraction or obstruction to the user of the browser, for example by opening popup windows or dialogue boxes, or by redirecting the browser to another webpage. Again, many browsers will restrict this behaviour by default.

Collaborating with other attack mechanisms. There are many attacks on web browsers that use security holes in browser plugins to perform actions that should not be allowed, such as executing arbitrary code with the privileges of an ordinary application. These attacks can usually only be initiated by the user viewing a webpage that holds malicious content. For example, suppose an attacker places a special PDF document on a webpage. The user happens to visit this page and the web browser attempts to open it using a PDF reader plugin. The plugin has a buffer overrun vulnerability that is exploited by the specially constructed PDF document, causing code at some unknown (and effectively random) address in memory to be run with the full privileges of the PDF reader, which will usually include filesystem access.

The PDF might include code that installs a program giving the attacker control of the computer, but the chances of that code being at the address in memory where the plugin incorrectly executes code are small. Malicious JavaScript code can collaborate with this attack to make it more likely to succeed. In a common form of attack, known as heap spraying [RLZ09], a JavaScript program creates many objects that, when stored in memory by the browser, have an encoding that matches the malicious code the attacker wishes the plugin to execute. Filling memory with malicious code increases the probability that a random address in memory holds that code, thereby increasing the probability of success for the attack. The JavaScript code can then redirect the browser to the PDF to initiate the attack.

Impersonating the user. If the user is logged into a website, such as a social networking website or shopping website, then visiting certain webpages may be sufficient to trigger actions that use the user's account privileges, such as editing the user's profile page or buying a book. If malicious JavaScript is placed on such a website, it can visit these webpages and perform these actions without the user's intervention. The Samy worm used this technique on the social networking website MySpace, copying itself to the personal page of any user who viewed a page holding the code [LC08].

```
// Benign: use keywords from a webpage to produce targetted adverts
var keywords = document.getElementById("keywords").textContent;
var img = document.getElementById("adimage");
img.src = 'http://ads.com/SelectAd.php?keywords='+keywords

// Malicious: steal a password instead of keywords
var password = document.getElementById("password").textContent;
var img = document.getElementById("adimage");
img.src = 'http://ads.com/SelectAd.php?keywords='+password
```

Figure 2.5: An example of an information leak in JavaScript

Leaking information. JavaScript code may have access to data from a variety of sources, such as: form controls filled in by the the user; the content of the webpage hosting the code; cookies stored by the web browser; the web browser's history; the user's mouse and keyboard use; and communication over the network with the server hosting the webpage. Some of this data may be sensitive, but the user typically has no way of knowing what data a script accesses and what it does with it.

It is quite possible for malicious code to harvest data from one of these sources and send it back to its hosting server. Indeed, a recent survey using a browser augmented with a run-time taint analysis found that many popular websites are guilty of this [JJLS10].

Figure 2.5 reproduces an example from Barthe and others [BCD+12] that demonstrates leaking of information. The first piece of code sends keywords from the webpage containing the code to an advert provider, so that the provider can display an advert relevant to the content of the webpage to the user. The second piece of code is syntactically very similar, but instead leaks the contents of a password field from the webpage to the provider, who may then use it to steal the user's account on the website. A suitable information flow analysis, informed that the password is high-security and the advert provider's server is low-security, would be able to detect this leak. In practice, the method by which the information is leaked might be far more subtle.

2.5 Metaprogramming

2.5.1 Styles of Metaprogramming

Metaprogramming is the ability to manipulate program code within a program. The term covers a broad range of programming language features [BT10]. Sometimes the language of the manipulated code (the *object language*) is different from that of the program that ma-

nipulates it (the *metalanguage*); this is known as *heterogeneous* metaprogramming. Typical applications include writing compilers or constructing SQL database queries from within another language. More commonly, metaprogramming refers to *homogeneous* metaprogramming, in which the object language and the metalanguage are the same.

Even within homogeneous metaprogramming, there is a wide range of styles and applications. For example, the macro system of C and the template system of C++ are forms of *compile-time* metaprogramming; they are resolved to ordinary code during compilation. In contrast, in some languages, such as JavaScript and Perl, **eval** provides the ability to execute an arbitrary *lexical* string, *generated* at *run-time*, as if it was a block of code; if the string is not a syntactically valid program, this usually generates an error.

Other languages, such as Lisp, include the ability to "quote" expressions: evaluation of expressions can be delayed and gaps within these expressions (denoted by "antiquotation") can be filled with ordinary expressions, which may themselves include quoted code; this approach ensures that any code is *syntactically* valid. It was often thought that this style of metaprogramming was only possible in Lisp because of its syntactic simplicity; it is only recently that it has been found in syntactically more complex languages, such as MetaML and Template Haskell [FT07]. Code quotations can typically be nested to arbitrarily many levels or *stages*; hence this style is sometimes referred to as *staged metaprogramming*. In some applications, there is a clear distinction between when different stages are executed; for example, one stage may be intended for execution at compilation and another at run-time. However, this is not central to the idea.

Finally, a few languages, such as ReFLect, allow code values to be deconstructed, exposing their *intensional* structure [GMO06].

2.5.2 A Metaprogramming Example

Metaprogramming can also be used for optimisation. For example, consider the common loop unrolling optimisation: a loop with n iterations can be expanded to n sequential executions of the loop body; this avoids the overhead of executing the loop's flow control. If n is a constant known at compile-time, then a compiler can implement this optimisation statically. However, if the loop is run several times with the same n, but this n is known only at run time, then it may be worthwhile to unroll the loop at run time by dynamically generating code for it. A popular demonstration of this is a curried version of a power function [BT10]. This might be written as follows in JavaScript (without loop unrolling):

```
function currypow (n) {
  return function (x) {
```

```
    var z = 1;
    for (var y = 0; y < n; y++) {
      z = z * x;
    }
    return z;
  }
}

> currypow(3)(5);
125
```

(> indicates a prompt in an interactive JavaScript shell.) However, an equivalent function that unrolls the exponentiation could be written thus:

```
function genpow (n) {
  var c = "1";
  for (var y = 0; y < n; y++) {
    c = "x * " + c;
  }
  return eval ("(function (x) { return " + c + "; })");
}

> genpow(3)(5);
125
```

The distinction is that, if one wishes to define a cube function, the second will be more efficient; the following pairs of definitions will be essentially equivalent:

```
var slowcube = currypow(3);
var fastcube = genpow(3);
```

and:

```
function slowcube (x) {
  var z = 1;
  for (var y = 0; y < 3; y++) {
    z = z * x;
```

```
  }
  return z;
}

function fastcube (x) {
  return x * x * x * 1;
}
```

In reality, it is unlikely that someone would care about this kind of low-level optimisation in a JavaScript application. JavaScript programmers are more likely to use **eval** for convenience or out of ignorance.

2.5.3 Abuses of Metaprogramming

As with any language feature, the power of code generation and evaluation is open to abuse. In languages where code strings can be parsed and evaluated, interpolation of untrusted strings into blocks of code is a common source of security holes. Meijer and Drayton claim that JavaScript's string evaluation is often used where some other, more restrictive language feature (either absent from the language or unknown to the programmer) could achieve the same result more elegantly [MD04]. For example, they say that inexperienced programmers often use **eval** instead of higher-order functions: rather than passing a function as a value and applying it, they pass a code string as a value and **eval** it. A large-scale survey by Richards and others [RHBV11] reveals many other uses. One of the most common is reading a field whose name is held in the variable x from the object o with `eval("o." + x)` instead of `o[x]`.

When **eval** is used maliciously, it seems to be mainly for obfuscation [DHF10]. Most sites that allow untrusted third-party code forbid use of **eval**, so in a context where an attacker can use **eval** (for example, because they have hacked the web server, or exploited some unrestricted aspect of the DOM), they can probably already use any JavaScript code they want. Hence the main motivation for using **eval** is to hide the malicious intent of code, for example from review by a human, or from automated checks present in the web server or browser.

Obfuscation techniques involving **eval** range from splitting code into two strings (to avoid simple checks for names) to use of cyphers. Some attempts at obfuscation mimic JavaScript code compressors (which are whitelisted by some security tools).

2.5.4 Semantics of Metaprogramming

The semantics of evaluation of dynamically generated code can be subtle. For example, variable identifiers may occur unbound within quoted code and are bound only when the code is evaluated. Exactly how they are bound depends on the scoping rules of the language. As a rather trivial example of how dynamically generated code can confound static analysis, observe that the following JavaScript code behaves the same as `y = x`; note that `x` does not occur in the code:

```
y = eval("X".toLowerCase());
```

A number of existing works consider the semantics of quoted code. However, the field is relatively unexplored; perhaps this is testament to the difficulty and subtlety of the area. I consider now some of the abstract languages with metaprogramming examined therein.

MetaML and λ-U. MetaML [TS00] was one of the first languages to feature both Lisp-style quoted code and a syntax more complex than that of Lisp. Taha explored its semantics by examining an untyped, call-by-name lambda calculus called λ-U [Tah00]. The language extends the standard untyped, call-by-name lambda calculus with three constructs for code quotation, antiquotation and evaluation.

The semantic subtleties of the language result in part from the behaviour of variables in open code values under substitution. Consequently, in order to formalise the denotational semantics of the language, a good technique for reasoning about name-binding would be needed. For example, an approach based on nominal sets might be successful [LP14].

The expressions of λ-U are stratified into stages, where a stage-n expression must be enclosed in n or fewer levels of quotation in order to match its antiquotes. If e is a stage-n expression, then its quotation $\langle e \rangle$ is a stage-$(n-1)$ expression. A program is a closed, stage-0 expression, but executing it may involve the evaluation of open or closed expressions at any stage.

An important phenomenon in the language is cross-stage persistence: evaluation may involve a stage-0 value being converted to a value at a higher stage. A simple example of this is the stage-0 expression $(\lambda x.\langle x \rangle)(\lambda y.y)$, which evaluates to the stage-0 value $\langle \lambda y.y \rangle$, where the stage-0 value $(\lambda y.y)$ occurs at stage 1; a function object is converted into a code object!

ReFLect and Parameterised Combinatory ReFLect. Melham and others consider the language ReFLect, which is an ML-like language developed by Intel for formal specification and verification of hardware [MCC13]. The language includes Lisp-style code quotations and antiquotations for splicing together code values. It is possible to pattern match over the structure of code and to evaluate code.

All code has a single type, *term*, regardless of the type of the value that results from evaluating the code. The language designers consciously made this choice so that functions manipulating code would be easily typeable. Unfortunately, the authors claim that this prevents the language from having a set-theoretic denotational semantics, which leads them to conclude that any standard higher order logic built on the language will be inconsistent. This is of particular concern with ReFLect, as the language is intended for use in a theorem-proving environment. The authors suggest rectifying this by parameterising *term* (by the type of the value that results from evaluating code) and extending the type system of the language with Generalised Algebraic Datatypes (GADTs).

An interesting aspect of this work is that the result is derived by considering a sublanguage of ReFLect called Combinatory ReFLect (CR). This language, based on the system of computation that uses only the combinators S, K and I of combinatory logic, does not have any variables. In this way, the authors cleverly avoid the need to deal with variable binding and hence to introduce any associated machinery (such as nominal sets). They also avoid subtle phenomena such as the capture of variables when code is spliced inside a lambda abstraction. The resulting language, which allows quoting and evaluation of code, but not pattern matching, is pleasingly simple. The language also includes a "lift" function, which takes a value and returns code that evaluates to that value.

The authors illustrate their proposed parameterisation of *term* in CR, resulting in the language Parameterised Combinatory ReFLect (PCR). As the language is simply typed in the style of the simply typed lambda calculus, it does not allow non-terminating programs. They outline a denotational semantics for PCR, which relies on terms being strongly normalising. To extend their approach to a Turing-powerful language, one would need to work around this limitation.

In searching for a denotational semantics of a language with pattern matching over code (such as ReFLect, but not CR or PCR), it is important to note that a fully abstract semantics cannot be found by extending a fully abstract semantics for a sublanguage without metaprogramming. This is because two syntactically distinct terms that are contextually equivalent in the sublanguage cease to be contextually equivalent in the presence of pattern matching, which allows the terms to be distinguished. Therefore their denotations must be the same in a fully abstract semantics for the sublanguage, but not in the full language. Observe that

in a language where arbitrary strings may be evaluated as code, such as JavaScript, pattern matching on code strings is possible through standard parsing techniques. I consider this to be evidence, if any was needed, that reasoning about arbitrary code strings semantically is hard.

The power of variables. It is straightforward to translate a term of lambda calculus into one of combinatory logic [HS08], maintaining a close correspondence between the high-level structure of the terms during execution and eliminating the use of variables without losing any expressive power. Following from the observations of Taha and Sheard [TS00] that it is not possible to translate a staged λ-U program into a lambda calculus program and maintain such a correspondence, it seems highly unlikely that a translation that removes both staging and use of variables would be able to do so. The question remains as to whether eliminating variables within a staged language reduces expressive power.

It may be worth considering the unstaging translation of Choi and others [CAYT11] from a staged language to lambda calculus extended with records. Here, the records are used to simulate environments. The power of records is that they allow a program to check equality between an unbounded number of constants (used to encode variable names), which may originate anywhere within a program. It seems unlikely that this behaviour can be encoded using combinators while maintaining the high-level structure of a program, as their behaviour is inherently bounded and local. Hence I suggest that, at least in the presence of variable-capturing substitution, variables are likely to contribute to the expressive power of a staged program.

PCF_{DP}. Berger and Tratt produce a Hoare-style total correctness logic for PCF_{DP} [BT10]. This language extends PCF with a quote construct and a form of let construct that combines anti-quotation and evaluation; the types are extended with a family of code types, parameterised by the type of the result of evaluation.

PCF_{DP} places some restrictions on how variables in quoted code may be used. As one might expect, open code terms (those having free variables) may not be executed. More significantly, variables free within quoted code may not be lambda abstracted outside the code; this is in direct contrast to λ-U. For example, the "lift" function $\lambda x.\langle x \rangle$ is not a well-typed PCF_{DP} expression. In fact, a consequence of this restriction is that the "lift" function is not in general expressible for function types.

In order to enforce these restrictions, the typing rules split the typing environment into two parts: the first is standard; the second holds "modal" variables, which are those occurring free inside quoted code. The type system builds on work by Davies and Pfen-

ning [DP01], in which code that evaluates to a value of type A has type $\Box A$. This extends to function types, enforcing type safety for code-manipulating programs. The structure of the typing rules mirrors the rules for the modal logic S4.

Berger and Tratt's logic has a number of desirable properties. Removing the rules for the metaprogramming constructs gives a logic for PCF. The logic is relatively complete: assuming the availability of an oracle for Peano arithmetic, all semantically true assertions expressible within the logic can be derived syntactically using its rules of inference. It is also observationally complete: the derivable logical assertions concerning the behaviour of two program terms are exactly the same if and only if those terms are contextually equivalent. Finally, the logic is descriptively complete: for any term, there exists a single logical assertion, syntactically derivable from the term, that is satisfied only by terms contextually equivalent to it.

The authors outline a number of open problems concerning metaprogramming and program logics. One is how to combine type-based techniques for metaprogramming with logical techniques for metaprogramming. This may be interesting as, while one may view types as simply a method of encoding logical assertions and vice versa, type systems tend in general to be less expressive than logical systems, but have better facilities for automated reasoning. Combining the two might lead to an expressive, semi-automated method of reasoning about metaprogramming.

Another open problem is how to handle open code; this is necessary in order to handle languages such as λ-U, where metaprogramming is less restrictive. A final problem is how to handle mutable state; this would be necessary in order to reason about any JavaScript-like language. Type safety for languages that can manipulate open code has been investigated by Pientka and others [Pie08, NPP08], extending the idea of using modal logic as the basis for types. Kameyama and others [KKS14] have also explored how to maintain type safety in a domain-specific language that allows manipulation of open code using mutable variables. However, these works do not show how to reason about the behaviour of metaprogramming beyond type safety.

Summary. Metaprogramming is semantically difficult, although there has been some progress in understanding and reasoning about it. Two important choices in the design of a language supporting metaprogramming are how to type quoted code and whether to allow lambda abstraction or capture of variables occurring free within quoted code from outside that code.

At one extreme, PCF_{DP} has quoted code typed according to the result of evaluating the code and forbids abstraction of variables free in quoted code. At the other extreme, λ-U

is entirely untyped and allows abstraction of variables free in quoted code. In between lies PCR, which has parameterised types for quoted code and no variables at all.

Chapter 3

Previous Work

3.1 Overview

I now survey the body of previous work relating to verification and enforcement of various information flow properties, including noninterference. I also look at some work relating to analysis of JavaScript and its dynamic features, even where it does not directly concern information flow. A comparison of the features covered by the various publications mentioned (and this thesis) is shown in Figure 3.1.

3.2 Verifying or Enforcing Information Flow Properties

There are various methods of verifying or otherwise enforcing information flow security properties [SM03a]: some are *dynamic* (performed at run-time); others are *static* (performed at compile-time).

A popular static approach for verifying that a program does not leak information is to construct a *type system* in which only programs that do not leak information can be well-typed, then to type-check the program within this system. A typical dynamic approach is to *instrument* a program to test whether it attempts to leak information at run-time and to stop it before it does; however, in contrast with data typing, dynamic approaches have few advantages over static ones.

3.2.1 Instrumentation and Monitoring

Historically, some of the earliest attempts at enforcing information flow security involved instrumentation or monitoring of a program to detect and prevent information leakage dynamically [Fen74]. The idea is simple: the execution environment of a program is aug-

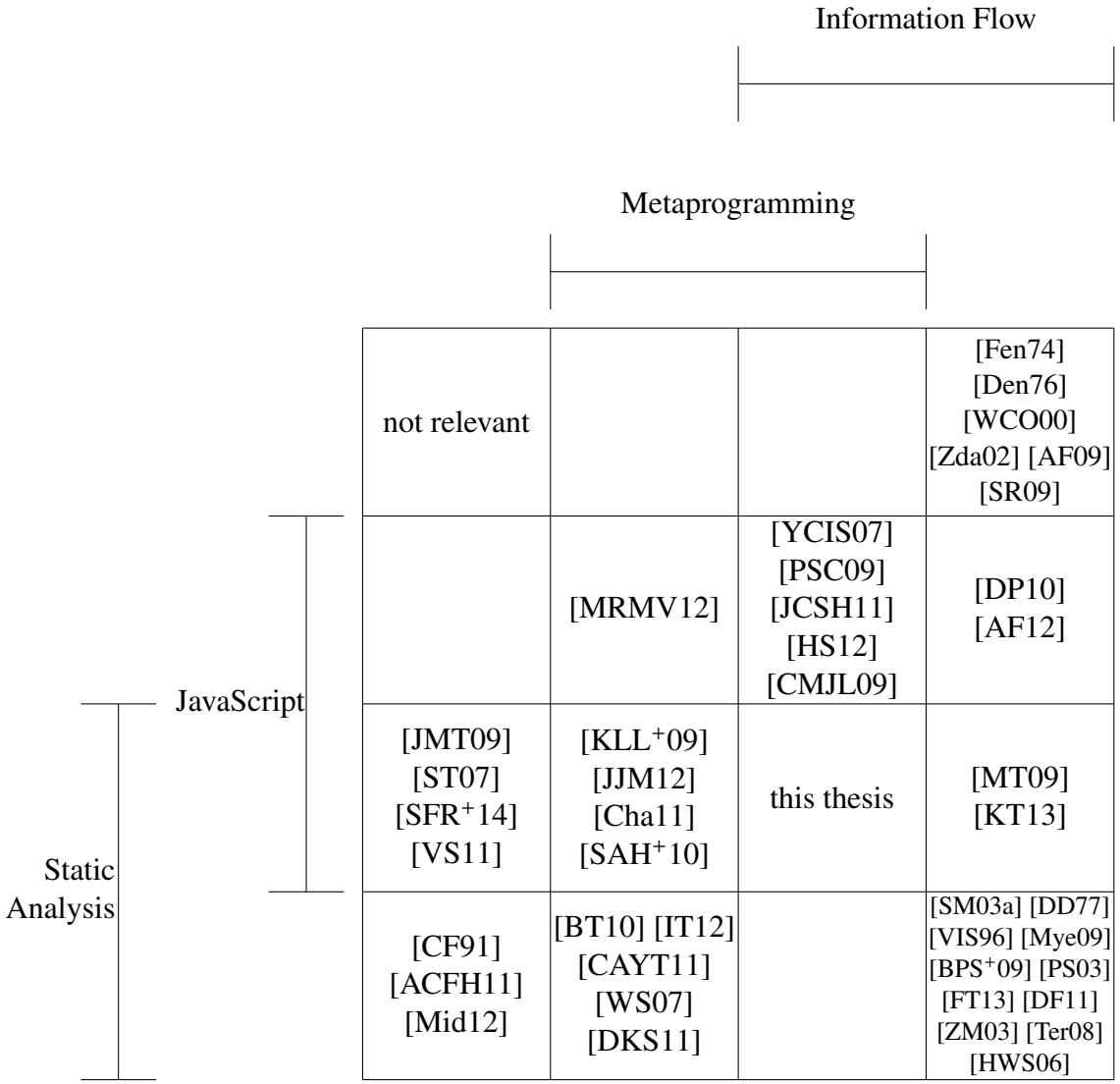


Figure 3.1: A graphical comparison of areas covered by previous work

mented with a *monitor* that checks all operations the program performs; if the program attempts to perform an operation that would leak information, the monitor terminates it.

The monitor could be implemented as part of the language interpreter or virtual machine that runs the program. Alternatively, the monitor could be implemented by adding *instrumentation* to the program at compile-time; then no special treatment would be needed at run-time.

These methods are largely unsatisfactory because they consider only a single execution path, while information flow security properties like noninterference rely on all paths; they tend to be weak at handling implicit flows. Furthermore, while it is true that terminating a program before it leaks information enforces termination-insensitive noninterference, it seems like cheating to change the behaviour of violating programs, rather than forbidding them entirely!

The only real advantage of the monitoring and instrumentation approach, apart from its simplicity, is that it can potentially handle complex situations that would be extremely difficult to handle statically, such as the use of JavaScript's **eval** construct, which allows execution of dynamically generated code. It is usually easy for a dynamic analysis or monitor to handle dynamic code generation: the executed generated code is immediately available and may be analysed or monitored as easily as any other code. Effectively, the dynamic method only has to handle the one code value that corresponds to the current run, rather than the infinity of code values that may be generated over all runs.

Nonetheless, perhaps because of the simplicity of the underlying idea and the ease of implementation, these methods have received some recent attention as a means of enforcing information flow security properties in JavaScript.

Extensional monitoring. There are a few variations that differ in terms of how the monitor is implemented and what it checks [Den76]. In the simplest case, the program is given a single security label, which is initially low. If the program inputs from a high channel, the monitor *upgrades* its label to match the channel. If the program attempts to output to a channel below its label, the monitor terminates it, as there is a potential information leak. In this case, the monitor can be implemented solely at the operating system level, for example as a wrapper around system calls; consequently, the overhead can be low. This is because the analysis only concerns the *extensional* behaviour of the program; the way it is implemented is irrelevant.

Unfortunately, this approach suffers badly from *label creep*: the program's security label increases monotonically during program execution, so many safe but useful programs

are prohibited; observe that as soon as any high input occurs, any low output is impossible. However, this method can enforce termination-insensitive noninterference.

Yu and others [YCIS07] present a language called CoreScript, which they claim is a fragment of JavaScript. Disappointingly, it seems to be little more than a simple imperative language with support for some form of I/O and a syntactic similarity to JavaScript; there is no appeal to a formal JavaScript semantics.

They describe how to instrument this language to enforce security policies described by *edit automata*. These policies limit the external actions of a program, but say nothing about internal information flow. For example, an edit automaton can specify a policy that, after allowing a program to read a (potentially sensitive) cookie belonging to a website, forbids the program from sending any data to any other website; however, it cannot forbid only communications that leak information about the content of the cookie.

Their key idea is that code produced during execution, for example by **eval**, can be *instrumented dynamically*: the initial instrumentation process wraps any I/O and any code generation; the wrapper for code generation repeats the instrumentation process on the generated code. They suggest that programs could be instrumented in JavaScript, as part of the program being executed, or (lacking any semantic guarantees against tampering) by a modified JavaScript implementation.

Phung and others [PSC09] develop this idea and enforce a similar range of security policies entirely within JavaScript, but without rewriting the original code. When JavaScript is executed within a browser, I/O can occur only through pre-defined, globally visible methods; these can be updated by programs running in the browser. The authors' approach is to add a monitor to the webpage containing the code, which immediately redefines all these methods, replacing them with *wrapped* versions that check the security policy being enforced. Through use of JavaScript's variable scoping, the monitor and its state are invisible from outside. Thus the original (unwrapped) methods can only be accessed via the new (wrapped) methods.

Once again, this paper does not use a formal semantics of JavaScript to verify that the monitor is safe; the authors acknowledge this as a weakness. Indeed, as they present workarounds for a number of possible tricks for tampering with the monitor, one wonders if they have missed one.

However, the approach does handle real JavaScript in its entirety, including **eval**, without any rewriting of the original code, using only a small monitor, and with only a modest performance hit. The paper claims that, in tests against real-world exploits, the monitor blocked almost all attacks. From a practical perspective, it is impressive that all this is

possible simply by adding some static code to the hosting webpage, without modifying the browser's JavaScript implementation.

Mandatory access control. A more precise approach is for the monitor to associate a security label with each value in the program [Fen74] to indicate its level of confidentiality. In the case of an imperative program, this means giving a label to each variable or memory location. Whenever a value is computed, the monitor assigns the resulting value a label not less than each of the existing labels of the values used in the computation; if the value is stored in a variable, the variable's label is upgraded. If the program attempts to output a high value to a low channel, the monitor terminates it; this blocks explicit information flows.

The scripting language Perl has a *taint mode* [WCO00] that uses a similar kind of monitoring. Perl scripts are commonly used to join together the input and output of different programs with some text processing and simple flow control. It is common to invoke these programs by constructing a string representing a shell command and executing it. If data is passed to a shell command in this way, an attacker can craft data that will be interpreted by the program as an option or interpreted by the shell as a separate command. The purpose of taint mode is to prevent use of data read from untrusted sources being used as part of code strings, thereby blocking such attacks. (Note that taint mode enforces information flow integrity rather than confidentiality: it blocks direct flows of information from low security inputs to high security outputs.)

It is tricky to extend this approach to handle implicit flows. Care has to be taken to track indirect information flow due to code that was *not* executed in the observed run. In an imperative language, such a flow occurs when a low update is made within a branch on a high value, so the monitor must record the security label or labels on which the current execution path depends and forbid updates at levels below those labels. However, it is difficult to determine when a branch ends if the program language allows complex or unstructured flow control (for example, if the source language allows **goto** statements or if the program has already been compiled to machine code); hence it is difficult to determine on which labels the current path depends.

No sensitive upgrade. Even if it is possible to determine when a branch ends, implicit flows can also occur when, within a high conditional branch, the label on a memory location is upgraded from low to high: the upgrade might not be made on the other branch. Strategies to handle this include, for instance, the *no sensitive upgrade* check [Zda02], which aborts execution if a public variable is assigned in code that is control dependent on

private data. Austin and Flanagan [AF09] similarly forbid upgrades within a branch on a high value. However, their *permissive upgrade* technique allows labels to be pre-emptively upgraded before the branch, using *annotations* inserted by the programmer or some static analysis tool; this increases the range of allowed programs. Examples of such analyses for JavaScript (including **eval**) are presented by Just and others [JCSH11] and Hedin and Sabelfeld [HS12]; the former uses some static analysis to identify implicit flows in the presence of unstructured flow control.

As a rule, however, such strategies are fairly coarse and could potentially abort many innocuous executions; thus it is commonly held that static analyses are superior to dynamic ones in their treatment of indirect flows [SM03a], although there has been a resurgence of interest in dynamic analyses [SR09].

Secure multi-execution. A slightly different approach to dynamic enforcement is offered by *secure multi-execution* [DP10]. The idea here is to run multiple copies of a program simultaneously, one at each security level of interest. Inputs from a level at or below a copy's own level are performed normally. However, inputs from a higher level are replaced with dummy data. Each copy outputs only at its own level; other outputs are discarded. Effectively, each copy can only access data visible from its own view, which guarantees noninterference (even termination-sensitive noninterference), but at the cost of potentially altering the behaviour of the program! The technique is potentially very inefficient, as the number of copies needed increases exponentially with the number of security levels; Austin and Flanagan [AF12] mitigate this with *faceted execution*, which collapses together levels that behave identically.

3.2.2 Isolation via Safe Subsets

A simple but rather restrictive way of statically preventing a program from leaking information is to identify a syntactic subset of the language in which I/O is impossible, then verify that the program lies within that subset.

A similar idea forms the basis of one technique used by websites such as Facebook to prevent harmful JavaScript programs from being served to users. Here, the aim is to combine several programs on a single webpage, without allowing them to communicate or otherwise interfere with each other. Maffeis and Taly [MT09] identify two subsets of JavaScript with properties useful in this regard.

Programs written in the first subset (which they call *Jt*) can only access properties that are named explicitly in the program code or belong to a limited set of special properties

(such as **toString**); this subset forbids the use of language features that would enable a string to be used as a property name, such as **eval** and expressions of the form **e**[**e**].

In addition to the restrictions of the first subset, programs written in the second subset (called *Js*) have the property that expressions cannot evaluate to *scope objects*; this is achieved by forbidding the use of language constructs such as **with** and **this**, in addition to those forbidden in the first subset. (In full JavaScript, various subtle tricks employing scope objects can be used to violate isolation mechanisms.) A consequence of this is that variables occurring in a program within this subset can be mechanically *renamed* in a way that preserves the behaviour of the original program; this is useful because it leads to a way of combining several programs (with possibly conflicting variable names) written within the subset into a single page, in such a way that the programs cannot interact with each other.

From a theoretical perspective, the security properties that are proved for these subsets of JavaScript are unremarkable. What makes this work interesting is that it considers real JavaScript, it does so using a formal semantics of JavaScript and it demonstrates failures in informal attempts to tackle the same problem.

3.2.3 Type Systems

Type systems have proved to be a popular method of verifying information flow security. Their advantages are that they tend to lead to syntax-directed static analyses and that a program's type can serve as a compact and readable description of its information flow behaviour. Observe that static analyses are almost always preferable to dynamic analyses, as they consider all possible execution paths; this is important because information flow security is a property of all execution paths. This contrasts, for example, with the situation for subtyping in an object-oriented language, where the choice between dynamic and static analysis usually reduces to a trade-off between precision and compile-time error detection.

Classical noninterference. The study of static analysis for information flow security can essentially be traced back to Denning and Denning, who developed a simple static information flow analysis that rejected programs with flows violating a security policy [DD77]. A similar type system for statically verifying noninterference, for the simple imperative language considered in Section 2.3.5, is shown in Figure 3.2. Use of the typing rules requires a permissions labelling $\Gamma \mathrel{\dot{\sqsupseteq}} P$ of all variables used in the program. If one interprets P as specifying the confidentiality of the input/output variables, then Γ specifies confidentiality

Security Type System

Typing expressions: $\Gamma \vdash e : l$

$$\frac{}{\Gamma \vdash n : \bot}\ (\textsc{Const})$$

$$\frac{}{\Gamma \vdash v : \Gamma(v)}\ (\textsc{Var})$$

$$\frac{\Gamma \vdash e_1 : l_1 \qquad \Gamma \vdash e_2 : l_2}{\Gamma \vdash e_1 \oplus e_2 : l_1 \sqcup l_2}\ (\textsc{Exp})$$

Typing commands: $\Gamma \vdash c : pc$

$$\frac{}{\Gamma \vdash \textbf{skip} : pc}\ (\textsc{Skip})$$

$$\frac{\Gamma \vdash c_1 : pc \qquad \Gamma \vdash c_2 : pc}{\Gamma \vdash c_1; c_2 : pc}\ (\textsc{Seq})$$

$$\frac{\Gamma \vdash e : l \qquad l \sqsubseteq \Gamma(v) \qquad pc \sqsubseteq \Gamma(v)}{\Gamma \vdash v := e : pc}\ (\textsc{Ass})$$

$$\frac{\Gamma \vdash e : l \qquad \Gamma \vdash c_1 : pc \sqcup l \qquad \Gamma \vdash c_2 : pc \sqcup l}{\Gamma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : pc}\ (\textsc{If})$$

$$\frac{\Gamma \vdash e : l \qquad \Gamma \vdash c : pc \sqcup l}{\Gamma \vdash \textbf{while } e \textbf{ do } c : pc}\ (\textsc{While})$$

Derived rule:

$$\frac{\Gamma \vdash c : pc \qquad pc' \sqsubseteq pc}{\Gamma \vdash c : pc'}\ (\textsc{Sub})$$

Typing programs: $\Gamma \vdash c$

$$\Gamma \vdash c \overset{\text{def}}{\Leftrightarrow} \Gamma \vdash c : \bot$$

Figure 3.2: A type system for statically verifying classical noninterference

levels for the temporary variables used in the computation; there may be more than one Γ under which the program type-checks.

The relation $\Gamma \vdash e : l$, defined by the rules Const, Var and Exp, types expressions. If $\Gamma \vdash e : l$, then the least upper bound of the labels of all variables in e (as assigned by Γ) is l.

The relation $\Gamma \vdash c : pc$ types commands. A judgement of this form means that command c has no side-effects visible below level pc and that c never writes a high value to a low variable. The name of pc may be a little confusing, as "pc" usually stands for "program counter", which (in the context of assembly language) is a register that holds the address of the program instruction that will be executed next. The convention of using this name originates from the idea of augmenting the program counter with a security label in order to track implicit flows, as discussed in Section 3.2.1.

The rule Skip is obvious: **skip** has no side-effects, so it is typeable at any level. Rule Seq says that if two commands with high side-effects only are sequenced, the combined command also has high side-effects only. If and While are similar to each other: they require that when execution of a command depends on a high value, that command has no low side-effects; this guards against implicit flows. Like Seq, they propagate any lower bound on side-effects to their component sub-commands. Finally, the rule Ass prevents both explicit and implicit flows from assignments: the condition on l prevents writes of high values to low variables, which would cause explicit flows; the condition on pc prevents writes to low variables within branches dependent on high values, which would cause implicit flows.

The typing rules for commands satisfy a *subsumption* property, as shown in the derived rule Sub: any command typeable at a high level is also typeable at a low level. Thus it is not really a restriction that, for example, the Seq rule requires both sub-commands to be typeable at the same level.

A program c is well-typed if it is typeable at level $\bot$ (where $\bot$ is the bottom element of the lattice of security labels); that is, it never writes a high value to a low variable. The type system is sound: a well-typed program satisfies termination-insensitive noninterference; proofs for similar systems are given for both big-step [VIS96] and small-step operational semantics [Mye09] in existing work.

Of course, the type system is not complete: there are programs that satisfy termination-insensitive noninterference, but are not typeable. Programs 6 and 7 in Figure 2.2 are examples of this; they fail to type-check because they contain low writes in high branches. Of the other example programs, 2 and 5 type-check correctly and 1, 3 and 4 fail to type-check as expected and desired.

Various authors have produced type systems for languages with richer sets of features. For example, in the setting of reactive systems, Bohannon and others [BPS+09] produce a similar type system for a similar language in a reactive setting, which features explicit input and output. Correspondingly, their formulation of noninterference considers not just the initial and final states of a program, but streams of input and output; their intent is to model something close to sequentially executing event handlers, as one might find in an interactive webpage using JavaScript.

Most notably, Pottier and Simonet [PS03] describe a type system that verifies termination-insensitive noninterference for an ML-like language with recursive functions, mutable references, first-class exceptions, algebraic datatypes and polymorphism. Unfortunately, the requirement that the program analysed follow a strict type discipline makes it impractical to apply these ideas to dynamically typed languages such as JavaScript. Perhaps as a consequence, information flow in untyped and dynamically typed languages is relatively poorly understood. However, there has been some work on how to apply information flow type systems to dynamically typed languages through the use of gradual types [FT13, DF11].

Observational determinism. Zdancewic and Myers [ZM03] prove observational determinism for a language allowing an unbounded number of threads and synchronisation and message-passing via first-class channels. They use a combination of a static *race freedom* analysis (to check for *timing channels*) and a type system (to check for explicit and implicit flows). The race freedom analysis requires that, if there are two memory accesses to the same location, either or both of which are a writes, then there must be a fixed ordering on those accesses. Because of race freedom, the type system then only needs to enforce similarity between traces of each low variable individually, rather than all of them simultaneously.

Terauchi [Ter08] presents a novel type system for proving observational determinism in a simple imperative language with threads and synchronisation over channels, which can verify some *non-confluent* programs. The type system is based around the idea of *fractional capabilities*, which were introduced by Boyland as a means of checking the safety of optimisations that reorder or parallelise code [Boy03]. In order to read from or write to a memory location, a thread must possess a relevant token or capability. Capabilities are represented by non-negative numbers. A positive capability allows reading, while a capability of 1 or more allows writing. If the sum of capabilities for a particular resource over all threads is limited to 1, then a memory location may only ever have one writer, but may have multiple readers.

Capabilities can be passed between threads over channels, but receiving capabilities requires a capability of 1 associated with that channel. Channel capabilities can also be passed over channels.

The type system also has rules in the usual style for detecting implicit and explicit flows, but uses the capabilities system to check determinism only where it is observable. Type-checking is implemented in three stages: the first stage generates constraints and resolves some of them by unification; the second and third stages resolve capability constraints by *linear programming*. Type-checking is possible in polynomial time.

3.2.4 Temporal Logic Model-Checking

As a language's mechanisms for indirection become more powerful, it becomes harder to reason about which values may be accessed at different places within the program. For example, JavaScript allows selection of object properties on the basis of any string computation. Thus, if an object has properties of different levels and a property of the object is chosen through computation, it is in general undecidable what the confidentiality level of the chosen property can be. This makes it difficult to perform a static information flow security analysis of such a program that is both sound and sufficiently precise to be useful.

Huisman and others [HWS06] sidestep this problem by expressing observational determinism as a CTL* or modal μ-calculus property of a trace of a program running in parallel with itself. After translating a program into a model that produces a suitable trace, they then use an existing model-checker on that model to verify observational determinism for the program. The advantage of this approach is that the heuristics implemented in the existing model-checker may enable it to verify many instances of problems that are in general undecidable.

In constructing the model, the *parallel self-composition* used permits either copy of the program to make a step by itself, or for both programs to make a step in synchronisation. The atomic propositions in the model record which variables changed in each copy of the program and whether the copies are equal. The CTL* property selects those traces where the two copies always update a variable at the same time. Within these traces it requires that: the variables remain equal; neither process unfairly steps forever by itself.

It is not clear to what extent this technique can be applied to *infinite state programs*, particularly where there is memory allocation, as then a naive interpretation of the construction would require an infinite number of atomic propositions.

3.2.5 Hybrid Approaches

Not all analyses fit neatly into the categorisation of static or dynamic. As previously mentioned, some dynamic analyses based on the *no sensitive upgrade* strategy use some static analysis to identify potential implicit flows before taking a branch, increasing the range of permitted programs.

Chugh and others consider how to perform a practical information flow analysis for a JavaScript-like language [CMJL09]. The language they consider, called Core JavaScript, has many of the features that make JavaScript interesting, including first-class functions, mutable references, objects implemented as associative arrays with prototype-based inheritance and an **eval** construct that executes a string as code. It would be misleading to claim that the language is a subset of JavaScript. Nonetheless, the authors use their work on Core JavaScript as the basis for an implementation that handles full JavaScript.

Their staged analysis has two distinct phases. The first phase occurs in advance of the user viewing the webpage, possibly on the web server, so it is permissible for it to be slow. It takes an information flow policy and performs a constraint-based flow analysis on the statically available JavaScript of a webpage; the analysis produces a set of residual constraints that must be satisfied by the dynamically loaded JavaScript of the webpage. The second phase occurs in the browser, when the webpage is viewed. Whenever new code is loaded, for example by an **eval**, it checks that the code satisfies the residual constraints; if it does not, it prevents the code from running. The details of the flow policies and the form of the constraints are motivated by the practical consideration that the second phase must be fast, so as not to slow the browser noticeably.

The flow policy comprises a sequence of integrity and confidentiality policies. A confidentiality policy states that the contents of a statically named variable must not influence any variable in a particular block of loaded code; this sort of policy is used to stop information leaks. An integrity policy states that no variable in a block of loaded code may influence the value of a statically named variable; this sort of policy can block code from changing the address viewed in a browser.

Observe that these policies are fairly simple; they are more like isolation policies. However, because of this, the residual constraints that must be checked in the second phase in order to enforce these policies are simple and essentially syntactic; thus the check is fast.

In contrast with the dynamic analyses that are enhanced with static analysis, this is a static analysis that is enhanced with dynamic analysis. However, in the context of this thesis on verifying metaprogramming, I would classify it as a dynamic analysis, as it makes no attempt to handle **eval** statically.

3.3 Verifying JavaScript

There are a number of mainly syntactic tools to assist with JavaScript development. For example, JSLint and JSHint are tools for enforcing coding standards in JavaScript. They parse JavaScript code and produce warnings if they find certain features in code, such as undeclared variables. While these are useful, I am interested in more advanced techniques that may be used as the basis for an information flow analysis. Two such techniques are dynamic type systems and control flow analysis.

3.3.1 Dynamic Type Systems

Type systems are a classic technique for static enforcement of program properties. As noted earlier, they are difficult to apply to a dynamically typed language like JavaScript, which does not follow a strict type discipline. Nonetheless, there have been some attempts to use types for reasoning about dynamically typed languages.

Soft typing. Cartwright and Fagan show that it is possible to use a static analysis on dynamically typed programs to omit run-time checks (and storage of the associated typing information) where safe, offering similar performance to static typing in many cases [CF91]. However, their technique does not stop typing errors at compile time.

Soft typing was one of the inspirations for the type system used in TAJS (Type Analyser for JavaScript) [JMT09], which combines a range of ideas from dynamic type systems and abstract interpretation to produce a practical JavaScript analysis. Indeed, TAJS appears to be one of the most advanced analysis tools for real-world JavaScript currently available. Keil and Thiemann have extended TAJS to perform dependency analysis [KT13], which can be used to prove noninterference.

Gradual typing. A new program is often initially written in a dynamically typed language. As the program matures and its design stabilises, sometimes it is rewritten in a statically typed language to gain compile-time guarantees of its behaviour. The rewriting process can be arduous and erroneous. Furthermore, if the program's requirements later change, its developers might wish initially to implement the changes in a dynamically typed language for convenience, which is difficult once the program has been rewritten.

It would be preferable if the initial programming language could incorporate both dynamically typed and statically typed code; Siek and Taha propose to implement this using gradual typing [ST07]. They consider a language with objects and structural subtyping and

describe how to implement static type-checking where possible in the presence of optional type annotations.

More recently, gradual typing has been applied to JavaScript, with the resulting language being called TypeScript. One difficulty with applying gradual typing to Web applications is that the code is often under continual development, so the design never stabilises. Consequently, there might never be a good time to introduce types even for part of the program.

By design, gradual type systems usually permit unsafe behaviours in the parts of a program that cannot be statically typed, which makes them less suitable for security-related analyses. Some recent work by Swamy and others [SFR+14] considers how to obtain security guarantees within the well-typed parts of a TypeScript program, regardless of the behaviour of the rest of the program.

Dynamic inference of static types. An and others [ACFH11] consider how to infer types for programs written in the dynamically typed language Ruby, using information gathered from test suites. The assumption here is that types are often used in a way that is compatible with static typing, even though it is not required. They implement wrappers around values in the language that track how the values are used. From this, they generate constraints on the types of the values, which they solve using a constraint solver.

The technique has the property that, provided every path of execution through a function (not the whole program) is covered at least once, the types inferred will be sound. Obviously, there is a limit to how precise types may be within their type system. The authors comment that their technique is particularly useful for languages with difficult or poorly specified semantics that make static analysis hard.

3.3.2 Control Flow Analysis

A control flow analysis (CFA) deduces an approximation to the flow of execution within a program [Mid12]. For a functional language, where higher order functions can influence the flow of control, this typically involves determining an approximation to the functions that may result from evaluation of expressions in a program. This family of analyses is a good match for JavaScript because they do not rely on a strict type discipline, although they can often be expressed in the form of a type system.

Perhaps the best-known variant is 0CFA [Shi88], which is relatively coarse (and tracks no information about calling context), but fast enough to be practical for large programs: it can be computed in time $O(n^3)$ using a suitable data structure. k-CFA extends 0CFA

by tracking k levels of calling context, which increases the precision, but is EXPTIME-complete even for $k = 1$ [HM13]. This is usually intractable for programs written in functional languages, but can remain practical for programs written in object-oriented languages, where the restricted way in which closures are generated means that the analysis remains computable in polynomial time. This observation led Might and others to develop m-CFA [MSH13], which tracks the top m stack frames of context. This seems to give precision close to k-CFA in practice, but in polynomial time, even for functional languages.

The analysis CFA2 was developed specifically for JavaScript [VS11]; it increases the precision of 0CFA by modelling function calls and returns as using a pushdown abstraction, while maintaining reasonable performance; it runs in exponential time in the worst case, but can be coarsened to run in polynomial time. Paradoxically, a higher degree of precision can make an analysis faster, as it may eliminate the need to analyse some spurious execution paths. CFA2 is the basis of the JavaScript analysis tool jsctags (formerly DoctorJS), which is integrated with various IDEs.

3.4 Verifying Metaprogramming

There has been relatively little work on analysing **eval** and other forms of metaprogramming. Existing work can be roughly split into that which handles staged metaprogramming and that which handles **eval**; there is also some work that analyses the structure of **eval**ed code, but not its behaviour.

3.4.1 Verifying Staged Metaprogramming

I have already discussed the work of Berger and Tratt on producing a Hoare-style logic for PCF_{DP} [BT10]. As noted, how to produce an automated analysis from such a logic is an open problem.

Inoue and Taha [IT12] consider the problem of reasoning about staged programs; in particular, they identify equivalences that fail to hold in the presence of staging, and develop a notion of bisimulation that can be used to prove extensionality of function abstractions, and work around some of the failing equivalences. Once again, while this is a useful foundation for reasoning about programs with metaprogramming, it does not really suggest how to produce an automated analysis.

Control flow analysis for a two-staged language has been investigated by Kim and others [KLL+09]. Their approach is based on abstract interpretation, However, they make a number of simplifications, including forbidding execution of code from generating more code. They put particular emphasis on inferring an over-approximation of all possible

pieces of code to which a code quotation may evaluate, but do not discuss how (if at all) they reason about the execution behaviour of the code thus generated.

Choi and others [CAYT11] propose a more general framework for static analysis of multi-staged programs, which is based on an unstaging translation that replaces staging constructs with function abstractions and applications. Under certain conditions, analysis results for the unstaged program can then be translated back to its staged version.

3.4.2 Verifying Eval

The most advanced tool for analysing **eval** is probably Unevalizer [JJM12], which is based around the JavaScript analysis tool TAJS. The aim of the tool is to analyse and remove uses of **eval**, replacing it with code that does not use metaprogramming. In addition to being able to transform constant strings, the tool can recognise certain fixed patterns of variable string usage, such as concatenating an argument into a function or an object access. The tool is practically motivated, meaning that it can handle full JavaScript taken from popular websites.

A different approach is taken by Meawad and others [MRMV12]. Their tool Evalorizer uses a proxy to intercept and log uses of JavaScript **eval** that occur during web browsing. It then advises a developer on how best to replace them with code that does not use **eval**. In order to do this safely and sensibly, it must first categorise the dynamically gathered code strings according to their structure and content. The aim of the tool is to aid migration of a website away from **eval**, with interaction from the developer, rather than to analyse its behaviour fully and automatically.

Some unpublished work by Charlton [Cha11], following a similar idea to Berger and Tratt [BT10], explores how to create a Hoare-style program logic for a language with **eval**. The main idea is to use a grammar-based string analysis to produce an over-approximation of the strings that may be passed to **eval**, then to use rules for induction over productions of the grammar. However, there is no suggestion of how to progress from this logic to an automated analysis.

3.4.3 String Analysis of Code

A common source of security vulnerabilities in Web applications is accidental execution of untrusted code resulting from careless splicing of user-supplied data into code templates. This is particularly common when building SQL queries through string concatenation in the server-side scripting language PHP.

String analyses are a popular method for detecting such vulnerabilities. For example, Wassermann and Su use a grammar-based analysis to detect when concatenation of untrusted data may change the syntactic structure of an SQL query [WS07] in PHP. Similarly, Doh and others [DKS11] use an abstract domain based on the structure of a parsing algorithm to check that HTML strings generated in PHP match a specified grammar.

Kudzu [SAH+10] is a tool that dynamically generates traces of JavaScript programs, then executes them symbolically to determine what other values variables may feasibly take when following the same trace. As JavaScript flow control often depends on the value of string variables (for example, from XML or from HTML form controls), this symbolic execution requires a string constraint solver, Kaluza. Kaluza is a custom-built solver for bounded integer and string variables with regular expression constraints, which itself uses an SMT solver for bit vectors. The combined system can detect when a user-supplied string might be **eval**ed and whether that string belongs to a particular regular language; this is sufficient to detect certain patterns of attack.

However, none of this work analyses the behaviour of the generated code. As Choi and others [CAYT11] point out, it is difficult to analyse the behaviour of code generated by a grammar-based abstraction, as it is not clear how to concretise it in a precise way.

Chapter 4

Outline of Research

4.1 Objectives

The new arena of Web applications presents many interesting challenges for information flow analysis. While there is an extensive body of research on information flow in statically typed languages [PS03], there is little tackling dynamically typed languages. The semantics of JavaScript are complex and poorly understood [MMT08], which makes any formal analysis difficult. Web applications frequently comprise code from multiple sources (including libraries and adverts), written by multiple authors in an ad-hoc style. They are often interactive (so cannot be viewed as a single execution with inputs and outputs) and it might not be known in advance which code will be loaded.

The **eval** construct of JavaScript takes a string and parses and executes it as program code. For several years, authors of static analyses for JavaScript argued that they could ignore it because it was rarely used or used only in trivial ways [GL09]. Their real reason was probably that analysis of such a powerful construct seemed utterly hopeless, particularly when considering the language's lack of protection mechanisms, as it allows arbitrary behaviours to result from an unstructured data value.

However, a recent survey [RHBV11] showed that a majority of the most popular websites use **eval** and that they use it in many varied (and often misguided) ways. Uses of **eval** vary widely from straightforward (loading data via JSON) through ill-informed (accessing fields of an object without using array notation) to subtle (changing scoping behaviour) and complex (emulating higher order functions).

There has been a surge of interest in techniques for dealing with **eval**. Some authors argue that static analysis of **eval** really is hopeless and instead develop dynamic analyses that monitor execution of **eval**ed code [HS12, SR09], enforcing security policies by terminating the program immediately before a violation. Indeed, this is probably the most reliable and

mature technology currently available [Fen74], but given the benefits of catching violations early in development or deployment, the attitude seems somewhat fatalistic.

Furthermore, there are fundamental limits to dynamic monitors: they can only enforce safety properties. For example, *noninterference* [GM82], which is one of the best known information flow security properties, is not a safety property [TA05]. Consequently, it can never be *precisely* enforced by a monitor [RS10], although it is possible to enforce more restrictive properties such as isolation of untrusted code [MT09]. I think that it is important to develop static techniques for analysing this notorious construct. In order to deal with this, I propose to investigate security analyses for languages with more principled forms of dynamic code generation.

Some recent work [JJM12, MRMV12] considers how to determine what arguments may be passed to uses of **eval** and hence how to replace occurrences of **eval** with code that does not use it. However, these approaches seem limited to handling certain fixed patterns of usage. I argue that reasoning about **eval** could better be viewed as two distinct problems: the first is determining what code may be executed; the second is reasoning about the behaviour of that code. As each is difficult in its own right, we are more likely to be successful if we focus on one problem at a time.

So that I might reasonably work formally, I show how to analyse **eval** in a simplified language called SLamJS. The language is heavily influenced by λ_{JS}, a "core calculus" for JavaScript [GSK10]. Like JavaScript, SLamJS is dynamically typed and features first-class functions and objects with prototype-based inheritance. Like JavaScript, it allows code to be constructed, passed around and executed at run-time. Unlike JavaScript, this is achieved using Lisp-style code quotations or *staged metaprogramming* rather than code strings [CAYT11]. The work of Jensen and others [JJM12] indicates that real-world usage of **eval** is often of a form that could be expressed using code quotations.

I begin the presentation of my research in Section 4.2 by explaining what staged metaprogramming is and why it is a suitable formalism for most uses of **eval**. Next, in Section 4.3, I present the syntax and semantics of SLamJS, including its staged metaprogramming constructs. Then, in Chapter 5, I explain what information flow means in SLamJS and formulate an information flow analysis for the language. Analysis of programs with executable code quotations is an important step towards analysis of programs with executable code strings, but in order to demonstrate applicability to JavaScript's **eval**, we need to be able to transform uses of **eval** into staged metaprogramming; I present a technique for this in Chapter 6.

4.2 Staged Metaprogramming

4.2.1 Constructs of Staged Metaprogramming

The language Lisp allows programs to construct code templates as data values, splice them together and run the resulting code. I refer to these features collectively as *staged metaprogramming*. Following the language λ_S of Choi and others [CAYT11], we can add these features to a programming language with three constructs:

- **box** e turns the expression e into a code value; it does not evaluate e.
- **run** e evaluates the code value e.
- **unbox** e may only occur inside a **box** expression. It forces evaluation of e; the resulting code value is spliced into the surrounding code template.

The language SLamJS, which will be introduced in Section 4.3, supports staged metaprogramming with **box**, **unbox** and **run**.

For example, in SLamJS, **box** $(x * 2)$ evaluates to a code value, which can be stored in a variable or passed to a function like any other value: **let** $y =$ **box** $(x * 2)$. When that code value is run, for example with **run** y, it evaluates to double the value of x in the current scope, which may be different from the scope in which the code value was defined. If we **let** $z =$ **box** 2, we can create the same code value with **let** $y =$ **box** $(x * ($**unbox** $z))$.

4.2.2 Expressivity of Metaprogramming

Although JavaScript's **eval** is only a single language construct, metaprogramming can increase the expressivity of a programming language in several distinct ways. In most cases, we find the expressivity is matched by staged metaprogramming:

Composing Code Templates In JavaScript, code templates can be encoded as strings and spliced together using string concatenation. For example:

```
var f = function(z) { return 3 * z };
var y = "2";
var x = "f(" + y + ")";
eval(x);
```

Adding **box**, **unbox** and **run** to JavaScript, we could express this as:

```
var f = function(z) { return 3 * z };
var y = box(2);
var x = box(f(unbox(y)));
run(x);
```

We will see how to automate this transformation in Chapter 6.

Changing scoping. When code is run with **eval**, it is evaluated in the scope of the **eval**, not the scope in which the code was defined (as is the case for functions). This example (left) returns 2, not 1; again, it can be modelled in staged metaprogramming (right):

```
(function(x){
 return (function(y) {
  return (function(x) {
   return eval(y)
  })(2)
 })("x")
})(1)
```

```
(function(x){
 return (function(y) {
  return (function(x) {
   return run(y)
  })(2)
 })(box(x))
})(1)
```

The precise choice of scoping rules may vary between languages. For example, λ_S has less permissive variable capture rules than SLamJS; the latter's are closer to those of JavaScript. However, JavaScript's **eval** has the peculiarity (copied by neither language) that, if it is called via an alias, it evaluates in the global scope, rather than the local one.

Changing evaluation order. It is folklore that metaprogramming facilities allow a programmer to mix uses of call-by-name and call-by-value evaluation in a single program. This intuition is made more concrete by Inoue and Taha [IT12]. While this is expressible in both JavaScript **eval** and staged metaprogramming, the analyses I consider are not sensitive to the evaluation order, so I do not discuss it further here.

Serialisation. In metaprogramming terminology, turning a data value into its corresponding code value is called *lifting* [Tra05]. This is not generally possible in JavaScript: there is no facility to turn a function into its source code. It is possible for primitive data values, as these can be converted into strings. In fact, JavaScript string concatenation does this implicitly.

To model this behaviour in staged metaprogramming, we need an operation **lift** e that evaluates e and, if it evaluates to a primitive value, turns it into the corresponding code

value. (Note that this is in contrast to **box** e, which does not evaluate e.) Using **lift**, this example:

```
var y = 2;
var x = "f(" + y + ")";
```

would become:

```
var y = 2;
var x = box(f(unbox(lift(y))));
```

I do not consider **lift** directly with SLamJS. Fortunately, for primitive datatypes, it can be added to the language and analysed in the same way as basic arithmetic operations, by defining a new primitive operation.

Deserialisation. If code can be treated as a data value, it can be read from files or the network and dynamically loaded using metaprogramming facilities during program execution. Some JavaScript libraries support incremental loading in this way. This is also possible using staged metaprogramming, although it requires support from the language runtime environment to turn data from the network into code values, in much the same way that it already provides support for turning strings passed to **eval** into code.

Static analysis of code using incrementally loaded libraries is difficult because not all of the code will necessarily be available at once. However, this is not a problem peculiar to **eval** or even JavaScript, although the haphazard infrastructure of Web applications does complicate it.

Aside: Reading the popular JSON (JavaScript Object Notation) [Cro06] format is one simple example of deserialisation; JSON encodes data as the JavaScript source code string that evaluates to that data. Because of the risk of evaluating code from an untrusted source, current "best practice" is not to **eval** JSON code, but instead to use a library function *JSON*.*parse*(). Perhaps a new best practice should be not to manipulate code templates with string concatenation, but to use (as yet unwritten) library functions *Stage*.*box*() and *Stage*.*unbox*()? This would provide a path for developers using **eval** to migrate to staged metaprogramming, making it less likely that they would unwittingly introduce security holes through sloppy use of **eval**. Indeed, some current work on adding "hygienic macros" to JavaScript is making this a possibility [DFHF14].

Booleans	b	::=	**true** | **false**
Strings	s	$\in$	*String*
Numbers	n	$\in$	*Number*
Names	x	$\in$	*Name*
Constants	k	::=	**undef** | **null** | b | s | n
Expressions	e	::=	k | $\{\overline{s : e}\}$ | x | **fun**$(x)\{e\}$ | $e(e)$ | **box** e | **unbox** e | **run** e
		|	**if**$(e)\{e\}$ **else**$\{e\}$ | $e[e]$ | $e[e] = e$ | **del** $e[e]$ | (e, ρ) | **run** e **in** ρ
Values…	v, v^0	::=	$(\textbf{fun}(x)\{e\}, \rho)$ … at stage 0 only
…at any stage	v^n	::=	k | $\{\overline{s : v^n}\}$ | $(\textbf{box}\ v^{n+1})$
…at higher	v^{n+1}	::=	x | $(\textbf{fun}(x)\{v^{n+1}\})$ | $(v^{n+1}(v^{n+1}))$ | $(\textbf{run}\ v^{n+1})$
stages only		|	$(\textbf{if}(v^{n+1})\{v^{n+1}\}\ \textbf{else}\{v^{n+1}\})$
		|	$(v^{n+1}[v^{n+1}])$ | $(v^{n+1}[v^{n+1}] = v^{n+1})$ | $(\textbf{del}\ v^{n+1}[v^{n+1}])$
	v^{n+2}	::=	$(\textbf{unbox}\ v^{n+1})$
Environments	ρ	$\in$	$\mathit{Name} \xrightarrow{\text{fin}} v^0$

Figure 4.1: Syntax of SLamJS

Intensional uses. Some metaprogramming formalisms, such as the SF calculus of Given-Wilson and Jay [GJ11], allow code values to be examined internally and broken into their component parts. This is not possible in the staged metaprogramming formalism. It is possible in JavaScript but is rare, presumably because it is difficult to achieve using unstructured strings. A JSON parser *written in JavaScript* using regexes and substring operations would be an example of an intensional use.

4.3 The Language SLamJS

4.3.1 Syntax

SLamJS is a functional language with atomic constants, records, branching, first-class functions and staged metaprogramming; the syntax is given in Figure 4.1.

The language has five types of atomic constant: booleans, strings, numbers and two special values (**undef** and **null**) to indicate undefined or null values. A record $\{\overline{s : v}\}$ is a finite mapping from fields (named by strings) to values. Fields can be read ($e[e]$), updated or replaced ($e[e] = e$) and deleted (**del** $e[e]$). Records support prototype-based lookup: a read from an undefined field of a record is redirected to the corresponding field on the record held in its `"__proto__"` field, if there is one.

Branching on boolean values is enabled by the **if**$(e)\{e\}$ **else**$\{e\}$ construct. Functions can be defined (**fun**$(x)\{e\}$) and applied ($e(e)$).

Staged metaprogramming is supported through use of the **box**, **unbox** and **run** constructs in the style of Choi and others [CAYT11]. **box** e_1 turns e_1 into a "quoted" or "boxed" code value, which can be executed using **run**. The use of **unbox** e_2 within a boxed expression e_1 forces evaluation of e_2 to a boxed value, which is spliced into e_1 *before it becomes a boxed value*.

Expressions of the form (e, ρ) and **run** e **in** ρ only arise as intermediate terms during execution: the former represents an explicit substitution [HM11, KYC06] where all free variables of the expression e are given their value by the environment ρ; the latter represents an expression to be unboxed and evaluated in environment ρ.

Values exist at all stages. Constants, records with constant fields and constant code quotations are values v^n at every stage n; closures are only values v^0 at stage zero. Other constructs may be values at higher stages (v^{n+1}, v^{n+2} for $n \geq 0$), provided that their subexpressions are values at the appropriate stage. We generally omit the stage superscript for values of stage zero (writing v instead of v^0).

4.3.2 Semantics

I give a small-step operational semantics with evaluation contexts and explicit substitutions for SLamJS. There are two reduction relations, $\overset{n}{\dashrightarrow}$ and $\overset{n}{\rightarrow}$, each annotated with a level n. The former is for top-level reduction, while the latter is for evaluation under a context.

Evaluation contexts. In a staged setting, evaluation contexts may straddle stage boundaries, hence they are annotated with stage subscripts and superscripts. A context C_n^m denotes a hole at stage n inside an expression at stage m. For a context C_n^m and an expression e, we denote by $C_n^m\langle e\rangle$ the expression obtained by plugging e into the hole contained in C_n^m. The grammar of evaluation contexts is given in Figure 4.2.

Reduction rules. Top-level reduction rules fall into two categories: environment propagation rules for pushing explicit substitutions inwards (Figure 4.3), and proper reduction rules (Figure 4.4). Note that explicit substitutions only apply at stage zero, hence (x, ρ) evaluates to x at level $n + 1$ without looking up x in ρ. Furthermore, observe that (**run** e, ρ) pushes its environment into e, allowing boxed code values to capture variables from outside.

The proper reduction rules are also quite standard [CAYT11], except for the field access rules, which are designed to mimic JavaScript semantics as far as possible.

In particular, every record is expected to have a `"_proto_"` field, which holds either the value **null** or another record, giving rise to a chain of prototype objects that ultimately ends

$$
\begin{array}{rcll}
C_n^m & ::= & [\,] & \in\ C_n^n \\
& | & (\{\overline{s : v^m}, s : C_n^m, \overline{s : e}\}) & \in\ C_n^m \\
& | & (\textbf{fun}(x)\{C_n^{m+1}\}) & \in\ C_n^{m+1} \\
& | & (C_n^m(e)) & \in\ C_n^m \\
& | & (v^m(C_n^m)) & \in\ C_n^m \\
& | & (\textbf{box}\ C_n^{m+1}) & \in\ C_n^m \\
& | & (\textbf{unbox}\ C_n^m) & \in\ C_n^{m+1} \\
& | & (\textbf{run}\ C_n^m) & \in\ C_n^m \\
& | & (\textbf{if}(C_n^m)\{e\}\,\textbf{else}\{e\}) & \in\ C_n^m \\
& | & (\textbf{if}(v^{m+1})\{C_n^{m+1}\}\,\textbf{else}\{e\}) & \in\ C_n^{m+1} \\
& | & (\textbf{if}(v^{m+1})\{v^{m+1}\}\,\textbf{else}\{C_n^{m+1}\}) & \in\ C_n^{m+1} \\
& | & (C_n^m[e]) & \in\ C_n^m \\
& | & (v^m[C_n^m]) & \in\ C_n^m \\
& | & (C_n^m[e] = e) & \in\ C_n^m \\
& | & (v^m[C_n^m] = e) & \in\ C_n^m \\
& | & (v^m[v^m] = C_n^m) & \in\ C_n^m \\
& | & (\textbf{del}\ C_n^m[e]) & \in\ C_n^m \\
& | & (\textbf{del}\ v^m[C_n^m]) & \in\ C_n^m \\
& | & (\textbf{run}\ C_n^m\ \textbf{in}\ \rho) & \in\ C_n^m
\end{array}
$$

Figure 4.2: Evaluation contexts

$$
\begin{array}{rcl}
(k, \rho) & \overset{n}{\dashrightarrow} & k \\
(\{\overline{s : e}\}, \rho) & \overset{n}{\dashrightarrow} & \{\overline{s : (e, \rho)}\} \\
(x, \rho) & \overset{n+1}{\dashrightarrow} & x \\
(\textbf{fun}(x)\{e\}, \rho) & \overset{n+1}{\dashrightarrow} & (\textbf{fun}(x)\{(e, \rho)\}) \\
(e_1(e_2), \rho) & \overset{n}{\dashrightarrow} & ((e_1, \rho)((e_2, \rho))) \\
(\textbf{box}\ e, \rho) & \overset{n}{\dashrightarrow} & (\textbf{box}\ (e, \rho)) \\
(\textbf{unbox}\ e, \rho) & \overset{n}{\dashrightarrow} & (\textbf{unbox}\ (e, \rho)) \\
(\textbf{run}\ e, \rho) & \overset{0}{\dashrightarrow} & (\textbf{run}\ (e, \rho)\ \textbf{in}\ \rho) \\
(\textbf{run}\ e, \rho) & \overset{n+1}{\dashrightarrow} & (\textbf{run}\ (e, \rho)) \\
(\textbf{if}(e_1)\{e_2\}\,\textbf{else}\{e_3\}, \rho) & \overset{n}{\dashrightarrow} & (\textbf{if}((e_1, \rho))\{(e_2, \rho)\}\,\textbf{else}\{(e_3, \rho)\}) \\
(e_1[e_2], \rho) & \overset{n}{\dashrightarrow} & ((e_1, \rho)[(e_2, \rho)]) \\
(e_1[e_2] = e_3, \rho) & \overset{n}{\dashrightarrow} & ((e_1, \rho)[(e_2, \rho)] = (e_3, \rho)) \\
(\textbf{del}\ e_1[e_2], \rho) & \overset{n}{\dashrightarrow} & (\textbf{del}\ (e_1, \rho)[(e_2, \rho)])
\end{array}
$$

Figure 4.3: Environment propagation rules

(Lookup)	(x, ρ)	$\overset{0}{\dashrightarrow}$	$\rho(x)$	
(Apply)	$((\textbf{fun}(x)\{e\}, \rho)(v))$	$\overset{0}{\dashrightarrow}$	$(e, \rho[x \mapsto v])$	
(Unbox)	$(\textbf{unbox}\ (\textbf{box}\ v^1))$	$\overset{1}{\dashrightarrow}$	(v^1)	
(Run)	$(\textbf{run}\ (\textbf{box}\ v^1)\ \textbf{in}\ \rho)$	$\overset{0}{\dashrightarrow}$	(v^1, ρ)	
(IfTrue)	$(\textbf{if}(\textbf{true})\{e_1\}\ \textbf{else}\{e_2\})$	$\overset{0}{\dashrightarrow}$	e_1	
(IfFalse)	$(\textbf{if}(\textbf{false})\{e_1\}\ \textbf{else}\{e_2\})$	$\overset{0}{\dashrightarrow}$	e_2	
(Read1)	$(\{\overline{s:v}, s_i : v_i, \overline{s:v}'\}[s_i])$	$\overset{0}{\dashrightarrow}$	v_i	
(Read2)	$(\{\overline{s:v}, \texttt{"_proto_"} : \{\overline{s:v}'\}, \overline{s:v}''\}[s_x])$	$\overset{0}{\dashrightarrow}$	$(\{\overline{s:v}'\}[s_x])$	if $s_x \notin \overline{s} \cup \overline{s}''$
(Read3)	$(\{\overline{s:v}, \texttt{"_proto_"} : \textbf{null}, \overline{s:v}''\}[s_x])$	$\overset{0}{\dashrightarrow}$	**undef**	if $s_x \notin \overline{s} \cup \overline{s}''$
(Write1)	$(\{\overline{s:v}, s_i : v_i, \overline{s:v}'\}[s_i] = v_i')$	$\overset{0}{\dashrightarrow}$	$\{\overline{s:v}, s_i : v_i', \overline{s:v}'\}$	
(Write2)	$(\{\overline{s:v}\}[s_x] = v_x)$	$\overset{0}{\dashrightarrow}$	$\{\overline{s:v}, s_x : v_x\}$	if $s_x \notin \overline{s}$
(Del1)	$(\textbf{del}\ \{\overline{s:v}, s_i : v_i, \overline{s:v}'\}[s_i])$	$\overset{0}{\dashrightarrow}$	$\{\overline{s:v}, \overline{s:v}'\}$	
(Del2)	$(\textbf{del}\ \{\overline{s:v}\}[s_x])$	$\overset{0}{\dashrightarrow}$	$\{\overline{s:v}\}$	if $s_x \notin \overline{s}$

Figure 4.4: Proper reduction rules

in **null**. Reading a record field follows this chain by rule (Read2), until the field is either found (Read1), or the top of the chain is reached, where (Read3) yields **undef**. Note that the reduction $\overset{0}{\dashrightarrow}$ can get stuck, for example, when applying a non-function, or branching on a non-boolean.

There is only a single rule for $\xrightarrow{m}$:

$$C_n^m\langle e\rangle \quad \xrightarrow{m} \quad C_n^m\langle e'\rangle \quad \text{if } e \overset{n}{\dashrightarrow} e'$$

We write $\xrightarrow{\sharp}$ for the union over all m of $\xrightarrow{m}$, and $\xrightarrow{\sharp}{}^*$ for its reflexive, transitive closure.

Example 1. Here is an evaluation trace of a simple **if** statement. We use ϵ to stand for the empty environment.

$$\begin{array}{ll} & (\textbf{if}(\textbf{true})\{\textbf{false}\}\ \textbf{else}\{1\}, \epsilon) \\ \overset{0}{\rightarrow} & \textbf{if}((\textbf{true}, \epsilon))\{(\textbf{false}, \epsilon)\}\ \textbf{else}\{(1, \epsilon)\} \\ \overset{0}{\rightarrow} & \textbf{if}(\textbf{true})\{(\textbf{false}, \epsilon)\}\ \textbf{else}\{(1, \epsilon)\} \\ \overset{0}{\rightarrow} & (\textbf{false}, \epsilon) \overset{0}{\rightarrow} \textbf{false} \end{array}$$

Example 2. The staging constructs in SLamJS allow fragments of code to be treated as

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & \ldots \mid \textbf{eval}\ e \\
\text{Contexts} & C_n^m & ::= & \ldots \mid (\textbf{eval}\ C_n^m) \in C_n^m \\
\text{Propagation} & & & (\textbf{eval}\ e, \rho) \overset{n}{\dashrightarrow} (\textbf{eval}\ (e, \rho)) \\
\text{Reduction} & (\textsc{Eval}) & & \textbf{eval}\ k \overset{0}{\dashrightarrow} \mathit{parse}\,(k)
\end{array}
$$

Figure 4.5: Adding **eval** to SLamJS

values and spliced together or evaluated at run-time, as shown in this evaluation trace.

$$
\begin{array}{ll}
 & (\textbf{run}\ (\textbf{box}\ (\textbf{if}(\textbf{unbox}\ (\textbf{box}\ (\textbf{true})))\{\textbf{false}\}\ \textbf{else}\{1\})), \epsilon) \\
\overset{0}{\to} & \textbf{run}\ (\textbf{box}\ (\textbf{if}(\textbf{unbox}\ (\textbf{box}\ (\textbf{true})))\{\textbf{false}\}\ \textbf{else}\{1\}), \epsilon)\ \textbf{in}\ \epsilon \\
\overset{0}{\to}^* & \textbf{run}\ (\textbf{box}\ (\textbf{if}(\textbf{unbox}\ (\textbf{box}\ (\textbf{true})))\{(\textbf{false}, \epsilon)\}\ \textbf{else}\{(1, \epsilon)\}))\ \textbf{in}\ \epsilon \\
\overset{0}{\to} & \textbf{run}\ (\textbf{box}\ (\textbf{if}(\textbf{true})\{(\textbf{false}, \epsilon)\}\ \textbf{else}\{(1, \epsilon)\}))\ \textbf{in}\ \epsilon \\
\overset{0}{\to}^* & \textbf{run}\ (\textbf{box}\ (\textbf{if}(\textbf{true})\{\textbf{false}\}\ \textbf{else}\{1\}))\ \textbf{in}\ \epsilon \\
\overset{0}{\to} & (\textbf{if}(\textbf{true})\{\textbf{false}\}\ \textbf{else}\{1\}, \epsilon) \\
\overset{0}{\to} & \textbf{if}(\textbf{true}, \epsilon)\{(\textbf{false}, \epsilon)\}\ \textbf{else}\{(1, \epsilon)\} \\
\overset{0}{\to} & \textbf{if}(\textbf{true})\{(\textbf{false}, \epsilon)\}\ \textbf{else}\{(1, \epsilon)\} \overset{0}{\to} (\textbf{false}, \epsilon) \overset{0}{\to} \textbf{false}
\end{array}
$$

Example 3. The staging constructs allow variables to be captured by code values originating outside their scope. Here, the code value **box** y is outside the scope of y, but captures it during evaluation.

$$
\begin{array}{ll}
 & (((\textbf{fun}(x)\{(\textbf{fun}(y)\{\textbf{run}\ x\})\})(\textbf{box}\ y))(\textbf{true}), \epsilon) \\
\overset{0}{\to}^* & (\textbf{fun}(y)\{\textbf{run}\ x\}, \langle x \mapsto \textbf{box}\ y\rangle)(\textbf{true}) \\
\overset{0}{\to} & (\textbf{run}\ x, \langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle) \\
\overset{0}{\to} & \textbf{run}\ (x, \langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle)\ \textbf{in}\ \langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle \\
\overset{0}{\to} & \textbf{run}\ (\textbf{box}\ y)\ \textbf{in}\ \langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle \\
\overset{0}{\to} & (y, \langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle) \\
\overset{0}{\to} & \textbf{true}
\end{array}
$$

This useful feature is vital for modelling certain uses of **eval**; the above code corresponds to this JavaScript:

```
((function (x) {return function (y) {return (eval(x));}})("y"))(true);
```

However, the power comes at a price: the usual alpha equivalence property of λ-calculus does not hold in SLamJS [KYC06], which makes reasoning about programs harder.

4.3.3 Eval in SLamJS

So that I may demonstrate how to transform **eval** into staged metaprogramming, I extend SLamJS with an **eval** keyword that takes a string argument, *parse*s it as program code and

executes that code. The extended syntax and semantics are shown in Figure 4.5. Here is an execution trace that uses **eval**:

$$
\begin{array}{rl}
 & (\textbf{fun}(c)\{\textbf{fun}(x)\{\textbf{eval}\ c\}(1)\},\epsilon)((\texttt{"x"},\epsilon)) \\
\xrightarrow{0} & (\textbf{fun}(c)\{\textbf{fun}(x)\{\textbf{eval}\ c\}(1)\},\epsilon)(\texttt{"x"}) \\
\xrightarrow{0} & (\textbf{fun}(x)\{\textbf{eval}\ c\}(1),\langle c\mapsto\texttt{"x"}\rangle) \\
\xrightarrow{0} & (\textbf{fun}(x)\{\textbf{eval}\ c\},\langle c\mapsto\texttt{"x"}\rangle)((1,\langle c\mapsto\texttt{"x"}\rangle)) \\
\xrightarrow{0} & (\textbf{fun}(x)\{\textbf{eval}\ c\},\langle c\mapsto\texttt{"x"}\rangle)(1) \\
\xrightarrow{0} & (\textbf{eval}\ c,\langle x\mapsto 1,c\mapsto\texttt{"x"}\rangle) \\
\xrightarrow{0} & \textbf{eval}\ (c,\langle x\mapsto 1,c\mapsto\texttt{"x"}\rangle)\ \textbf{in}\ \langle x\mapsto 1,c\mapsto\texttt{"x"}\rangle \\
\xrightarrow{0} & \textbf{eval}\ \texttt{"x"}\ \textbf{in}\ \langle x\mapsto 1,c\mapsto\texttt{"x"}\rangle \\
\xrightarrow{0} & (x,\langle x\mapsto 1,c\mapsto\texttt{"x"}\rangle) \\
\xrightarrow{0} & 1
\end{array}
$$

Note how x is captured by parsing the code string "x", even though the string originated from outside the scope of x.

Chapter 5

Verifying Information Flow with Staged Metaprogramming

5.1 Overview

In this chapter, I present a static information flow analysis for SLamJS, a dynamically typed functional language with prototype-based inheritance and staged metaprogramming. I prove its soundness, implement it and test it on various examples designed to show its relevance to proving security properties, such as noninterference, in JavaScript.

To my knowledge, this is the first fully static information flow analysis for a language with staged metaprogramming, and the first formal soundness proof of a CFA-based information flow analysis for a functional programming language.

The analysis is based on the idea of extending a constraint-based formulation of the analysis 0CFA [Shi88] with constraints to track information flow. I believe that the idea could be extended to other CFA-style analyses (such as CFA2 [VS11]) for improved precision.

Before I can define an information flow analysis, I need to define what information flow is. I begin in Section 5.2 by presenting an augmented semantics for SLamJS with dependency markers that track the origin of an expression.

Following Pottier and Conchon [PC00], I use the idea that if information does not flow from a marked expression into a value resulting from evaluation, then erasing that marked expression or replacing it with a dummy value should not affect the result of evaluation. (I use only their proof technique; their type-based analysis is not applicable to SLamJS.) In Section 5.3 I define erasure and establish some results about its behaviour.

The information flow analysis is built on top of a 0CFA-style analysis capable of handling staging constructs. Two variants of such an analysis are explained in Section 5.4; mechanised correctness proofs in Coq are available online.

$$
\begin{array}{lrcl}
(\textsc{Lift-App}) & ((\mathfrak{m} : e), \rho)(v) & \xrightarrow{0} & (\mathfrak{m} : ((e, \rho)(v))) \\
(\textsc{Lift-If}) & (\textbf{if}(\mathfrak{m} : v)\{e_1\}\ \textbf{else}\{e_2\}) & \xrightarrow{0} & (\mathfrak{m} : (\textbf{if}(v)\{e_1\}\ \textbf{else}\{e_2\})) \\
(\textsc{Lift-Unbox}) & \textbf{unbox}\ (\mathfrak{m} : v) & \xrightarrow{1} & (\mathfrak{m} : (\textbf{unbox}\ v)) \\
(\textsc{Lift-RunIn}) & \textbf{run}\ (\mathfrak{m} : v)\ \textbf{in}\ \rho & \xrightarrow{0} & (\mathfrak{m} : (\textbf{run}\ v\ \textbf{in}\ \rho)) \\
(\textsc{Lift-ReadSel}) & (v_1[\mathfrak{m} : v_2]) & \xrightarrow{0} & (\mathfrak{m} : (v_1[v_2])) \\
(\textsc{Lift-ReadRec}) & ((\mathfrak{m} : v_1)[v_2]) & \xrightarrow{0} & (\mathfrak{m} : (v_1[v_2])) \\
(\textsc{Lift-WriteSel}) & (v_1[\mathfrak{m} : v_2] = v_3) & \xrightarrow{0} & (\mathfrak{m} : (v_1[v_2] = v_3)) \\
(\textsc{Lift-WriteRec}) & ((\mathfrak{m} : v_1)[v_2] = v_3) & \xrightarrow{0} & (\mathfrak{m} : (v_1[v_2] = v_3)) \\
(\textsc{Lift-DelSel}) & (\textbf{del}\ v_1[\mathfrak{m} : v_2]) & \xrightarrow{0} & (\mathfrak{m} : (\textbf{del}\ v_1[v_2])) \\
(\textsc{Lift-DelRec}) & (\textbf{del}\ (\mathfrak{m} : v_1)[v_2]) & \xrightarrow{0} & (\mathfrak{m} : (\textbf{del}\ v_1[v_2]))
\end{array}
$$

Figure 5.1: Semantic rules for lifts

In Section 5.5, I present the information flow analysis itself. A key idea in CFA is that control flow influences data flow and vice versa. Information flow depends on control and data flow, but the reverse is not true. Therefore it is possible to treat information flow analysis as an addition to CFA, rather than a completely new combined analysis. I have two versions of the CFA, each of which yields an information flow analysis. I sketch a correctness proof of the simpler analysis; complete mechanised proofs of both are available online [Les15] and in Appendix A.

Finally, in Section 5.6, I prove soundness of the information flow analysis. I also discuss its relationship with noninterference.

5.2 Augmented Semantics of SLamJS

The result of a program can depend on its component values in essentially two different ways. Consider programs operating on two variables l and h. The program $(\textbf{if}(l)\{h\}\ \textbf{else}\{1\})$ may evaluate to the value of h (if l is **true**); we say that there is a *direct flow* from h to the result. Conversely, the program $(\textbf{if}(h)\{\textbf{false}\}\ \textbf{else}\{1\})$ cannot evaluate to h. However, the result of evaluation tells us whether h was **true** or **false** because h influences the control flow of the program; there is an *indirect flow* from h to the result of the program.

In order to track the dependency of a result on its component subexpressions, we augment the language with explicit *dependency markers* [PC00, ABHR99]. We also introduce new rules for lifting markers into their parent expressions to avoid losing information about dependencies. The augmented semantics is not intended for use in the execution of programs; rather, we use it for analysing and reasoning about dependencies in the original

language. We begin by adding markers to the syntax:

$$\begin{array}{llll} \text{Markers} & \mathfrak{m} & \in & \mathit{Marker} \\ \text{Expressions} & e & ::= & \ldots \mid (\mathfrak{m} : e) \\ \text{Values} & v^n & ::= & \ldots \mid (\mathfrak{m} : v^n) \end{array}$$

We extend contexts to allow evaluation within a marked expression:

$$C_n^m \quad ::= \quad \ldots \quad \mid \quad (\mathfrak{m} : C_n^m) \quad \in \quad C_n^m$$

We allow propagation of environments within marked expressions:

$$(\mathfrak{m} : e, \rho) \quad \overset{n}{\dashrightarrow} \quad (\mathfrak{m} : (e, \rho))$$

In Figure 5.1 we introduce lifts to maintain a record of indirect flows. Note there is no need for a lift rule on the right of an assignment (such as $v_1[v_2] = (\mathfrak{m} : v_3) \overset{0}{\dashrightarrow} (\mathfrak{m} : v_1[v_2] = v_3)$), since the flow from v_3 is direct.

Example 4. Recall Example 1. Suppose we add markers to each of the components of the **if**. The evaluation trace now becomes:

$$\begin{array}{ll} & (\textbf{if}((\textsc{h} : \textbf{true}))\{(\textsc{l} : \textbf{false})\}\,\textbf{else}\{(\textsc{i} : 1)\}, \epsilon) \\ \overset{0}{\rightarrow}^* & \textbf{if}((\textsc{h} : \textbf{true}))\{((\textsc{l} : \textbf{false}), \epsilon)\}\,\textbf{else}\{((\textsc{i} : 1), \epsilon)\} \\ \overset{0}{\rightarrow} & (\textsc{h} : (\textbf{if}(\textbf{true})\{((\textsc{l} : \textbf{false}), \epsilon)\}\,\textbf{else}\{((\textsc{i} : 1), \epsilon)\})) \\ \overset{0}{\rightarrow} & (\textsc{h} : ((\textsc{l} : \textbf{false}), \epsilon)) \\ \overset{0}{\rightarrow}^* & (\textsc{h} : (\textsc{l} : \textbf{false})) \end{array}$$

Note how the markers H and L in the result indicate that it depends on the marked values (H : **true**) and (L : **false**).

Example 5. Here is an example of marked evaluation with functions:

$$(((\textbf{fun}(x)\{\textsc{i} : (\textbf{fun}(y)\{x\})\})(\textsc{h} : 1))(\textsc{l} : 2), \epsilon) \overset{0}{\rightarrow}^* (\textsc{i} : (\textsc{h} : 1))$$

Observe that the result depends on I because the function (I : (**fun**$(y)\{x\}$)) was used to compute it, but not on L, as (L : 2) is discarded by that function.

Simulation. Consider a function *unmark*, defined in the obvious way, which strips an expression of all markers. Clearly if $\mathit{unmark}(e_1) = f_1 \overset{n}{\rightarrow} f_2$, then for some e_2 such that $e_1 \overset{n}{\rightarrow}^* e_2$, we have $\mathit{unmark}(e_2) = f_2$.

$$
\begin{array}{rcll}
\lfloor _ \rfloor_M & = & _ & \\
\lfloor k \rfloor_M & = & k & \\
\lfloor \{\overline{s : e}\} \rfloor_M & = & \{\overline{s : \lfloor e \rfloor_M}\} & \\
\lfloor x \rfloor_M & = & x & \\
\lfloor \mathbf{fun}(x)\{e\} \rfloor_M & = & \mathbf{fun}(x)\{\lfloor e \rfloor_M\} & \\
\lfloor e_1(e_2) \rfloor_M & = & \lfloor e_1 \rfloor_M(\lfloor e_2 \rfloor_M) & \\
\lfloor \mathbf{box}\ e \rfloor_M & = & \mathbf{box}\ \lfloor e \rfloor_M & \\
\lfloor \mathbf{unbox}\ e \rfloor_M & = & \mathbf{unbox}\ \lfloor e \rfloor_M & \\
\lfloor \mathbf{run}\ e \rfloor_M & = & \mathbf{run}\ \lfloor e \rfloor_M & \\
\lfloor \mathbf{if}(e_1)\{e_2\}\,\mathbf{else}\{e_3\} \rfloor_M & = & \mathbf{if}(\lfloor e_1 \rfloor_M)\{\lfloor e_2 \rfloor_M\}\,\mathbf{else}\{\lfloor e_3 \rfloor_M\} & \\
\lfloor e_1[e_2] \rfloor_M & = & \lfloor e_1 \rfloor_M[\lfloor e_2 \rfloor_M] & \\
\lfloor e_1[e_2] = e_3 \rfloor_M & = & \lfloor e_1 \rfloor_M[\lfloor e_2 \rfloor_M] = \lfloor e_3 \rfloor_M & \\
\lfloor \mathbf{del}\ e_1[e_2] \rfloor_M & = & \mathbf{del}\ \lfloor e_1 \rfloor_M[\lfloor e_2 \rfloor_M] & \\
\lfloor (e, \rho) \rfloor_M & = & (\lfloor e \rfloor_M, \lfloor \rho \rfloor_M) & \\
\lfloor \mathbf{run}\ e\ \mathbf{in}\ \rho \rfloor_M & = & \mathbf{run}\ \lfloor e \rfloor_M\ \mathbf{in}\ \lfloor \rho \rfloor_M & \\
\lfloor \rho \rfloor_M(x) & = & \lfloor \rho(x) \rfloor_M & \\
\lfloor \mathfrak{m} : e \rfloor_M & = & \mathfrak{m} : \lfloor e \rfloor_M & \text{if } \mathfrak{m} \in M \\
\lfloor \mathfrak{m} : e \rfloor_M & = & _ & \text{if } \mathfrak{m} \notin M
\end{array}
$$

Figure 5.2: Definition of $\lfloor e \rfloor_M$, the M-erasure of e

5.3 Erasure and Stability

Erasure. We extend the language with a "hole" that behaves like an unbound variable:

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & \ldots \mid _ \\
\text{Values} & v^n & ::= & \ldots \mid _ \\
\text{Propagation} & & & (_, \rho) \overset{n}{\dashrightarrow} _
\end{array}
$$

Now for $M \subseteq \mathit{Marker}$, define the *$M$-erasure* of e, written $\lfloor e \rfloor_M$, to be: e with any subexpression $(\mathfrak{m} : e')$ where $\mathfrak{m} \notin M$ replaced by $_$. A full definition is given in Figure 5.2.

Prefixing and monotonicity. We say that e_1 is a *prefix* of e_2 or write $e_1 \preccurlyeq e_2$ if replacing some subexpressions of e_2 with $_$ gives e_1.

Evaluation is monotonic with respect to prefixing: if $e_1 \preccurlyeq e_2$ and $e_1 \xrightarrow{\square}{}^* f$, where f contains no $_$, then $e_2 \xrightarrow{\square}{}^* f$.

Lemma 1 (Step Stability)**.** *If $e_1 \overset{n}{\dashrightarrow} e_2$, then either $\lfloor e_1 \rfloor_M \overset{n}{\dashrightarrow} \lfloor e_2 \rfloor_M$ or the reduction rule applied to derive this is a lift (Lift-*) of a marker $\mathfrak{m} \notin M$.*

Proof. By induction over the rules defining $\overset{n}{\dashrightarrow}$. □

Theorem 1 (Stability)**.** *Consider an expression e_1 (which may use _) and a _-free expression e_2 such that $e_1 \xrightarrow{\square}{}^* e_2$. Then for every $M \subseteq$ Marker such that $\lfloor e_2 \rfloor_M = e_2$, it follows that $\lfloor e_1 \rfloor_M \xrightarrow{\square}{}^* \lfloor e_2 \rfloor_M$.*

Proof. Consider any e_2 and M with $\lfloor e_2 \rfloor_M = e_2$. Aim to prove, for any e_1 with $e_1 \xrightarrow{\square}{}^* e_2$, that $\lfloor e_1 \rfloor_M \xrightarrow{\square}{}^* e_2$. Argue by induction over the length k of derivations of $e_1 \xrightarrow{\square}{}^* e_2$.

Base case: $k = 0$. So $e_1 = e_2$. We have $\lfloor e_2 \rfloor_M = e_2$, so trivially $\lfloor e_1 \rfloor_M = e_2$.

Inductive step: $k = k' + 1$. Given $e_1 \xrightarrow{n} e \xrightarrow{\square}{}^{k'} e_2$, aim to prove $\lfloor e_1 \rfloor_M \xrightarrow{\square}{}^* e_2$. Assume by the induction hypothesis that $\lfloor e \rfloor_M \xrightarrow{\square}{}^{k'} e_2$. Let $e_1 = C_n^m\langle f_1 \rangle$ and $e = C_n^m\langle f \rangle$ with $f_1 \overset{n}{\dashrightarrow} f$. Case split on whether $f_1 \overset{n}{\dashrightarrow} f$ is a lift of a marker $\mathfrak{m} \notin M$.

If it is such a lift, then let $f = (\mathfrak{m} : f')$. Now $\lfloor f \rfloor_M = _$, so $\lfloor f \rfloor_M \preccurlyeq \lfloor f_1 \rfloor_M$. Thus $\lfloor C_n^m\langle f \rangle \rfloor_M \preccurlyeq \lfloor C_n^m\langle f_1 \rangle \rfloor_M$; that is, $\lfloor e \rfloor_M \preccurlyeq \lfloor e_1 \rfloor_M$. We already have (from the induction hypothesis) that $\lfloor e \rfloor_M \xrightarrow{\square}{}^{k'} e_2$. Now, applying Monotonicity, we get $\lfloor e_1 \rfloor_M \xrightarrow{\square}{}^* e_2$.

Otherwise, apply the Step Stability Lemma to get $\lfloor f_1 \rfloor_M \overset{n}{\dashrightarrow} \lfloor f \rfloor_M$. It follows that $\lfloor C_n^m\langle f_1 \rangle \rfloor_M \xrightarrow{n} \lfloor C_n^m\langle f \rangle \rfloor_M$; that is, $\lfloor e_1 \rfloor_M \xrightarrow{n} \lfloor e \rfloor_M$. Using the induction hypothesis gives $\lfloor e_1 \rfloor_M \xrightarrow{n} \lfloor e \rfloor_M \xrightarrow{\square}{}^{k'} e_2$, as required. □

Example 6. Recall that in Example 5, the result depended on H and I, but not L. Applying $\lfloor - \rfloor_{\{\mathrm{H},\mathrm{I}\}}$ and evaluating the initial expression gives:

$$(((\textsf{fun}(x)\{\mathrm{I} : (\textsf{fun}(y)\{x\})\})(\mathrm{H} : 1))(_), \epsilon) \xrightarrow{0}{}^* (\mathrm{I} : (\mathrm{H} : 1))$$

That is, the result of evaluation is unchanged.

5.4 0CFA for SLamJS

I use a context-insensitive, flow-insensitive control flow analysis (0CFA [Shi88]) to approximate statically the set of values to which individual expressions in a program may evaluate at runtime.

As far as 0CFA is concerned, the only non-standard feature of SLamJS is its staging constructs. Roughly speaking, **box** and **unbox/run** act like function abstraction and application, except that they use a dynamic (instead of static) scoping discipline. (This intuition is made more precise in work by Choi and others on static analysis of staged programs [CAYT11], where staging constructs are translated into function abstraction and application; I prefer to work directly on the staged language for simplicity.)

I present two variants of 0CFA for SLamJS: a simple, but somewhat imprecise formulation that does not distinguish like-named variables bound by different abstractions, and a more complicated one that does.

Abstract values	$v \in \mathit{AbsVal}$	::=	$\texttt{NULL} \mid \texttt{UNDEF} \mid \texttt{BOOL} \mid \texttt{NUM} \mid \texttt{STR}$
		$\mid$	$\texttt{FUN}(x, e) \mid \texttt{BOX}(e) \mid \texttt{REC}(\ell)$
Abstract variables	$\xi \in \mathit{AbsVar}$	::=	$x \mid \ell.p$
Abstract caches	Γ	:	$\mathit{Label} \to \mathcal{P}(\mathit{AbsVal})$
Abstract environments	ϱ	:	$\mathit{AbsVar} \to \mathcal{P}(\mathit{AbsVal})$

Figure 5.3: Abstract domains (simple analysis)

5.4.1 Simple Analysis

Following Nielson, Nielson and Hankin [NNH99], I formalise the analysis by means of an *acceptability judgement* of the form $\Gamma, \varrho \models e$, where Γ is an *abstract cache* associating sets of abstract values with labelled program points, and ϱ is an *abstract environment* mapping local variables and record fields to sets of abstract values. Intuitively, the purpose of this judgement is to ensure that $\Gamma(\ell)$ soundly over-approximates all possible values to which the expression at program point ℓ can evaluate, and ϱ does the same for variables and record fields.

More precisely, the analysis assumes that all expressions in the program are labelled with labels drawn from a set *Label*. An abstract cache is a mapping $\mathit{Label} \to \mathcal{P}(\mathit{AbsVal})$ associating a set of abstract values with every program point; similarly, an abstract environment $\varrho\colon \mathit{AbsVar} \to \mathcal{P}(\mathit{AbsVal})$ maps abstract variables to sets of abstract values, where an abstract variable is either a simple name x (representing a function parameter), or a field name of the form $\ell.p$, where ℓ is a label representing a record, and p is the name of a field of that record.

The domain of abstract values (Figure 5.3) is mostly standard, with, for example, an abstract value `NULL` to represent the concrete **null** value, an abstract `NUM` value representing any number, and abstract values $\texttt{FUN}(x, e)$, $\texttt{BOX}(e)$ and $\texttt{REC}(\ell)$ representing, respectively, a function value, a quoted piece of code, and a record allocated at program point ℓ. For an abstract environment ϱ and a label ℓ, define $\texttt{proto}(\ell)_\varrho$ to be the smallest set $P \subseteq \mathit{Label}$ such that $\ell \in P$ and for every $p \in P$ and $\texttt{REC}(\ell') \in \varrho(p.\texttt{"_proto_"})$ also $\ell' \in P$.

The acceptability judgement is now defined using syntax-directed rules, which are shown in Figure 5.4.

I write t^ℓ to represent an expression of the syntactic form t, labelled with ℓ. Thus, k^ℓ means an expression consisting of a literal k labelled ℓ, and the first rule simply says that in order for Γ and ϱ to constitute an acceptable analysis of k^ℓ, $\Gamma(\ell)$ must contain the abstract value $\lceil k \rceil$ representing k. Similarly, the second rule requires Γ and ϱ to be consistent in the abstract values they assign to variables and references to them.

$$
\begin{array}{lll}
\Gamma,\varrho \models k^{\ell} & \text{if} & \lceil k \rceil \in \Gamma(\ell) \\
\Gamma,\varrho \models x^{\ell} & \text{if} & \varrho(x) \subseteq \Gamma(\ell) \\
\Gamma,\varrho \models \{\overline{s : e}\}^{\ell} & \text{if} & \forall i.\Gamma,\varrho \models e_i \\
 & \text{and} & \exists \mathtt{REC}(\ell') \in \Gamma(\ell).\forall i.\Gamma(lbl(e_i)) \subseteq \varrho(\ell'.s_i) \\
\Gamma,\varrho \models (\textbf{fun}(x)\{e\})^{\ell} & \text{if} & \Gamma,\varrho \models e \\
 & \text{and} & \exists v \in \Gamma(\ell).\Gamma,\varrho \models v \approx \textbf{fun}(x)\{e\} \\
\Gamma,\varrho \models (t_1^{\ell_1}(t_2^{\ell_2}))^{\ell} & \text{if} & \Gamma,\varrho \models t_1^{\ell_1} \wedge \Gamma,\varrho \models t_2^{\ell_2} \\
 & \text{and} & \forall \mathtt{FUN}(x,t_3^{\ell_3}) \in \Gamma(\ell_1).\Gamma(\ell_2) \subseteq \varrho(x) \wedge \Gamma(\ell_3) \subseteq \Gamma(\ell) \\
\Gamma,\varrho \models (\textbf{box}\ e)^{\ell} & \text{if} & \Gamma,\varrho \models e \\
 & \text{and} & \exists v \in \Gamma(\ell).\Gamma,\varrho \models v \approx \textbf{box}\ e \\
\Gamma,\varrho \models (\textbf{unbox}\ t^{\ell})^{\ell_0} & \text{if} & \Gamma,\varrho \models t^{\ell} \\
 & \text{and} & \forall \mathtt{BOX}(t'^{\ell'}) \in \Gamma(\ell).\Gamma(\ell') \subseteq \Gamma(\ell_0) \\
\Gamma,\varrho \models (\textbf{run}\ t^{\ell})^{\ell_0} & \text{if} & \Gamma,\varrho \models t^{\ell} \\
 & \text{and} & \forall \mathtt{BOX}(t'^{\ell'}) \in \Gamma(\ell).\Gamma(\ell') \subseteq \Gamma(\ell_0) \\
\Gamma,\varrho \models (\textbf{run}\ t^{\ell}\ \textbf{in}\ \rho)^{\ell_0} & \text{if} & \Gamma,\varrho \models t^{\ell} \wedge \Gamma,\varrho \models \rho \\
 & \text{and} & \forall \mathtt{BOX}(t'^{\ell'}) \in \Gamma(\ell).\Gamma(\ell') \subseteq \Gamma(\ell_0) \\
\Gamma,\varrho \models (\textbf{if}(t_1^{\ell_1})\{t_2^{\ell_2}\}\,\textbf{else}\{t_3^{\ell_3}\})^{\ell_4} & \text{if} & \Gamma,\varrho \models t_1^{\ell_1} \wedge \Gamma,\varrho \models t_2^{\ell_2} \wedge \Gamma,\varrho \models t_3^{\ell_3} \\
 & \text{and} & \Gamma(\ell_2) \subseteq \Gamma(\ell_4) \wedge \Gamma(\ell_3) \subseteq \Gamma(\ell_4) \\
\Gamma,\varrho \models (t,\rho)^{\ell} & \text{if} & \Gamma,\varrho \models t^{\ell} \wedge \Gamma,\varrho \models \rho \\
\Gamma,\varrho \models (t_1^{\ell_1}[t_2^{\ell_2}])^{\ell} & \text{if} & \Gamma,\varrho \models t_1^{\ell_1} \wedge \Gamma,\varrho \models t_2^{\ell_2} \\
 & \text{and} & \forall \mathtt{REC}(\ell') \in \Gamma(\ell_1).\forall s,\ell'' \in \mathtt{proto}(\ell')_{\varrho}.\rho(\ell''.s) \subseteq \Gamma(\ell) \\
 & \text{and} & \mathtt{UNDEF} \in \Gamma(\ell) \\
\Gamma,\varrho \models (t_1^{\ell_1}[t_2^{\ell_2}] = t_3^{\ell_3})^{\ell} & \text{if} & \Gamma,\varrho \models t_1^{\ell_1} \wedge \Gamma,\varrho \models t_2^{\ell_2} \wedge \Gamma,\varrho \models t_3^{\ell_3} \\
 & \text{and} & \forall s \in \mathit{String},\mathtt{REC}(\ell') \in \Gamma(\ell_1).\Gamma(\ell_3) \subseteq \varrho(\ell'.s) \\
 & \text{and} & \Gamma(\ell_1) \subseteq \Gamma(\ell) \\
\Gamma,\varrho \models (\textbf{del}\ t_1^{\ell_1}[t_2^{\ell_2}])^{\ell} & \text{if} & \Gamma,\varrho \models t_1^{\ell_1} \wedge \Gamma,\varrho \models t_2^{\ell_2} \\
 & \text{and} & \Gamma(\ell_1) \subseteq \Gamma(\ell) \\
\Gamma,\varrho \models (\mathfrak{m} : t_1^{\ell_1})^{\ell} & \text{if} & \Gamma,\varrho \models t_1^{\ell_1} \\
 & \text{and} & \Gamma(\ell_1) \subseteq \Gamma(\ell) \\
 & & \\
\Gamma,\varrho \models \rho & \text{if} & \forall x \in dom(\rho).\Gamma,\varrho \models \rho(x) \wedge \Gamma(lbl(\rho(x))) \subseteq \varrho(x)
\end{array}
$$

Figure 5.4: Rules for the 0CFA acceptability judgement (simple analysis)

$$
\begin{array}{lll}
\Gamma, \varrho \models \lceil k \rceil \approx k & & \text{for any literal } k \\
\Gamma, \varrho \models \texttt{FUN}(x, e) \approx \textbf{fun}(x)\{e\} & & \\
\Gamma, \varrho \models \texttt{BOX}(e) \approx \textbf{box}\ e & & \\
\Gamma, \varrho \models \texttt{REC}(\ell') \approx \{\overline{s : t^{\ell}}\} & \text{if} & \forall i. \exists v_i \in \varrho(\ell'.s_i). \Gamma, \varrho \models v_i \approx t_i \\
\Gamma, \varrho \models v \approx t' & \text{if} & \Gamma, \varrho \models v \approx t \wedge t^{\ell} \xrightarrow{n} t'^{\ell} \\
\Gamma, \varrho \models v \approx (t, \rho) & \text{if} & \Gamma, \varrho \models v \approx t \wedge \Gamma, \varrho \models \rho \quad \text{(simple analysis)} \\
\Gamma, \varrho \models v \approx (t, \rho) & \text{if} & \exists \Lambda. \Gamma, \varrho \models v \approx t \wedge \Gamma, \varrho, \Lambda \models \rho \quad \text{(improved analysis)}
\end{array}
$$

Figure 5.5: The approximation judgement $\Gamma, \varrho \models v \approx t$

$$
\text{For a literal } k, \text{ define } \lceil k \rceil \text{ by: } \left\{ \begin{array}{rcll}
\lceil \textbf{null} \rceil & = & \texttt{NULL} & \\
\lceil \textbf{undef} \rceil & = & \texttt{UNDEF} & \\
\lceil b \rceil & = & \texttt{BOOL} & \text{for boolean } b \\
\lceil n \rceil & = & \texttt{NUM} & \text{for number } n \\
\lceil s \rceil & = & \texttt{STR} & \text{for string } s
\end{array} \right.
$$

Figure 5.6: The abstract value operation $\lceil k \rceil$

The rules for dealing with function abstractions and records are standard, so I do not motivate them in detail. Note that as the analysis is flow insensitive, the rule for **del** introduces no interesting constraints. That is, if the analysis has determined that a field of an object may contain a value at some point during execution of a program, it behaves as if it may contain that value at any point during execution, including before it has been written and after it has been deleted.

Indeed, the analysis is very coarse in its handling of records in general: if a value is written to one field of an object, the analysis behaves as if it could be read from any field. In practice, it would be easy to address this using a finer abstraction; Liang and Might show how to do this for a similar (but unstaged) language within 0CFA [LM12]. I choose not to do so as the issue is essentially orthogonal to my focus of handling information flow for staged code, but would significantly complicate the correctness proofs for the analysis.

The rule for **box** e requires Γ and ϱ to be an acceptable analysis of the single subexpression e, and for $\Gamma(\ell)$ to include an abstract value v approximating **box** e, which is written as $\Gamma, \varrho \models v \approx \textbf{box}\, e$. This judgement holds if $v = \texttt{BOX}(e)$, but we must be slightly more flexible: during evaluation, unboxing may splice new code fragments into e, changing its syntactic shape to some new expression e'. In order for the flow analysis to be effectively computable, we want the set of abstract values to be finite, so we cannot expect every such $\texttt{BOX}(e')$ to be part of the abstract domain. Instead, we close the approximation judgement under reduction, that is, if $\Gamma, \varrho \models v \approx t$ and $t^{\ell} \xrightarrow{n} t'^{\ell'}$, then also $\Gamma, \varrho \models v \approx t'$; the full

definition of the approximation judgement appears in Figures 5.5–5.6.

The rule for **unbox** e is surprisingly simple: all that is required is that for any abstract value $\texttt{BOX}(e')$ that the analysis thinks can flow into $lbl(e)$ (that is, the label of expression e) every abstract value flowing into its body e' also flows into the unboxing expression. Note that this models the name capture associated with dynamic scoping, since the abstract environment ϱ does not distinguish between different variables of the same name.

The rule for **run** is the same as for **unbox**, as is the intuition. This may seem surprising, but note that the reduction rules for **run** and **unbox** are almost identical, except for the stage at which they may occur; that is, the practical difference between them is how they affect evaluation order. Thus, in a flow insensitive analysis such as 0CFA, which makes no attempt to account for evaluation order, it makes sense that they would have the same analysis rules.

The rule for **if** is standard: any abstract value that either of the branches can evaluate to is also a possible result of the entire **if** expression.

Note that the rule for environments ϱ is never needed to analyse a program, as environments arise only during evaluation. However, we still need to define a rule in order to reason about judgements holding during a sequence of intermediate steps in evaluation.

To show this acceptability judgement makes sense, I prove its coherence with evaluation:

Theorem 2 (Simple CFA Coherence)**.** *If* $\Gamma, \varrho \models e$ *and* $e \xrightarrow{n} e'$*, then* $\Gamma, \varrho \models e'$.

The proof of this theorem is fairly technical and is elided here. A full formalisation in Coq is available online in the supporting material.

Owing to its syntax-directed nature, the definition of the acceptability relation can quite easily be recast as constraint rules; by generating and solving all constraints for a given program, an acceptable flow analysis can be derived.

There may be some concern about how to solve the constraints for records involving quantification over all strings finitely, particularly if a program uses string operations, which would allow it to generate infinitely many different strings. Given the coarseness of the record abstraction, it suffices to introduce "wildcard" abstract variables $\ell.*$ and explicitly conflate accesses to different fields of a record.

Note that, while there may be infinitely many abstract values of the form $\texttt{BOX}(e)$ and $\texttt{FUN}(e)$ that are relevant to a particular program, the closure of the approximation judgement under reduction means that the analysis need only consider those corresponding to subexpressions e of the original program, not those that may arise during execution. That is, the analysis need only solve a finite set of constraints over a finite set of abstract values

$$
\begin{array}{lrcl}
\text{Abstract values} & v \in \mathit{AbsVal} & ::= & \texttt{NULL} \mid \texttt{UNDEF} \mid \texttt{BOOL} \mid \texttt{NUM} \mid \texttt{STR} \\
 & & \mid & \texttt{FUN}(x,e) \mid \texttt{BOX}(e) \mid \texttt{REC}(\ell) \\
\text{Abstract caches} & \Gamma & : & \mathit{Label} \to \mathcal{P}(\mathit{AbsVal}) \\
\text{Abstract environments} & \varrho & : & \mathit{Label} \times (\mathit{Name} \uplus \mathit{String}) \to \mathcal{P}(\mathit{AbsVal}) \\
\text{Abstract frames} & \Lambda & : & \mathit{Name} \to \mathit{Label} \\
\text{Abstract contexts} & \Xi & ::= & \Lambda \mid \Xi + (\ell, \Lambda)
\end{array}
$$

Figure 5.7: Abstract domains (improved analysis)

$$
\begin{array}{lrcl}
\text{Context...} & & & \\
\text{...height} & \mathit{height}\,(\Lambda_0 + (\ell_1,\Lambda_1) + \cdots + (\ell_n,\Lambda_n)) & \stackrel{def}{=} & n \\
\text{...extension} & (\Xi + (\ell,\Lambda))[x \mapsto \ell'] & \stackrel{def}{=} & (\Xi + (\ell, \Lambda[x \mapsto \ell'])) \\
\text{...base replacement} & \Lambda_0 + (\ell_1,\Lambda_1) + \cdots + (\ell_n,\Lambda_n) \upharpoonright \Lambda' & \stackrel{def}{=} & \Lambda' + (\ell_1,\Lambda_1) + \cdots + (\ell_n,\Lambda_n) \\
\text{...as partial function} & (\Xi + (\ell,\Lambda))(x) & \stackrel{def}{=} & \begin{cases} \Lambda(x) & \text{if } x \in \mathit{dom}(\Lambda) \\ \ell & \text{otherwise} \end{cases} \\
\text{Hence:} & \Xi \upharpoonright \Lambda(x) & = & \begin{cases} \Lambda(x) & \text{if } \mathit{height}\,(\Xi) = 0 \\ \Xi(x) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 5.8: Operations on abstract contexts

and a finite set of labels and abstract variables, so it can be guaranteed to terminate.

Example 7. Recall again Example 5. My implementation of the analysis labels the expression as follows:

$$(((\textsf{fun}(x)\{(\textsc{i} : (\textsf{fun}(y)\{x^0\})^1)^2\})^3(\textsc{h} : 1^4)^5)^6(\textsc{l} : 2^7)^8)^9$$

By generating and solving constraints it gives the following solution for Γ:

$$
\begin{array}{lllllllll}
0 & \mapsto & \{\texttt{NUM}\} & 1 & \mapsto & \{\texttt{FUN}(y,(x)^0)\} & 2 & \mapsto & \{\texttt{FUN}(y,(x)^0)\} \\
3 & \mapsto & \{\texttt{FUN}(x,((\textsc{i} : (\textsf{fun}(y)\{(x)^0\})^1)^2)\} & & & & 4 & \mapsto & \{\texttt{NUM}\} \\
5 & \mapsto & \{\texttt{NUM}\} & 6 & \mapsto & \{\texttt{FUN}(y,(x)^0)\} & 7 & \mapsto & \{\texttt{NUM}\} \\
8 & \mapsto & \{\texttt{NUM}\} & 9 & \mapsto & \{\texttt{NUM}\} & & &
\end{array}
$$

while $\varrho = \{x \mapsto \{\texttt{NUM}\}, y \mapsto \{\texttt{NUM}\}\}$. As expected, the result of evaluation (labelled 9) is a number.

5.4.2 Improved Analysis

The analysis presented so far is not very precise, since abstract environments do not distinguish identically named parameters of different functions. Ordinarily, this is not a problem,

$$
\begin{array}{lll}
\Gamma,\varrho,\Xi \models k^{\ell} & \text{if} & \exists \nu \in \Gamma(\ell).\Gamma,\varrho \models \nu \approx k \\
\Gamma,\varrho,\Xi \models x^{\ell} & \text{if} & x \notin dom(\Xi) \vee \varrho(\Xi(x),x) \subseteq \Gamma(\ell) \\
\Gamma,\varrho,\Xi \models \{\overline{s:e}\}^{\ell} & \text{if} & \forall e_i.\Gamma,\varrho,\Xi \models e_i \\
 & \text{and} & \exists \mathtt{REC}(\ell') \in \Gamma(\ell).\forall e_i.\Gamma(lbl(e_i)) \subseteq \varrho(\ell',s_i) \\
\Gamma,\varrho,\Xi \models (\textbf{fun}(x)\{t^{\ell}\})^{\ell_0} & \text{if} & \Gamma,\varrho,\Xi[x \mapsto \ell] \models t^{\ell} \\
 & \text{and} & \exists \nu \in \Gamma(\ell_0).\Gamma,\varrho \models \nu \approx \textbf{fun}(x)\{e\} \\
\Gamma,\varrho,\Xi \models (t_1^{\ell_1}(t_2^{\ell_2}))^{\ell} & \text{if} & \Gamma,\varrho,\Xi \models t_1^{\ell_1} \wedge \Gamma,\varrho,\Xi \models t_2^{\ell_2} \\
 & \text{and} & \forall \mathtt{FUN}(x,t_3^{\ell_3}) \in \Gamma(\ell_1).\Gamma(\ell_2) \subseteq \varrho(\ell_3,x) \wedge \Gamma(\ell_3) \subseteq \Gamma(\ell) \\
\Gamma,\varrho,\Xi \models (\textbf{box}\ t^{\ell})^{\ell_0} & \text{if} & \Gamma,\varrho,\Xi + (\ell,\epsilon) \models t^{\ell} \\
 & \text{and} & \exists \nu \in \Gamma(\ell).\Gamma,\varrho \models \nu \approx \textbf{box}\ t^{\ell} \\
\Gamma,\varrho,\Xi \models (\textbf{unbox}\ t^{\ell})^{\ell_0} & \text{if} & \exists \Xi',\ell_1,\Lambda.\Xi = \Xi' + (\ell_1,\Lambda) \\
 & \text{and} & \Gamma,\varrho,\Xi' \models t^{\ell} \\
 & \text{and} & \forall \mathtt{BOX}(t'^{\ell'}) \in \Gamma(\ell).(\forall x.\varrho(\Xi(x),x) = \varrho(\ell',x)) \\
 & & \qquad \wedge \Gamma(\ell') \subseteq \Gamma(\ell_0) \\
\Gamma,\varrho,\Xi \models (\textbf{run}\ t^{\ell})^{\ell_0} & \text{if} & \Gamma,\varrho,\Xi \models t^{\ell} \\
 & \text{and} & \forall \mathtt{BOX}(t'^{\ell'}) \in \Gamma(\ell).(\forall x \in dom(\Xi).\varrho(\Xi(x),x) = \varrho(\ell',x)) \\
 & & \qquad \wedge \Gamma(\ell') \subseteq \Gamma(\ell_0) \\
\Gamma,\varrho,\Xi \models (\textbf{run}\ t^{\ell}\ \textbf{in}\ \rho)^{\ell_0} & \text{if} & \exists \Lambda.\Gamma,\varrho,\Xi \restriction \Lambda \models t^{\ell} \wedge \Gamma,\varrho,\Lambda \models \rho \\
 & \text{and} & \forall \mathtt{BOX}(t'^{\ell'}) \in \Gamma(\ell). \\
 & & \quad (\forall x \in dom(\Xi \restriction \Lambda).\varrho(\Xi \restriction \Lambda(x),x) = \varrho(\ell',x)) \\
 & & \quad \wedge \Gamma(\ell') \subseteq \Gamma(\ell_0) \\
\Gamma,\varrho,\Xi \models (\textbf{if}(t_1^{\ell_1})\{t_2^{\ell_2}\}\,\textbf{else}\{t_3^{\ell_3}\})^{\ell_4} & \text{if} & \Gamma,\varrho,\Xi \models t_1^{\ell_1} \wedge \Gamma,\varrho,\Xi \models t_2^{\ell_2} \wedge \Gamma,\varrho,\Xi \models t_3^{\ell_3} \\
 & \text{and} & \Gamma(\ell_2) \subseteq \Gamma(\ell_4) \wedge \Gamma(\ell_3) \subseteq \Gamma(\ell_4) \\
\Gamma,\varrho,\Xi \models (t,\rho)^{\ell} & \text{if} & \exists \Lambda.\Gamma,\varrho,\Xi \restriction \Lambda \models t^{\ell} \wedge \Gamma,\varrho,\Lambda \models \rho \\
\Gamma,\varrho,\Xi \models (t_1^{\ell_1}[t_2^{\ell_2}])^{\ell} & \text{if} & \Gamma,\varrho,\Xi \models t_1^{\ell_1} \wedge \Gamma,\varrho,\Xi \models t_2^{\ell_2} \\
 & \text{and} & \forall \mathtt{REC}(\ell') \in \Gamma(\ell_1).\forall s,\ell'' \in \mathtt{proto}(\ell')_{\varrho}.\varrho(\ell'',s) \subseteq \Gamma(\ell) \\
 & \text{and} & \mathtt{UNDEF} \in \Gamma(\ell) \\
\Gamma,\varrho,\Xi \models (t_1^{\ell_1}[t_2^{\ell_2}] = t_3^{\ell_3})^{\ell} & \text{if} & \Gamma,\varrho,\Xi \models t_1^{\ell_1} \wedge \Gamma,\varrho,\Xi \models t_2^{\ell_2} \wedge \Gamma,\varrho,\Xi \models t_3^{\ell_3} \\
 & \text{and} & \forall s,\mathtt{REC}(\ell') \in \Gamma(\ell_1).\Gamma(\ell_3) \subseteq \varrho(\ell',s) \\
 & \text{and} & \Gamma(\ell_1) \subseteq \Gamma(\ell) \\
\Gamma,\varrho,\Xi \models (\textbf{del}\ t_1^{\ell_1}[t_2^{\ell_2}])^{\ell} & \text{if} & \Gamma,\varrho,\Xi \models t_1^{\ell_1} \wedge \Gamma,\varrho,\Xi \models t_2^{\ell_2} \\
 & \text{and} & \Gamma(\ell_1) \subseteq \Gamma(\ell) \\
 & & \\
\Gamma,\varrho,\Lambda \models \rho & \text{if} & \forall x \in dom(\rho).x \in dom(\Lambda) \\
 & & \qquad \wedge \Gamma,\varrho,\Lambda \models \rho(x) \\
 & & \qquad \wedge \Gamma(lbl(\rho(x))) \subseteq \varrho(\Lambda(x),x)
\end{array}
$$

Figure 5.9: Rules for the 0CFA acceptability judgement (improved analysis)

as one can rename them apart, but this is not possible for SLamJS, which does not enjoy alpha conversion.

To restore analysis precision in the absence of alpha conversion, we introduce an *abstract context* Ξ that keeps track of name bindings (Figure 5.7) and various operations on it (Figure 5.8). In a single-staged language, such an abstract context would simply map a name x to the innermost enclosing function abstraction whose parameter is x. In a multi-staged setting, we need to distinguish between bindings at different stages, hence the abstract context maintains one such mapping per stage. Thus Ξ is a stack of *abstract frames* Λ, one for each stage; a frame maps each variable name to the label of its binding function abstraction; each frame in Ξ (except the base) is paired with the label of the **box** expression corresponding to the stage.

For instance, the two uses of x in the SLamJS expression:

$$\textbf{fun}(x)\ \{\textbf{box}(\textbf{fun}(x)\ \{\ (\textbf{unbox}\ x)(x)\ \})\}$$

are at different stages, and hence bound by different abstractions: the first x by the outer abstraction, the second by the inner one.

The *height* of an abstract context is the level of its topmost abstract frame; that is, one less than the total number of frames in the context.

Having enhanced the analysis by recording where variables are bound, we can use this information to improve the precision of our abstract environment ϱ, which is now a binary function. For a label ℓ labelling the body of a function abstraction with parameter x, $\varrho(\ell, x)$ overapproximates all values this parameter may be bound to in any invocation of the function. Similarly, for ℓ labelling the body of a **box** expression, $\varrho(\ell, x)$ overapproximates the values x may have in any evaluation of that body. Finally, for ℓ labelling a record, $\varrho(\ell, s)$ overapproximates all values that may be stored in field s of that record.

The acceptability judgement for the improved analysis is now of the form $\Gamma, \varrho, \Xi \models e$, and the derivation rules include additional bookkeeping to adjust Ξ when analysing subexpressions at different stages. While the change is conceptually simple, the rules are now syntactically somewhat more complex, as shown in Figure 5.9. Observe that they are structurally very similar to those for the simple analysis.

The rule for **fun** now updates the top frame of Ξ, adding a link from the variable x to its enclosing function abstraction. The rule for **box** pushes a new empty frame ϵ (with a link to the label ℓ of the **box** expression) to the stack Ξ; this enlarged stack is used to analyse the contents of the **box** expression, which are at a higher stage. Conversely, the rule for **unbox** pops the top frame off the stack Ξ and analyses its contents at a lower stage. The equality constraint in this rule (which is implemented as two subset constraints) effectively aliases

variables in scope of the **unbox** to variables of the same name that may be spliced in from an external **box** code value. As with the simple analysis, the rule for **run** is much like that for **unbox**, except that it does not pop off the top frame of Ξ, as its contents are at the same stage.

As for the simple analysis, I demonstrate correctness of the improved analysis by proving its coherence with evaluation:

Theorem 3 (Improved CFA Coherence)**.** *If* $\Gamma, \varrho, \Xi \models e$ *with* Ξ *of height n, and* $e \xrightarrow{n} e'$*, then* $\Gamma, \varrho, \Xi \models e'$.

Once again, a full formalisation in Coq is available in the online supporting material [Les15] and in Appendix A.

5.5 Information Flow for SLamJS

Assume we have already analysed a program using 0CFA and found environments Γ, ϱ that over-approximate the values flowing to each labelled expression. We use information about which functions and boxed values may occur to assist in determining what direct and indirect flows occur between labels of the expression.

By recursing over the structure of an expression, we generate and solve constraints on a relation $\rightsquigarrow$:

$$\rightsquigarrow : (\mathit{Label} \uplus \mathit{Name} \uplus \mathit{Marker}) \to (\mathit{Label} \uplus \mathit{Name} \uplus \mathit{Marker})$$

As an expression, the labels, variable names and markers occurring within an expression and the abstract values in the results of 0CFA for an expression are all finite, the process will terminate.

We express constraints between labels, variable names and markers as either direct flows ($x \rightsquigarrow y \implies x \rightsquigarrow y$) or indirect flows ($x \nrightarrow y \implies x \rightsquigarrow y$). (The distinction between direct and indirect is for clarity of exposition; there is no practical difference between them with regard to the resulting analysis.)

Note that if instead we interpret $x \rightsquigarrow y$ and $x \nrightarrow y$ as (elements of) relations and define $\rightsquigarrow = \rightsquigarrow \cup \nrightarrow$, then $\rightsquigarrow$ satisfies the constraints.

We say that $\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} e$ if $\Gamma, \varrho \models e$ and the conditions in Figure 5.10 hold. As Γ, ϱ and $\rightsquigarrow$ are constant throughout the definition, we abbreviate $\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} e$ to $\models_{\mathrm{IF}} e$ for clarity.

Note that a direct flow essentially occurs whenever the rules for 0CFA introduce a subset constraint: $\Gamma(\ell_1) \subseteq \Gamma(\ell_2)$ induces the flow $\ell_1 \rightsquigarrow \ell_2$. Conversely, an indirect flow occurs when a data value may somehow influence the control flow of a program. In the case

Expression e $\models_{\text{IF}} e$ holds:	*Subexpressions* if:	*Direct Flows* and:	*Indirect Flows* and:
k^{ℓ}	–	–	–
x^{ℓ}	–	$x \rightsquigarrow \ell$	–
$\textsf{fun}(x)\{t^{\ell_1}\}^{\ell_2}$	$\models_{\text{IF}} t^{\ell_1}$	–	–
$(t_1^{\ell_1}(t_2^{\ell_2}))^{\ell}$	$\models_{\text{IF}} t_1^{\ell_1} \wedge \models_{\text{IF}} t_2^{\ell_2}$	$\forall \texttt{FUN}(x, t_3^{\ell_3}) \in \Gamma(\ell_1)\,.$ $\ell_2 \rightsquigarrow x \wedge \ell_3 \rightsquigarrow \ell$	$\ell_1 \hookrightarrow \ell$
$(\textsf{if}(t_1^{\ell_1})\{t_2^{\ell_2}\}\,\textsf{else}\{t_3^{\ell_3}\})^{\ell_4}$	$\bigwedge_{i=1}^{3} \models_{\text{IF}} t_i^{\ell_i}$	$\ell_2 \rightsquigarrow \ell_4 \wedge \ell_3 \rightsquigarrow \ell_4$	$\ell_1 \hookrightarrow \ell_4$
$(t_1, \rho)^{\ell_1}$	$\models_{\text{IF}} {t_1}^{\ell_1} \wedge$ $\bigwedge_{(x \mapsto t^{\ell}) \in \rho} \models_{\text{IF}} t^{\ell}$	–	–
$(\mathfrak{m} : t^{\ell_1})^{\ell_2}$	$\models_{\text{IF}} t^{\ell_1}$	$\ell_1 \rightsquigarrow \ell_2 \wedge \mathfrak{m} \rightsquigarrow \ell_2$	–
$(t_1^{\ell_1}[t_2^{\ell_2}])^{\ell}$	$\models_{\text{IF}} t_1^{\ell_1} \wedge \models_{\text{IF}} t_2^{\ell_2}$	$\forall \texttt{REC}(\ell') \in \Gamma(\ell_1)\,.\forall \ell'' \in \texttt{proto}(\ell')_{\varrho}\,.$ $\forall s\,.\ell''.s \rightsquigarrow \ell$	$\ell_1 \hookrightarrow \ell$ $\ell_2 \hookrightarrow \ell$
$\{s_1 : t_1^{\ell_1}, \ldots, s_n : t_n^{\ell_n}\}^{\ell}$	$\bigwedge_{i=1}^{n} \models_{\text{IF}} e_n$	$\exists \texttt{REC}(\ell') \in \Gamma(\ell)\,.\forall i\,.\ell_i \rightsquigarrow \ell'.s_i$	–
$(t_1^{\ell_1}[t_2^{\ell_2}] = t_3^{\ell_3})^{\ell}$	$\bigwedge_{i=1}^{3} \models_{\text{IF}} t_i^{\ell_i}$	$\ell_1 \rightsquigarrow \ell \wedge$ $\forall \texttt{REC}(\ell') \in \Gamma(\ell_1)\,.\forall s\,.\ell_3 \rightsquigarrow \ell'.s$	$\ell_2 \hookrightarrow \ell$
$(\textsf{del}\ t_1^{\ell_1}[t_2^{\ell_2}])^{\ell}$	$\models_{\text{IF}} t_1^{\ell_1} \wedge \models_{\text{IF}} t_2^{\ell_2}$	$\ell_1 \rightsquigarrow \ell$	$\ell_2 \hookrightarrow \ell$
$(\textsf{box}\ t^{\ell_1})^{\ell_2}$	$\models_{\text{IF}} t^{\ell_1}$	–	–
$(\textsf{unbox}\ t^{\ell_1})^{\ell_2}$	$\models_{\text{IF}} t^{\ell_1}$	$\forall \texttt{BOX}(t'^{\ell'}) \in \Gamma(\ell_1)\,.\ell' \rightsquigarrow \ell_2$	$\ell_1 \hookrightarrow \ell_2$
$(\textsf{run}\ t^{\ell_1})^{\ell_2}$	$\models_{\text{IF}} t^{\ell_1}$	$\forall \texttt{BOX}(t'^{\ell'}) \in \Gamma(\ell_1)\,.\ell' \rightsquigarrow \ell_2$	$\ell_1 \hookrightarrow \ell_2$
$(\textsf{run}\ t^{\ell_1}\ \textsf{in}\ \rho)^{\ell_2}$	$\models_{\text{IF}} t^{\ell_1} \wedge \models_{\text{IF}} \rho$	$\forall \texttt{BOX}(t'^{\ell'}) \in \Gamma(\ell_1)\,.\ell' \rightsquigarrow \ell_2$	$\ell_1 \hookrightarrow \ell_2$

Figure 5.10: Rules for generating information flow constraints

of records, the field selector in a record access (for example, t_2 in $\mathbf{del}\, t_1[t_2]$) controls which field in a record is read from or written to, introducing an indirect flow to the resulting record.

I now prove the coherence of the information flow analysis with evaluation. Like the corresponding proof for 0CFA for SLamJS, this is lengthy and technical, so I only sketch it here. A mechanisation of the proof is available online [Les15] and in Appendix A.

Lemma 2 (Reduction Preserves Satisfaction)**.** *If we have* $\Gamma, \varrho, \rightsquigarrow \models_{IF} t_1^{\ell_1}$ *and* $t_1^{\ell_1} \overset{n}{\dashrightarrow} t_2^{\ell_2}$*, then* $\Gamma, \varrho, \rightsquigarrow \models_{IF} t_2^{\ell_2}$*. Furthermore,* $\ell_2 \rightsquigarrow^* \ell_1$*.*

Proof. By case analysis on the rules defining $\overset{n}{\dashrightarrow}$. □

Theorem 4 (Information Flow Coherence)**.** *If we have* $\Gamma, \varrho, \rightsquigarrow \models_{IF} e_1$ *and* $e_1 \xrightarrow{m} e_2$*, then* $\Gamma, \varrho, \rightsquigarrow \models_{IF} e_2$*. Furthermore,* $lbl(e_2) \rightsquigarrow^* lbl(e_1)$*.*

Proof. Sketch: Unfolding the definition of $\xrightarrow{m}$, we let $e_1 = C_n^m\langle t_1^{\ell_1}\rangle$ and $e_2 = C_n^m\langle t_2^{\ell_2}\rangle$ with $t_1^{\ell_1} \overset{n}{\dashrightarrow} t_2^{\ell_2}$.

Observe that $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t_1^{\ell_1}$ and hence, applying Lemma 2, we have $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t_2^{\ell_2}$, with $\ell_2 \rightsquigarrow^* \ell_1$. Observe further that constraints generated by C_n^m and the contents of its hole interact only at that hole, labelled ℓ_2 or ℓ_1. Thus, using $\ell_2 \rightsquigarrow^* \ell_1$, they must be satisfied in the conclusion, giving $\Gamma, \varrho \rightsquigarrow \models_{\text{IF}} C_n^m\langle t_2^{\ell_2}\rangle$ as required.

The claim that $lbl(e_2) \rightsquigarrow^* lbl(e_1)$ is trivial for all non-empty contexts, as $lbl(e_2) = lbl(e_1)$. For the empty context, it follows directly from the similar claim in Lemma 2. □

Note that, while the 0CFA and information flow analysis phases are conceptually distinct, correctness of the latter depends on correctness of the former. Therefore, for the sake of simplicity, the mechanisation of the proof concerns a combined formulation of the analyses in which both phases are performed simultaneously.

Example 8. Recall once more Example 5. Using the results of 0CFA, the implementation generates the relations $\rightsquigarrow$ and $\hookrightarrow$ as depicted in Figure 5.11. Plain arrows indicate direct flows ($\rightsquigarrow$); dotted arrows indicate indirect flows ($\hookrightarrow$).

Setting $\rightsquigarrow = \rightsquigarrow \cup \hookrightarrow$, we have $\text{H} \rightsquigarrow^* 9$ and $\text{I} \rightsquigarrow^* 9$ and $\text{L} \not\rightsquigarrow^* 9$. As expected, this means the result (labelled 9) has information flows from H and I, but not L.

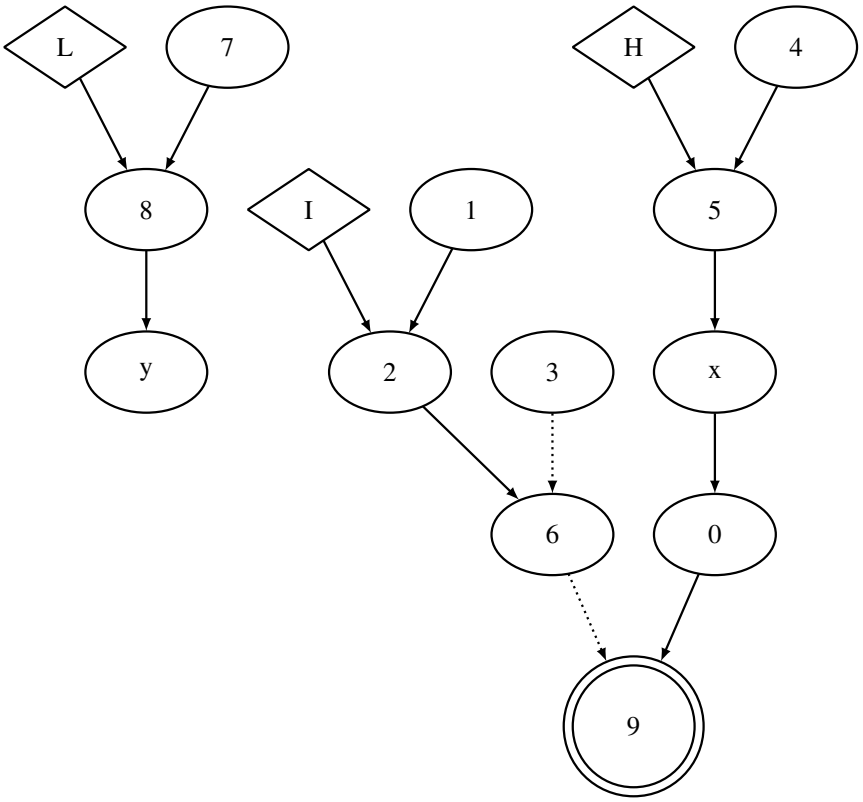


Figure 5.11: Information flow constraints for Example 5

5.6 Information Flow Soundness

It remains to show that the information flow analysis does indeed identify information flows. Consider some term t^ℓ, which eventually evaluates to a value $v^{\ell'}$. If the evaluation depended on some marker $\mathfrak{m}$ present in t (and hence, there is a flow of information from $\mathfrak{m}$ into v) then, when evaluating t using the augmented semantics, $\mathfrak{m}$ will remain in v. So, looking at some M-erasure of v, if $\mathfrak{m} \notin M$, then $\lfloor v \rfloor_M \neq v$. Conversely, if $\lfloor v \rfloor_M = v$, then necessarily $\mathfrak{m} \in M$. Thus, for the information flow analysis to be correct, it must be able to identify every marker that must be in M in order to obtain $\lfloor v \rfloor_M = v$. That is, it must identify every marker on which evaluation of t could depend. Of course, when analysing t^ℓ, we do not know what $v^{\ell'}$ will be, but we do know ℓ and (from the results of the analysis) can easily find all markers $\mathfrak{m}$ such that $\mathfrak{m} \rightsquigarrow^* \ell$; I now prove that these are the markers that must be in M.

Theorem 5 (Information Flow Soundness)**.** *Suppose* $\Gamma, \varrho, \rightsquigarrow \models_{IF} t^\ell$. *Then if* $t^\ell \xrightarrow{\square}{}^* v^{\ell'}$, *where* v *is a stage-0 value composed only of markers and constants, then* $\lfloor v \rfloor_M = v$ *where* $M = \{\mathfrak{m} \in \mathit{Marker} \mid \mathfrak{m} \rightsquigarrow^* \ell\}$.

Proof. First show that $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} v^{\ell'}$ with $\ell' \rightsquigarrow^* \ell$. Argue by a simple induction over the derivation of $t^{\ell} \xrightarrow{\square}{}^* v$.

Base case: $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t^{\ell}$ follows immediately from the theorem's premise.

Inductive step: Assume that $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} e_1$ and $lbl(e_1) \rightsquigarrow^* \ell$, with $e_1 \xrightarrow{\square} e_2$ the next step in the derivation. Apply Theorem 4 to show that $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} e_2$ and $lbl(e_2) \rightsquigarrow^* lbl(e_1)$; hence $lbl(e_2) \rightsquigarrow^* \ell$.

Now we have $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} v$ and $\ell' \rightsquigarrow^* \ell$. Observe from the definition of $\lfloor v \rfloor_M$ that if for every marker $\mathfrak{m}$ that occurs in v we have $\mathfrak{m} \in M$, then $\lfloor v \rfloor_M = v$.

But v is a value composed only of markers and constants, so for every marker $\mathfrak{m}$ that occurs in v (by examination of the $\models_{\text{IF}}$ constraint rules) it must be the case that $\mathfrak{m} \rightsquigarrow^* \ell'$. Thus, as $\ell' \rightsquigarrow^* \ell$, $\mathfrak{m} \rightsquigarrow^* \ell$. Hence, from the definition of M, $\mathfrak{m} \in M$. So it is indeed true that $\lfloor v \rfloor_M = v$. □

Relationship with noninterference. The information flow analysis could perhaps more accurately be called a dependency analysis [ABHR99], as it determines which expressions in a program depend on which other expressions. This kind of analysis is dual to strictness analysis [ALL96]: if a strictness analysis says that a function is strict in one of its arguments, it definitely depends on it; if a dependency analysis says that a function is independent of one of its arguments, it definitely does not depend on it; otherwise, for either analysis, we cannot conclude whether the function will depend on the argument or not.

Such an information flow analysis can be used to verify the security property noninterference [KT13]. Noninterference asserts that the values of any "high-security" inputs must not affect the values of any "low-security" outputs. In order for this assertion to be meaningful, we must have notions of input, output and high- and low-security levels.

Suppose elements of *Marker* represent different levels of security, such as L for low security and H for high security. We can model input by putting dummy values in a term t, then having a relation that replaces those dummy values with the "input" values. For example, we might have $t = (\textsf{fun}(x)\{\textsf{fun}(y)\{x\}\})(L : 0)(H : 0)$, with the dummy values 0 indicating inputs. Now suppose we have a relation $\xrightarrow{\text{low}}$ such that $t \xrightarrow{\text{low}} t'$ only when t' can be obtained from t just by replacing constants marked as L with constants of the same type (integer, boolean or string); suppose we also have a similar relation $\xrightarrow{\text{high}}$ that replaces only constants marked as H. Then:

$$\begin{array}{ll} & (\textsf{fun}(x)\{\textsf{fun}(y)\{x\}\})(L : 0)(H : 0) \\ \xrightarrow{\text{low}} & (\textsf{fun}(x)\{\textsf{fun}(y)\{x\}\})(L : 1)(H : 0) \\ \xrightarrow{\text{high}} & (\textsf{fun}(x)\{\textsf{fun}(y)\{x\}\})(L : 1)(H : 2) \end{array}$$

models a low input of 1 and a high input of 2. We can model low-security output as the value to which an expression evaluates.

Now say that expression t^ℓ satisfies noninterference analysis if $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t^\ell$ and $\text{H} \not\rightsquigarrow^* \ell$. Further, require that $\xrightarrow{\text{low}}$ and $\xrightarrow{\text{high}}$ satisfy the following conditions:

$$\begin{aligned}
\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t^\ell \wedge t \xrightarrow{\text{low}} t' &\implies \Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t'^\ell \\
\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t^\ell \wedge t \xrightarrow{\text{high}} t' &\implies \Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t'^\ell \\
\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t^\ell \wedge t \xrightarrow{\text{high}} t' &\implies \lfloor t \rfloor_{\{\text{L}\}} = \lfloor t' \rfloor_{\{\text{L}\}}
\end{aligned}$$

Claim: If t^ℓ satisfies noninterference analysis, then in the following situation:

$$t^\ell \xrightarrow{\text{low}} t'^\ell \qquad t'^\ell \xrightarrow{\text{high}} t_1^\ell \qquad t'^\ell \xrightarrow{\text{high}} t_2^\ell \qquad t_1^\ell \xrightarrow{\square}{}^* u^{\ell'}$$

where $u^{\ell'}$ is a value composed only of markers and constants, it follows that $t_2^\ell \xrightarrow{\square}{}^* u^{\ell'}$. That is, the low output u is independent from the values of the high inputs for t selected using $\xrightarrow{\text{high}}$.

Proof: By the condition on $\xrightarrow{\text{low}}$, observe we have $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t'$. By the first condition on $\xrightarrow{\text{high}}$, it then follows that $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t_1^\ell$ and $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t_2^\ell$. As $\text{H} \not\rightsquigarrow^* \ell$, by soundness of information flow, we have $u = \lfloor u \rfloor_{\{\text{L}\}}$. So using stability, we get $\lfloor t_1 \rfloor_{\{\text{L}\}} \xrightarrow{\square}{}^* u$. But, by the second condition on $\xrightarrow{\text{high}}$, we have $\lfloor t' \rfloor_{\{\text{L}\}} = \lfloor t_1 \rfloor_{\{\text{L}\}} = \lfloor t_2 \rfloor_{\{\text{L}\}}$. So $\lfloor t_2 \rfloor_{\{\text{L}\}} \xrightarrow{\square}{}^* u$. Then by monotonicity, $t_2 \xrightarrow{\square}{}^* u$.

Note that the examples of $\xrightarrow{\text{low}}$ and $\xrightarrow{\text{high}}$ discussed above meet the required conditions, but the proof would hold for any other $\xrightarrow{\text{low}}$ and $\xrightarrow{\text{high}}$ that meet these conditions.

5.7 Implementation and Examples

I have implemented the analysis in OCaml and tested it on a range of examples. The most expensive part of the analysis computationally is 0CFA, which runs in time $O(n^3)$ in the size of the program [HM97]; consequently, it runs quickly on all my examples and I expect it to scale well to large programs. The source code for my analysis tool and the examples are available online [Les15]. I now present some of these examples.

For each example, I list the markers on which the simple analysis says the result may *depend*. Where the *improved* analysis gives a more precise result, I list that too. To improve readability, I write **let** $x = v$ **in** e as a shorthand for **fun**$(x)\{e\}v$. My implementation extends SLamJS (and its analysis) as presented here with primitive arithmetic, equality and **typeof** operators, which I use in some of our examples. It can also handle mutable references in the style of λ_{JS} . Many of my examples are inspired by patterns of **eval** usage common in

Web applications, as surveyed by Richards and others [RHBV11] and discussed by Jensen and others [JJM12].

Example 9. *Depends on*: H, L.

if(H : **true**){L : **false**} **else**{1}

I begin with a classic example where branching on a value introduces an indirect flow from it. As the analysis does not track specific boolean values, it would give the same result if the branch were on (H : **false**). I could resolve this imprecision by extending the abstract value domain with abstract values for **true** and **false**.

Example 10. *Depends on*: H, I, L. *Depends (improved)*: H, L.

let *ctrue* = **fun**(x){**fun**(y){x}} **in**

let *cif* = **fun**(x){**fun**(y){**fun**(z){(x(y))(z)}}} **in**

((*cif*(H : *ctrue*))(L : **false**))(I : 1))

Conversely, if we present the previous example using the standard Church-encodings of **if** and **true** as functions, the analysis is precise enough to determine that the result does not depend on I. Note that we need the improved analysis to distinguish the bindings of x and y in *ctrue* and *cif*.

Example 11. *Depends on*: L.

let x = **if**(**true**){**box** f} **else**{**box** g} **in**

let f = **fun**(y){1} **in**

let g = **fun**(z){L : **true**} **in**

run (**box** ((**unbox** x)(H : **undef**)))

This is modelled on the following JavaScript usage [JJM12]:

```
if (...)  x = "f"; else x = "g";
eval(x + "()");
```

f and g are bound to functions; x is set to a code value of either f or g; a function argument is added to the code value and the result executed. In this example, both f and g ignore their argument (H : **undef**), so the result does not depend on H; the analysis correctly identifies this.

Example 12. *Depends on*: H, L. *Depends (improved)*: L.

let c = **box** x **in**

let x = L : 1 **in**

let *eval* = **fun**(b){**run** b} **in**

let x = H : 2 **in**

eval(c)

JavaScript programmers sometimes use **eval** to execute code within a different scope. SLamJS does not aim to emulate all the quirks of **eval**, but scoping of staged code can still have interesting behaviour, as shown in this example. In the scope of the definition of the function bound to *eval*, x is 1. So when it evaluates the code value c, which contains just the variable x, this is the value it returns; note that x was not bound at all where c was defined. The second binding of x is unused; the analysis correctly determines this.

Example 13. *Depends on*: H, I, L.

let $i = \text{I} : \{\texttt{"_proto_"} : \textbf{null}, \texttt{"}x\texttt{"} : (\text{H} : 1), \texttt{"}y\texttt{"} : (\text{L} : 2)\}$ **in**
let $s = \textbf{fun}(id)\{\textbf{let}\ f = \textbf{box}\ (i[\textbf{unbox}\ id])\ \textbf{in}\ \textbf{run}\ f\}$ **in**
$s(\textbf{box}\ \texttt{"}y\texttt{"})$

Some programmers use **eval** to construct variable names. For example, they might use (`var n = 5; eval ("f_" + n);`) to access `f_5`. We cannot express this directly in SLamJS as there are no facilities to manipulate variable names. Another common practice is to use **eval** to access object properties, often because of the programmer's ignorance of JavaScript's indirect object field access syntax; this example models that practice in SLamJS. Because the analysis does not model the values strings may take, its handling of field reads and writes is rather coarse, so it cannot tell the result will not depend on H; this could be addressed refining the abstract value domain.

Example 14. *Depends on*: H.

let $fst = \textbf{fun}(x)\{\textbf{fun}(y)\{x\}\}$ **in**
let $f = \textbf{if}(\textbf{false})\{fst\}\ \textbf{else}\{\textbf{box}\, fst\}$ **in**
let $x = (\text{H} : 1)$ **in**
let $y = (\text{L} : \textbf{true})$ **in**
$\textbf{if}(\textbf{typeof}\ f = \texttt{"function"})\{(f(x))(y)\}$
$\textbf{else}\{\textbf{run}\ (\textbf{box}\ (((\textbf{unbox}\ f)(x))(y)))\}$

This example models the JavaScript usage pattern:

```
if (f instanceof Function) f(x);
else eval (f + "(x)");
```

which may arise when using **eval** to emulate higher-order functions. Here, the analysis shows the same precision on a boxed value representing a function as when dealing with a real function.

Example 15. *Depends on:* H,L. *Depends (improved)*: L.

let $pair$ = **fun**$(x)\{$**fun**$(y)\{$**fun**$(z)\{$**run** $z\}\}\}$ **in**
let fst = **fun**$(z)\{z($**box** $x)\}$ **in**
let snd = **fun**$(z)\{z($**box** $y)\}$ **in**
let bp = **box** $((pair(\text{L} : (\textbf{box}\ (1))))(\text{H} : (\textbf{box}\ (\textbf{true}))))$ **in**
let $boxfst$ = **box** $((fst)($**unbox** $bp))$ **in**
run (**run** $(boxfst)$)

Most examples of staged metaprogramming in the literature do not use more than one level of staging. This example, which pairs and unpairs two values in a rather roundabout way, illustrates that the analysis can handle higher levels too.

Example 16. *Depends on*: H.

fun$(n)\{($**fun**$(x)\{(x(x))(n)\})$
$($**fun**$(x)\{$**fun**$(y)\{$**if**$(y = 0)\{$**true**$\}$ **else**$\{(x(x))(sub(y,1))\}\}\})$
$\}(\text{H} : 5)$

This program loops n times (where n is $(\text{H} : 5)$ in this instance) before returning **true**. In this sense, the result is independent of n: if n were a high-security input and the output low, the program would satisfy noninterference, although the duration of execution may leak information about n. However, n must be examined in order to execute the program, so there is an information flow from n to the result, in the sense captured by the augmented semantics. That is, no noninterference analysis based on a sound over-approximation of the behaviour of such a semantics could ever show the program to be noninterfering [RS10].

Example 17. *Depends on:* L.

let fst = **fun**$(x)\{$**fun**$(y)\{x\}\}$ **in**
let a = **box** x **in**
let b = **box** $($**fun**$(x)\{$**fun**$(y)\{fst($**unbox** $a)(y)\}\})$ **in**
$($**run** $b)(\text{L} : 1)(\text{H} : 2)$

This program, based on an example from Choi and others [CAYT11], splices a variable name into a code template to produce code that takes two arguments and returns the first. The analysis correctly determines that the result depends only on the first.

Example 18. *Depends on:* L,H.

let fst = **fun**$(x)\{$**fun**$(y)\{x\}\}$ **in**
let a = **fun**$(p)\{p[$`"x"`$]\}$ **in**
let $b = ($**fun**$(h)\{$**fun**$(p)\{$**fun**$(x)\{$**fun**$(y)\{$ $fst\ (h((p[$`"x"`$] = x)[$`"y"`$] = y))\ (y)$ $\}\}\}\})(a)$ **in**
$b(\{$`"__proto__"` : **null**$\})(\text{L} : 1)(\text{H} : 2)$

By applying Choi and others' unstaging translation to the core of the previous example, we

obtain this unstaged one. Note that while the result of the program is the same, we lose precision by analysing this version instead of working directly on the staged version.

Example 19. *Depends on:* L,H. *Depends (improved):* L.

let $blank$ = **fun**(get){get(**null**)(**null**)} **in**
let $getx$ = **fun**(x){**fun**(y){x}} **in**
let $gety$ = **fun**(x){**fun**(y){y}} **in**
let $setx$ = **fun**(env){**fun**($newx$){
fun(get){get($newx$)(env($gety$))}}} **in**
let $sety$ = **fun**(env){**fun**($newy$){
fun(get){get(env($getx$))($newy$)}}} **in**
let fst = **fun**(x){**fun**(y){x}} **in**
let a = **fun**(p){p($getx$)} **in**
let b = (**fun**(h){**fun**(p){**fun**(x){**fun**(y){ fst (h($sety$($setx$(p)(x))(y))) (y) }}}})(a) **in**
b($blank$)(L : 1)(H : 2)

Here I have applied the unstaging translation, as in the previous example, but using higher order functions to encode environments instead of records. In this case, we can recover the lost precision, but at the cost of an $O(n^2)$ increase in the size of the source program, making the combined analysis $O(n^6)$ instead of $O(n^3)$.

Chapter 6

The Boxing Algorithm: Transforming Eval to Staged Metaprogramming

6.1 Overview

In developing an information flow analysis for a language with staged metaprogramming, the aim was to advance the analysis of JavaScript programs using **eval**. In order to demonstrate the applicability of the analysis, I now describe the *Boxing Algorithm*, an algorithm for transforming a program that uses string-based **eval** into one that uses the **box** and **unbox** of staged metaprogramming. I illustrate the algorithm with examples in SLamJS, although I believe that the idea is sufficiently general that it could be applied to many other languages with an **eval** construct, such as Perl and PHP.

Given a program e_1 that uses **eval**, the aim is to transform it into an equivalent program e_2 that instead uses staged metaprogramming. We can then meaningfully apply the information flow analysis of Chapter 5 to e_2. An overview of the algorithm is given in Figure 6.1; a more detailed illustration of dataflow within the algorithm is shown in Figure 6.2.

The transformation is based on the assumption that an **eval**-using program uses string concatenation to join together well-formed code templates. Hence constant strings can be transformed into **box** expressions, uses of concatenation into **unbox** expressions, and **eval** into **run**. This assumption may be violated if, for example: the structure of concatenation in the program does not correspond closely enough to the syntactic structure of expressions; the program makes decisions about flow control on the basis of the syntactic content of code strings; or the program uses substring operations to deconstruct code templates (as **unbox** only describes composition). In these cases, the transformation will fail with an error.

In Section 6.2, I explain some of the key ideas and technical points of the algorithm. The algorithm operates via a fixed point process; in Section 6.3, I explain the steps in a single

– *Input:* an **eval**-using program e_1.
– *Output:* a transformed staged program e_2.

While a fixed point has not been reached:

1. Generate constraints α for e_1 ...
   – ... and (if not first iteration) any code previously generated in step 4.
2. Solve constraints α ...
   – ... and (if not first iteration) any constraints γ previously generated in step 5.
3. Search for a transformation.
   – If the search fails, terminate unsuccessfully.
4. Splice **box** expressions into e_1 according to transformation, generating candidate e_2.
   – If (not first iteration and) candidate e_2 is the same as previous iteration's, a fixed point has been reached; terminate successfully and return e_2.
5. Generate, solve and resolve staged constraints γ on spliced code.

Figure 6.1: Control flow in the Boxing Algorithm

cycle. As the whole process is quite intricate, in Section 6.4, I give a worked example of how the transformation proceeds. Next, in Section 6.5, I argue for the soundness of the transformation. Lastly, in Section 6.6, I give some examples of programs that the algorithm can and cannot transform.

6.2 Key Ideas

6.2.1 Prerequisites

The basic idea is that the algorithm will transform:

- code constants into **box** expressions;
- concatenation of code strings into splicing using **unbox**;
- **eval** into **run**.

For example:	becomes:
let x = "y" **in**	**let** x = **box** y **in**
eval x	**run** x
while:	becomes:
let f = **fun**$(z)\{mul(3,z)\}$ **in**	**let** f = **fun**$(z)\{mul(3,z)\}$ **in**
let y = "2" **in**	**let** y = **box** 2 **in**
let x = "f(" + y + ")" **in**	**let** x = **box**$(f($**unbox** $y))$ **in**
eval x	**run** x

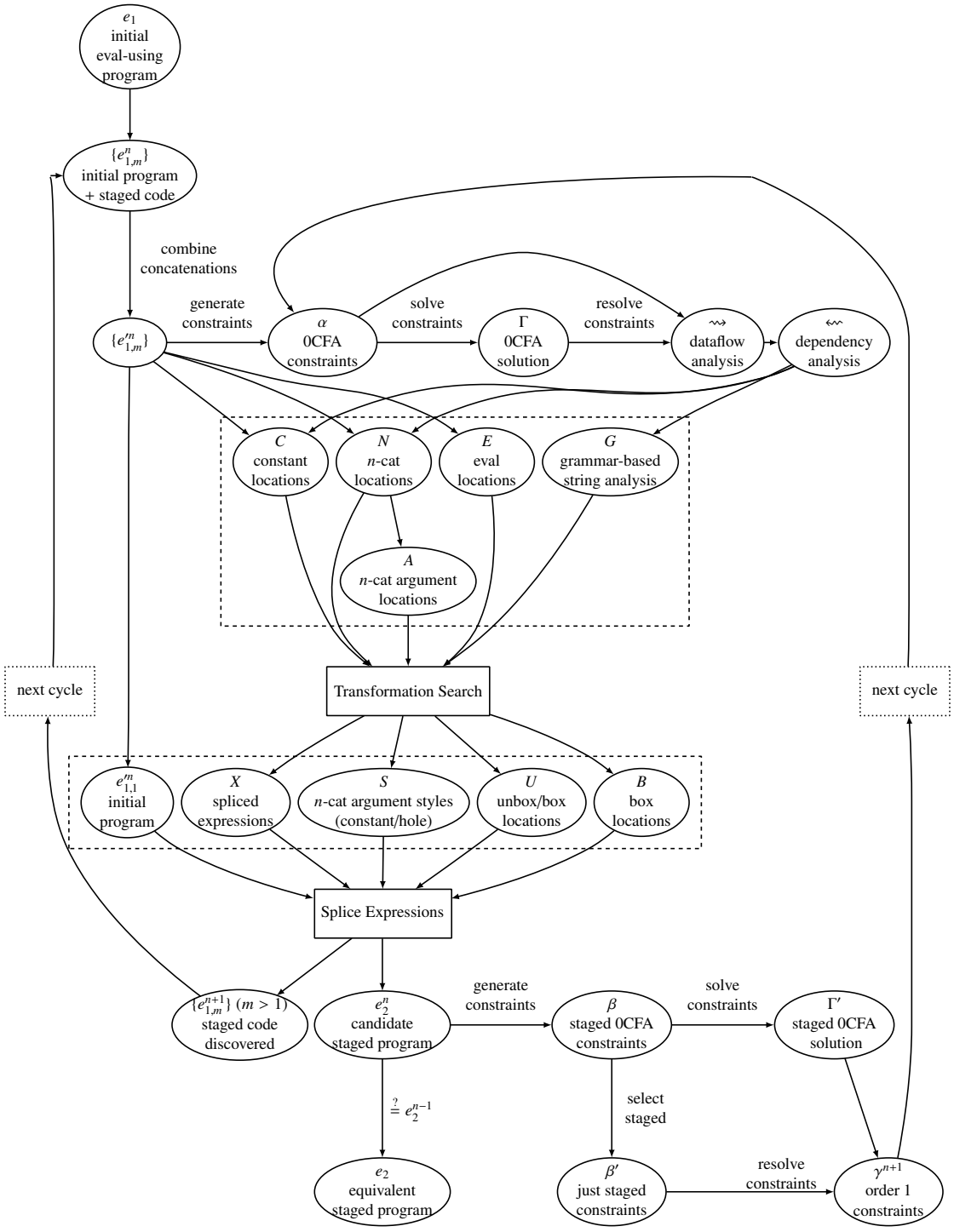


Figure 6.2: Data flow in the Boxing Algorithm

But in order to use it, certain conditions must hold:

1. we need a sound dataflow analysis for the target language, including metaprogramming constructs;
2. we need a string analysis for the target language that will produce a sound over-approximation of the string values that may occur at different program points or be bound to different variables;
3. the language must be parseable using `lex` and `yacc` or similar tools.

In Chapter 5, we saw how to apply 0CFA to SLamJS, so we reuse that analysis to satisfy condition 1. As 0CFA over-approximates the flow control of a program with a regular graph, it is easy to extract a grammar-based string analysis from the results; that satisfies condition 2, although there are many techniques that could improve upon this [CLKD06]. The lexer and parser for SLamJS are implemented using OCamlLex and OCamlYacc, which satisfies condition 3. INRIA's Prosecco has produced an unambiguous Menhir (`yacc` for OCaml) grammar for JavaScript, so it also meets this condition.

6.2.2 Building a Sequence of Program Approximations

In most static analyses, the goal is to construct an over-approximation of the behaviours of a program. In contrast, the goal of the Boxing Algorithm is to produce a staged program that exactly captures the behaviours of the original **eval**-using program. Although we cannot immediately analyse the behaviour of a program that uses **eval**, if we under-approximate the behaviour of **eval** by supposing it does nothing, we can analyse the behaviour of other parts of the program and use this as a starting point for building a sequence of increasingly accurate approximations that will hopefully converge to a fixed point that has exactly the behaviour of the whole program.

In particular, any information supplied by the analysis about the program's behaviour prior to the first execution of **eval** will be sound. This includes information about the content of the first strings executed as code. By analysing this code, we can construct a new approximation that will either reveal new information about the code executed, or show that we have enough information to capture the program's behaviour exactly.

We cannot simply analyse individual code strings, as there may be infinitely many possible such strings arising over all executions of the program. Fortunately, the staged metaprogramming analysis allows us to analyse the behaviour of possibly infinite sets of code values built using staged metaprogramming.

The approximation produced in the nth cycle of the fixed point process, if run, would behave the same as the original program up to the nth execution of **eval**. After that, the behaviours of the original and approximate program may diverge; in particular, the approximation may contain an odd mixture of staged metaprogramming and code string-based metaprogramming. If the algorithm reaches a fixed point, the transformed program, if executed, would behave the same as the original program.

6.2.3 Parsing Expressions Out-Of-Order

The Boxing Algorithm relies on using the existing parser for a language for part of the transformation; this makes the technique easily applicable to other languages. Typically, languages are parsed in two phases. The first phase, lexical analysis, splits the program text into a sequence of lexemes and transforms these into a sequence of tokens; the tool `lex` generates a *lexer* that does this using a deterministic finite state automaton. In the second phase, the actual parser turns the sequence of tokens into an abstract syntax tree; `yacc` generates a LALR (Look Ahead, Left-to-right, Rightmost derivation) *parser* to do this with a restricted form of pushdown automaton.

Normally, the lexer processes the characters in the program text in order from beginning to end. Similarly, the parser processes the token sequence in order from left to right (hence the second L in LALR). The Boxing Algorithm abuses these tools to process **eval**ed text out of order in fragments. Effectively, this gives us a finite way of parsing a string abstraction, which may encode infinitely many concrete strings of unbounded length. However, there is a price: we risk changing the meaning of the **eval**ed code. To avoid this (and hence keep the transformation sound), we must check that the lexing and parsing phases are unaffected by the change of order.

Parsing. In order to build an approximate analysis of an **eval**-using program, the algorithm transforms string operations on code values into uses of staged metaprogramming. (Note that the intermediate approximations may combine uses of **eval** with uses of staged metaprogramming.) The difficulty here is that, while an **eval**-using program might construct code strings using concatenation from left to right (or some other order), staged metaprogramming splices one well-formed expression into another, without respect for this order.

For example, consider a language (unlike SLamJS) with conventional arithmetic expressions. Using the usual precedence of arithmetic operators, the expression $1 + 2 * 3$ would evaluate to 7. We could use string concatenation to construct a string representation

of this code and **eval** it:

$$\begin{array}{l}\textbf{let}\ x = \texttt{"1 + 2"}\ \textbf{in}\\ \textbf{let}\ y = \texttt{"3"}\ \textbf{in}\\ \textbf{eval}(x + \texttt{" * "} + y)\end{array}$$

We might be tempted to transform this into the following representation using staged metaprogramming:

$$\begin{array}{l}\textbf{let}\ x = \textbf{box}(1 + 2)\ \textbf{in}\\ \textbf{let}\ y = \textbf{box}\ 3\ \textbf{in}\\ \textbf{run}(\textbf{box}((\textbf{unbox}\ x) * (\textbf{unbox}\ y)))\end{array}$$

But this would be incorrect, as it would change the bracketing of the expression from $1 + 2 * 3 = 1 + (2 * 3) = 7$ (according to the usual rules of precedence in arithmetic) to $(1 + 2) * 3 = 9$. (Arguably this may have been the intent of the author of the program.)

The problem arises because the grammar of expressions in the language is ambiguous. The proposed transformation corresponds to one possible parsing of `1 + 2 * 3` but not the one we would expect the language's parser to choose, assuming it was implemented to follow the usual rules of precedence.

If the language we wish to analyse is parseable using a deterministic context free grammar, we can parse fragments of program code out-of-order without changing the resulting expression. Hence we can safely transform a sequence of concatenations into the splicing of an expression into a template.

This is not an onerous requirement as, while `yacc` accepts ambiguous grammar specifications, it resolves the grammar ambiguity (perhaps arbitrarily) and produces a LALR parser for a more restricted, unambiguous grammar, which would itself be valid as a grammar specification.

Disambiguation in grammar specifications is often achieved by adding extra syntactic classes of expression. The example above might not be transformable in this case, as while x and y should clearly encode valid expressions, it might no longer be permissible to multiply arbitrary expressions. In this case, replacing the final concatenation with `"("+`x`+ ") * ("+`y`+")"` would allow the transformation. So we may be unable to transform some ostensibly reasonable programs because of the restriction to deterministic grammars, but it seems a reasonable sacrifice in order to ensure sound transformation of other programs.

We can easily modify the `yacc` grammar for SLamJS to support parsing of incomplete expressions containing holes by adding a new token named `HOLE`, representing a hole in an expression, and a new rule stating that `HOLE` is an expression. We write $\langle\bullet\rangle$ to mean a string that lexes to `HOLE`. In order to avoid misinterpreting text produced within a program as $\langle\bullet\rangle$, it ought to be treated a special character that cannot occur textually within a program or be produced by the string operations within the language.

Consider again the second example at the start of Section 6.2.1. Having added $\langle\bullet\rangle$ and `HOLE` to the lexer and parser respectively, we can treat the string generated by the concatenation `"f("` + y + `")"` as `"f(`$\langle\bullet\rangle$`)"`, which, as $\langle\bullet\rangle$ parses as an expression, also parses as an expression. In order to fill the hole, we also need to parse `"2"` as an expression. The algorithm's use of the parser is fairly straightforward: it still only parses complete expressions; the trick is that sometimes the parsing of their subexpressions occurs separately, those subexpressions having been replaced with holes. In contrast, other work on string analysis of code (see Section 3.4.3) tracks the parse stack produced by partially parsing code string fragments. This leads to a way of determining whether the code string must be a valid expression, but not what the content of that expression is.

Lexing. A further complication in parsing expressions out-of-order is that concatenation of code strings may change the boundaries and types of tokens produced during their lexical analysis.

This is unlikely to cause problems in SLamJS because the grammar is quite restrictive, but suppose we allowed function application to be written as juxtaposition of expressions without brackets, as in a typical functional language, and consider the following:

```
let x = if(y){"(g(3))"} else{"g(3)"} in
let z = "f" + x in
eval z
```

We might be tempted to transform it to:

```
let x = if(y){box (g(3))} else{box (g(3))} in
let z = box (f(unbox z)) in
run z
```

But this would be wrong in the case where y is false. Depending on the value of y, the values of z in the original program and its tokenisations might be:

```
"f(g(3))" ↦ VAR(f) LP VAR(g) LP INT(3) RP RP
"fg(3)"   ↦ VAR(fg) LP INT(3) RP
```

In transforming the concatenation assigned to z, we observe that x is not constant, so we tokenise it as `HOLE(x)`; the concatenation then tokenises as `VAR(f) HOLE(x)`. Treating `HOLE(x)` as a wildcard, this matches the first case, but not the second.

The problem in this case is that the use of the hole has changed the lexeme boundaries within the string and hence its tokenisation. Conceptually, we can view the lexer as a deterministic finite state transducer T that reads the string and emits tokens on lexeme boundaries. In order to avoid changing the tokenisation of code strings, we need to check

that, whenever code strings x and y are concatenated, it is true that $T(x \cdot y) = T(x) \cdot T(y)$. (If it is not true, it might be deliberate on the part of the programmer, but is more likely to indicate an error or a fragile piece of code.)

This hides some details of the problem, as in practice tokens that are identical in the automaton model often carry some data that distinguishes them (as with `VAR(f)` and `VAR(fg)` in the above example). What we really need to check is that concatenation does not change the positions of lexeme boundaries in the code string.

My solution is to look at the finite state transducer produced by the lexer generator in combination with the string analysis. Whenever we wish to treat a concatenation argument as a hole, we compute (an over-approximation to): the final states of the lexer having processed the last character of the hole string; and all possible first characters of the following argument. We then require that, for each possible final state: either it is one that emits a token (and hence the following character is irrelevant in how the hole string is lexed); or, for all possible following characters, the lexer immediately emits the same token. That is, each final state must correspond to a lexeme boundary. If the check fails, then the argument may not safely be treated as a hole.

6.2.4 Constraint Solution and Resolution

Recall that the staged metaprogramming analysis for SLamJS is formulated as a system of constraints. The constraints express the abstract values that may arise when evaluating a particular subexpression of a program. Solution of the constraints yields a function Γ such that $\Gamma(\ell)$ is a sound over-approximation of the abstract values that may occur at program point ℓ during execution.

The majority of constraints are "order 0" constraints of the form $a \in \Gamma(\ell)$ or "order 1" constraints of the form $\Gamma(\ell_1) \subseteq \Gamma(\ell_2)$. But in order to express the behaviour of function and code values, we need "order 2" constraints of the form $\forall BOX(e^{\ell'}) \in \Gamma(\ell_1).\Gamma(\ell') \subseteq \Gamma(\ell_2)$.

It is possible (although not most efficient [PS95]) to find the smallest solution to these constraints iteratively using a fixed point computation: initialise each $\Gamma(\ell)$ to be empty; consider each constraint in turn, adding more abstract values to some relevant $\Gamma(\ell)$ until it is satisfied; repeat the process until no constraint adds new values (and hence every constraint is satisfied).

Note that, for a fixed Γ, an order 2 constraint can be expressed as a set of order 1 constraints and an order 1 constraint can be expressed as a set of order 0 constraints. Obviously, when solving the constraints iteratively, Γ is not fixed, but this observation that higher order constraints can be *resolved* to lower order ones becomes important when considering how to use staged constraints in an **eval**-based program.

6.2.5 Combining Concatenations

The main premise of the Boxing Algorithm is that the syntactic structure of concatenation in reasonable **eval**-using programs will be similar to composition of templates via staged metaprogramming. However, they are unlikely to match exactly. In particular, if we consider an expression like $f(x)$ built with string concatenation by `"f(" + "x" + ")"`, the implicit bracketing of the concatenation is `("f(" + "x") + ")"`. As "$f($" is not a grammatically valid expression, we cannot represent this with staged metaprogramming.

The solution is to replace syntactically adjacent instances of binary concatenation in an expression with a single n-ary concatenation (or *n-cat*). This will not change the behaviour of the program in SLamJS, as concatenation is associative, although in other languages (such as full JavaScript) there may be subtleties arising from implicit string conversion or other peculiarities.

6.3 Cycle Description

Let us now consider the steps in a single cycle of the fixed point process that produces a transformation. Each cycle operates on the original program (plus, in later cycles, fragments of staged code and associated constraints).

We begin by combining adjacent concatenations in the original program to get e'_1. Next, we run our original analysis on the program e'_1 to obtain the relation $\rightsquigarrow$ that describes direct data flows between program points ℓ. If $\ell_1 \rightsquigarrow \ell_2$, then there is a direct flow from ℓ_1 to ℓ_2. We can view $\rightsquigarrow$ as an edge relation for a directed dataflow graph with program points as vertices. By reversing every edge in the graph, we obtain a dependency graph with edge relation $\leftsquigarrow$. We can use this to determine the program points from which code strings flow into **eval**s. We also use it to produce a naive, grammar-based string analysis G. (More advanced analyses have been widely studied and could be used, but for the purposes of this transformation, namely determining whether a program point ℓ yields a constant string, this seems to be adequate.)

We are now able to determine which program points are candidates for transformation:

- E — all occurrences of **eval** within e'_1;
- C — all constant strings on which an argument of an **eval** depends;
- N — all n-cats on which the argument of an **eval** depends;
- A — all arguments to n-cats in N.

E is determined syntactically from e'_1; the rest are determined using $\leftsquigarrow$.

For every path from an **eval** argument to a program point that produces a code string, the goal is then to transform the expression at that program point into one that produces an equivalent staged metaprogramming code value. If the code string produced is a constant and that constant parses correctly to an expression, then this is simply a matter of replacing it with the corresponding **box** expression. If the code string is produced by concatenation, it might not be constant. In this case, we attempt to turn it into a **box** template with **unbox** expressions to fill any holes in the template.

A code concatenation expression will consist of several subexpressions joined together. For each of these subexpressions, we must choose whether to treat it as a constant (which is obviously only possible if the subexpression is indeed constant, as determined by G), or to treat it as a hole. If we treat it as a constant, it becomes part of the code template, which we parse to produce the **box** expression. If we treat it as a hole, we introduce the obligation to transform any program point on a path from the subexpression that produces a code string.

Hence the process of choosing what and how to transform can be viewed as a depth-first search rooted at the occurrences of **eval**. If the search is successful, we produce:

- B — a set of program points where we may transform string constants into **box** expressions;
- U — a set of program points where we may transform string concatenations into **unbox** expressions;
- $X : (B \cup U) \to T$ — a mapping from transformed program points into code templates.
- $S : A \to \{h, c\}$ — a "style" specifying whether a concatenation argument should be turned into a *h*ole or treated as a *c*onstant.

A more formal description on the conditions that B, U, X and S must satisfy is given in Figure 6.3. (In addition, every template must satisfy the lexing check described in Section 6.2.3.)

It is possible that we are unable to find B, U, X and S satisfying these conditions, in which case the analysis fails. Otherwise, we transform e'_1 into a candidate e_2 by splicing in all the code templates from X and turning occurrences of **eval** into **run**.

In order to make explicit the relationship between the original and transformed subexpressions, we maintain the labels on the program points. For example, in transforming $\texttt{"f(x)"}^1$ into $(\textbf{box}(f^2(x^3))^4)^1$ we preserve the outermost label 1. Similarly, in transforming $(\texttt{"f("}^1 + y^2 + \texttt{")"}^3)^4$ into $(\textbf{box}(f^5(\textbf{unbox}\ y^2)^6)^7)^4$ we preserve the label 4 on the n-ary concatenation and the label 2 on the spliced argument.

Domain anti-restriction of a relation:	$x\ (A \square R)\ y$	$\overset{\text{def}}{\Leftrightarrow}$	$x\ R\ y \wedge x \notin A$
Restricted dependencies of a program point:	$\hookleftarrow$	$\overset{def}{=}$	$(N \cup C)\square \nleftarrow$
Cut of a program point:	$cut(\ell)$	$\overset{def}{=}$	$\{\ell' \mid \ell \hookleftarrow^* \ell'\} \cap (N \cup C)$
Cut lifted to a set of program points:	$cut(L)$	$\overset{def}{=}$	$\bigcup\{cut(\ell) \mid \ell \in L\}$
Arguments to **eval**:	D	$\overset{def}{=}$	$\{\ell \mid \exists \ell' \in E, e.\textbf{eval}(e^{\ell})^{\ell'}$ occurs in $e'_1\}$
Require that the transformed **box**/**unbox** locations cover the combined cuts of all arguments to **eval**:	$cut(D)$	$\subseteq$	$B \cup U$
Concatenated arguments treated as holes:	V	$\overset{def}{=}$	$\{\ell \mid S(\ell) = h\}$
Require that transformed locations cover cuts of all arguments to concatenation treated as holes:	$cut(V)$	$\subseteq$	$B \cup U$
n-cat arguments treated as constants:	W	$\overset{def}{=}$	$\{\ell \mid S(\ell) = c\}$
String analysis (for identifying constants):	$G(\ell)$	$\overset{def}{=}$	$\begin{cases} s & \text{if, according to analysis, } \ell \\ & \text{yields the constant string } s \\ undef & \text{otherwise} \end{cases}$
Require that locations transformed as constant parts of templates are indeed constant:	W	$\subseteq$	$dom(G)$
Parsing function:	$parse(s)$	$\overset{def}{=}$	$\begin{cases} e & \text{if } s \text{ parses to expression } e \\ undef & \text{otherwise} \end{cases}$
Require that locations transformed to constant **box**es agree with string analysis:	$\forall \ell \in B.X(\ell)$	$=$	$parse(G(\ell))$
Template fragment for a program point:	$frag\ (\ell)$	$\overset{def}{=}$	$\begin{cases} \langle\bullet\rangle & \text{if } S(\ell) = h \\ G(\ell) & \text{if } S(\ell) = c \end{cases}$
Template fragment lifted to n-cat arguments:	$frag\ (\ell)_i$	$\overset{def}{=}$	$\begin{cases} frag(\ell') & \text{if } \ell' \text{ is } i\text{th argument to } \ell \in N \\ undef & \text{otherwise} \end{cases}$
n-ary concatenation:	$\bigoplus \overline{s_i}$	$\overset{def}{=}$	$s_1 \ldots s_n$
Template for an n-cat :	$T(\ell)$	$\overset{def}{=}$	$\bigoplus \overline{frag\ (\ell)_i}$
Require that templates generated at n-cat locations agree with string analysis:	$\forall \ell \in U.X(\ell)$	$=$	$parse(T(\ell))$

Figure 6.3: Conditions on B, U, X and S in a cycle of the Boxing Algorithm

We can now analyse our approximation e_2 using our existing analysis for staged metaprogramming. This may reveal more information about the staging behaviour of the program. For example, we may find that there are new occurrences of **eval**, there are new strings used as code values, or that certain program points take on a wider range of code values than previously assumed. We need to transfer this information back to e'_1, so that we can transform it again, generating a new, more accurate approximation e'_2.

What we would like to do is augment the constraints generated in the analysis of e'_1 with the new constraints β (some of them staged) generated by the analysis of the candidate e_2. We cannot simply union the two sets of constraints, as this would introduce staged code values to the analysis of an **eval**-based program, which would be meaningless.

However, observing that the staged constraints are order 2 constraints, after solving β in our analysis Γ' of e_2, we can filter out just the staged constraints $\beta' \subseteq \beta$, then resolve them (relative to Γ') to order 1 set-inclusion constraints γ.

As the staged constraints generated in cycle n express the interaction between the unstaged parts of $e_1 = e^{n+1}_{1,1}$ and the staged code $e^{n+1}_{1,2}, e^{n+1}_{1,3}, \ldots, e^{n+1}_{1,k} = \{e^{n+1}_{1,m} \mid m > 1\}$ introduced in its transformation to e_2, these order 1 constraints will refer to labelled program points in both e_1 and the staged code. Consequently, in order to combine the staged constraints meaningfully with the constraints for e_1, we must also include the constraints generated by analysis of these new expressions.

So when we repeat the transformation cycle, we consider not just the original program e_1, but also the transformed code introduced in e_2, augmented with the resolved staged constraints γ' from the analysis of e_2. Note that it is now possible that we will have to transform expressions outside our original e_1.

We repeat the process until (hopefully) we reach a fixed point; that is, until the transformed expression is identical in two consecutive cycles. The rationale is that in cycle n, the transformed expression accurately models the behaviour of e_1 up to (at least) the $n+1$th use of **eval**. If we reach a fixed point at cycle k, the result of any cycle $k' > k$ will be identical. Hence, by an inductive argument, the transformed expression in cycle k accurately models any number of uses of **eval**. Note that, in contrast to simply executing the program up to the $n+1$th use of **eval**, cycle $n+1$ may accurately model infinitely many more uses of **eval** than cycle n. For example, if in the original program an **eval** occurs in a loop with an unbounded number of iterations (or one that does not terminate at all), the transformed program in some (finitely reachable) cycle $n+1$ may accurately model the loop up to any number of iterations.

6.3.1 Termination

It is not clear that the fixed point computation will terminate. In the case of 0CFA on a purely staged program, termination is guaranteed because we need only consider finitely many program points and finitely many abstract code values (representative of infinitely many possible constructed code values). However, both of these conditions are violated when we introduce new transformed code. That said, it seems unlikely that this would ever occur; it is more likely that the increasing density of constraints in the original program would lead to a cycle where no transformation was possible, so the algorithm would fail.

If possible non-termination is an issue, the algorithm can simply be set to terminate after a fixed number of cycles; it seems unlikely that a useful, realistic program will feature a pathological sequence of extracted constant code values, although it may be possible to construct one in a similar style to a Quine (self-replicating program). In fact, in the vast majority of examples, where the first cycle of analysis of **eval** does not reveal any new uses of **eval** or new code strings, two cycles are sufficient.

6.3.2 Complexity

Worst case. Within a cycle of the algorithm, there are two potential sources of exponential time complexity related to parsing. Parsing with an LALR grammar is $O(n)$ in the size of the string. It is possible to construct a program that assigns a constant string to a variable that is exponential in the size of the program, with none of the concatenated strings being parseable. For example, in the following program, *xn* holds a string of length $O(2^n)$:

let *x0* = "`fun(giant){`" **in**
let *x1* = *x0* + *x0* **in**
let *x2* = *x1* + *x1* **in**
...

An attempt to parse this string as a constant would take time $O(n2^n)$.

A further possible source of exponential time usage is the process of deciding whether to treat a concatenation argument as a hole or a constant. For an n-argument concatenation, there are 2^n possible choices of hole/constant. If many arguments are concatenated together in a single place, trying all choices may be very slow. In particular, combining this with the previous observation that parsing could take time $O(n2^n)$, this might take time $O(n2^{2n})$

Typical case. However, these exponential worst cases are unlikely to occur in practice. Concerning the size of parsed strings, note that a similar construction to the one above leads to exponential time usage in the widely-used Hindley-Milner type inference algorithm.

As for the choice of whether to treat strings as holes or constants, in practice, the number of strings concatenated in one place is usually small, so it does not cause a problem. I use the heuristic that as many arguments should be treated as holes as possible; this involves trying all possible choices in decreasing order of the number of holes. There are two benefits to this heuristic. Firstly, it produces a staged program that more closely follows the structure of the original program; treating a concatenation argument as a constant changes the program structure. Secondly, it gives a good chance of success without the need for backtracking. If we initially treat a code string that might or might not be constant as if it is constant, but later discover code that shows it not to be constant, we must return to where we originally made that choice, wasting any work done since. However, if we initially treat the code string as a hole, then even if we later discover evidence that it is not constant, it does not invalidate our original decision.

Of the other operations performed in a cycle, the most expensive asymptotically is the $O(n^3)$ cost of 0CFA; my experience is that in practice it is this cost that dominates the running time of the transformation. The rest of a cycle can be implemented as efficiently or more efficiently than this. For example, computing cuts of program points depends on a restricted form of reachability over a graph $O(n)$ in the size of the program and hence is also possible in $O(n^3)$ time.

In summary, while there are two possible sources of exponential time complexity, they are unlikely to be encountered on normal programs. Nonetheless, this may be of concern in a security setting, where an attacker might deliberately craft a program to trigger the worst case behaviour. So it would be important in any realistic application to ensure the algorithm can time out and report failure.

6.3.3 Interaction of Concerns

The algorithm brings together a wide range of ideas and techniques, including: lexical analysis, parsing and grammars (and the associated automaton models); control flow analysis; string analysis; the use of set constraints to express an analysis; heuristic search; and the use of fixed points to derive an accurate analysis.

Furthermore, all of these concerns interact. I use control flow analysis expressed using set constraints to produce a string analysis, which forms part of a heuristic search that is constrained by the parser of the language; this leads to the next approximation in a sequence of approximations produced as part of a fixed point process, with some of the set constraints produced by its analysis being fed back into the next round of control flow analysis.

It is unusual for an analysis technique to have to handle so many distinct interacting concerns simultaneously. However, it is necessary because **eval** allows essentially unconstrained behaviour to result from the value of a data string, with that behaviour being highly sensitive to changes in the string.

6.4 Worked Examples

I now illustrate in detail how the Boxing Algorithm transforms two **eval**-using programs. The first example focuses on how string concatenation is transformed into **box** and **unbox**; the second example demonstrates why multiple cycles are sometimes needed. I use the simple version of staged 0CFA as it is easier to follow than the improved version and is sufficiently accurate for these examples.

6.4.1 Handling String Concatenation

Look ahead to Example 20 on page 110. Recall that **let** $x = v$ **in** e is a shorthand for **fun**$(x)\{e\}v$. Desugaring **let** and combining adjacent concatenations gives the code shown in Figure 6.4 (with labels, as assigned by my implementation, on the left). Also shown is a graph of data and information flow in the program, as determined by the information flow analysis. Direct flows that pass values unchanged are shown by solid arrows; direct flows resulting from concatenation (which changes values) are shown by dotted arrows, as are indirect flows resulting from the use of a value influencing flow control.

The site of the **eval** has label 10 and its argument has label 9. That is, $E = \{10\}$ and $D = \{9\}$. The argument depends on the n-cat at 8, which in turn depends on the constant strings at 1, 2, 4 and 7. That is, $N = \{8\}$ and $C = \{1,2,4,7\}$.

Now we need to decide which locations B and U to transform. We need to transform strings that may flow to the argument of **eval**. As $D = \{9\}$ can only take string values that flow out of the n-cat at 8, $cut(D) = cut(9) = \{8\}$. We require that $cut(D) \subseteq B \cup U$, so (as $8 \in N$ and $U \subseteq N$ but $B \subseteq C$) we must have $8 \in U$.

As $8 \in U$, we need to find a code template $X(8) = parse(T(8))$ for the n-cat. In order to find the code template, we need to decide whether to treat each of the arguments in $A = \{4,5,7\}$ as constants or holes, so that we can decide on the corresponding code string fragments. That is, we need to find values for $S(4)$, $S(5)$ and $S(7)$.

As x^5 can be either "0" or "1", it is not a constant and must be treated as a hole; that is, $G(5) = undef$ and thus necessarily $S(5) = h$. (Otherwise, we would have $S(5) = c$ and $5 \in W$, which would violate the requirement that $W \subseteq dom(G)$.)

Original code:

```
      fun(x){
        fun(y){
          eval (
 9          y
10        )
11      }
        (
          ncat (
 4          "add("
          ,
 5          x
          ,
 7          ", 2)"
 8        )
12      )
13    }
      (
        if (
 0        true
        ) {
 1        "0"
        } else {
 2        "1"
 3      }
14    )
```

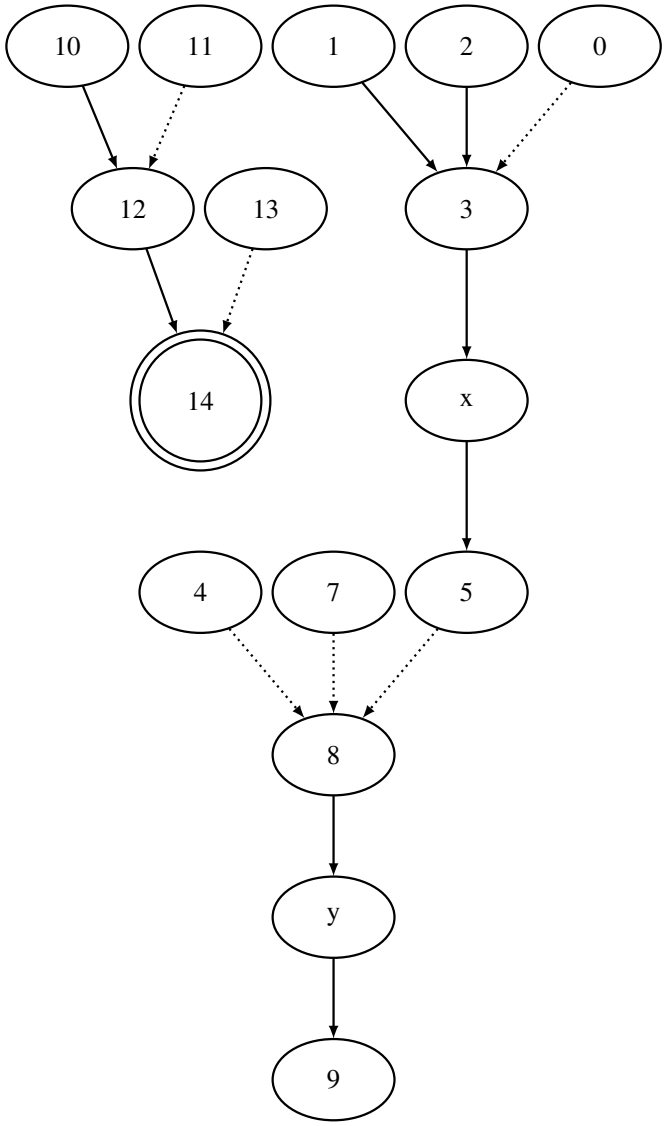


Figure 6.4: Boxing Algorithm concatenation example — cycle 1 input

Candidate staged code:

```
    fun(x){
      fun(y){
        run (
 9        y
10      )
11    }
      (
        box (
          add (
            unbox (
 5            x
18          )
            ,
901           2
902         )
 8        )
12    )
13  }
    (
      if (
 0      true
      ) {
        box (
200       0
 1      )
      } else {
        box (
300       1
 2      )
 3    }
14  )
```

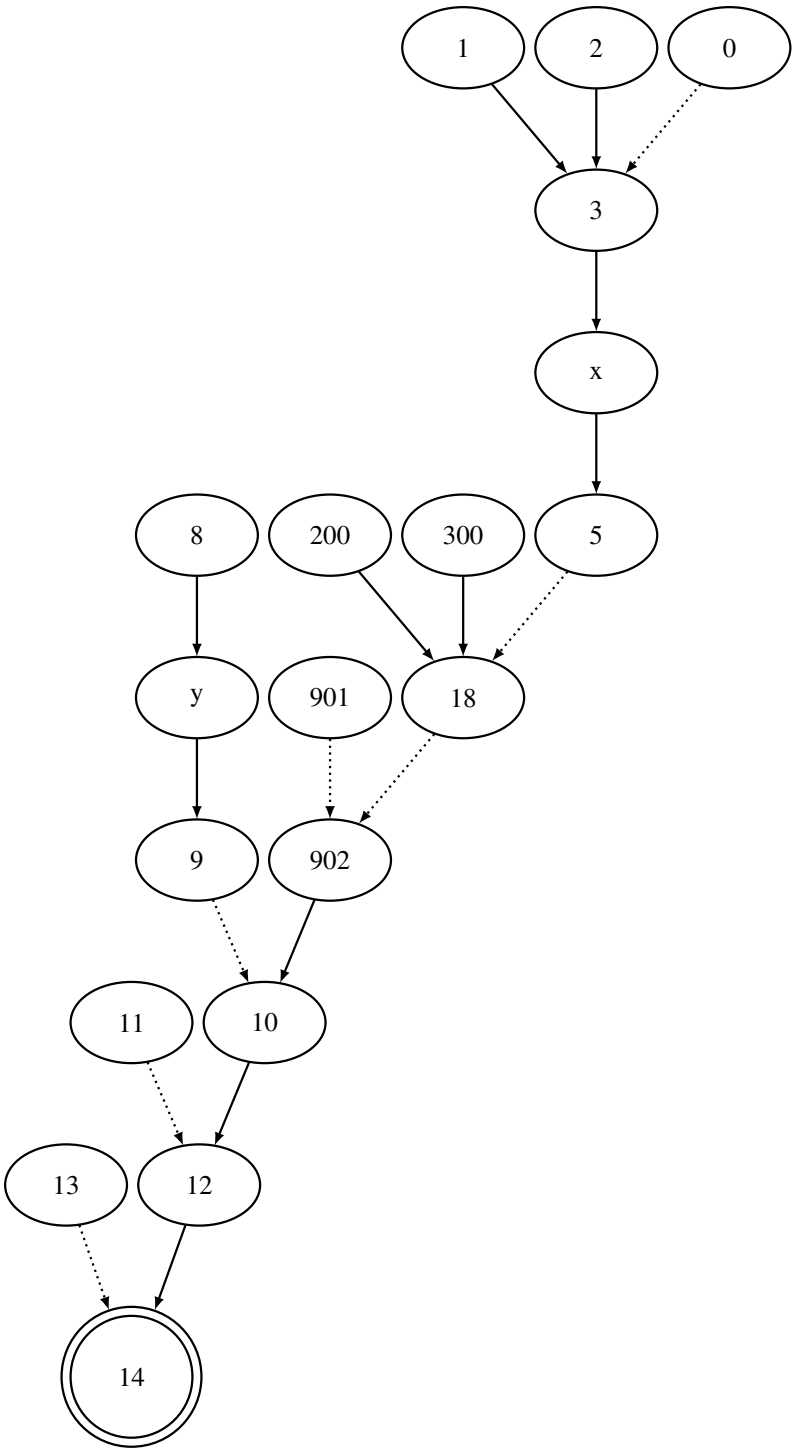


Figure 6.5: Boxing Algorithm concatenation example — cycle 1 output

Original code:

```
       fun(x){
         fun(y){
           eval (
 9           y
10         )
11       }
         (
           ncat (
 4           "add("
           ,
 5           x
           ,
 7           ", 2)"
 8         )
12       )
13     }
       (
         if (
 0         true
         ) {
 1         "0"
         } else {
 2         "1"
 3       }
14     )
```

Staged code:

```
       add (
18       <*>
       ,
901      2
902    )
```

and:

```
200    0
```

and:

```
300    1
```

Staged constraints:

$\Gamma(902) \subseteq \Gamma(10)$
$\Gamma(200) \subseteq \Gamma(18)$
$\Gamma(300) \subseteq \Gamma(18)$

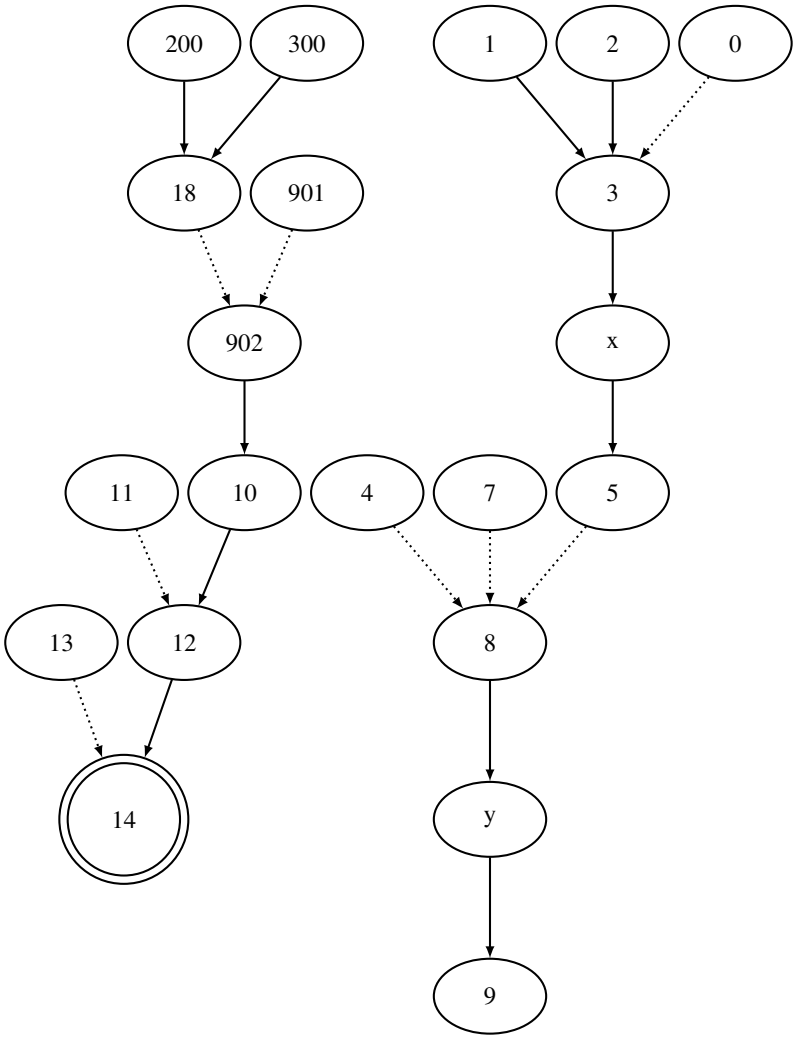


Figure 6.6: Boxing Algorithm concatenation example — cycle 2 input

As "add("^{4} and ",2"^{7} are constants, $G(4)$ = "add(" $\neq$ *undef* and $G(7)$ = ",2)" $\neq$ *undef*, so we have the option of treating them as constants and setting $S(4) = c$ and $S(7) = c$. However, heuristically, we usually get better transformation results by treating as many n-cat arguments as holes as is possible. But if we set $S(4) = h$ or $S(7) = h$, we would eventually introduce the requirement to parse these constants as expressions. This would fail, but we might not discover this until later, at which point we would either have to give up or backtrack to the point where the decision was made and try an alternative; neither is desirable. However, because 4 and 7 are constants, we can immediately check whether $G(4)$ and $G(7)$ can be parsed as expressions; they cannot, so we set $S(4) = c$ and $S(7) = c$.

Having decided on $S(4) = c$, $S(5) = h$ and $S(7) = c$, we obtain the fragments of the template at 8:

$$\begin{aligned} frag(8)_1 = frag(4) = G(4) &= \texttt{"add("} \\ frag(8)_2 = frag(5) &= \langle\bullet\rangle \\ frag(8)_3 = frag(7) = G(7) &= \texttt{",2)"} \end{aligned}$$

Hence $T(8)$ = "add($\langle\bullet\rangle$,2)" and $X(8) = parse(T(8)) = add(\langle\bullet\rangle, 2)$.

Now that we have decided whether to treat each argument as a hole or constant, we have the holes $V = \{5\}$ and the constants $W = \{4,7\}$. As remarked earlier, $G(4)$ and $G(7)$ are defined, so $W \subseteq dom(G)$ is satisfied and no further work is needed on that front.

But treating x^5 as a hole means we must transform the code strings that may flow into it, namely "0"^{1} and "1"^{2}. That is, we must have $cut(V) = cut(5) = \{1,2\} \subseteq B \cup U$. As they are both constant strings, we will transform them into **box** statements; that is, as $1 \in C$ and $2 \in C$, we must set $1 \in B$ and $2 \in B$. This means we have to find code templates $X(1)$ and $X(2)$ for these constant strings. Our choice is constrained to:

$$X(1) = parse(G(1)) = parse(\texttt{"0"}) = 0 \quad \text{and} \quad X(2) = parse(G(2)) = parse(\texttt{"1"}) = 1$$

We have found a transformation. Applying it to the original code gives our candidate e_2, as shown in Figure 6.5.

In order to check whether a single cycle is sufficient, we transfer the staged code and constraints back to the original program and perform the transformation again. Both the original and the staged code are shown in Figure 6.6. In this case, the staged constraints are:

$$\forall \texttt{BOX}(t'^{\ell'}) \in \Gamma(9).\Gamma(\ell') \subseteq \Gamma(10) \quad \text{and} \quad \forall \texttt{BOX}(t'^{\ell'}) \in \Gamma(5).\Gamma(\ell') \subseteq \Gamma(18)$$

These resolve to the unstaged constraints also shown in Figure 6.6, together with the resulting information flow analysis.

In this case, the new code and constraints have no effect on the transformation: the first staged constraint only introduces new values to the result of the **eval**; the other staged constraints only affect one of the arguments to the (staged) addition. As the result of the second cycle is the same as the first, we have reached a fixed point; the first candidate e_2 was correct, so we return it as the result of the algorithm.

6.4.2 Handling Nested Evals

In the previous example, a single cycle was sufficient to uncover all staged code. The second cycle was necessary only to check that the first cycle was correct. Let us now consider Example 23, in which the result of the first cycle reveals new code that must be transformed in the second cycle. The desugared, labelled code and the information flow graph are shown in Figure 6.7.

Initially, there is only one **eval** visible to the analysis, with label 3 and argument y labelled 2. Thus $E = \{3\}$ and $D = \{2\}$. The only constant that flows to the argument is the string `"eval x"` at label 1. There is no concatenation. So $C = \{1\}$ and $N = A = \{\}$, with $cut(2) = \{1\}$. It is straightforward to conclude that boxing the string at 1 will give a valid transformation: we have $B = \{1\}$, $U = \{\}$, $X(1) = (\textbf{eval}\ x)$ and (as A is empty) S undefined.

Applying the transformation gives the code shown in Figure 6.8 with corresponding information flow analysis.

Clearly, the transformation is not yet complete, as the code contains a use of **eval**. The analysis generates the staged constraint:

$$\forall \texttt{BOX}(t'^{\ell'}) \in \Gamma(2).\Gamma(\ell') \subseteq \Gamma(3)$$

which we resolve to $\Gamma(203) \subseteq \Gamma(3)$. Combining this with the original code and the code fragment **eval** x, we get the input to the second cycle, as shown in Figure 6.9. Once again, the graph shows the corresponding information flow analysis.

The staged code fragments and resolved constraints reveal new code to be transformed: an **eval** at 201 with argument x at 200, into which flows the constant string `"1"` at 0. Hence we now have $E = \{3, 201\}$, $D = \{2, 200\}$ and $C = \{0, 1\}$ with $cut(200) = \{0\}$. Again, it is easy to find a transformation, which in this case extends that of the previous cycle with $B = \{0, 1\}$ and $X(0) = 1$. Applying the transformation, we get the staged program shown (with information flow analysis) in Figure 6.10.

This looks more promising, as there are now no uses of **eval**. Still, we need to try another cycle to be sure. Again, we resolve the staged constraints:

$$\forall \texttt{BOX}(t'^{\ell'}) \in \Gamma(2).\Gamma(\ell') \subseteq \Gamma(3) \quad \text{and} \quad \forall \texttt{BOX}(t'^{\ell'}) \in \Gamma(200).\Gamma(\ell') \subseteq \Gamma(201)$$

Original code:

```
      fun(x){
        fun(y){
          eval (
2           y
3         )
4       }
        (
1         "eval x"
5       )
6     }
      (
0       "1"
7     )
```

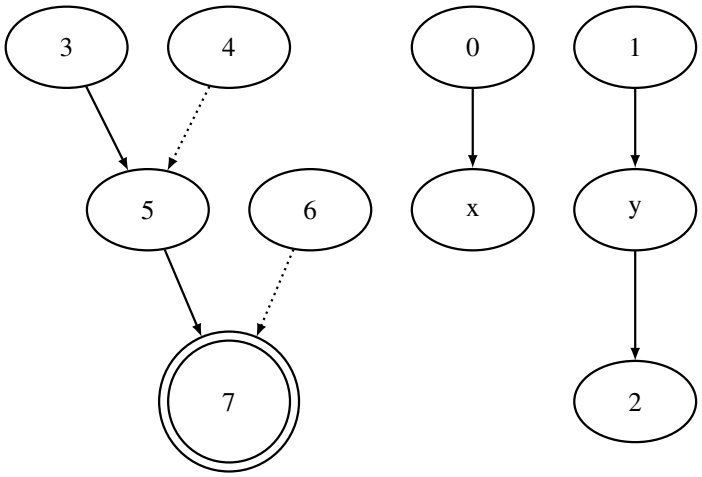


Figure 6.7: Boxing Algorithm nested example — cycle 1 input

Candidate staged code:

```
    fun(x){
      fun(y){
        run (
2         y
3       )
4     }
      (
        box (
          eval (
200         x
201       )
1       )
5     )
6   }
    (
0     "1"
7   )
```

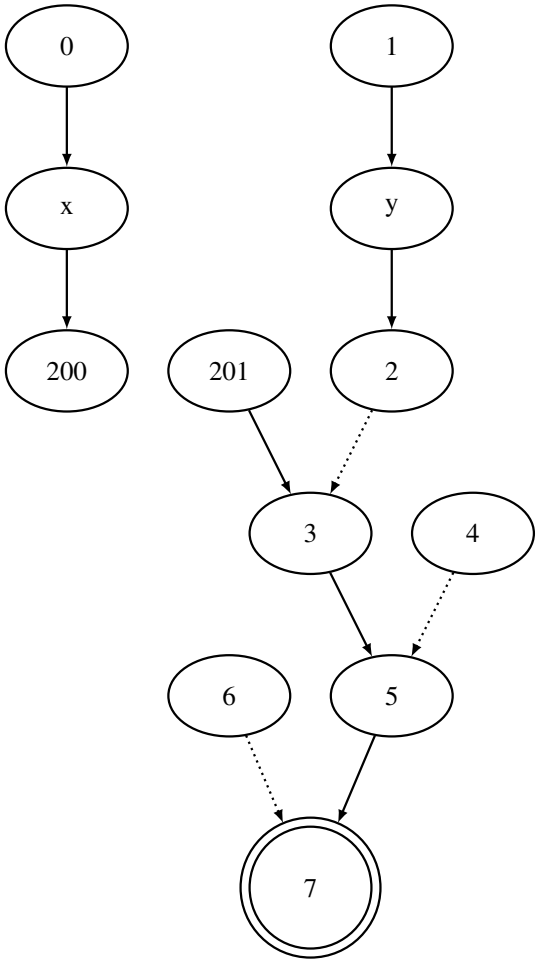


Figure 6.8: Boxing Algorithm nested example — cycle 1 output

Original code:

```
      fun(x){
        fun(y){
          eval (
2           y
3         )
4       }
        (
1         "eval x"
5       )
6     }
      (
0       "1"
7     )
```

Staged code:

```
      eval (
200     x
201   )
```

Staged constraints:

$\Gamma(203) \subseteq \Gamma(3)$

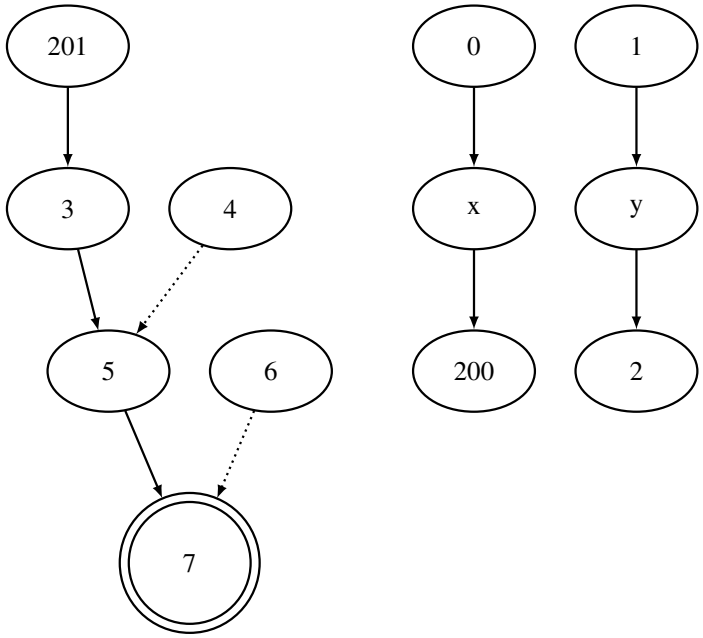


Figure 6.9: Boxing Algorithm nested example — cycle 2 input

Candidate staged code:

```
    fun(x){
      fun(y){
        run (
 2        y
 3      )
 4    }
      (
        box (
          run (
200           x
201         )
 1      )
 5    )
 6  }
    (
      box (
100      1
 0    )
 7  )
```

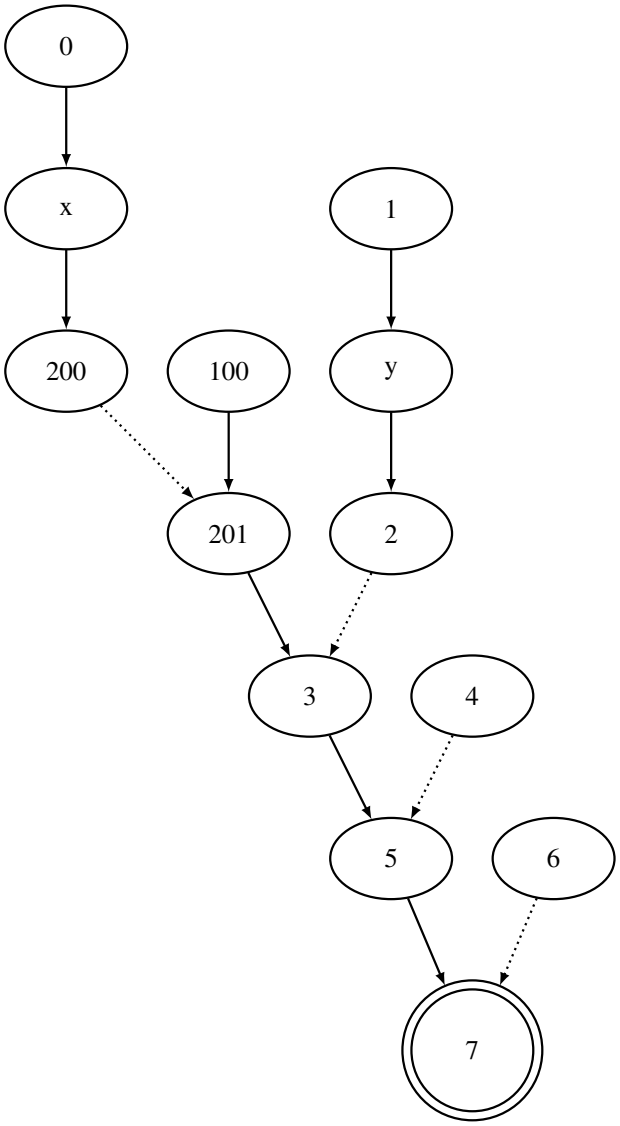


Figure 6.10: Boxing Algorithm nested example — cycle 2 output

Original code:

```
      fun(x){
        fun(y){
          eval (
2           y
3         )
4       }
        (
1         "eval x"
5       )
6     }
      (
0       "1"
7     )
```

Staged code:

```
      eval (
200     x
201   )
```

and:

```
100    1
```

Staged constraints:

$\Gamma(201) \subseteq \Gamma(3)$
$\Gamma(100) \subseteq \Gamma(201)$

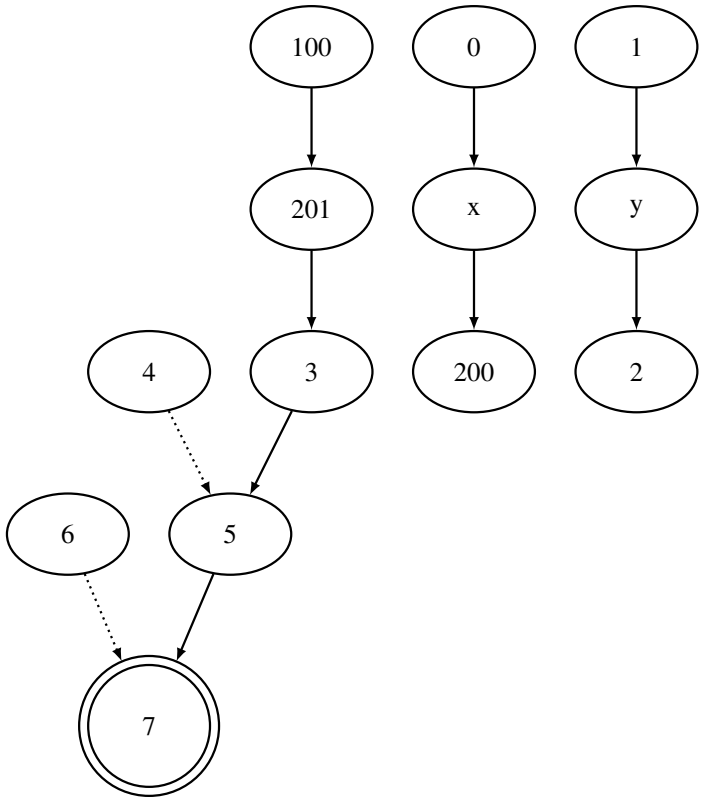


Figure 6.11: Boxing Algorithm nested example — cycle 3 input

producing $\Gamma(201) \subseteq \Gamma(3)$ and $\Gamma(100) \subseteq \Gamma(201)$. We combine them with the original code and staged code fragments, as shown with resulting information flow graph in Figure 6.11.

For cycle 3, both the transformation and output are as in cycle 2, so the output of cycle 2 was correct; we return it and terminate successfully.

6.5 Soundness

I now sketch a proof of the soundness of the transformation produced by the Boxing Algorithm. It would be desirable to give a detailed proof of the correctness of the algorithm, perhaps even going as far as the mechanised proof of the information flow analysis in the previous chapter. However, as discussed earlier, the algorithm tackles an unusually broad range of concerns. Correspondingly, a more formal and detailed proof would need to invoke many results about their behaviours. Furthermore, a significant amount of technical machinery would have to be introduced to track the interaction between these techniques.

In particular, it would require: a reformulation of the semantics of SLamJS in terms of a graph model (rather than the current term tree model) in order to track the correspondence between the initial expression and its staged approximations; and a modified proof of the information flow analysis to take account of the meaning of the resolved staged constraints and the staged code fragments. This would be a significant undertaking, but would be unlikely to aid significantly in the understanding of the algorithm or the clarity of its exposition, so I leave it for future work.

The transformation is developed through a sequence of approximations. So in order to argue about the correctness of the final result, we must argue about the correctness of the intermediate steps.

The input to the algorithm is an expression e_1 to be transformed. But cycle n takes as input not just $e^n_{1,1} \stackrel{def}{=} e_n$, but also (for cycle $n > 1$) fragments of staged code $e^n_{1,m}$ (where $m > 1$) and 0CFA constraints γ^n; note that the fragments of staged code do not themselves include any staging constructs. The cycle then produces, in addition to a candidate transformed program e^n_2, the fragments of staged code $e^{n+1}_{1,m}$ and constraints γ^{n+1} for the following cycle. So we must argue about the relationship of all these entities.

Lemma 3 (Boxing Cycle Soundness)**.** *In cycle n of the Boxing Algorithm:*

- *the candidate e^n_2 simulates the execution of e_1 up to and including at least the nth use of* **eval***;*
- *the fragments of staged code $e^{n+1}_{1,m}$ are those that, when spliced into e_1, yield e^n_2;*
- *the constraints γ^{n+1}, combined with the constraints generated by analysing $e^{n+1}_{1,m}$, yield all staged flows within e^n_2, and hence all flows in e_1 resulting from uses of* **eval** *up to and including its nth use.*

Proof. Cycle k of the algorithm begins by folding together adjacent string concatenations in the code $\{e^k_{1,m}\}$ to produce $\{e'^k_{1,m}\}$. As string concatenation in SLamJS is associative and the language does not have side effects, this will not change the result of evaluation; in particular, the result of evaluating $e^k_{1,1}$ and $e'^k_{1,1}$ is the same.

Next, the algorithm performs 0CFA on the code and (as we are considering only successful transformations) finds a transformation meeting the conditions in Figure 6.3. Applying that transformation to $e'^k_{1,1}$ gives the candidate e^k_2.

We now have to show a correspondence between configurations in the execution traces of these programs. The simulation is weak in the sense that several steps in the execution of one program may correspond to multiple steps in the other. In order to make the

correspondence more explicit, instead of considering ${e'}^k_{1,1}$ directly, we first replace all n-cat arguments treated as constants with their corresponding string; that is, we replace all subexpressions ℓ where $S(\ell) = c$ with $G(\ell)$. Next, merge any adjacent constant arguments to an n-cat; so $ncat("\texttt{x}","\texttt{y}", z)$ becomes $ncat("\texttt{xy}", z)$. Call the resulting expression f^k_1; it evaluates to the same result as ${e'}^k_{1,1}$ by correctness of the string analysis.

Our goal is now to show a correspondence between f^k_1 and e^k_2. We can achieve this by considering an augmented semantics in which we maintain the transformation $\boxtimes$ between the two expressions, tracking explicitly the correspondence $\boxdot$ between values that are code strings in f^k_1 and their equivalent staged values in e^k_2.

In order to formalise this idea fully, we would need a significant amount of technical machinery. For example, we would need a notion of expression template with multiple holes at arbitrary positions (as opposed to our existing notion of execution context, which permits a single hole that points to the next redex). In order to combine this with tracking the shared structure of the original and transformed program in the presence of nested operations on code strings or staged code, we would also have to switch to a graph-based formulation of the semantics [FT90]. (Simonet and Pottier use a succinct paired formalism to track common structure, but this does not permit nesting [PS03].) As this is unlikely to improve the clarity of my soundness argument, I instead focus on explaining the correspondence between f^k_1 and e^k_2 for reductions that manipulated code strings or staged code.

Write $\langle f, \boxdot, \boxtimes\rangle \sim e$ to mean: that the expression f occurring in the execution of f^k_1 corresponds to the expression e in the execution of e^k_2; and that $\boxtimes(f) = e$ (meaning that $\boxtimes$ transforms f into e); with $f' \boxdot e' = \textbf{box}\ (parse(f'))$ for any subexpression f' of f in the domain of $\boxdot$. Permit f' and e' to range over expression templates (as well as expressions).

Set $\boxdot_0$ to relate constant string expressions in f^k_1 to their transformed **box** expressions in e^k_2. That is, for all ℓ in B, the subexpression s^ℓ of f^k_1 has $s^\ell \boxdot_0 e^\ell$ for the subexpression $e^\ell = \textbf{box}\ X(\ell)$ of e^k_2. In addition, set $\boxdot_0$ to relate n-cat expressions in f^k_1 to their transformed **box** expression templates. That is, for all ℓ in U, the subexpression template $f^\ell = ncat(\overline{frag(\ell)_i})^\ell$ of f^k_1 has $f^\ell \boxdot_0 e^\ell$ for the subexpression $e^\ell = \textbf{box}\ X(\ell)$ of e^k_2. Set $\boxtimes_0$ to be the transformation from f^k_1 to e^k_2, so that $\boxtimes_0(f^k_1) = e^k_2$. Assert $\langle f^k_1, \boxdot_0, \boxtimes_0\rangle \sim e^k_2$ as the correspondence between the initial configurations of both programs.

Now, suppose for some $f, \boxdot, \boxtimes, e$ we have $\langle f, \boxdot, \boxtimes\rangle \sim e$ with f being at any point before the kth invocation of **eval** in the execution of f^k_1. We need to show that there are f', $\boxdot'$, $\boxtimes'$ and e' with $\langle f', \boxdot', \boxtimes'\rangle \sim e'$ and $f' \neq f$ or $e' \neq e$. There are several cases, depending on whether the next step in the execution of f' reduces an expression that is transformed and, if so, whether it was transformed to a **box**, **unbox** or **run**.

Case: Reduced subexpression was not transformed. If the reduced subexpression was not transformed, as f and e will otherwise have the same structure, the same reduction will apply to both. The reduction may copy code strings or staged code values respectively, but otherwise cannot interact with them. (Here we use the requirement that code strings are not used intensionally.) So $f \xrightarrow{\square} f'$ and $e \xrightarrow{\square} e'$. The construction of $\boxdot'$ and $\boxtimes'$ depends on the reduction rule (which may restructure the expressions), but is otherwise straightforward.

Case: Reduced subexpression originated from $\ell \in B$. The redex in f is a code string and in e is the corresponding **box** expression. The only rule affecting constants is environment propagation, so $f \xrightarrow{\square} f'$. We pick e' with $e \xrightarrow{\square}{}^* e'$ based on how many reductions are needed to propagate the environment throughout the **box** expression. The initialisation of $\boxdot_0$ means that the code string and staged value are already appropriately related, so as in the previous case, $\boxdot'$ and $\boxtimes'$ only change to reflect any change in structure.

Case: Reduced subexpression originated from $\ell \in A \cup N$. The redex in f is an n-cat or one of its arguments. The corresponding redex in e occurs either in a **box** expression that contains **unbox** expressions or within one of those **unbox** subexpressions. Recall that we constructed f_1^k by replacing constant arguments with their eventual values. So the only arguments to the n-cat that may require evaluation (before performing the concatenation) are those that correspond to **unbox** expressions. In fact, the evaluation of the arguments themselves requires no special consideration beyond this observation.

The reductions that do require consideration are the application of the rule (Unbox) in e ($\ell \in A$) and the evaluation of the concatenation in f ($\ell \in N$). Both of these correspond to the act of splicing together code templates, but in the case of the code string all the splicing happens after the spliced code has been evaluated, whereas in the staged case the splicing is interspersed with evaluation of the spliced code for each hole. In the first case, $f' = f$ and $e \xrightarrow{\square} e'$. $\boxdot'$ is constructed from $\boxdot$ by removing the mapping from f''^{ℓ} to $(\textbf{box}\ e'')^{\ell}$ and replacing the mapping from $ncat(f_1, \ldots, \bullet^{\ell}, \ldots, f_j)$ to **box** T with one from $ncat(f_1, \ldots, f''^{\ell}, \ldots, f_j)$ to **box** $T[e'']$. In the second case, $f \xrightarrow{\square} f'$ and $e' = e$. The reduction in f is $ncat(\overline{s_i})^{\ell} \xrightarrow{0} \oplus(\overline{s_i})^{\ell}$. Hence $\boxdot'$ is constructed from $\boxdot$ by replacing the mapping from $ncat(\overline{s_i})^{\ell}$ to $(\textbf{box}\ e'')^{\ell}$ with one from $\oplus(\overline{s_i})^{\ell}$ to $(\textbf{box}\ e'')^{\ell}$. In both cases, $\boxtimes'$ need only take account of the changed structure of the expression.

Case: Reduced subexpression originated from $\ell \in E$. The redex in f is an **eval** $f''^{\ell'}$ with $\ell' \in D$; the redex in e is a **run** $e''^{\ell'}$. $f''^{\ell'}$ must be a valid code string (or else f_1^k would not evaluate to a value). Furthermore, as $f''^{\ell'} \boxdot e''^{\ell'}$, it follows that $e'' = \textbf{box}\ (parse(f''))$. Hence **eval** $f''^{\ell'} \xrightarrow{0} parse(f'')$ and **run** $e''^{\ell'} \xrightarrow{0} parse(f'')$, giving $f \xrightarrow{0} f'$ and $e \xrightarrow{0} e'$ respectively. □

Theorem 6 (Boxing Algorithm Soundness)**.** *Consider a program e_1 written in SLamJS extended with* **eval** *but without staging constructs. If the Boxing Algorithm transforms e_1 to e_2 and $e_1 \xrightarrow{\square}{}^* v$, then $e_2 \xrightarrow{\square}{}^* v$. Furthermore, e_2 is a SLamJS program without* **eval**.

Proof. If the algorithm terminates, then $e_2 = e_2^k$ for some k, with $e_2^k = e_2^{k-1}$. Hence $\forall n \geq k . e_2^k = e^n$. Thus by Lemma 3, e_2 simulates the execution of e_1 up to any number of uses of **eval**. There are only finitely many uses of **eval** in the execution of $e_1 \xrightarrow{\square}{}^* v$, so e_2 simulates the entirety of the execution of e_1 and $e_2 \xrightarrow{\square}{}^* v$. The fact that e_2 does not use **eval** follows from the construction of the transformation. □

6.6 Implementation and Examples

I now consider some example programs on which my implementation of the transformation works and some on which it does not. As we are not immediately concerned with information flow analysis, but rather on turning **eval** into staged metaprogramming, my examples do not feature any dependency markers.

Example 20.

Original program:

```
let x = if(true){"0"} else{"1"} in
let y = "add(" + x + ", 2)" in
eval y
```

Transformed program:

```
let x = if(true){box 0} else{box 1} in
let y = box(add((unbox x),2)) in
run y
```

This example illustrates the basic concept that we turn constant strings (in this case, representing the numbers 0 and 1) into **box** expressions and concatenation (in this case, inside a template that adds 2) into **box** and **unbox** expressions.

Example 21.

Original program:

```
let id = "fun (x) { x } " in
let arg = if(true){"(1)"} else{"(2)"} in
eval(id + arg)
```

Transformation fails.

Although this is a seemingly reasonably program, in this case, the transformation fails because the concatenation does not match the syntactic structure of the language. As *arg* is not a constant, it must be treated as a hole for a code expression. According to the grammar of SLamJS expressions, function application has the form $e_1(e_2)$. So while $\textbf{fun}(x)\{x\}$, (1) and (2) are all valid expressions, the concatenation of two expressions in the final line does not constitute an expression; there is no grammatical form $e_1 e_2$ in the language. If the final concatenation had been *id* + "(" + *arg* + ")", it would have worked. The source of this problem in general is that parsing is not compositional, in that it is not true (for some $\cdot$) that $parse(s_1 \cdot s_2) = parse(s_1) \cdot parse(s_2)$.

Example 22.

Original program:	Transformed program:
let id = `"fun (x) { x } "` **in**	**let** id = `"fun (x) { x } "` **in**
let arg = `"(1)"` **in**	**let** arg = `"(1)"` **in**
eval$(id + arg)$	**run**(**box**(**fun**$(x)\{x\}(1)$))

Where possible, the transformation prefers to use **box** and **unbox** to preserve the structure of the original program. However, where it is not possible, it will try to use constant **box** expressions instead. This example is very similar to the previous one, but because the **eval**ed string is constant, the transformation can handle it, even though the concatenation does not follow the syntactic structure of the language. Here, the transformation determines exactly what the constant string passed to **eval** is and parses it directly, rather than trying to build it from its component subexpressions.

Note how, as *id* and *arg* are not transformed, they remain in the transformed program, but are unused. If we wish to preserve information flow in the transformed program, rather than just the final result, then the **box** expression must adopt any dependency markers that were on *id* and *arg*.

A similar concern applies if we wish to extend the transformation to handle a language with side effects, especially as they may affect which code strings are constructed. Suppose our original program contains an expression e that yields a code string s, but also has some side effects, such as incrementing a mutable variable used as a loop counter elsewhere in the source program. If we wish to transform e to a code value **box** e', we must be careful to preserve these side effects. We can do this by executing e and discarding the result; that is, transforming e not to **box** e' but to **let** $x = e$ **in box** e', where x is fresh.

Example 23.

Original program:	Transformed program:
let x = `"1"` **in**	**let** x = **box** 1 **in**
let y = `"eval x"` **in**	**let** y = **box** (**run** x) **in**
eval y	**run** y

All of the examples so far reach a fixed point in a single cycle of the algorithm, although a second cycle is required in order to check that a fixed point has been reached. This program requires two cycles to reach a fixed point: the first finds that the string in y is a code string for **eval** x; the second finds that the string in x is also a code string.

Example 24.

Original program:

```
let init = "0" in
let double = 5 in
let build = fun(loop){fun(n){fun(c){
 if(n = 0){c} else{((loop(loop))
 (sub(n,1))("add(" + c + ", 2)")}}}} in
let code = (build(build))(double)(init) in
eval code
```

Transformed program:

```
let init = box 0 in
let double = 5 in
let build = fun(loop){fun(n){fun(c){
 if(n = 0){c} else{((loop(loop))
 (sub(n,1))(box(add((unbox c),2)))}}}} in
let code = (build(build))(double)(init) in
run code
```

This is a functional implementation of a motivating example of Choi and others [CAYT11]. It builds the arithmetic expression `2 + ...+ 2` of unbounded size (determined here by the value of *double*) and executes it. The purpose of that example was to demonstrate the difficulty of handling string-based metaprogramming, in contrast with template-based staged metaprogramming, as methods for analysing the former often introduce imprecision (for example, in the form of an infinite expression). As we claim our analysis is applicable to JavaScript's **eval**, it is important that we can transform it exactly. As this example shows, we can.

Example 25.

Original program:

```
let gen_power =
   let f = fun(p){fun(n){
      if(n <= 0){"1"} else
      {let q = (p(p))(sub(n,1)) in
        "mul(x, " + q + ")"}
   }} in
   f(f) in
let power = fun(y){eval(
 "fun(x) { " + (gen_power(y)) + " }")} in
let raise5 = power(5) in
raise5(2)
```

Transformed program:

```
let gen_power =
   let f = fun(p){fun(n){
      if(n <= 0){box 1} else
      {let q = (p(p))(sub(n,1)) in
        (box(mul(x,unbox q)))}
   }} in
   f(f) in
let power = fun(y){run(
 box(fun(x){unbox(gen_power(y))})} in
let raise5 = power(5) in
raise5(2)
```

The staged power function is a staple of literature on metaprogramming [BT10]. Given a number (in this case 5), it generates code for a function that takes an argument and raises it to that power. This can be useful in performance-critical situations, as it avoids the overhead of using a loop. While it is unlikely that anyone would use JavaScript in such a situation, the ubiquity of the example demands that we should be able to handle it.

Note that, in general, any program that only uses constant code strings (with no concatenation or any attempts to perform intensional operations) and does not feature nested uses of **eval** can be transformed successfully by the algorithm. There are many more interesting programs that can also be successfully transformed, but I do not have a succinct characterisation of them.

Chapter 7

Conclusions

7.1 Summary of Results

I have presented a fully static information flow analysis based on 0CFA for a dynamically typed language with staged metaprogramming, implemented it and formally proved its soundness in Coq. I have shown how to apply the analysis to a language with string-based **eval** via a transformation to staged metaprogramming; this transformation has also been implemented. I believe the approach is transferable to other CFA-style analyses and applicable to JavaScript.

7.2 Comparison with Previous Work

As noted in Chapter 3, there is a wide body of previous work in each of the individual areas relevant to my research. However, as shown in Figure 3.1, there is little in the intersection of these areas. In order to demonstrate the contribution presented by my work, I now compare it with some of the most similar previous work.

7.2.1 Dynamic Analyses

While there is much work on dynamic analysis of JavaScript and information flow, some of it including support for **eval**, I consider it largely irrelevant to my work on static analysis. As discussed in Chapter 3, dynamic methods are fundamentally ill-suited to information flow analysis, even when these are mitigated with the *no sensitive upgrade* check [Zda02] and its ilk. Furthermore, it is far preferable to detect potential information leakage before deployment or execution of a program, rather than at run-time, even though this is may be a significantly harder problem.

The hybrid approach of Chugh and others [CMJL09] has some relevance to my analysis; they combine a static information flow analysis with a dynamic component that performs additional checks at runtime when dynamically generated code becomes available. The static part of their analysis is similar to mine (minus staging), although they do not formally state or prove its soundness. Their study of JavaScript on popular websites suggests the static part is precise enough to be useful. Because the additional checks on dynamically generated code occur at runtime, they must necessarily be quick and simple to avoid performance degradation. Consequently, these checks are limited to purely syntactic isolation properties, with a corresponding loss of precision. My fully static analysis does not suffer from these limitations.

7.2.2 Static Analysis of Staged Metaprogramming

Many different approaches to staged metaprogramming have been proposed. The staging constructs of SLamJS are modelled after the language λ_S of Choi and others [CAYT11]. However, the semantics of variable capture are different. For example, SLamJS allows the program:

$$(\textbf{fun}(x)\{\textbf{run}\ (\textbf{box}\ x)\}(1))$$

which behaves much like this JavaScript program:

```
(function (x) {return eval("x")})(1);
```

Meanwhile, the work of Inoue and Taha avoids variable capture [IT12] entirely.

Recall that Choi and others propose an unstaging translation that replaces staged metaprogramming with function abstractions and applications. There are some limitations to their unstaging translation. Most significantly, many interesting programs, such as the one above, are not valid in λ_S and hence cannot be unstaged using their translation; this limits its applicability to JavaScript. Furthermore, as shown in Examples 17–19, the precision of the resulting combined analysis is highly sensitive to the target language encoding used in the translation and the behaviour of the target language analysis. While their approach is useful as a quick way of adding staging to an existing language and analysis, I argue that staging constructs are sufficiently important and complex that we should aim to analyse them directly.

Kim and others [KLL+09] also consider control flow analysis with staged metaprogramming. However, as they do not discuss how to analyse staged code and do not allow staged code to generate new code, their treatment neglects the more difficult aspects of the problem.

7.2.3 Analysing Eval

There has been relatively little work on analysing **eval**. Evalorizer [MRMV12] is not really comparable as it collects uses of **eval** dynamically and suggests equivalent code interactively to its user, rather than providing fully automatic reasoning or transformation.

Unevalizer [JJM12], which is based around TAJS, seems to be the only current tool for analysing **eval** fully automatically. In contrast to my approach of transforming **eval** into a better-behaved form and then analysing that, it aims to recognise common patterns of **eval** usage and remove them in a single step, replacing them with code that does not use metaprogramming. Unlike my approach, it lacks generality: new usage patterns must be manually added and justified. Unfortunately, the combination of JavaScript's semantic peculiarities and the ad-hoc approach means that it takes considerable effort to reason about the correctness of each new transformation pattern. However, because it is based on TAJS, it is able to handle all of JavaScript. Of the examples in Section 6.6, Unevalizer is only able to handle Example 22.

Concurrently with the development of my analysis, Keil and Thiemann [KT13] showed how to extend TAJS to perform a dependency analysis. They suggest that this could be combined with the work on Unevalizer to produce a dependency analysis for JavaScript with **eval**, although they do not yet seem to have done this.

Compared with my discussion of the Boxing Algorithm, the unpublished work of Charlton [Cha11] gives a more thorough account of how the grammar of a language affects reasoning about **eval**. However, it has not led to any automated analysis tools.

7.3 Future Work

Progressing from here, there are a few obvious lines of work:

1. improve the precision of the analysis;
2. broaden the range of JavaScript language features that it can handle;
3. develop the infrastructure in which the analysis could be applied;
4. investigate further the behaviour of the Boxing Algorithm.

I think the major theoretical problems in the first two lines have already been solved, but it would take significant time effort to combine them all in a single analysis tool. Conversely, the question of how best to build infrastructure around such an analysis is still open, as are some questions about the behaviour of the Boxing Algorithm.

7.3.1 Improving Precision

At present, the information flow analysis for SLamJS is relatively coarse. It should be possible to improve the precision of the analysis by applying its ideas to CFA2 or using results from abstract interpretation. As these are the approaches taken by the existing JavaScript analysis tools jsctags [VS11] and TAJS [JMT09], this ought to be both possible and effective. Indeed, CFA2 is likely to be particularly useful, as it was developed with JavaScript in mind.

The handling of object reads and writes is particularly bad; some work by Liang and Might on analysis of programs using Python hash maps [LM12], which are conceptually similar to JavaScript objects, may prove instructive in improving this aspect of the precision of the analysis.

7.3.2 Handling More Features

The language SLamJS incorporates only a subset of JavaScript's features; it does not include imperative flow control or exceptions. As jsctags and TAJS handle full JavaScript (excluding **eval**), it is clearly possible to handle the remaining features, and looking at these tools would be a good starting point for work in that direction.

Recall that SLamJS was developed from λ_{JS} by removing features and adding staged metaprogramming. λ_{JS} was chosen because, in contrast to most other "core calculi" for JavaScript, there is a compositional, semantics-preserving translation from JavaScript (excluding **with** and **eval**) into λ_{JS} [GSK10]. Consequently, an obvious way to proceed would be to add the features removed from λ_{JS} back into SLamJS.

The problem with this is that, in order to handle **eval**, we would somehow have to relate the **eval**ed JavaScript code to the corresponding λ_{JS} code at analysis time; in essence, the problem would become how to handle heterogeneous metaprogramming, rather than homogeneous metaprogramming. As the translation is compositional, it ought to be the case that the translation of JavaScript into λ_{JS} commutes with the composition of code templates, but this is a claim that would need to be reasoned about very carefully.

One area that would require particular attention in handling full JavaScript is the range of different possible scoping behaviours for **eval**. SLamJS currently models how **eval** executes code with dynamic variable scoping, but does not consider quirks such as running code in the global scope when **eval** is called indirectly via an alias. The latest version of JavaScript complicates matters by introducing a new "strict" mode of **eval** (which generally behaves better) but retaining the old behaviours for backwards-compatibility. The latest version of λ_{JS}, called S5 [Kri12], goes to some lengths to model the different variations

in scoping rules. To take account of the different scoping rules, the boxing transformation would need to include a static analysis to determine in which modes each instance of **eval** can be run.

7.3.3 Infrastructural Issues

In considering applications of a JavaScript information flow analysis, one should also consider when it will be run, by whom and in what environment. So let us now discuss more generally the infrastructural requirements for static analysis of Web applications.

Where to use an information flow analysis. In the context of information flow analysis of JavaScript, the precision of dynamic analysis is good enough for many practical situations, even though there are inherent limitations. By definition, dynamic analysis occurs when code is run. For client-side JavaScript, that means being run by a user, in a browser; this usually requires a modified browser, although monitoring within JavaScript is possible [MT09].

Hence the primary advantage of static analysis of code is its ability to identify possible errors or security problems before code is run, during development or at deployment. So while one could imagine a static analysis being part of a browser, it would make more sense for it to be used on the webserver or by the developer.

This scenario introduces further challenges, as a complete Web application is unlikely to be a single, monolithic piece of code. For example, it may use libraries, which may be loaded incrementally during execution. This is not especially problematic, as the developer presumably knows which libraries could be loaded, has access to their code and may have control over the server from which they are loaded. So in principle, the application could be treated as a single piece of code (including all libraries), although this might not scale well to large libraries.

In the event that the developer does not control which version of the libraries is served, the application would need a mechanism for identifying which library has been loaded and a policy to follow if an unknown library is loaded. For example, the application code could include hashes of libraries against which it has been analysed. The browser would be responsible for checking hashes of libraries as they are loaded. In the event that an unknown library is loaded, either the application could fail or it could fall back to some dynamic analysis.

The situation becomes more difficult when third-party content (such as adverts or user-supplied plugins) is involved. In this case, some of the code might not even be written until after development of the core application. If all code, including adverts and plugins,

is served from the same webserver, then it is still the case that in principle the application could be treated as a single piece of code. However, authors of adverts and plugins might need to take some responsibility for ensuring their code was amenable to static analysis.

Once again, if the code is not centrally distributed, the application needs a mechanism for identifying well-behaved code, but it is no longer possible to use a whitelist. Instead, the application would have to include some form of software contract specifying acceptable behaviours of third-party content and the browser would need to stage static analysis of code as it was loaded to check this contract. In order for this to be practical, either the contract would need to be easily verifiable syntactically [CMJL09], or the content would need to include an easily verifiable certificate witnessing that it satisfies the contract. It is not clear how to achieve this while still supporting expressive information flow policies.

The environment of execution. As Web applications execute in the context of a webpage in a browser, a practical analysis must also model how a webpage interacts with code via the DOM. Webpage elements provide new channels for information to flow between fragments of JavaScript code. They can also be used to add new JavaScript code, in a manner similar to **eval**. TAJS models the DOM to some degree [JMM11], but from a security perspective, the problem is complicated somewhat by differences between browser implementations; we need a model of the DOM that is accurate and accounts for the behaviours of both the relevant specifications and various browser implementations.

The choice of information flow policy. My work has focused mainly on information flow analysis as a means for enforcing noninterference. However, this may be too strict a policy for a practical Web application; we may need to allow for limited declassification [LZ05]. Furthermore, a practical Web application is not simply a program that takes inputs, runs once, then gives output: it may interleave input and output throughout its execution. From this perspective, it may be more appropriate to model an application as a reactive system [BPS+09]. While these aspects are largely orthogonal to my work, in the sense that it seems relatively straightforward to extend existing analyses to accommodate them (once the relevant theory has been developed), they are a further complication that must be considered in a practical setting.

7.3.4 Investigating the Boxing Algorithm

There are a few unsolved questions concerning the behaviour of the Boxing Algorithm. Firstly, while I believe that non-termination is unlikely, and have an intuition to support this belief, it would be good to prove or disprove it. Secondly, while I have identified a

very restrictive class of **eval**-using programs that the algorithm can handle, it can clearly go far beyond this (as shown by some of the examples); a less restrictive characterisation of programs for which a successful transformation is guaranteed would strengthen the appeal of the algorithm.

From a more theoretical perspective, it would be desirable to produce a more formal (and possibly mechanised) proof of the soundness of the algorithm, particularly if it is to be used as the basis for a security analysis. It may also be interesting to investigate whether restricting the transformation to a simpler class of programs (for example, those without nested uses of **eval**) would lead to a simpler algorithm and hence a simpler proof.

There is also some scope for improving the algorithm, for example by identifying better heuristics when searching for a transformation, or predicting when the worst case exponential time complexity may occur and avoiding it.

7.4 Conclusion

It is both necessary and possible to analyse statically the behaviour of JavaScript's metaprogramming construct **eval**, but the complexity of the task means it is best split into two problems: a transformation to a more principled form of metaprogramming and an analysis of this more principled form. I have now developed the theory to handle both problems.

All the pieces are now in place for an interesting, sound and principled information flow analysis of JavaScript with **eval**, but it will take significant effort to bring them together and handle the full language in a practical setting.

Appendix A

Coq Formalisations of Proofs

A.1 Library Functions — lib.v

```
(** Some utility definitions. *)
Set Implicit Arguments.
Require Import String List.

(** A Haskell-like Eq type class. *)
Class Eq A := {
  cmp : forall (a a': A), {a=a'}+{a<>a'}
}.
Infix "==" := cmp (at level 30).
Notation "x =?= y" := (if cmp x y then true
    else false) (at level 30).

Instance StringEq : Eq string := { cmp :=
    string_dec }.
Instance NatEq : Eq nat := { cmp := Peano_dec.
    eq_nat_dec }.

(** A lightweight formalisation of
    finite maps as alists. *)
Section FinMap.
  Variables A B : Type.
  Context `{Eq A}.

  Inductive finmap : Type :=
  | NilMap : finmap
  | ConsMap : A → B → finmap → finmap.

  Fixpoint ext_map m a b :=
  match m with
  | NilMap ⇒ ConsMap a b NilMap
  | ConsMap a' b' m' ⇒ if a == a' then
      ConsMap a b m' else ConsMap a' b' (
      ext_map m' a b)
  end.
  Functional Scheme ext_map_ind :=
      Induction for ext_map Sort Prop.

Fixpoint lookup (m:finmap) (a:A) :=
match m with
| NilMap ⇒ None
| ConsMap a' b' m' ⇒ if a == a' then Some b'
    else lookup m' a
end.

Fixpoint codom (m:finmap) : list B :=
match m with
| NilMap ⇒ nil
| ConsMap _ b m' ⇒ b :: codom m'
end.

Fixpoint mapp (m m':finmap) :=
match m with
| NilMap ⇒ m'
| ConsMap a b m ⇒ ConsMap a b (mapp m m')
end.

Inductive mforall (P:A→ B→ Prop) : finmap
    → Prop :=
| MForallNil : mforall P NilMap
| MForallCons : forall a b m, P a b →
    mforall P m → mforall P (ConsMap a b m)
    .

Fixpoint del_map m a :=
match m with
| NilMap ⇒ NilMap
| ConsMap a' b m ⇒ if a == a' then m else
    ConsMap a' b (del_map m a)
end.
```

```
Lemma lookup_ext : forall m a b, lookup (
    ext_map m a b) a = Some b.
Proof.
  intros; functional induction (ext_map m
      a b); simpl.
    destruct (a == a); congruence.
    destruct (a == a); congruence.
    destruct (a == a'); congruence.
Qed.

Lemma lookup_ext2 : forall m a a' b, a <> a'
    → lookup (ext_map m a b) a' = lookup m a
    '.
Proof.
  intros; functional induction (ext_map m
      a b); simpl.
    destruct (a' == a); congruence.

    destruct (a' == a); congruence.

    specialize (IHf H0).
    destruct (a' == a'0); congruence.
Qed.

Lemma ext_map_ext : forall m a b b',
    ext_map (ext_map m a b) a b' = ext_map
    m a b'.
Proof.
  intros; functional induction (ext_map m
      a b); simpl.
    destruct (a == a); congruence.
    destruct (a == a); congruence.
    destruct (a == a'); congruence.
Qed.

Lemma mforall_app : forall P m m', mforall P
    (mapp m m') ↔ mforall P m ∧ mforall P
    m'.
Proof.
  Hint Constructors mforall.
  induction m; simpl.
    constructor; eauto.
    destruct 1; auto.

    constructor.
      inversion 1; subst.
      rewrite IHm in H5.
      destruct H5; auto.

      destruct 1.
      inversion H0; subst.
      constructor; auto.
      apply IHm; auto.
Qed.

Lemma mforall_lookup : forall P m a b,
    lookup m a = Some b → mforall P m → P a
    b.
Proof.
  intros.
  Functional Scheme lookup_ind :=
      Induction for lookup Sort Prop.
  functional induction (lookup m a);
      inversion H0; subst.
    inversion H1; auto.
    inversion H1; auto.
Qed.

Lemma mforall_monotonic : forall (P P':A→
    B→ Prop), (forall a b, P a b → P' a b) →
    forall m, mforall P m → mforall P' m.
Proof.
  induction 2; constructor; auto.
Qed.

Lemma mforall_ext : forall P m, mforall P m
    → forall a b, P a b → mforall P (
    ext_map m a b).
Proof.
  intros.
  functional induction (ext_map m a b);
      auto.
    inversion H0; subst; auto.
    inversion H0; subst; auto.
Qed.

Lemma mforall_del : forall P m, mforall P m
    → forall a, mforall P (del_map m a).
Proof.
  induction 1; simpl.
    constructor.

    intro a'.
    destruct (a' == a); auto.
  Qed.
End FinMap.
Implicit Arguments NilMap [[A] [B]].

Section MapMap.
  Variables A B C : Type.
  Variable f : B → C.
  Context `{Eq A}.

  Fixpoint mmap (m:finmap A B) :=
  match m with
  | NilMap ⇒ NilMap
  | ConsMap a b m' ⇒ ConsMap a (f b) (mmap m')
  end.
```

```
Lemma mforall_map : forall (Q:A→ C→ Prop)
    m,
  mforall Q (mmap m) ↔ mforall (fun a b ⇒ Q
      a (f b)) m.
Proof.
  constructor.
    induction m.
      constructor.

      inversion 1; subst.
      constructor; auto.

    induction m.
      constructor.

      inversion 1; subst.
      constructor; auto.
Qed.

End MapMap.

Section MapRed.
  Variables A B C : Type.
  Variable f : A → B → C → C.
  Variable e : C.
  Context `{Eq A}.

  Fixpoint mfold_right (m:finmap A B) :=
  match m with
  | NilMap ⇒ e
  | ConsMap a b m' ⇒ f a b (mfold_right m')
  end.
End MapRed.

Notation "𝒫  A" := (list A) (at level 20).
Infix "∈ " := In (at level 40).
Infix "⊆ " := incl (at level 40).
```

A.2 Language Definition — lambdajs.v

```
(** Formalisation of syntax and
    semantics of a staged subset of
    LambdaJS.

    To facilitate the correctness proof
        of the binding analysis,
    this formalisation works with
        labelled expressions. *)
Set Implicit Arguments.

Require Import Eqdep Bool String List.
Require Import Program.Equality.
Require Import lib.

(** For concreteness, we define labels
    to be lists of natural
    numbers. Any definition would do,
        however, since we only carry
    labels around and never need to
        create new ones. *)
Definition label := list nat.

(** There are five kinds of constant
    literals: booleans, numbers,
    strings, and singleton literals
        representing the undefined and
    null values. *)
Inductive literal :=
| Undef
| Null
| BoolLit :> bool → literal
| NumLit :> nat → literal
| StrLit :> string → literal.

(* Encode security markers using nats.
    *)
Definition marker := nat.

(** An expression is a labelled term. *)
Inductive expr :=
| Expr : term → label → expr

with term :=
| TLit :> literal → term
| TRec : finmap string expr → term
| TVar : string → term
| TAbs : string → expr → term
| TApp : expr → expr → term
| TBox : expr → term
| TUnbox : expr → term
| TRun : expr → term
| TIf : expr → expr → expr → term
| TRecRead : expr → expr → term
| TRecWrite : expr → expr → expr → term
| TRecDel : expr → expr → term
| TClos : term → finmap string (value 0) →
    term
| TRunIn : expr → finmap string (value 0) →
    term
| TMark : marker → expr → term
```

```
(** Values are indexed by their level.
    *)
with value : nat → Type :=
| VLit : forall n, label → literal → value
    n
| VRec : forall n, label → finmap string (
    value n) → value n
| VVar : forall n, label → string → value (S
    n)
| VAbs : forall n, label → string → value (S
    n) → value (S n)
| VApp : forall n, label → value (S n) →
    value (S n) → value (S n)
| VBox : forall n, label → value (S n) →
    value n
| VUnbox : forall n, label → value (S n) →
    value (S (S n))
| VRun : forall n, label → value (S n) →
    value (S n)
| VIf : forall n, label → value (S n) → value
    (S n) → value (S n) → value (S n)
| VClos : label → string → expr → finmap
    string (value 0) → value 0
| VRecRead : forall n, label → value (S n) →
    value (S n) → value (S n)
| VRecWrite : forall n, label → value (S n)
    → value (S n) → value (S n) → value (S n)
| VRecDel : forall n, label → value (S n) →
    value (S n) → value (S n)
| VMark : forall n, label → marker → value n
    → value n
.

(* begin hide *)
Section ValueAltInd.
  Variable P : forall n, value n → Prop.

  Variable H1 : forall n ℓ k, P (VLit n ℓ k).
  Variable H2 : forall n ℓ (xvs:finmap string
      (value n)), mforall (fun x v ⇒ P v) xvs
      → P (VRec ℓ xvs).
  Variable H3 : forall n ℓ x, P (VVar n ℓ x).
  Variable H4 : forall n ℓ x (v:value (S n)),
      P v → P (VAbs ℓ x v).
  Variable H5 : forall n ℓ (v1 v2:value (S n)),
      P v1 → P v2 → P (VApp ℓ v1 v2).
  Variable H6 : forall n ℓ (v:value (S n)), P
      v → P (VBox ℓ v).
  Variable H7 : forall n ℓ (v:value (S n)), P
      v → P (VUnbox ℓ v).
  Variable H8 : forall n ℓ (v:value (S n)), P
      v → P (VRun ℓ v).
  Variable H9 : forall n ℓ (v1 v2 v3:value (S
      n)), P v1 → P v2 → P v3 → P (VIf ℓ v1 v2
      v3).
  Variable H10 : forall ℓ x e r, P (VClos ℓ x
      e r).
  Variable H11 : forall n ℓ (v1 v2:value (S n))
      , P v1 → P v2 → P (VRecRead ℓ v1 v2).
  Variable H12 : forall n ℓ (v1 v2 v3:value (S
      n)), P v1 → P v2 → P v3 → P (VRecWrite
      ℓ v1 v2 v3).
  Variable H13 : forall n ℓ (v1 v2:value (S n))
      , P v1 → P v2 → P (VRecDel ℓ v1 v2).
  Variable H14 : forall n ℓ m (v:value n), P v
      → P (VMark ℓ m v).

  Lemma val_altind : forall n (v:value n), P v.
  Proof.
    fix 2.
    destruct v.
      apply H1.
      apply H2. induction f; constructor;
          auto.
      apply H3.
      apply H4; auto.
      apply H5; auto.
      apply H6; auto.
      apply H7; auto.
      apply H8; auto.
      apply H9; auto.
      apply H10; auto.
      apply H11; auto.
      apply H12; auto.
      apply H13; auto.
      apply H14; auto.
  Qed.
End ValueAltInd.
(* end hide *)

Notation "t ^ ℓ" := (Expr t ℓ).

Definition get_term (e:expr) := let (t,_) :=
    e in t.
Definition get_label (e:expr) := let (_,ℓ) :=
    e in ℓ.

Definition relabel e ℓ:= (get_term e)^ℓ.

Definition env := finmap string (value 0).

(** Function [embed] re-interprets a
    value as an expression. *)
Fixpoint embed n (v:value n) : expr :=
match v with
| VLit _ ℓ k ⇒ (TLit k)^ℓ
| VRec _ ℓ bs ⇒ (TRec (mmap (·embed _) bs))^ℓ
```

```
| VVar _ ℓ x ⇒ (TVar x)^ℓ
| VAbs _ ℓ x v ⇒ (TAbs x (embed v))^ℓ
| VApp _ ℓ v1 v2 ⇒ (TApp (embed v1) (embed v2))
    ^ℓ
| VBox _ ℓ v ⇒ (TBox (embed v))^ℓ
| VUnbox _ ℓ v ⇒ (TUnbox (embed v))^ℓ
| VRun _ ℓ v ⇒ (TRun (embed v))^ℓ
| VIf _ ℓ v1 v2 v3 ⇒ (TIf (embed v1) (embed v2
    ) (embed v3))^ℓ
| VClos ℓ x e r ⇒ (TClos (TAbs x e) r)^ℓ
| VRecRead _ ℓ v1 v2 ⇒ (TRecRead (embed v1) (
    embed v2))^ℓ
| VRecWrite _ ℓ v1 v2 v3 ⇒ (TRecWrite (embed
    v1) (embed v2) (embed v3))^ℓ
| VRecDel _ ℓ v1 v2 ⇒ (TRecDel (embed v1) (
    embed v2))^ℓ
| VMark _ ℓ m v ⇒ (TMark m (embed v))^ℓ
end.

(** The grammar of evaluation contexts.
    Evaluation contexts are doubly
    indexed to indicate their staging
        behaviour. A context of type
    [ctxt m n] identifies a hole at
        level [n] inside an expression of
    level [m]. *)
Inductive ctxt : nat → nat → Type :=
| CId : forall n, ctxt n n
| CRec : forall m n, label → finmap string (
    value m) → string → ctxt m n → finmap
    string expr → ctxt m n
| CAbs : forall m n, label → string → ctxt (
    S m) n → ctxt (S m) n
| CAppL : forall m n, label → ctxt m n →
    expr → ctxt m n
| CAppR : forall m n, label → value m → ctxt
     m n → ctxt m n
| CBox : forall m n, label → ctxt (S m) n →
    ctxt m n
| CUnbox : forall m n, label → ctxt m n →
    ctxt (S m) n
| CRun : forall m n, label → ctxt m n → ctxt
    m n
| CIf1 : forall m n, label → ctxt m n → expr
    → expr → ctxt m n
| CIf2 : forall m n, label → value (S m) →
    ctxt (S m) n → expr → ctxt (S m) n
| CIf3 : forall m n, label → value (S m) →
    value (S m) → ctxt (S m) n → ctxt (S m) n
| CRecRead1 : forall m n, label → ctxt m n →
    expr → ctxt m n
| CRecRead2 : forall m n, label → value m →
    ctxt m n → ctxt m n
| CRecWrite1 : forall m n, label → ctxt m n
    → expr → expr → ctxt m n
| CRecWrite2 : forall m n, label → value m →
    ctxt m n → expr → ctxt m n
| CRecWrite3 : forall m n, label → value m →
    value m → ctxt m n → ctxt m n
| CRecDel1 : forall m n, label → ctxt m n →
    expr → ctxt m n
| CRecDel2 : forall m n, label → value m →
    ctxt m n → ctxt m n
| CRunIn : forall m n, label → ctxt m n →
    finmap string (value 0) → ctxt m n
| CMark : forall m n, label → marker → ctxt
    m n → ctxt m n
.

(** A standard plugging function for
    filling in a hole identified by a
    context. *)
Fixpoint plug m n (C:ctxt m n) (e:expr) : expr
    :=
match C with
| CId _ ⇒ e
| CRec _ _ ℓ xvs x C' xes ⇒ (TRec (mapp (mmap
    (·embed _) xvs) (ConsMap x (plug C' e) xes)
    ))^ℓ
| CAbs _ _ ℓ x C' ⇒ (TAbs x (plug C' e))^ℓ
| CAppL _ _ ℓ C' e' ⇒ (TApp (plug C' e) e')^ℓ
| CAppR _ _ ℓ v C' ⇒ (TApp (embed v) (plug C'
    e))^ℓ
| CBox _ _ ℓ C' ⇒ (TBox (plug C' e))^ℓ
| CUnbox _ _ ℓ C' ⇒ (TUnbox (plug C' e))^ℓ
| CRun _ _ ℓ C' ⇒ (TRun (plug C' e))^ℓ
| CIf1 _ _ ℓ C' e1 e2 ⇒ (TIf (plug C' e) e1 e2)
    ^ℓ
| CIf2 _ _ ℓ v C' e' ⇒ (TIf (embed v) (plug C'
    e) e')^ℓ
| CIf3 _ _ ℓ v1 v2 C' ⇒ (TIf (embed v1) (embed
     v2) (plug C' e))^ℓ
| CRecRead1 _ _ ℓ C' e' ⇒ (TRecRead (plug C' e
    ) e')^ℓ
| CRecRead2 _ _ ℓ v C' ⇒ (TRecRead (embed v) (
    plug C' e))^ℓ
| CRecWrite1 _ _ ℓ C' e1 e2 ⇒ (TRecWrite (
    plug C' e) e1 e2)^ℓ
| CRecWrite2 _ _ ℓ v C' e' ⇒ (TRecWrite (
    embed v) (plug C' e) e')^ℓ
| CRecWrite3 _ _ ℓ v1 v2 C' ⇒ (TRecWrite (
    embed v1) (embed v2) (plug C' e))^ℓ
| CRecDel1 _ _ ℓ C' e' ⇒ (TRecDel (plug C' e)
    e')^ℓ
| CRecDel2 _ _ ℓ v C' ⇒ (TRecDel (embed v) (
    plug C' e))^ℓ
| CRunIn _ _ ℓ C' r ⇒ (TRunIn (plug C' e) r)^ℓ
| CMark _ _ ℓ m C' ⇒ (TMark m (plug C' e))^ℓ
```

```
end.

(** This is the reduction relation *e &
    rarr;<sup>n</sup></sup> e'*e -->^n e. *)
Inductive red : expr → nat → expr → Prop :=
| PushLit : forall n k r ℓ,
            red ((TClos (TLit k) r)^ℓ) n ((
                TLit k)^ℓ)
| PushRec : forall n xes r ℓ,
            red ((TClos (TRec xes) r)^ℓ) n
                ((TRec
                  (mmap (fun e ⇒ (TClos (
                      get_term e) r)^(
                      get_label e)) xes))^ℓ
                  )
| PushVar : forall n x r ℓ,
            red ((TClos (TVar x) r)^ℓ) (S n)
                ((TVar x)^ℓ)
| PushApp : forall n t1 t2 ℓ1 ℓ2 r ℓ,
            red ((TClos (TApp (t1^ℓ1) (t2^ℓ2))
                 r)^ℓ)
                n
                ((TApp ((TClos t1 r)^ℓ1) ((
                    TClos t2 r)^ℓ2))^ℓ)
| PushAbs : forall x t r ℓ1 ℓ2 n,
            red ((TClos (TAbs x (t^ℓ1)) r)^ℓ2)
                (S n)
                ((TAbs x ((TClos t r)^ℓ1))^ℓ2
                  )
| PushBox : forall t ℓ1 ℓ2 r n,
            red ((TClos (TBox (t^ℓ1)) r)^ℓ2)
                n
                ((TBox ((TClos t r)^ℓ1))^ℓ2)
| PushUnbox : forall t ℓ1 ℓ2 r n,
            red ((TClos (TUnbox (t^ℓ1)) r)^ℓ2)
                n
                ((TUnbox ((TClos t r)^ℓ1))^ℓ2
                  )
| PushRunZ : forall t ℓ1 ℓ2 r,
            red ((TClos (TRun (t^ℓ1)) r)^ℓ2)
                0
                ((TRunIn ((TClos t r)^ℓ1) r)^
                    ℓ2)
| PushRunS : forall t ℓ1 ℓ2 r n,
            red ((TClos (TRun (t^ℓ1)) r)^ℓ2)
                (S n)
                ((TRun ((TClos t r)^ℓ1))^ℓ2)
| PushIf : forall t1 t2 t3 ℓ1 ℓ2 ℓ3 ℓ r n,
            red ((TClos (TIf (t1^ℓ1) (t2^ℓ2) (
                t3^ℓ3)) r)^ℓ)
                n
                ((TIf ((TClos t1 r)^ℓ1) ((
                    TClos t2 r)^ℓ2) ((TClos
                        t3 r)^ℓ3))^ℓ)
| PushRecRead : forall t1 t2 ℓ1 ℓ2 ℓ3 r n,
            red ((TClos (TRecRead (t1^ℓ1) (t2
                ^ℓ2)) r)^ℓ3)
                n
                ((TRecRead ((TClos t1 r)^ℓ1)
                    ((TClos t2 r)^ℓ2))^ℓ3)
| PushRecWrite : forall t1 t2 t3 ℓ1 ℓ2 ℓ3 ℓ4 r
    n,
            red ((TClos (TRecWrite (t1^ℓ1) (t
                2^ℓ2) (t3^ℓ3)) r)^ℓ4)
                n
                ((TRecWrite ((TClos t1 r)^ℓ1
                    ) ((TClos t2 r)^ℓ2) ((
                    TClos t3 r)^ℓ3))^ℓ4)
| PushRecDel : forall t1 t2 ℓ1 ℓ2 ℓ3 r n,
            red ((TClos (TRecDel (t1^ℓ1) (t2^
                ℓ2)) r)^ℓ3)
                n
                ((TRecDel ((TClos t1 r)^ℓ1)
                    ((TClos t2 r)^ℓ2))^ℓ3)
| Lookup : forall x r ℓ v, lookup r x = Some v
    →
            red ((TClos (TVar x) r)^ℓ) 0 (
                relabel (embed v) ℓ)
| Beta : forall x t r ℓ1 ℓ2 ℓ3 v,
            red ((TApp ((TClos (TAbs x (t^ℓ1))
                r)^ℓ2) (embed v))^ℓ3)
                0
                ((TClos t (ext_map r x v))^ℓ1
                  )
| UnboxBox : forall (v:value 1) ℓ1 ℓ2,
            red ((TUnbox ((TBox (embed v))^ℓ1)
                )^ℓ2)
                1
                (relabel (embed v) ℓ2)
| RunBox : forall (v:value 1) ℓ1 ℓ2 r,
            red ((TRunIn ((TBox (embed v))^ℓ1)
                r)^ℓ2)
                0
                (Expr (TClos (get_term (
                    embed v)) r) ℓ2)
| IfTrue : forall t2 t3 ℓ1 ℓ2 ℓ3 ℓ,
            red ((TIf (true^ℓ1) (t2^ℓ2) (t3^ℓ3
                ))^ℓ)
                0
                (t2^ℓ2)
| IfFalse : forall t2 t3 ℓ1 ℓ2 ℓ3 ℓ,
            red ((TIf (false^ℓ1) (t2^ℓ2) (t3^
                ℓ3))^ℓ)
                0
                (t3^ℓ3)
| RecReadHere : forall x (xvs:finmap string (
    value 0)) ℓ1ℓ2 ℓ v, lookup xvs x = Some v
```

```
    →
          red ((TRecRead (embed (VRec ℓ1
              xvs)) ((TLit x)^ℓ2))^ℓ)
               0
               (relabel (embed v) ℓ)
| RecReadProto : forall x (xvs xvs':finmap
    string (value 0)) ℓ0ℓ1 ℓ2 ℓ, lookup xvs x
    = None →
          lookup xvs ("__proto__")%
              string = Some (VRec ℓ0xvs')
              →
          red ((TRecRead (embed (VRec ℓ1
              xvs)) ((TLit x)^ℓ2))^ℓ)
               0
               ((TRecRead (embed (VRec ℓ0
                   xvs')) ((TLit x)^ℓ2))^ℓ)
| RecReadNone : forall x (xvs:finmap string (
    value 0)) ℓ0ℓ1 ℓ2 ℓ, lookup xvs x = None
    →
          lookup xvs ("__proto__")%
              string = Some (VLit _ ℓ0Null
              ) →
          red ((TRecRead (embed (VRec ℓ1
              xvs)) ((TLit x)^ℓ2))^ℓ)
               0
               (Undef^ℓ)
| RecWrite : forall (x:string) (xvs:finmap
    string (value 0)) (v:value 0) ℓ1 ℓ2 ℓ,
          red ((TRecWrite (embed (VRec ℓ1
              xvs)) ((TLit x)^ℓ2) (embed v)
              )^ℓ)
               0
               (embed (VRec ℓ (ext_map xvs
                   x v)))
| RecDel : forall (x:string) (xvs:finmap
    string (value 0)) ℓ1 ℓ2 ℓ,
          red ((TRecDel (embed (VRec ℓ1 xvs
              )) ((TLit x)^ℓ2))^ℓ)
               0
               (embed (VRec ℓ (del_map xvs
                   x)))
| PushMark : forall m t1 ℓ1 ℓ r n,
          red ((TClos (TMark m (t1^ℓ1)) r)^ℓ
              )
               n
               ((TMark m ((TClos t1 r)^ℓ1))^
                   ℓ)
| LiftIf : forall m t1 t2 t3 ℓ0 ℓ1 ℓ2 ℓ3 ℓ,
          red ((TIf ((TMark m (t1^ℓ1))^ℓ0) (
              t2^ℓ2) (t3^ℓ3))^ℓ)
              0
              ((TMark m ((TIf (t1^ℓ1) (t2^ℓ2
                  ) (t3^ℓ3))^ℓ))^ℓ)
| LiftApp : forall m t r ℓ1 ℓ2 ℓ3 (v : value
    0),
          red ((TApp ((TClos (TMark m (t^ℓ1)
              ) r)^ℓ2) (embed v))^ℓ3)
               0
               ((TMark m ((TApp ((TClos t r)
                   ^ℓ1) (embed v))^ℓ3))^ℓ3)
| LiftUnbox : forall m (v:value 0) ℓ1 ℓ2,
          red ((TUnbox ((TMark m (embed v))
              ^ℓ1))^ℓ2)
               1
               ((TMark m ((TUnbox (embed v)
                   )^ℓ2))^ℓ2)
| LiftRunIn : forall m (v:value 0) ℓ1 ℓ2 r,
          red ((TRunIn ((TMark m (embed v))
              ^ℓ1) r)^ℓ2)
               0
               ((TMark m ((TRunIn (embed v)
                   r)^ℓ2)^ℓ2))
| LiftReadSel : forall m (v1:value 0) (v2:
    value 0) ℓ1 ℓ2,
          red ((TRecRead (embed v1) ((TMark
              m (embed v2))^ℓ1))^ℓ2)
               0
               ((TMark m ((TRecRead (embed
                   v1) (embed v2))^ℓ2)^ℓ2))
| LiftReadRec : forall m (v1:value 0) (v2:
    value 0) ℓ1 ℓ2,
          red ((TRecRead ((TMark m (embed v
              1))^ℓ1) (embed v2))^ℓ2)
               0
               ((TMark m ((TRecRead (embed
                   v1) (embed v2))^ℓ2)^ℓ2))
| LiftWriteSel : forall m (v1:value 0) (v2:
    value 0) (v3:value 0) ℓ1 ℓ2,
          red ((TRecWrite (embed v1) ((
              TMark m (embed v2))^ℓ1) (
              embed v3))^ℓ2)
               0
               ((TMark m ((TRecWrite (
                   embed v1) (embed v2) (
                   embed v3))^ℓ2)^ℓ2))
| LiftWriteRec : forall m (v1:value 0) (v2:
    value 0) (v3:value 0) ℓ1 ℓ2,
          red ((TRecWrite ((TMark m (embed
              v1))^ℓ1) (embed v2) (embed v3)
              )^ℓ2)
               0
               ((TMark m ((TRecWrite (
                   embed v1) (embed v2) (
                   embed v3))^ℓ2)^ℓ2))
| LiftDelSel : forall m (v1:value 0) (v2:value
    0) ℓ1 ℓ2,
          red ((TRecDel (embed v1) ((TMark
              m (embed v2))^ℓ1))^ℓ2)
```

```
                0
                ((TMark m ((TRecDel (embed v
                    1) (embed v2))^ℓ2)^ℓ2))
| LiftDelRec : forall m (v1:value 0) (v2:value
    0) ℓ1 ℓ2,
        red ((TRecDel ((TMark m (embed v1
            ))^ℓ1) (embed v2))^ℓ2)
            0
            ((TMark m ((TRecDel (embed v
                1) (embed v2))^ℓ2)^ℓ2))
.

(** The evaluation relation; currently
    just the context closure of the
    reduction relation. *)
Inductive eval : expr → nat → expr → Prop
    :=
| Ctxt : forall m n (C:ctxt m n) e e',
        red e n e' → eval (plug C e) m (
            plug C e')
.

Lemma plug_label : forall m n (C:ctxt m n) e e
    ', get_label e = get_label e' →
    get_label (plug C e) = get_label (plug C
    e').
Proof.
  induction C; auto.
Qed.

Lemma eval_label : forall e n e', eval e (S n)
    e' → get_label e = get_label e'.
Proof.
  intros.
  dependent destruction H.
  dependent induction C; simpl; eauto.
  dependent destruction H; simpl; auto.
Qed.

(* begin hide *)
(* a view datatype useful for inverting
    the eval relation *)
Inductive eval_view : expr → nat → expr →
    Prop :=
| EVRec : forall xes ℓ xes' n, eval_view (
    Expr (TRec xes) ℓ) n (Expr (TRec xes') ℓ)
| EVAbs : forall x t t' ℓ1 ℓ2 n, eval_view (
    Expr (TAbs x (Expr t ℓ1)) ℓ2) (S n) (Expr (
    TAbs x (Expr t' ℓ1)) ℓ2)
| EVAppZ : forall e e' t ℓ ℓ', eval_view (
    Expr (TApp e e') ℓ) 0 (Expr t ℓ')
| EVAppS : forall e1 e1' e2 e2' ℓ n,
    eval_view (Expr (TApp e1 e2) ℓ) (S n) (
    Expr (TApp e1' e2') ℓ)
| EVBox : forall t t' ℓ1 ℓ2 n, eval_view (
    Expr (TBox (Expr t ℓ1)) ℓ2) n (Expr (TBox (
    Expr t' ℓ1)) ℓ2)
| EVUnbox : forall e t ℓ n, eval_view (Expr (
    TUnbox e) ℓ) n (Expr t ℓ)
| EVRun : forall e ℓ e' n, eval_view (Expr (
    TRun e) ℓ) n (Expr (TRun e') ℓ)
| EVRunInZ : forall e ℓ e' r, eval_view (Expr
    (TRunIn e r) ℓ) 0 e'
| EVRunInS : forall e ℓ e' r n, eval_view (
    Expr (TRunIn e r) ℓ) (S n) (Expr (TRunIn e
    ' r) ℓ)
| EVIfZ : forall e1 e2 e3 e4 ℓ, eval_view (
    Expr (TIf e1 e2 e3) ℓ) 0 e4
| EVIfS : forall e1 e1' e2 e2' e3 e3' ℓ n,
    eval_view (Expr (TIf e1 e2 e3) ℓ) (S n) (
    Expr (TIf e1' e2' e3') ℓ)
| EVClosLit : forall k r ℓ n, eval_view (Expr
    (TClos (TLit k) r) ℓ) n (Expr (TLit k) ℓ)
| EVClosRec : forall xes xes' r ℓ n,
    eval_view (Expr (TClos (TRec xes) r) ℓ) n
    (Expr (TRec xes') ℓ)
| EVClosVarZ : forall x r ℓ e, eval_view (
    Expr (TClos (TVar x) r) ℓ) 0 e
| EVClosVarS : forall x r ℓ n, eval_view (
    Expr (TClos (TVar x) r) ℓ) (S n) (Expr (
    TVar x) ℓ)
| EVClosAbs : forall x t r ℓ1 ℓ2 n, eval_view
    (Expr (TClos (TAbs x (Expr t ℓ1)) r) ℓ2) (S
     n) (Expr (TAbs x (Expr (TClos t r) ℓ1)) ℓ
    2)
| EVClosApp : forall e1 e1' e2 e2' r ℓ n,
    eval_view (Expr (TClos (TApp e1 e2) r) ℓ)
    n (Expr (TApp e1' e2') ℓ)
| EVClosBox : forall t r ℓ1 ℓ2 n, eval_view (
    Expr (TClos (TBox (Expr t ℓ1)) r) ℓ2) n (
    Expr (TBox (Expr (TClos t r) ℓ1)) ℓ2)
| EVClosUnbox : forall e e' r ℓ n, eval_view (
    Expr (TClos (TUnbox e) r) ℓ) n (Expr (
    TUnbox e') ℓ)
| EVClosRunZ : forall e e' r ℓ, eval_view (
    Expr (TClos (TRun e) r) ℓ) 0 (Expr (TRunIn
     e' r) ℓ)
| EVClosRunS : forall e e' r ℓ n, eval_view (
    Expr (TClos (TRun e) r) ℓ) (S n) (Expr (
    TRun e') ℓ)
| EVClosIf : forall e1 e2 e3 e1' e2' e3' r ℓ n,
     eval_view (Expr (TClos (TIf e1 e2 e3) r)
    ℓ) n (Expr (TIf e1' e2' e3') ℓ)
| EVClosRecRead : forall e1 e2 r ℓ n e1' e2',
    eval_view (Expr (TClos (TRecRead e1 e2) r
    ) ℓ) n (Expr (TRecRead e1' e2') ℓ)
| EVClosRecWrite : forall e1 e2 e3 r ℓ n e1' e
    2' e3', eval_view (Expr (TClos (
```

```
    TRecWrite e1 e2 e3) r) ℓ) n (Expr (
    TRecWrite e1' e2' e3') ℓ)
| EVClosRecDel : forall e1 e2 r ℓ n e1' e2',
    eval_view (Expr (TClos (TRecDel e1 e2) r)
    ℓ) n (Expr (TRecDel e1' e2') ℓ)
| EVRecReadZ : forall t1 t2 ℓ t' ℓ',
    eval_view (Expr (TRecRead t1 t2) ℓ) 0 (
    Expr t' ℓ')
| EVRecWriteZ : forall t1 t2 t3 ℓ t' ℓ',
    eval_view (Expr (TRecWrite t1 t2 t3) ℓ) 0
    (Expr t' ℓ')
| EVRecReadS : forall t1 t2 t1' t2' ℓ n,
    eval_view (Expr (TRecRead t1 t2) ℓ) (S n)
    (Expr (TRecRead t1' t2') ℓ)
| EVRecWriteS : forall t1 t2 t3 t1' t2' t3' ℓ
    n, eval_view (Expr (TRecWrite t1 t2 t3) ℓ)
     (S n) (Expr (TRecWrite t1' t2' t3') ℓ)
| EVRecDelZ : forall t1 t2 ℓ t' ℓ', eval_view
    (Expr (TRecDel t1 t2) ℓ) 0 (Expr t' ℓ')
| EVRecDelS : forall t1 t2 t1' t2' ℓ n,
    eval_view (Expr (TRecDel t1 t2) ℓ) (S n) (
    Expr (TRecDel t1' t2') ℓ)
| EVMark : forall m e e' ℓ n, eval_view (Expr
    (TMark m e) ℓ) n (Expr (TMark m e') ℓ)
| EVClosMark : forall m e e' r ℓ n, eval_view
    (Expr (TClos (TMark m e) r) ℓ) n (Expr (
    TMark m e') ℓ)
.
Hint Constructors eval_view.

Lemma eval_view_labels : forall t ℓn t' ℓ',
    eval_view (t^ℓ) (S n) (t'^ℓ') → ℓ'=ℓ.
Proof.
  intros.
  dependent induction H; auto.
Qed.

Lemma view_eval : forall e n e', eval e n e'
    → eval_view e n e'.
Proof.
  destruct 1.
  induction C; simpl; try (solve [destruct m;
      auto | auto]).
    destruct H; auto.
      destruct (embed v); unfold relabel;
          simpl; auto.

      unfold relabel; destruct (embed v);
          auto.

    simpl; auto.

    simpl; auto.

simpl.

    destruct (plug C e).
    destruct (plug C e').
    rewrite (eval_view_labels (IHC H)); auto.

    destruct (plug C e).
    destruct (plug C e').
    rewrite (eval_view_labels (IHC H)); auto.
Qed.
(* end hide *)

Lemma red_eval : forall e n e', red e n e' →
    eval e n e'.
Proof.
  intros; apply (Ctxt (·CId n) H).
Qed.
Hint Resolve red_eval.
```

A.3 Simple 0CFA and Information Flow Analysis for SLamJS — simple_cfa.v

```
(** A simple binding analysis for Staged
    LambdaJS. *)
Set Implicit Arguments.

Require Import Eqdep Bool String List.
Require Import Program.Equality.
Require Import lib.

Require Import lambdajs.

Require Import Arith.

Section Analysis.
  Inductive absval := NULL | UNDEF | BOOL |
      NUM | STR
    | LAM : string → expr → absval
    | BOX : expr → absval
    | REC : label → absval
    | MARK : marker → absval.

  Inductive absvar :=
  | AVar :> string → absvar
```

```
| AProp : label → string → absvar.

Variable Γ : label → 𝒫 absval.
Variable ρ : absvar → 𝒫 absval.

Definition is_mark (a : absval) : bool :=
  match a with
  | MARK _ ⇒ true
  | _ ⇒ false
  end.

Definition filter_mark := filter is_mark.

Lemma filter_subset : forall A (f : A →
    bool) (l: list A),
  filter f l ⊆ l.
Proof.
  intros. induction l.
    auto with datatypes.
    simpl. destruct (f a); auto with
        datatypes.
Qed.
Hint Resolve filter_subset.

Lemma filter_mark_subset : forall ℓ,
  filter_mark(Γ  ℓ) ⊆ Γ ℓ.
Proof.
  intros. unfold filter_mark. auto.
Qed.
Hint Resolve filter_mark_subset.

Lemma direct__indirect : forall ℓ1ℓ2,
  Γ ℓ1  ⊆ Γ ℓ2  → filter_mark (Γ  ℓ1) ⊆ Γ ℓ2.
Proof.
  intros. eauto with ∗.
Qed.
Hint Resolve direct__indirect.

Lemma subset_nil__nil : forall A (l : list A
    ),
  l ⊆ nil → l = nil.
Proof.
  intros. destruct l.
    auto.
    unfold incl in H.
    specialize H with (a0 := a). simpl in H.
        intuition.
Qed.

Lemma nil_subset_all : forall A (l : list A),
  nil ⊆ l.
Proof.
  intros. unfold incl. intros. auto with ∗.
Qed.

Lemma filter_mono : forall A (f : A → bool)
    (l1 : list A) (l2 : list A),
  l1 ⊆ l2 → (filter f l1) ⊆ (filter f l2).
Proof.
  induction l1.
    intros. simpl. apply nil_subset_all.
    intros.
    assert (In a l2). auto with datatypes.
    assert (l1 ⊆ (a :: l1)). auto with
        datatypes.
    assert (l1 ⊆ l2). eauto with datatypes.
    apply IHl1 in H2.
    simpl. remember (f a). destruct b; auto.
    assert (In a (filter f l2)). apply
        filter_In. auto.
    auto with datatypes.
Qed.

Lemma direct__indirect_mono : forall ℓ1ℓ2,
  Γ ℓ1  ⊆ Γ ℓ2  → filter_mark (Γ  ℓ1) ⊆
      filter_mark (Γ  ℓ2).
Proof.
  intros. apply filter_mono. auto.
Qed.
Hint Resolve direct__indirect_mono.

Lemma mark_in_filter_mark : forall m ℓ,
  MARK m ∈ Γ ℓ  → MARK m ∈ filter_mark (Γ  ℓ)
      .
Proof.
  intros. apply filter_In. auto.
Qed.
Hint Resolve mark_in_filter_mark.

Lemma mark_propogates_indirectly :
    forall m ℓ1ℓ2,
  MARK m ∈ Γ ℓ1  → filter_mark (Γ  ℓ1) ⊆ Γ ℓ2
      → MARK m ∈ Γ ℓ2.
Proof.
  intros. auto.
Qed.
Hint Resolve mark_propogates_indirectly.

Inductive protos : label → label → Prop
    :=
| ProtoRefl : forall ℓ, protos ℓ ℓ
| ProtoStep : forall ℓℓ' ℓ'',  REC ℓ' ∈ ρ (
    AProp ℓ ("__proto__")%string) →
    protos ℓ' ℓ'' → protos ℓℓ''.

Inductive accept : expr → Prop :=
| AcceptLit : forall v k ℓ,  v ∈ Γ ℓ  →
    approx v (TLit k) →
```

```
                accept (Expr (TLit k) ℓ)
| AcceptVar : forall (x:string) ℓ, ρ x ⊆ Γ ℓ
    →
                accept (Expr (TVar x) ℓ)
| AcceptRec : forall xes ℓ ℓ',
              mforall (fun x e ⇒
                         accept e ∧ Γ (
                             get_label e) ⊆
                             ρ (AProp ℓ' x))
                             xes →
              REC ℓ' ∈ Γ ℓ →
                accept (Expr (TRec xes) ℓ)
| AcceptAbs : forall v x e ℓ, accept e → v
    ∈ Γ ℓ → approx v (TAbs x e) →
                accept (Expr (TAbs x e) ℓ)
| AcceptApp : forall e1 e2 ℓ, accept e1 →
    accept e2 →
              (forall x e, LAM x e ∈ Γ (
                  get_label e1) →
                Γ (get_label e2) ⊆ ρ x ∧
                Γ (get_label e) ⊆ Γ ℓ) →
            filter_mark (Γ (get_label e1))
              ⊆ Γ ℓ →
                accept (Expr (TApp e1 e2) ℓ)
| AcceptBox : forall v e ℓ, accept e → v ∈
    Γ ℓ → approx v (TBox e) →
                accept (Expr (TBox e) ℓ)
| AcceptUnbox : forall e ℓ, accept e →
              (forall e', BOX e' ∈ Γ (
                  get_label e) →
                Γ (get_label e') ⊆ Γ ℓ) →
            filter_mark (Γ (get_label e)) ⊆
              Γ ℓ →
                accept (Expr (TUnbox e) ℓ)
| AcceptRun : forall e ℓ, accept e →
              (forall e', BOX e' ∈ Γ (
                  get_label e) →
                Γ (get_label e') ⊆ Γ ℓ) →
            filter_mark (Γ (get_label e)) ⊆
              Γ ℓ →
                accept (Expr (TRun e) ℓ)
| AcceptIf : forall e1 e2 e3 ℓ, accept e1
    → accept e2 → accept e3 →
              Γ (get_label e2) ⊆ Γ ℓ →
              Γ (get_label e3) ⊆ Γ ℓ →
              filter_mark (Γ (get_label e1
                  )) ⊆ Γ ℓ→
                accept (Expr (TIf e1 e2 e3) ℓ
                    )
| AcceptRecRead : forall e1 e2 ℓ,
              accept e1 → accept e2 →
              (forall ℓ' ℓ'' x, REC ℓ' ∈ Γ (
                  get_label e1) →
                protos ℓ' ℓ'' → ρ (AProp ℓ''
                    x) ⊆ Γ ℓ) →
              UNDEF ∈ Γ ℓ →
              filter_mark (Γ (get_label e1
                  )) ⊆ Γ ℓ→
              filter_mark (Γ (get_label e2
                  )) ⊆ Γ ℓ→
                accept (Expr (TRecRead e1 e2
                    ) ℓ)
| AcceptRecWrite : forall e1 e2 e3 ℓ,
              accept e1 → accept e2 →
                  accept e3 →
              (forall ℓ' x, REC ℓ' ∈ Γ (
                  get_label e1) →
                Γ (get_label e3) ⊆ ρ (AProp
                    ℓ' x)) →
              Γ (get_label e1) ⊆ Γ ℓ →
              filter_mark (Γ (get_label e2
                  )) ⊆ Γ ℓ→
                accept (Expr (TRecWrite e1
                    e2 e3) ℓ)
| AcceptRecDel : forall e1 e2 ℓ,
              accept e1 → accept e2 → Γ (
                  get_label e1) ⊆ Γ ℓ→
              filter_mark (Γ (get_label e2
                  )) ⊆ Γ ℓ→
                accept (Expr (TRecDel e1 e2)
                    ℓ)
| AcceptClos : forall t r ℓ, accept (Expr t
    ℓ) → accept_env r →
                accept (Expr (TClos t r) ℓ)
| AcceptRunIn : forall e r ℓ, accept e →
    accept_env r →
              (forall e', BOX e' ∈ Γ (
                  get_label e) →
                Γ (get_label e') ⊆ Γ ℓ) →
            filter_mark (Γ (get_label e)) ⊆
              Γ ℓ →
                accept (Expr (TRunIn e r) ℓ)
| AcceptMark : forall e m ℓ, accept e →
                Γ (get_label e) ⊆ Γ ℓ →
                (MARK m) ∈ Γ ℓ →
                  accept (Expr (TMark m e) ℓ)

with accept_env : env → Prop :=
| AcceptNil : accept_env NilMap
| AcceptCons : forall r (x:string) v, accept
    (embed v) → accept_env r → Γ (
    get_label (embed v)) ⊆ ρ x →
    accept_env (ConsMap x v r)
with approx : absval → term → Prop :=
| ApproxNull : approx NULL Null
| ApproxUndef : approx UNDEF Undef
| ApproxBool : forall b, approx BOOL (
    BoolLit b)
```

```
| ApproxNum : forall n, approx NUM (NumLit
    n)
| ApproxStr : forall s, approx STR (StrLit
    s)
| ApproxLam : forall x e, approx (LAM x e) (
    TAbs x e)
| ApproxBox : forall e, approx (BOX e) (
    TBox e)
| ApproxRec : forall ℓ xes, mforall (fun x
    e ⇒ exists v, v ∈ ρ (AProp ℓ x) ∧
    approx v (get_term e)) xes → approx (
    REC ℓ) (TRec xes)
| ApproxRed : forall v t ℓ n t', approx v t
    → eval (Expr t ℓ) n (Expr t' ℓ) →
    approx v t'
| ApproxClos : forall v t r, approx v t →
    accept_env r → approx v (TClos t r).
Hint Constructors accept accept_env
    approx.

Inductive approx_view : absval → term →
    Prop :=
| AVNull : approx_view NULL Null
| AVUndef : approx_view UNDEF Undef
| AVBool : forall b, approx_view BOOL (
    BoolLit b)
| AVNum : forall n, approx_view NUM (
    NumLit n)
| AVStr : forall s, approx_view STR (
    StrLit s)
| AVLam : forall x t t' ℓ, approx_view (
    LAM x (Expr t ℓ)) (TAbs x (Expr t' ℓ))
| AVBox : forall t t' ℓ, approx_view (BOX (
    Expr t ℓ)) (TBox (Expr t' ℓ))
| AVRec : forall ℓ xes, approx_view (REC ℓ)
    (TRec xes)
| AVClos : forall v t r, approx_view v t →
    approx_view v (TClos t r).
Hint Constructors approx_view.

Lemma view_approx : forall v t, approx v t
    → approx_view v t.
Proof.
  induction 1; auto.
    solve [destruct e; auto].

    solve [destruct e; auto].

    generalize (view_eval H0); intro K.
    destruct IHapprox; try (solve [
        inversion K; subst; auto]).
      solve [inversion K; subst; inversion
          IHapprox; subst; auto].
Qed.

Lemma accept_env_inv : forall r,
    accept_env r →
  forall x v, lookup r x = Some v → accept (
      embed v) ∧ Γ (get_label (embed v)) ⊆
      ρ x.
Proof.
  induction 1; inversion 1; subst.
  destruct (string_dec x0 x); eauto.
  inversion H4; subst; auto.
Qed.
Hint Resolve accept_env_inv.

Lemma accept_ext_env : forall r (x:string)
    v, accept (embed v) → accept_env r →
    Γ (get_label (embed v)) ⊆ ρ x →
    accept_env (ext_map r x v).
Proof.
  intros.
  functional induction (ext_map r x v);
      auto.
    inversion H0; auto.
    inversion H0; auto.
Qed.
Hint Resolve accept_ext_env.

Lemma accept_monotonic : forall t ℓ,
    accept (Expr t ℓ) → forall ℓ', Γ ℓ ⊆ Γ ℓ'
     → accept (Expr t ℓ').
Proof.
  assert (forall e, accept e → forall ℓ, Γ (
      get_label e) ⊆ Γ ℓ→ accept (Expr (
      get_term e) ℓ)).
    induction 1; simpl; eauto with
        datatypes.
      constructor; auto.
      intros; edestruct H1; eauto with
          datatypes.
      eauto with datatypes.
      constructor; eauto with datatypes.
  intros; apply (H (Expr t ℓ)); auto.
Qed.
Hint Resolve accept_monotonic.

Lemma eval_monotonic : forall e n e', eval e
     n e' → accept e → Γ (get_label e') ⊆
    Γ (get_label e).
Proof.
  destruct 1.
  induction C; simpl; auto with datatypes.
  induction H; simpl in *; try (solve [auto
      with datatypes]).
    inversion 1; subst.
    inversion H3; subst.
```

```
    inversion H2; subst.
    generalize (view_approx H11);
        inversion 1; subst; simpl in *.
    edestruct H5; solve [eauto].

    inversion 1; subst; solve [auto].

    inversion 1; subst; solve [auto].
Qed.
Hint Resolve eval_monotonic.

Lemma plug_eval : forall m n (C:ctxt m n) e
    n e', accept e →
  eval e n e' → Γ (get_label (plug C e')) ⊆
      Γ (get_label (plug C e)).
Proof.
  induction C; eauto with datatypes.
Qed.
Hint Resolve plug_eval.

Lemma accept_plug : forall m n (C:ctxt m n)
    e, accept (plug C e) → accept e.
Proof.
  induction C; simpl; (solve [auto] ||
      inversion 1; subst; auto).
    rewrite mforall_app in H2.
    destruct H2.
    inversion H1; subst.
    destruct H5; auto.
Qed.
Hint Resolve accept_plug.

Lemma soundness_help : forall e n e', red e
    n e' → accept e → accept e'.
Proof.
  induction 1.
    solve [inversion 1; subst; auto with
        datatypes].

    inversion 1; subst.
    inversion H2; subst.
    econstructor; eauto.
    eapply mforall_map; eauto.
    eapply mforall_monotonic; eauto.
    intros a (e,ℓ'') (K1,K2); simpl in *.
    constructor; solve [eauto].

    solve [inversion 1; inversion H2; subst
        ; auto with datatypes].

    inversion 1; subst.
    inversion H2; subst.
    solve [auto with datatypes].

    inversion 1; subst.
    inversion H2; subst.
    econstructor; eauto.
    apply ApproxRed with (t := TClos (TAbs
        x (Expr t ℓ1)) r) (ℓ:=ℓ2) (n:=S n);
        eauto.
    solve [eauto using red].

    inversion 1; subst.
    inversion H2; subst.
    econstructor; eauto.
    apply ApproxRed with (t := TClos (TBox (
        Expr t ℓ1)) r) (ℓ:=ℓ2) (n:=S n);
        eauto.
    solve [eauto using red].

    inversion 1; subst.
    inversion H2; subst.
    solve [auto with datatypes].

    inversion 1; subst.
    inversion H2; subst.
    solve [auto with datatypes].

    inversion 1; subst.
    inversion H2; subst.
    solve [auto with datatypes].

    inversion 1; subst.
    inversion H2; subst.
    solve [auto with datatypes].

    inversion 1; subst.
    inversion H2; subst.
    solve [auto with datatypes].

    inversion 1; subst.
    inversion H2; subst.
    solve [auto with datatypes].

    inversion 1; subst.
    inversion H2; subst.
    solve [auto with datatypes].

    inversion 1; inversion H3; subst.
    edestruct accept_env_inv; eauto.
    destruct (embed v); unfold relabel;
        simpl in *.
    solve [eauto with datatypes].

    inversion 1; subst.
    inversion H3; subst.
    inversion H2; subst; simpl in *.
    generalize (view_approx H11);
```

```
    inversion 1; subst.
edestruct H5; solve [eauto].

destruct (embed v); unfold relabel.
inversion 1; subst.
inversion H2; subst.
generalize (view_approx H7);
    inversion 1; subst.
specialize (H3 _ H6); solve [eauto].

destruct (embed v).
inversion 1; subst.
inversion H3; subst.
generalize (view_approx H8);
    inversion 1; subst.
econstructor; eauto.
eapply accept_monotonic; eauto.
exact (H5 _ H7).

inversion 1; subst; solve [eauto].

inversion 1; subst; solve [eauto].

inversion 1; subst; simpl in *.
inversion H4; subst.
rewrite mforall_map in H3.
destruct (mforall_lookup _ H H3).
specialize (H6 _ ℓ' x H10).
destruct (embed v).
eapply accept_monotonic; solve [eauto
    using protos with datatypes].

inversion 1; subst; simpl in *.
inversion H5; subst.
constructor; auto.
  rewrite mforall_map in H4.
  destruct (mforall_lookup _ H0 H4);
      solve [auto].

  intros.
  eapply H7; eauto.
  rewrite mforall_map in H4.
  destruct (mforall_lookup _ H0 H4);
      simpl in *.
  solve [eauto using protos].

  simpl.
  rewrite mforall_map in H4.
  destruct (mforall_lookup _ H0 H4).
      simpl in *.
  eapply H7 in H11.
  eauto with datatypes.
  solve [eauto using protos].

inversion 1; subst.
solve [eauto].

inversion 1; subst.
inversion H4; subst.
econstructor; eauto.
rewrite mforall_map in H2.
apply mforall_map.
apply mforall_ext; solve [auto].

inversion 1; subst.
inversion H3; subst; simpl in *.
rewrite mforall_map in H2.
econstructor; eauto.
apply mforall_map.
solve [eauto using mforall_del].

(* PushMark *)
inversion 1; subst.
inversion H2; subst.
solve [auto with datatypes].

(* LiftIf *)
inversion 1. subst.
inversion H4. subst.
constructor; auto.
  constructor; auto. simpl in H9. simpl.
      simpl in H10. solve [eauto with
      datatypes].
  simpl. solve [auto with datatypes].

(* LiftApp *)
inversion 1; subst.
inversion H3; subst.
inversion H2; subst; simpl in *.
constructor; eauto with datatypes.
solve [constructor; eauto with
    datatypes].

(* LiftUnbox *)
inversion 1; subst.
inversion H2; subst.
constructor; eauto with datatypes.
solve [constructor; eauto with
    datatypes].

(* LiftRunIn *)
inversion 1; subst.
inversion H3; subst.
constructor; eauto with datatypes.
solve [constructor; eauto with
    datatypes].

(* LiftReadSel *)
```

```
    inversion 1; subst.
    inversion H4; subst.
    solve [constructor; eauto with
        datatypes].

    (* LiftReadRec *)
    inversion 1; subst.
    inversion H3; subst.
    constructor; eauto with datatypes.
    solve [constructor; eauto with
        datatypes].

    (* LiftWriteSel *)
    inversion 1; subst.
    inversion H5; subst.
    solve [constructor; eauto with
        datatypes].

    (* LiftWriteRec *)
    inversion 1; subst.
    inversion H4; subst.
    constructor; eauto with datatypes.

    (* LiftDelSel *)
    inversion 1; subst.
    inversion H4; subst.
    solve [constructor; eauto with
        datatypes].

    (* LiftWriteRec *)
    inversion 1; subst.
    inversion H3; subst.
    constructor; eauto with datatypes.
Qed.

Lemma soundness: forall e n e', eval e n e'
    → accept e → accept e'.
Proof.
  destruct 1.
  induction C; simpl; auto.
    solve [eauto using soundness_help].

    inversion 1; subst.
    rewrite mforall_app in H3.
    destruct H3.
    inversion H2; subst.
    destruct H6.
    econstructor; eauto.
    apply mforall_app.
    constructor; auto.
    constructor; auto.
    assert (accept e) by eauto.
    assert (H7 : eval e n e') by eauto.
    generalize (plug_eval C H6 H7).
    solve [eauto with datatypes].

    inversion 1; subst.
    assert (K : eval (plug (CAbs l s C) e) (
        S m) (plug (CAbs l s C) e')).
      eauto using eval.
    solve [eauto].

    inversion 1; subst.
    assert (accept e) by eauto.
    assert (K : eval e n e') by eauto.
    generalize (plug_eval C H1 K).
    constructor; auto.
    solve [eauto with datatypes].

    inversion 1; subst.
    constructor; auto.
    intros; edestruct H6; eauto.
    assert (accept e) by eauto.
    assert (K : eval e n e') by eauto.
    generalize (plug_eval C H8 K).
    solve [eauto with datatypes].

    inversion 1; subst.
    assert (K : eval (plug (CBox l C) e) m (
        plug (CBox l C) e')).
      eauto using eval.
    solve [eauto].

    inversion 1; subst.
    assert (accept e) by eauto.
    assert (K : eval e n e') by eauto.
    generalize (plug_eval C H1 K).
    constructor; auto.
    solve [eauto with datatypes].

    inversion 1; subst.
    assert (accept e) by eauto.
    assert (K : eval e n e') by eauto.
    generalize (plug_eval C H1 K).
    constructor; auto.
    solve [eauto with datatypes].

    inversion 1; subst.
    assert (accept e) by eauto.
    assert (K : eval e n e') by eauto.
    generalize (plug_eval C H1 K).
    solve [eauto with datatypes].

    inversion 1; subst.
    assert (accept e) by eauto.
    assert (eval e n e') by eauto.
    constructor; solve [eauto with
        datatypes].
```

```
    inversion 1; subst.
    assert (accept e) by eauto.
    assert (eval e n e') by eauto.
    constructor; solve [eauto with
        datatypes].

    inversion 1; subst.
    assert (accept e) by eauto.
    assert (K : eval e n e') by eauto.
    generalize (plug_eval C H1 K).
    constructor; solve [eauto with
        datatypes].

    inversion 1; subst.
    assert (accept e) by eauto.
    constructor; solve [eauto with
        datatypes].

    inversion 1; subst.
    assert (accept e) by eauto.
    assert (Γ (get_label (plug C e')) ⊆ Γ (
        get_label (plug C e))) by eauto.
    constructor; solve [eauto with
        datatypes].

    inversion 1; subst.
    assert (accept e) by eauto.
    assert (K : eval e n e') by eauto.
    generalize (plug_eval C H1 K).
    constructor; solve [eauto with
        datatypes].

    inversion 1; subst.
    assert (accept e) by eauto.
    assert (Γ (get_label (plug C e')) ⊆ Γ (
        get_label (plug C e))) by eauto.
    constructor; solve [eauto with
        datatypes].

    inversion 1; subst.
    constructor; auto.
    assert (accept e) by eauto.
    assert (K : eval e n e') by eauto.
    generalize (plug_eval C H1 K); solve [
        eauto with datatypes].

    inversion 1; subst.
    constructor; auto.
    assert (accept e) by eauto.
    assert (K : eval e n e') by eauto.
    generalize (plug_eval C H1 K); solve [
        eauto with datatypes].

    inversion 1; subst.
    assert (accept e) by eauto.
    assert (K : eval e n e') by eauto.
    generalize (plug_eval C H1 K).
    solve [constructor; eauto with
        datatypes].

    (* CMark *)
    inversion 1; subst.
    constructor; auto.
    assert (accept e) by eauto.
    assert (K : eval e n e') by eauto.
    generalize (plug_eval C H1 K); solve [
        eauto with datatypes].

  Qed.
End Analysis.
```

A.4 Improved 0CFA and Information Flow Analysis for SLamJS — cfa.v

```
(** A more sophisticated binding
    analysis for Staged LambdaJS. *)
Set Implicit Arguments.

Require Import Eqdep Le Bool String List.
Require Import Program.Equality.
Require Import lib.
Require Import lambdajs.

Section Analysis.
  Inductive absval := NULL | UNDEF | BOOL |
      NUM | STR
    | LAM : string → expr → absval
    | BOX : expr → absval
    | REC : label → absval
    | MARK : marker → absval.

  Inductive absctxt : nat → Type :=
  | Base : finmap string label → absctxt 0
  | Boxed : forall n, absctxt n → label →
      finmap string label → absctxt (S n).
```

```
Definition ctxt_lookup n (Ξ :absctxt n) (x:
    string) :=
match Ξ with
| Base φ ⇒ lookup φx
| Boxed _ _ ℓ φ ⇒
  match lookup φx with
  | Some ℓ ⇒ Some ℓ
  | None ⇒  Some ℓ
  end
end.
Coercion ctxt_lookup : absctxt >→
    Funclass.

Definition ext n (Ξ :absctxt n) x ℓ:=
match Ξ in absctxt n return absctxt n with
  | Base φ ⇒ Base (ext_map φx ℓ)
  | Boxed _ Ξ ' ℓ0 φ ⇒ Boxed Ξ ' ℓ0(ext_map
      φx ℓ)
end.

Fixpoint get_base n (Ξ :absctxt n) :=
match Ξ with
| Base φ ⇒ φ
| Boxed _ Ξ _ _ ⇒ get_base Ξ
end.

Fixpoint set_base n (Ξ :absctxt n) φ:=
match Ξ in absctxt n return absctxt n with
| Base _ ⇒ Base φ
| Boxed _ Ξ ℓ φ' ⇒ Boxed (set_base Ξ φ) ℓφ'
end.

Lemma set_base_lookup : forall (Ξ :absctxt
    0) φx, (set_base Ξ φ) x = lookup φx.
Proof.
  intros.
  dependent destruction Ξ ; auto.
Qed.

Lemma set_base_lookup2 : forall n (Ξ :
    absctxt (S n)) φx, (set_base Ξ φ) x = Ξ x
    .
Proof.
  intros.
  dependent destruction Ξ ; auto.
Qed.

Lemma set_set_base : forall n (Ξ :absctxt n
    ) φφ', set_base (set_base Ξ φ) φ' =
    set_base Ξ φ'.
Proof.
  induction Ξ ; simpl; congruence.
Qed.

Lemma set_get_base : forall n (Ξ :absctxt n
    ), set_base Ξ (get_base Ξ ) = Ξ .
Proof.
  induction Ξ ; simpl; congruence.
Qed.

Lemma lookup_ctxt_ext : forall n (Ξ :
    absctxt n) x ℓ, (ext Ξ x ℓ) x = Some ℓ.
Proof.
  destruct Ξ ; simpl; intros.
    rewrite lookup_ext; auto.
    rewrite lookup_ext; auto.
Qed.

Lemma lookup_ctxt_ext2 : forall n (Ξ :
    absctxt n) x x' ℓ, x <> x' → (ext Ξ x ℓ) x
    ' = Ξ x'.
Proof.
  destruct Ξ ; simpl; intros.
    rewrite lookup_ext2; auto.
    rewrite lookup_ext2; auto.
Qed.

Variable Γ : label → 𝒫 absval.
Variable ρ : label → string → 𝒫 absval.


Definition is_mark (a : absval) : bool :=
  match a with
  | MARK _ ⇒ true
  | _ ⇒ false
  end.

Definition filter_mark := filter is_mark.

Lemma filter_subset : forall A (f : A →
    bool) (l: list A),
  filter f l ⊆ l.
Proof.
  intros. induction l.
    auto with datatypes.
    simpl. destruct (f a);  auto with
        datatypes.
Qed.
Hint Resolve filter_subset.

Lemma filter_mark_subset : forall ℓ,
  filter_mark(Γ  ℓ) ⊆ Γ ℓ.
Proof.
  intros. unfold filter_mark. auto.
Qed.
Hint Resolve filter_mark_subset.

Lemma direct__indirect : forall ℓ1ℓ2,
```

```
  Γ ℓ1 ⊆ Γ ℓ2 → filter_mark (Γ ℓ1) ⊆ Γ ℓ2.
Proof.
  intros. eauto with *.
Qed.
Hint Resolve direct__indirect.

Lemma subset_nil__nil : forall A (l : list A
    ),
  l ⊆ nil → l = nil.
Proof.
  intros. destruct l.
    auto.
    unfold incl in H.
    specialize H with (a0 := a). simpl in H.
        intuition.
Qed.

Lemma nil_subset_all : forall A (l : list A),
  nil ⊆ l.
Proof.
  intros. unfold incl. intros. auto with *.
Qed.

Lemma filter_mono : forall A (f : A → bool)
    (l1 : list A) (l2 : list A),
  l1 ⊆ l2 → (filter f l1) ⊆ (filter f l2).
Proof.
  induction l1.
    intros. simpl. apply nil_subset_all.
    intros.
    assert (In a l2). auto with datatypes.
    assert (l1 ⊆ (a :: l1)). auto with
        datatypes.
    assert (l1 ⊆ l2). eauto with datatypes.
    apply IHl1 in H2.
    simpl. remember (f a). destruct b; auto.
    assert (In a (filter f l2)). apply
        filter_In. auto.
    auto with datatypes.
Qed.

Lemma direct__indirect_mono : forall ℓ1ℓ2,
  Γ ℓ1 ⊆ Γ ℓ2 → filter_mark (Γ ℓ1) ⊆
      filter_mark (Γ ℓ2).
Proof.
  intros. apply filter_mono. auto.
Qed.
Hint Resolve direct__indirect_mono.

Lemma mark_in_filter_mark : forall m ℓ,
  MARK m ∈ Γ ℓ → MARK m ∈ filter_mark (Γ ℓ)
      .
Proof.
  intros. apply filter_In. auto.
Qed.
Hint Resolve mark_in_filter_mark.

Lemma mark_propogates_indirectly :
    forall m ℓ1ℓ2,
  MARK m ∈ Γ ℓ1 → filter_mark (Γ ℓ1) ⊆ Γ ℓ2
      → MARK m ∈ Γ ℓ2.
Proof.
  intros. auto.
Qed.
Hint Resolve mark_propogates_indirectly.

Inductive protos : label → label → Prop
    :=
| ProtoRefl : forall ℓ, protos ℓ ℓ
| ProtoStep : forall ℓℓ' ℓ'', REC ℓ' ∈ ρ ℓ (
    "__proto__")%string → protos ℓ' ℓ''
    → protos ℓℓ''.

Inductive accept : forall n, absctxt n →
    expr → Prop :=
| AcceptLit : forall n (Ξ :absctxt n) v k ℓ,
    v ∈ Γ ℓ → approx v (TLit k) →
        accept Ξ (Expr (TLit k) ℓ)
| AcceptVar : forall n (Ξ :absctxt n) (x:
    string) ℓ,
        (forall β, Ξ x = Some β → ρ β
            x ⊆ Γ ℓ) →
        accept Ξ (Expr (TVar x) ℓ)
| AcceptRec : forall n (Ξ :absctxt n) xes ℓℓ
    ',
        mforall (fun x e ⇒
            accept Ξ e ∧ Γ (
                get_label e) ⊆
                ρ ℓ' x) xes →
        REC ℓ' ∈ Γ ℓ →
        accept Ξ (Expr (TRec xes) ℓ)
| AcceptAbs : forall n (Ξ :absctxt n) v x e ℓ,
    accept (ext Ξ x (get_label e)) e →
        v ∈ Γ ℓ → approx v (TAbs x e)
            →
        accept Ξ (Expr (TAbs x e) ℓ)
| AcceptApp : forall n (Ξ :absctxt n) e1 e2 ℓ,
    accept Ξ e1 → accept Ξ e2 →
        (forall x e, LAM x e ∈ Γ (
            get_label e1) →
        Γ (get_label e2) ⊆ ρ (
            get_label e) x ∧
        Γ (get_label e) ⊆ Γ ℓ) →
      filter_mark (Γ (get_label e1))
          ⊆ Γ ℓ →
        accept Ξ (Expr (TApp e1 e2) ℓ
            )
| AcceptBox : forall n (Ξ :absctxt n) v e ℓ,
```

```
    accept (Boxed Ξ (get_label e) NilMap) e
    →
            v ∈ Γ ℓ → approx v (TBox e)
                →
              accept Ξ (Expr (TBox e) ℓ)
| AcceptUnbox : forall n (Ξ :absctxt n) ℓ1φ
    e ℓ2, accept Ξ e →
            (forall e', BOX e' ∈ Γ (
                get_label e) →
              (forall x β, (Boxed Ξ ℓ1 φ)
                  x = Some β → ρ βx = ρ (
                  get_label e') x) ∧
              Γ (get_label e') ⊆ Γ ℓ2) →
          filter_mark (Γ (get_label e)) ⊆
              Γ ℓ2→
              accept (Boxed Ξ ℓ1 φ) (Expr (
                  TUnbox e) ℓ2)
| AcceptRun : forall n (Ξ :absctxt n) e ℓ,
    accept Ξ e →
            (forall e', BOX e' ∈ Γ (
                get_label e) →
              (forall x β, Ξ x = Some β →
                  ρ βx = ρ (get_label e')
                  x) ∧
              Γ (get_label e') ⊆ Γ ℓ) →
          filter_mark (Γ (get_label e)) ⊆
              Γ ℓ →
              accept Ξ (Expr (TRun e) ℓ)
| AcceptIf : forall n (Ξ :absctxt n) e1 e2 e
    3 ℓ, accept Ξ e1 → accept Ξ e2 →
    accept Ξ e3 →
            Γ (get_label e2) ⊆ Γ ℓ →
            Γ (get_label e3) ⊆ Γ ℓ →
            filter_mark (Γ (get_label e1
                )) ⊆ Γ ℓ→
              accept Ξ (Expr (TIf e1 e2 e3)
                  ℓ)
| AcceptRecRead : forall n (Ξ :absctxt n) e1
    e2 ℓ,
            accept Ξ e1 → accept Ξ e2 →
            (forall ℓ' ℓ'' x, REC ℓ' ∈ Γ (
                get_label e1) →
              protos ℓ' ℓ'' → ρ ℓ'' x ⊆ Γ ℓ
                ) →
            UNDEF ∈ Γ ℓ →
            filter_mark (Γ (get_label e1
                )) ⊆ Γ ℓ→
            filter_mark (Γ (get_label e2
                )) ⊆ Γ ℓ→
              accept Ξ (Expr (TRecRead e1
                  e2) ℓ)
| AcceptRecWrite : forall n (Ξ :absctxt n) e
    1 e2 e3 ℓ,
            accept Ξ e1 → accept Ξ e2 →
                accept Ξ e3 →
            (forall ℓ' x, REC ℓ' ∈ Γ (
                get_label e1) →
              Γ (get_label e3) ⊆ ρ ℓ' x)
                →
            Γ (get_label e1) ⊆ Γ ℓ →
            filter_mark (Γ (get_label e2
                )) ⊆ Γ ℓ→
              accept Ξ (Expr (TRecWrite e
                  1 e2 e3) ℓ)
| AcceptRecDel : forall n (Ξ :absctxt n) e1
    e2 ℓ,
            accept Ξ e1 → accept Ξ e2 →
                Γ (get_label e1) ⊆ Γ ℓ→
            filter_mark (Γ (get_label e2
                )) ⊆ Γ ℓ→
              accept Ξ (Expr (TRecDel e1 e
                  2) ℓ)
| AcceptClos : forall n (Ξ :absctxt n) φt r ℓ
    ,
            accept_env φr → accept (
                set_base Ξ φ) (Expr t ℓ)
                →
              accept Ξ (Expr (TClos t r) ℓ)
| AcceptRunIn : forall n (Ξ :absctxt n) φr e
    ℓ,
            accept_env φr → accept (
                set_base Ξ φ) e →
            (forall e', BOX e' ∈ Γ (
                get_label e) →
              (forall x β, (set_base Ξ φ)
                  x = Some β→ ρ βx = ρ (
                  get_label e') x) ∧
              Γ (get_label e') ⊆ Γ ℓ) →
          filter_mark (Γ (get_label e)) ⊆
              Γ ℓ →
              accept Ξ (Expr (TRunIn e r) ℓ
                  )
| AcceptMark : forall n (Ξ :absctxt n) e m ℓ,
    accept Ξ e →
            Γ (get_label e) ⊆ Γ ℓ →
            (MARK m) ∈ Γ ℓ →
              accept Ξ (Expr (TMark m e)
                  ℓ)

with accept_env : finmap string label →
    env → Prop :=
| AcceptEnv : forall φr,
            (forall x v, lookup r x = Some
                v →
              exists β, lookup φx = Some
                  β ∧
                    accept (Base φ) (
                        embed v) ∧
```

```
                                    Γ (get_label (
                                        embed v)) ⊆ ρ
                                        βx) →
                                accept_env φr
with approx : absval → term → Prop :=
| ApproxNull : approx NULL Null
| ApproxUndef : approx UNDEF Undef
| ApproxBool : forall b, approx BOOL (
    BoolLit b)
| ApproxNum : forall n, approx NUM (NumLit
    n)
| ApproxStr : forall s, approx STR (StrLit
    s)
| ApproxLam : forall x e, approx (LAM x e) (
    TAbs x e)
| ApproxBox : forall e, approx (BOX e) (
    TBox e)
| ApproxRec : forall ℓ xes, mforall (fun x
    e ⇒ exists v, v ∈ ρ ℓ x ∧ approx v (
    get_term e)) xes → approx (REC ℓ) (TRec
    xes)
| ApproxRed : forall v t ℓ m t' ℓ', approx
    v t → eval (t^ℓ) m (t'^ℓ') → approx v t'
| ApproxClos : forall n (Ξ :absctxt n) v t r,
    approx v t → accept_env (get_base Ξ )
    r → approx v (TClos t r).
Hint Constructors accept accept_env
    approx.

Inductive approx_view : absval → term →
    Prop :=
| AVNull : approx_view NULL Null
| AVUndef : approx_view UNDEF Undef
| AVBool : forall b, approx_view BOOL (
    BoolLit b)
| AVNum : forall n, approx_view NUM (
    NumLit n)
| AVStr : forall s, approx_view STR (
    StrLit s)
| AVLam : forall ℓ x t t', approx_view (
    LAM x (t^ℓ)) (TAbs x (t'^ℓ))
| AVBox : forall ℓ t t', approx_view (BOX (
    t^ℓ)) (TBox (t'^ℓ))
| AVRec : forall ℓ xes, approx_view (REC ℓ)
    (TRec xes)
| AVClos : forall v t r, approx_view v t →
    approx_view v (TClos t r).
Hint Constructors approx_view.

Lemma view_approx : forall v t, approx v t
    → approx_view v t.
Proof.
  induction 1; auto.
    destruct e; auto.

    destruct e; auto.

    generalize (view_eval H0); intro K.
    destruct IHapprox; try (solve [
        inversion K; subst; auto]).
      solve [inversion K; subst; inversion
          IHapprox; subst; auto].
Qed.

Lemma accept_literal : forall n (Ξ :
    absctxt n) (k:literal) ℓ, accept Ξ (k^ℓ)
    → forall n' (Ξ ':absctxt n'), accept Ξ '
    (k^ℓ).
Proof.
  intros.
  dependent destruction H.
  set (K := view_approx H0).
  inversion K; subst; eauto.
Qed.
Hint Resolve accept_literal.

Lemma eval_monotonic : forall e n e', eval e
    n e' → forall (Ξ :absctxt n), accept Ξ
    e → Γ (get_label e') ⊆ Γ (get_label e).
Proof.
  destruct 1.
  induction C; simpl; auto with datatypes.
  induction H; simpl in *; try (solve [auto
      with datatypes]).
    intros.
    dependent destruction H.
    dependent destruction H.
    dependent destruction H3.
    generalize (view_approx H5);
        inversion 1; subst; simpl in *.
    edestruct H1; solve [eauto].

    inversion 1; solve [auto].

    inversion 1; solve [auto].
Qed.
Hint Resolve eval_monotonic.

Lemma plug_eval : forall m n (C:ctxt m n) e
    e' (Ξ :absctxt n), accept Ξ e →
  eval e n e' → Γ (get_label (plug C e')) ⊆
      Γ (get_label (plug C e)).
Proof.
  induction C; eauto with datatypes.
Qed.
Hint Resolve plug_eval.

Lemma accept_plug : forall m n (C:ctxt m n) (
```

```
    Ξ :absctxt m) e, accept Ξ (plug C e) →
    exists Ξ ':absctxt n, accept Ξ ' e.
Proof.
  induction C; simpl; intros.
    eauto.

    dependent destruction H.
    rewrite mforall_app in H.
    destruct H.
    inversion H1; subst.
    destruct H4; solve [eauto].

    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
    dependent destruction H; solve [eauto].
Qed.

Lemma lookup_similar : forall n (Ξ :
    absctxt (S n)) φx, (set_base Ξ φ) x = Ξ x
    .
Proof.
  assert (forall n (Ξ :absctxt n),
    match n as n return absctxt n → Prop
        with
    | 0 ⇒ fun _ ⇒ True
    | S n ⇒ fun Ξ ⇒ forall φx, (set_base
        Ξ φ) x = Ξ x
    end Ξ ).
    induction Ξ ; auto.
  intros.
  exact (H _ Ξ φ x).
Qed.

Lemma trivial_relabel : forall e, e =
    relabel e (get_label e).
Proof.
  destruct e; auto.
Qed.

Lemma accept_embed : forall n (v:value n) (
    Ξ :absctxt n), accept Ξ (embed v) →
  forall φℓ,  Γ (get_label (embed v)) ⊆ Γ ℓ
      → accept (set_base Ξ φ) (relabel (
      embed v) ℓ).
Proof.
  induction v using val_altind; unfold
      relabel; simpl; intros.
    dependent destruction H.
    econstructor; solve [eauto].

    dependent destruction H0.
    econstructor; eauto.
    rewrite mforall_map in H0.
    apply mforall_map.
    induction xvs; constructor.
      inversion H; subst.
      inversion H0; subst.
      destruct H6.
      constructor; auto.
      erewrite trivial_relabel; solve [
          eauto with datatypes].

      inversion H; subst.
      inversion H0; subst; solve [auto].

    dependent destruction Ξ .
    dependent destruction H.
    solve [eauto with datatypes].

    dependent destruction Ξ .
    dependent destruction H.
    econstructor; eauto.
    assert (accept (set_base (ext (Boxed Ξ
        l f) x (get_label (embed v))) φ)
                (embed v)).
      rewrite trivial_relabel; solve [
          eauto with datatypes].
    solve [auto].

    dependent destruction H.
    econstructor; eauto with datatypes.
      rewrite trivial_relabel; solve [
          eauto with datatypes].

      rewrite trivial_relabel; solve [
          eauto with datatypes].

      intros.
      edestruct H2; solve [eauto with
          datatypes].

    dependent destruction H.
    econstructor; eauto.
```

```
rewrite trivial_relabel.
eapply (IHv _ H _ _);  solve [eauto with
    datatypes].

dependent destruction H.
econstructor; eauto with datatypes.
  rewrite trivial_relabel; solve [
      eauto with datatypes].

  intros.
  edestruct H1; solve [eauto with
      datatypes].

dependent destruction H.
econstructor; eauto with datatypes.
  rewrite trivial_relabel; solve [
      eauto with datatypes].

  intros.
  edestruct H1; eauto.
  constructor; eauto with datatypes.
  intros.
  rewrite lookup_similar in *.
  solve [eauto].

dependent destruction H.
econstructor; eauto with datatypes.
  rewrite trivial_relabel; solve [
      eauto with datatypes].

  rewrite trivial_relabel; solve [
      eauto with datatypes].

  rewrite trivial_relabel; solve [
      eauto with datatypes].

dependent destruction H.
dependent destruction H1.
generalize (view_approx H3);
    inversion 1; subst.
econstructor; eauto.
rewrite set_set_base; solve [eauto].

dependent destruction H.
econstructor; eauto with datatypes.
  rewrite trivial_relabel; solve [
      eauto with datatypes].

  rewrite trivial_relabel; solve [
      eauto with datatypes].

dependent destruction H.
econstructor; eauto with datatypes.
  rewrite trivial_relabel; solve [
      eauto with datatypes].

  rewrite trivial_relabel; solve [
      eauto with datatypes].

  rewrite trivial_relabel; solve [
      eauto with datatypes].

    dependent destruction H.
    econstructor; eauto with datatypes.
      rewrite trivial_relabel; solve [
          eauto with datatypes].

      rewrite trivial_relabel; solve [
          eauto with datatypes].

    dependent destruction H.
    econstructor; eauto with datatypes.
      rewrite trivial_relabel; solve [
          eauto with datatypes].
Qed.

Lemma set_base_base : forall (Ξ :absctxt 0)
    φ, set_base Ξ φ= Base φ.
Proof.
  intros.
  dependent destruction Ξ ; auto.
Qed.

Lemma accept_ext_env : forall r φφ' (x:
    string) v ℓ,
  accept_env φr → accept (Base φ') (embed
      v) →
  Γ (get_label (embed v)) ⊆ ρ ℓ x →
    accept_env (ext_map φx ℓ) (ext_map r x
        v).
Proof.
  constructor.
  intros x' v' K.
  destruct (x' == x); subst.
    rewrite lookup_ext in K.
    inversion K; subst.
    assert (K' :  accept (Base (ext_map φx ℓ)
        ) (embed v')).
      rewrite trivial_relabel.
      apply (accept_embed _ H0 (ext_map φx
          ℓ));  solve [auto with datatypes].
    solve [eauto using lookup_ext].

    rewrite lookup_ext2 in K; auto.
    destruct H.
    destruct (H _ _ K)  as (β'&?&?&?).
    assert (K' :  accept (Base (ext_map φx ℓ)
        ) (embed v')).
```

```
      rewrite trivial_relabel.
      apply (accept_embed _ H3 (ext_map φx
          ℓ)); solve [auto with datatypes].
    exists β'.
    constructor; eauto.
    rewrite lookup_ext₂; solve [auto].
Qed.

Inductive compat : forall m n, absctxt n →
    absctxt m → Prop :=
| CompatBase : forall φ, compat (Base φ) (
    Base φ)
| CompatBoxedBase : forall m (Ξ :absctxt m)
    ℓφφ',
    (forall x β, lookup φx = Some β →
        lookup φ' x = Some β) →
    (forall x β, lookup φ' x = Some β → ρ β
        x = ρ ℓ x ∨ lookup φx = Some β) →
      compat (Boxed Ξ ℓ φ) (Base φ')
| CompatBoxedBoxed : forall m n (Ξ :absctxt
    m) (Ξ ':absctxt n) ℓℓ' φφ',
    compat Ξ Ξ ' →
    (forall x β, lookup φx = Some β →
        lookup φ' x = Some β) →
    (forall x β, (Boxed Ξ ' ℓ' φ') x = Some β
        → ρ βx = ρ ℓ x ∨ lookup φx = Some
        β) →
      compat (Boxed Ξ ℓ φ) (Boxed Ξ ' ℓ' φ').

Lemma compat_refl : forall n (Ξ :absctxt n),
    compat Ξ Ξ .
Proof.
  induction Ξ ; constructor; auto.
  intros x β; simpl.
  destruct (lookup f x); inversion 1; subst;
      auto.
Qed.

Lemma accept_compat : forall n (v:value n) (
    Ξ :absctxt n) m (Ξ ':absctxt m),
  n > 0 → S m >= n →
  accept Ξ (embed v) → compat Ξ Ξ ' →
  forall ℓ'', Γ (get_label (embed v)) ⊆ Γ ℓ
      '' →
    accept Ξ ' (relabel (embed v) ℓ'').
Proof.
  induction v using val_altind.
    intros; simpl.
    dependent destruction H1.
    econstructor; solve [eauto].

    unfold relabel; simpl; intros.
    dependent destruction H2.
    econstructor; eauto.
    rewrite mforall_map in H2.
    apply mforall_map.
    induction xvs; constructor.
      inversion H; subst.
      inversion H2; subst.
      destruct H9.
      constructor; auto.
      destruct (embed b).
      replace (t^l) with (relabel (t^l) l);
          auto.
      eapply H8; solve [eauto with
          datatypes].

      inversion H; subst.
      inversion H2; subst.
      solve [auto].

    unfold relabel; simpl; intros.
    constructor; auto.
    dependent destruction H2.
      intros βK.
      dependent destruction H1; simpl in *.
      specialize (H2 x).
      specialize (H4 x).
      destruct (lookup φx).
        erewrite H2 in *; solve [eauto with
            datatypes].

        edestruct H4; eauto.
          rewrite H5; solve [eauto with
              datatypes].

          solve [discriminate H5].

      intros βK.
      dependent destruction H1; simpl in *.
      specialize (H4 x).
      specialize (H5 x).
      destruct (lookup φx).
        erewrite H4 in *; solve [eauto with
            datatypes].

        edestruct H5; eauto.
          rewrite H6; solve [eauto with
              datatypes].

          solve [discriminate H6].

    unfold relabel; simpl; intros.
    dependent destruction H1.
    dependent destruction H2.
      econstructor; eauto.
      destruct (embed v).
      replace (t^l) with (relabel (t^l) l);
```

```
      auto.
  eapply IHv; eauto with datatypes.
  simpl; constructor; auto.
    intros y β'.
    destruct (y == x); subst.
      rewrite ! lookup_ext; solve [auto
          ].

      rewrite ! lookup_ext₂; solve [
          auto].

    intros y β'; simpl.
    destruct (y == x); subst.
      rewrite ! lookup_ext; solve [auto
          ].

      rewrite ! lookup_ext₂; solve [
          auto].

  econstructor; eauto.
  destruct (embed v).
  replace (t^l) with (relabel (t^l) l);
      auto.
  eapply IHv; eauto with datatypes.
  simpl; constructor; auto.
    intros y β'.
    destruct (y == x); subst.
      rewrite ! lookup_ext; solve [auto
          ].

      rewrite ! lookup_ext₂; solve [
          auto].

    intros y β'; simpl.
    destruct (y == x); subst.
      rewrite ! lookup_ext; solve [auto
          ].

      rewrite ! lookup_ext₂; solve [
          auto].

unfold relabel; simpl; intros.
dependent destruction H1.
econstructor; eauto with datatypes.
  destruct (embed v1) as (t, l).
  replace (t^l) with (relabel (t^l) l);
      auto.
  eapply IHv1; solve [eauto with
      datatypes].

  destruct (embed v2) as (t, l).
  replace (t^l) with (relabel (t^l) l);
      auto.
  eapply IHv2; solve [eauto with
      datatypes].

  intros.
  edestruct H1; solve [eauto with
      datatypes].

unfold relabel; simpl; intros.
dependent destruction H1.
econstructor; eauto.
destruct (embed v) as (t,l).
replace (t^l) with (relabel (t^l) l);
    auto.
apply IHv with (Ξ := Boxed Ξ l NilMap);
    auto with arith datatypes.
unfold relabel; simpl.
constructor; simpl; auto.
inversion 1; subst; solve [eauto].

unfold relabel; simpl; intros.
dependent destruction H1.
dependent destruction H2.
  inversion H0.
  solve [inversion H8].

  constructor; eauto with datatypes.
    destruct (embed v) as (t,l).
    replace (t^l) with (relabel (t^l) l);
        auto.
    eapply IHv; solve [eauto with arith
        datatypes].

    intros.
    edestruct H4; eauto.
    constructor; eauto with datatypes.
    intro x; simpl in *.
    specialize (H6 x).
    specialize (H7 x).
    specialize (H9 x).
    destruct (lookup φx).
      erewrite H6; solve [eauto].

      intros.
      edestruct H7; eauto.
        rewrite H12; solve [eauto].

        discriminate H12.

unfold relabel; simpl; intros.
dependent destruction H1.
constructor; eauto with datatypes.
  destruct (embed v) as (t, l).
  replace (t^l) with (relabel (t^l) l);
      auto.
  eapply IHv; solve [eauto with
```

```
      datatypes].

  dependent destruction H2.
    intros.
    edestruct H4; eauto with datatypes
        .
    constructor; eauto with datatypes.
    intro x; simpl in *.
    specialize (H2 x).
    specialize (H6 x).
    specialize (H8 x).
    destruct (lookup φx).
      erewrite H2; solve [eauto].

      intros.
      edestruct H6; eauto.
        rewrite H11; solve [eauto].

        discriminate H11.

    intros.
    edestruct H4; eauto with datatypes
        .
    constructor; eauto with datatypes.
    intro x; simpl in *.
    specialize (H6 x).
    specialize (H7 x).
    specialize (H9 x).
    destruct (lookup φx).
      erewrite H6; solve [eauto].

      intros.
      edestruct H7; eauto.
        rewrite H12; solve [eauto].

        discriminate H12.

unfold relabel; simpl; intros.
dependent destruction H1.
econstructor; eauto with datatypes.
  destruct (embed v1) as (t, l).
  replace (t^l) with (relabel (t^l) l);
      auto.
  eapply IHv1; solve [eauto with
      datatypes].

  destruct (embed v2) as (t, l).
  replace (t^l) with (relabel (t^l) l);
      auto.
  eapply IHv2; solve [eauto with
      datatypes].

  destruct (embed v3) as (t, l).
  replace (t^l) with (relabel (t^l) l);
      auto.
  eapply IHv3; solve [eauto with
      datatypes].

solve [inversion 1].

unfold relabel; simpl; intros.
dependent destruction H1.
econstructor; eauto with datatypes.
  destruct (embed v1) as (t, l).
  replace (t^l) with (relabel (t^l) l);
      auto.
  eapply IHv1; solve [eauto with
      datatypes].

  destruct (embed v2) as (t, l).
  replace (t^l) with (relabel (t^l) l);
      auto.
  eapply IHv2; solve [eauto with
      datatypes].

unfold relabel; simpl; intros.
dependent destruction H1.
econstructor; eauto with datatypes.
  destruct (embed v1) as (t, l).
  replace (t^l) with (relabel (t^l) l);
      auto.
  eapply IHv1; solve [eauto with
      datatypes].

  destruct (embed v2) as (t, l).
  replace (t^l) with (relabel (t^l) l);
      auto.
  eapply IHv2; solve [eauto with
      datatypes].

  destruct (embed v3) as (t, l).
  replace (t^l) with (relabel (t^l) l);
      auto.
  eapply IHv3; solve [eauto with
      datatypes].

unfold relabel; simpl; intros.
dependent destruction H1.
econstructor; eauto with datatypes.
  destruct (embed v1) as (t, l).
  replace (t^l) with (relabel (t^l) l);
      auto.
  eapply IHv1; solve [eauto with
      datatypes].

  destruct (embed v2) as (t, l).
  replace (t^l) with (relabel (t^l) l);
      auto.
```

```
      eapply IHv2; solve [eauto with
          datatypes].

    unfold relabel; simpl; intros.
    dependent destruction H1.
    econstructor; eauto with datatypes.
      destruct (embed v) as (t, l).
      replace (t^l) with (relabel (t^l) l);
          auto.
      eapply IHv; solve [eauto with
          datatypes].

Qed.

Lemma accept_compat1 : forall (Ξ :absctxt
    0) (v:value 1) ℓ,
  accept (Boxed Ξ ℓ NilMap) (embed v) →
  forall ℓ' φ' ℓ'',
    (forall x β, (Boxed Ξ ℓ' φ') x = Some β
        → ρ βx = ρ ℓ x) →
    Γ (get_label (embed v)) ⊆ Γ ℓ'' →
    accept (Boxed Ξ ℓ' φ') (relabel (embed v
        ) ℓ'').
Proof.
  intros.
  eapply (accept_compat v); eauto with
      arith.
  constructor; auto.
    apply compat_refl.

    inversion 1.
Qed.

Lemma accept_compat2 : forall (Ξ :absctxt
    0) (v:value 1) ℓ,
  accept (Boxed Ξ ℓ NilMap) (embed v) →
  forall ℓ'',
    (forall x β, Ξ x = Some β → ρ βx = ρ ℓ
        x) →
    Γ (get_label (embed v)) ⊆ Γ ℓ'' →
    accept Ξ (relabel (embed v) ℓ'').
Proof.
  intros.
  eapply (accept_compat v); eauto with
      arith datatypes.
  dependent destruction Ξ .
  constructor; auto.
  solve [inversion 1].
Qed.

Lemma soundness_help : forall e n e', red e
    n e' → forall (Ξ :absctxt n), accept Ξ
    e → accept Ξ e'.
Proof.
induction 1; intros.
  solve [inversion H; subst; eauto].

  dependent destruction H.
  dependent destruction H0.
  econstructor; eauto.
  eapply mforall_map; eauto.
  eapply mforall_monotonic; eauto.
  intros a (e,ℓ'') (K1,K2); simpl in *.
  constructor; solve [eauto].

  dependent destruction H.
  dependent destruction H0.
  rewrite set_base_lookup2 in H0.
  solve [eauto].

  dependent destruction H.
  dependent destruction H0.
  solve [eauto].

  dependent destruction H.
  dependent destruction H0.
  econstructor; eauto.
    econstructor; eauto.
    dependent destruction Ξ ; solve [
        auto].

    apply ApproxRed with (t := TClos (
        TAbs x (Expr t ℓ1)) r) (ℓ:=ℓ2) (ℓ
        ':=ℓ2) (m:=S n).
      apply ApproxClos with (Ξ :=Base φ);
          solve [auto].

      apply red_eval.
      solve [constructor].

  dependent destruction H.
  dependent destruction H0.
  econstructor; eauto.
  apply ApproxRed with (t := TClos (TBox (
      Expr t ℓ1)) r) (ℓ:=ℓ2) (ℓ':=ℓ2) (m:=S
      n).
    apply ApproxClos with (Ξ :=Base φ);
        solve [auto].

    apply red_eval.
    solve [constructor].

  dependent destruction H.
  dependent destruction H0.
  dependent destruction Ξ .
  dependent destruction n.
  solve [eauto]. solve [eauto].
```

```
dependent destruction H.
dependent destruction H0.
econstructor; eauto.
econstructor; eauto.
rewrite set_set_base.
solve [eauto].

dependent destruction H.
dependent destruction H0.
econstructor; eauto.
intros; edestruct H1; eauto.
constructor; auto.
intros; apply H4.
rewrite set_base_lookup_2; solve [auto
    ].

dependent destruction H.
dependent destruction H0.
solve [eauto].

dependent destruction H.
dependent destruction H0.
solve [eauto].

dependent destruction H.
dependent destruction H0.
solve [eauto].

dependent destruction H.
dependent destruction H0.
solve [eauto].

dependent destruction H0.
dependent destruction H0.
dependent destruction H1.
edestruct H0 as (ζ'&?&?&?); eauto.
dependent destruction Ξ .
apply accept_embed with (φ:=f) (ℓ:=ℓ)
    in H3; solve [eauto with datatypes
    ].

dependent destruction H.
dependent destruction H.
dependent destruction H3.
generalize (view_approx H5);
    inversion 1; subst; simpl in *.
edestruct H1; eauto.
dependent destruction Ξ ; simpl in *.
replace (Base (ext_map φx ℓ_1)) with (
    set_base (Base f) (ext_map φx ℓ_1))
    in H3; auto.
econstructor; eauto.
eapply accept_ext_env; solve [eauto].

dependent destruction H.
dependent destruction H.
generalize (view_approx H3);
    inversion 1; subst; simpl
    get_label in *.
edestruct H0; eauto.
rewrite H7 in *.
eapply accept_compat_1; eauto.
  rewrite ← H7; solve [auto].

  rewrite ← H7; solve [auto].

dependent destruction H.
dependent destruction H0.
generalize (view_approx H4);
    inversion 1; subst.
edestruct H1; eauto; simpl in *.
econstructor; eauto.
rewrite ← H8 in H0; simpl get_label in
    *.
rewrite H8 in *.
replace (t' ^ ℓ_2) with (relabel (embed v
    ) ℓ_2).
  eapply accept_compat_2; eauto.
  rewrite ← H8; solve [eauto].

  rewrite ← H8; solve [trivial].

dependent destruction H.
dependent destruction H; solve [auto].

dependent destruction H.
dependent destruction H; solve [auto].

dependent destruction H0.
dependent destruction H0_.
rewrite mforall_map in H4.
destruct (mforall_lookup _ H H4).
replace Ξ with (set_base Ξ (get_base
    Ξ )).
  eapply accept_embed; solve [eauto
      using protos with datatypes].

  solve [apply set_get_base].

dependent destruction H1.
dependent destruction H1_.
rewrite mforall_map in H5.
destruct (mforall_lookup _ H0 H5).
simpl.
constructor; auto.
intros.
eapply H1; eauto.
destruct (mforall_lookup _ H0 H5);
```

```
    simpl in *.
solve [eauto using protos].
solve [eauto using protos with
    datatypes].

dependent destruction H1.
econstructor; solve [eauto].

dependent destruction H.
dependent destruction H.
rewrite mforall_map in H.
econstructor; eauto.
apply mforall_map.
apply mforall_ext; solve [auto].

dependent destruction H.
dependent destruction H.
rewrite mforall_map in H.
econstructor; eauto.
apply mforall_map; solve [eauto using
    mforall_del].

dependent destruction H.
dependent destruction H0.
solve [constructor; eauto].

dependent destruction H.
dependent destruction H.
solve [constructor; eauto with
    datatypes].

dependent destruction H.
dependent destruction H.
dependent destruction H3.
constructor; eauto with datatypes.
solve [constructor; eauto with
    datatypes].

dependent destruction H.
dependent destruction H.
constructor; eauto with datatypes.
solve [constructor; eauto with
    datatypes].

dependent destruction H.
dependent destruction H0.
constructor; eauto with datatypes.
solve [econstructor; eauto with
    datatypes].

dependent destruction H.
dependent destruction H0.
solve [constructor; eauto with
    datatypes].

    dependent destruction H.
    dependent destruction H.
    constructor; eauto with datatypes.
    solve [constructor; eauto with
        datatypes].

    dependent destruction H.
    dependent destruction H0.
    solve [constructor; eauto with
        datatypes].

    dependent destruction H.
    dependent destruction H.
    solve [constructor; eauto with
        datatypes].

    dependent destruction H.
    dependent destruction H0.
    solve [constructor; eauto with
        datatypes].

    dependent destruction H.
    dependent destruction H.
    solve [constructor; eauto with
        datatypes].
Qed.

Lemma soundness: forall e n e', eval e n e'
    →
  forall (Ξ :absctxt n), accept Ξ e →
      accept Ξ e'.
Proof.
  destruct 1.
  induction C; intros; simpl in *; auto.
    solve [eauto using soundness_help].

    dependent destruction H0.
    rewrite mforall_app in H0.
    destruct H0.
    inversion H2; subst.
    destruct H5.
    econstructor; eauto.
    apply mforall_app.
    constructor; auto.
    constructor; auto.
    constructor; eauto.
    edestruct accept_plug as (Ξ '&K);
        eauto.
    solve [eauto with datatypes].

    dependent destruction H0.
    assert (get_label (plug C e) =
        get_label (plug C e')).
```

```
  assert (eval (plug C e) (S m) (plug C
      e')) by eauto using eval.
  eapply eval_label; eauto.
assert (approx v (TAbs s (plug C e'))).
  assert (eval (plug (CAbs l s C) e) (S
      m) (plug (CAbs l s C) e')).
    eauto using eval.
  eauto.
econstructor; eauto.
rewrite ← H3; solve [eauto].

dependent destruction H0.
edestruct accept_plug as (Ξ' &K);
    eauto.
generalize (plug_eval C K (red_eval H))
    .
constructor; solve [eauto with
    datatypes].

dependent destruction H0.
constructor; auto.
intros.
intros; edestruct H0; eauto.
edestruct accept_plug as (Ξ' &K);
    eauto.
generalize (plug_eval C K (red_eval H))
    .
solve [eauto with datatypes].

dependent destruction H0.
assert (get_label (plug C e) =
    get_label (plug C e')).
  assert (eval (plug C e) (S n0) (plug C
      e')) by eauto using eval.
  eapply eval_label; eauto.
assert (approx v (TBox (plug C e'))).
  assert (eval (plug (CBox l C) e) n0 (
      plug (CBox l C) e')).
    eauto using eval.
  eauto.
econstructor; eauto.
rewrite ← H3; solve [eauto].

dependent destruction H0.
edestruct accept_plug as (Ξ' &K);
    eauto.
generalize (plug_eval C K (red_eval H))
    .
constructor; eauto with datatypes.

dependent destruction H0.
edestruct accept_plug as (Ξ' &K);
    eauto.
generalize (plug_eval C K (red_eval H))
    .
constructor; solve [eauto with
    datatypes].

dependent destruction H0.
constructor; auto.
edestruct accept_plug as (Ξ' &K);
    eauto.
generalize (plug_eval C K (red_eval H))
    .
solve [eauto with datatypes].

dependent destruction H0.
constructor; auto.
edestruct accept_plug as (Ξ' &K);
    eauto.
generalize (plug_eval C K (red_eval H))
    ; solve [eauto with datatypes].

dependent destruction H0.
constructor; auto.
edestruct accept_plug as (Ξ' &K);
    eauto.
generalize (plug_eval C K (red_eval H))
    ; solve [eauto with datatypes].

dependent destruction H0.
edestruct accept_plug as (Ξ' &K);
    eauto.
generalize (plug_eval C K (red_eval H))
    .
constructor; solve [eauto with
    datatypes].

dependent destruction H0.
edestruct accept_plug as (Ξ' &K);
    eauto.
generalize (plug_eval C K (red_eval H))
    .
constructor; solve [eauto with
    datatypes].

dependent destruction H0.
edestruct accept_plug as (Ξ' &K);
    eauto.
generalize (plug_eval C K (red_eval H))
```

```
    .
constructor; solve [eauto with
    datatypes].

dependent destruction H0.
constructor; auto.
edestruct accept_plug as (Ξ '&K);
    eauto.
generalize (plug_eval C K (red_eval H))
    ; solve [eauto with datatypes].

dependent destruction H0.
constructor; auto.
edestruct accept_plug as (Ξ '&K);
    eauto.
generalize (plug_eval C K (red_eval H))
    ; solve [eauto with datatypes].

dependent destruction H0.
constructor; auto.
edestruct accept_plug as (Ξ '&K);
    eauto.
generalize (plug_eval C K (red_eval H))
    ; solve [eauto with datatypes].

dependent destruction H0.
edestruct accept_plug as (Ξ '&K);
    eauto.
generalize (plug_eval C K (red_eval H))
    .
econstructor; solve [eauto with
    datatypes].

dependent destruction H0.
constructor; auto.
edestruct accept_plug as (Ξ '&K);
    eauto.
generalize (plug_eval C K (red_eval H))
    .
solve [eauto with datatypes].
Qed.
End Analysis.
```

Bibliography

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 147–160. ACM, 1999.

[ACFH11] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for Ruby. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 459–472. ACM, 2011.

[AF09] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In Stephen Chong and David A. Naumann, editors, *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, pages 113–124. ACM, 2009.

[AF12] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 165–178. ACM, 2012.

[ALL96] Martín Abadi, Butler W. Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In Robert Harper and Richard L. Wexelblat, editors, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96), Philadelphia, Pennsylvania, May 24-26, 1996.*, pages 83–91. ACM, 1996.

[BCD+12] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. Secure multi-execution through static program transformation. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, volume 7273 of *Lecture Notes in Computer Science*, pages 186–202. Springer, 2012.

[BCF+14] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 87–100. ACM, 2014.

[Boy03] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[BPS+09] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 79–90. ACM, 2009.

[BT10] Martin Berger and Laurence Tratt. Program logics for homogeneous metaprogramming. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2010.

[CAYT11] Wontae Choi, Baris Aktemur, Kwangkeun Yi, and Makoto Tatsuta. Static analysis of multi-staged programs via unstaging translation. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 81–92. ACM, 2011.

[CF91] Robert Cartwright and Mike Fagan. Soft typing (with retrospective). In Kathryn S. McKinley, editor, *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*, pages 412–428. ACM, 1991.

[Cha11] Nathaniel Charlton. Reasoning about string-based runtime code generation. http://www.billiejoecharlton.com/wp-content/uploads/2012/02/rasbrcg.pdf, 2011. Retrieved from author's website on 2015–07–09.

[CLKD06] Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung-Goo Doh. A practical string analyzer by the widening approach. In Naoki Kobayashi, editor, *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings*, volume 4279 of *Lecture Notes in Computer Science*, pages 374–388. Springer, 2006.

[CMJL09] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 50–62. ACM, 2009.

[Cro06] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[DF11] Tim Disney and Cormac Flanagan. Gradual information flow typing. In *Workshop on Script-to-Program Evolution (STOP)*, 2011.

[DFHF14] Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. Sweeten your JavaScript: hygienic macros for ES5. In Andrew P. Black and Laurence Tratt, editors, *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 35–44. ACM, 2014.

[DHF10] Andreas Dewald, Thorsten Holz, and Felix C. Freiling. ADSandbox: sandboxing JavaScript to fight malicious websites. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 1859–1864. ACM, 2010.

[DKS11] Kyung-Goo Doh, Hyunha Kim, and David A. Schmidt. Abstract LR-parsing. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *Lecture Notes in Computer Science*, pages 90–109. Springer, 2011.

[DP01] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001.

[DP10] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 109–124. IEEE Computer Society, 2010.

[Fen74] J. S. Fenton. Memoryless subsystems. *Comput. J.*, 17(2):143–147, 1974.

[FT90] John Field and Tim Teitelbaum. Incremental reduction in the lambda calculus. In *LISP and Functional Programming*, pages 307–322, 1990.

[FT07] Fabien Fleutot and Laurence Tratt. Contrasting compile-time meta-programming in Metalua and Converge. In *Workshop on Dynamic Languages and Applications*, July 2007.

[FT13] Luminous Fennell and Peter Thiemann. Gradual security typing with references. In *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*, pages 224–239. IEEE Computer Society, 2013.

[GJ11] Thomas Given-Wilson and Barry Jay. A combinatory account of internal structure. *J. Symb. Log.*, 76(3):807–826, 2011.

[GL09] Salvatore Guarnieri and V. Benjamin Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. In Fabian Monrose, editor, *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 151–168. USENIX Association, 2009.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[GMO06] Jim Grundy, Thomas F. Melham, and John W. O'Leary. A reflective functional language for hardware design and theorem proving. *J. Funct. Program.*, 16(2):157–196, 2006.

[GSK10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 126–150. Springer, 2010.

[HM97] Nevin Heintze and David A. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*, pages 342–351. IEEE Computer Society, 1997.

[HM11] David Van Horn and Matthew Might. An analytic framework for JavaScript. *CoRR*, abs/1109.4467, 2011.

[HM13] David Van Horn and Harry G. Mairson. Deciding k-CFA is complete for EXPTIME. *CoRR*, abs/1311.5810, 2013.

[HS08] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, New York, NY, USA, 2 edition, 2008.

[HS12] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of JavaScript. In Stephen Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 3–18. IEEE Computer Society, 2012.

[HWS06] Marieke Huisman, Pratik Worah, and Kim Sunesen. A temporal logic characterisation of observational determinism. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*, page 3. IEEE Computer Society, 2006.

[Int99] ECMA International. ECMAScript language specification. Standard ECMA-262, 3rd edition, 1999.

[Int01] ECMA International. ECMAScript 3rd edition compact profile. Standard ECMA-327, 2001.

[IT12] Jun Inoue and Walid Taha. Reasoning about multi-stage programs. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 357–376. Springer, 2012.

[JCSH11] Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer. Information Flow Analysis for JavaScript. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, PLASTIC '11, pages 9–18, New York, NY, USA, 2011. ACM.

[JJLS10] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 270–283. ACM, 2010.

[JJM12] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In Mats Per Erik Heimdahl and Zhendong Su, editors, *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 34–44. ACM, 2012.

[JMM11] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 59–69. ACM, 2011.

[JMT09] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In Jens Palsberg and Zhendong Su, editors, *Static Analysis,*

16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009.

[KKS14] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Combinators for impure yet hygienic code generation. In Wei-Ngan Chin and Jurriaan Hage, editors, *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation, PEPM 2014, January 20-21, 2014, San Diego, California, USA*, pages 3–14. ACM, 2014.

[KLL+09] Taeksu Kim, Chunwoo Lee, Kiljoo Lee, Soohyun Baik, Chisu Wu, and Kwangkeun Yi. Test coverage metric for two-staged language with abstract interpretation. In Shahida Sulaiman and Noor Maizura Mohamad Noor, editors, *16th Asia-Pacific Software Engineering Conference, APSEC 2009, 1-3 December 2009, Batu Ferringhi, Penang, Malaysia*, pages 301–308. IEEE Computer Society, 2009.

[Kri12] Shriram Krishnamurthi. Semantics and analyses for JavaScript and the web. In Antoine Miné and David Schmidt, editors, *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, volume 7460 of *Lecture Notes in Computer Science*, page 4. Springer, 2012.

[KT13] Matthias Keil and Peter Thiemann. Type-based dependency analysis for JavaScript. In Prasad Naldurg and Nikhil Swamy, editors, *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013, Seattle, WA, USA, June 20, 2013*, pages 47–58. ACM, 2013.

[KYC06] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 257–268. ACM, 2006.

[LC08] V. Benjamin Livshits and Weidong Cui. Spectator: Detection and containment of JavaScript worms. In Rebecca Isaacs and Yuanyuan Zhou, editors,

2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings, pages 335–348. USENIX Association, 2008.

[Les13] Martin Mariusz Lester. Position paper: the science of boxing. In Prasad Naldurg and Nikhil Swamy, editors, *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013, Seattle, WA, USA, June 20, 2013*, pages 83–88. ACM, 2013.

[Les15] Martin Mariusz Lester. Verifying information flow and metaprogramming in dynamically typed languages: Mechanised Coq proofs and analysis source code supporting thesis. Oxford University Research Archive, 2015. Available from http://ora.ox.ac.uk.

[LM12] Shuying Liang and Matthew Might. Hash-flow taint analysis of higher-order programs. In Sergio Maffeis and Tamara Rezk, editors, *Proceedings of the 2012 Workshop on Programming Languages and Analysis for Security, PLAS 2012, Beijing, China, 15 June, 2012*, page 8. ACM, 2012.

[LOS13] Martin Lester, Luke Ong, and Max Schäfer. Information flow analysis for a dynamically typed language with staged metaprogramming. In *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*, pages 209–223. IEEE Computer Society, 2013.

[LP14] Steffen Lösch and Andrew M. Pitts. Denotational semantics with nominal Scott domains. *J. ACM*, 61(4):27:1–27:46, 2014.

[LZ05] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 158–170. ACM, 2005.

[MCC13] Tom Melham, Raphael Cohn, and Ian Childs. On the semantics of reflect as a basis for a reflective theorem prover. *CoRR*, abs/1309.5742, 2013.

[MD04] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[Mid12] Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10, 2012.

[MMT08] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In G. Ramalingam, editor, *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2008.

[MRMV12] Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. Eval begone!: semi-automated removal of eval from JavaScript programs. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 607–620. ACM, 2012.

[MSH13] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox. *CoRR*, abs/1311.4231, 2013.

[MT09] Sergio Maffeis and Ankur Taly. Language-based isolation of untrusted JavaScript. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 77–91. IEEE Computer Society, 2009.

[Mye09] A. Myers. A proof of noninterference for a while-language using small-step operational semantics. 2009.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[NPP08] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3), 2008.

[PC00] François Pottier and Sylvain Conchon. Information flow inference for free. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 46–57. ACM, 2000.

[Pie08] Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In George C. Necula and

Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 371–382. ACM, 2008.

[PS95] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Inf. Comput.*, 118(1):128–141, 1995.

[PS03] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, 2003.

[PSC09] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting JavaScript. In Wanqing Li, Willy Susilo, Udaya Kiran Tupakula, Reihaneh Safavi-Naini, and Vijay Varadharajan, editors, *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2009, Sydney, Australia, March 10-12, 2009*, pages 47–60. ACM, 2009.

[RHBV11] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - A large-scale study of the use of eval in JavaScript applications. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2011.

[RLBV10] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 1–12. ACM, 2010.

[RLZ09] Paruj Ratanaworabhan, V. Benjamin Livshits, and Benjamin G. Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In Fabian Monrose, editor, *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 169–186. USENIX Association, 2009.

[RS10] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pages 186–199. IEEE Computer Society, 2010.

[RS13] Arun Ranganathan and Jonas Sicking. File API. Last call WD, W3C, September 2013. http://www.w3.org/TR/2013/WD-FileAPI-20130912/.

[RWW96] A. W. Roscoe, Jim Woodcock, and L. Wulf. Non-interference through determinism. *Journal of Computer Security*, 4(1):27–54, 1996.

[SAH+10] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 513–528. IEEE Computer Society, 2010.

[SFR+14] Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin M. Bierman. Gradual typing embedded securely in JavaScript. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 425–438. ACM, 2014.

[Shi88] Olin Shivers. Control-flow analysis in Scheme. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 164–174. ACM, 1988.

[SM03a] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[SM03b] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2003.

[SR09] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009,*

Novosibirsk, Russia, June 15-19, 2009. Revised Papers, volume 5947 of *Lecture Notes in Computer Science*, pages 352–365. Springer, 2009.

[ST07] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2007.

[TA05] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.

[Tah00] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of MetaML is non-trivial (extended abstract). In Julia L. Lawall, editor, *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00), Boston, Massachusetts, USA, January 22-23, 2000*, pages 34–43. ACM, 2000.

[Ter08] Tachio Terauchi. A type system for observational determinism. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008*, pages 287–300. IEEE Computer Society, 2008.

[Tra05] Laurence Tratt. Compile-time meta-programming in a dynamically typed OO language. In Roel Wuyts, editor, *Proceedings of the 2005 Symposium on Dynamic Languages, DLS 2005, October 18, 2007, San Diego, California, USA*, pages 49–63. ACM, 2005.

[Tra09] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, 2009.

[TS00] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[TT07] ECMA International TC39-TG1. Proposed ECMAScript 4th edition - language overview. http://www.ecmascript.org/es4/spec/overview.pdf, 2007.

[VIS96] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[VS11] Dimitrios Vardoulakis and Olin Shivers. CFA2: a context-free approach to control-flow analysis. *Logical Methods in Computer Science*, 7(2), 2011.

[WCO00] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl - there's more than one way to do it (3. ed.)*. O'Reilly, 2000.

[WS07] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 32–41. ACM, 2007.

[YCIS07] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 237–249. ACM, 2007.

[Zda02] Steve Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.

[ZM03] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA*, page 29. IEEE Computer Society, 2003.