# IMPROVING CONTROL-FLOW ANALYSIS OF HIGHER-ORDER LANGUAGES

by

Steven Lyde

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2015

The University of Utah Graduate School

# STATEMENT OF DISSERTATION APPROVAL

The dissertation of Steven Lyde
has been approved by the following supervisory committee members:

| | |
|---|---|
| Matthew Might, Chair | 10/20/2015 |
| | Date Approved |
| Ganesh Gopalakrishnan, Member | 10/12/2015 |
| | Date Approved |
| Matthew Flatt, Member | 10/12/2015 |
| | Date Approved |
| Zvonimir Rakamarić, Member | 10/9/2015 |
| | Date Approved |
| Andrew W. Keep, Member | 10/20/2015 |
| | Date Approved |

and by Ross Whitaker, Chair of the School of Computing
and by David B. Kieda, Dean of The Graduate School.

# ABSTRACT

Control-flow analysis of higher-order languages is a difficult problem, yet an important one. It aids in enabling optimizations, improved reliability, and improved security of programs written in these languages.

This dissertation explores three techniques to improve the precision and speed of a small-step abstract interpreter: using a priority work list, environment unrolling, and strong function call. In an abstract interpreter, the interpreter is no longer deterministic and choices can be made in how the abstract state space is explored and trade-offs exist. A priority queue is one option. There are also many ways to abstract the concrete interpreter. Environment unrolling gives a slightly different approach than is usually taken, by holding off abstraction in order to gain precision, which can lead to a faster analysis. Strong function call is an approach to clean up some of the imprecision when making a function call that is introduced when abstractly interpreting a program.

An alternative approach to building an abstract interpreter to perform static analysis is through the use of constraint solving. Existing techniques to do this have been developed over the last several decades. This dissertation maps these constraints to three different problems, allowing control-flow analysis of higher-order languages to be solved with tools that are already mature and well developed. The control-flow problem is mapped to pointer analysis of first-order languages, SAT, and linear-algebra operations. These mappings allow for fast and parallel implementations of control-flow analysis of higher-order languages.

A recent development in the field of static analysis has been pushdown control-flow analysis, which is able to precisely match calls and returns, a weakness in the existing techniques. This dissertation also provides an encoding of pushdown control-flow analysis to linear-algebra operations. In the process, it demonstrates that under certain conditions (monovariance and flow insensitivity) that in terms of precision, a pushdown control-flow analysis is in fact equivalent to a direct style constraint-based formulation.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

In higher-order languages, it is not always clear from the syntax which functions are being called at which call sites. This is because a higher-order language allows functions as arguments, and also allows functions to return other functions.

Take for example the simple Racket program that implements Turner's tautology checker and invokes it two times [19]. The details of the example are not important, just the illustration it gives. The function `taut` takes a curried function `f` that represents a boolean expression. The function `taut` then checks if all possible assignments of either `#t` or `#f` to the parameters of `f` result in `f` evaluating to `#t`.

```
(define (taut f n)
  (if (= n 0) f
      (and (taut (f #t) (- n 1))
           (taut (f #f) (- n 1)))))

(define g (lambda (x)
            (lambda (y)
              (or (and x (not y))
                  (or (not x) y)))))

(define h (lambda (z) z))

(taut g 2)
(taut h 1)
```

Because functions are passed around as arguments, it might not be immediately clear which functions can flow to `f` and thus what functions are invoked when calling `f`. There are two call sites which recursively invoke `taut`, but the first argument is the result of

invoking `f` itself. In order to answer the question of which functions can flow to `f`, we need a control-flow analysis. A higher-order control-flow analysis conservatively bounds the set of functions that are applied at a higher-order call site.

A control-flow analysis is also needed for determining which methods are called in an object-oriented setting.

## 1.1 Thesis

Increases in speed and precision are feasible for control-flow analyses of higher-order languages—including pushdown analyses—via management of the abstract heap and by reduction to problems for which efficient implementations exist, such as pointer analysis, SAT, and linear-algebra operations.

## 1.2 Overview

A control-flow analysis determines the control-flow of a program. This is a difficult problem in higher-order languages, because data flow affects control flow and control flow affects data flow. To address this issue, much work has been done. Midtgaard surveys work on control-flow analysis of functional languages over the last 30 years, citing almost two hundred works [35].

A popular family of algorithms for solving control-flow of higher-order languages is $k$-CFA [49]. It is a family of algorithms where the chosen value of $k$ determines the precision of the analysis. A higher value of $k$ gives greater precision but at the cost of a greater runtime. In creating $k$-CFA, Shivers' original hope was to create an analysis that would allow programs written in higher-order languages to be as efficient as those written in C [49]. Might continued this line of work by solving the environment problem, enabling even more powerful optimizations when compiling higher-order languages [37].

However, $k$-CFA is an exponential algorithm [54], and even the most efficient polynomial formulation when $k = 0$, more commonly known as 0CFA, remains cubic [55]. Almost linear formulations exists, but maybe at the cost of too much precision [4].

Building further upon these works, Vardoulakis and Shivers developed a context-free approach to control flow analysis, CFA2, that allows for matching calls and returns more precisely [57]. Even though CFA2 results in an order of magnitude reduction in analysis time and an order of magnitude in precision, the algorithm remains exponential naively implemented, and remains polynomial with widening. Earl et al. developed their own approach to solving the problem of properly matching function calls and returns via abstracting the stack with a pushdown system [11].

This dissertation demonstrates that the cost of running these analyses can be brought down even further.

## 1.3 Contributions

First, this dissertation makes three contributions in carefully managing the abstract heap in a small-step abstract interpreter framework.

- **Prioritizing the Work List**

  In general, how the state space is explored for an abstract interpretation of higher-order languages is unimportant. The final result will be the same. However, when you use global store widening and subsumption testing, the way you explore the state space matters. Each transition contributes values to the store, but some transitions contribute more values than others. Thus, whether we explore the work list in a bread-first manner or depth-first manner can affect how many states we visit before we reach a fixed point.

  However, neither of these two search strategies may be optimal, but it is unlikely that the optimal search strategy can be known a priori. Nevertheless, cheap heuristics exist that will be closer to optimal than either breadth-first or depth-first searches [32].

- **Environment Unrolling**

  Generally, the state space of an abstract-machine for the lambda calculus is infinite because environments refer to closures and closures refer to environments [49]. Model checkers for imperative languages will often apply loop unrolling to make their state space finite in the presence of loops and recursion. Environments in higher-order languages can be handled in a similar fashion by putting a bound on the number of environments to which a given closure can transitively refer [30].

- **Strong Function Call**

  In an abstract interpretation, after a function call, we know which abstract closure was called and can deduce that any other values found at the same abstract address in the abstract store could not possibly exist in the corresponding concrete state [31]. Thus they can be removed from the abstract store without loss of soundness. However, this technique requires that we use abstract counting [39] to ensure that our abstract address is only abstracting one concrete address. Otherwise, the analysis would be unsound. This approach is similar to that of strong update [20].

Second, this dissertation makes three contributions in describing how to map a constraint based formulation of control-flow analysis into three unique problems.

- **Pointer Analysis**

  Mapping to a pointer analysis allowed us to achieve one of the main goals of this dissertation, parallelizing a control-flow analysis. Tools exist which both parallelize pointer analysis on the GPU and on multicore CPUs. Through this encoding, we can run our control-flow analysis in parallel on both of these platforms [27].

- **SAT**

  This dissertation demonstrates an encoding for a control-flow analysis of higher-order programs to SAT [29]. It shows that in some cases, modern SAT solvers, using this encoding, can perform better than an optimized solution.

- **Linear Algebra**

  Mapping the constraints to linear-algebra operations shows how it is possible to solve the constraints on the GPU, but also helps with an important result on how a pushdown control-flow analysis can be equivalent to a constraint-based formulation of the direct style lambda calculus.

Third, this dissertation makes the contribution of giving a linear algebra encoding of pushdown control-flow analysis [28] and shows how the previous encodings can also be used to perform the same analysis.

One of the original goals of this dissertation was to parallelize a pushdown control-flow analysis. However, the path to get there was a little circuitous. First, we map a control-flow analysis to a pointer analysis. Then, we demonstrate that the mapped control-flow analysis is actually equivalent to a pushdown control-flow analysis. To do this, this dissertation presents a linear encoding of pushdown control-flow analysis which makes it suitable to be run on a GPU, but also gives intuition on how to run the program. The style of this encoding takes a similar approach as that of EigenCFA [46].

## 1.4 Outline

This dissertation demonstrates the thesis as follows.

Chapter 2 gives a brief overview of abstract interpretation. It gives a brief overview of the language used for a majority of this dissertation, continuation-passing style lambda calculus. It gives a concrete semantics and then an abstract semantics. The subsequent sections then

present techniques for adjusting the abstract interpretation. It describes choices that can be made while exploring that abstract state space, which have consequence on the amount of memory used and the speed of the analysis. It also describes how to soundly change the abstract domains to achieve greater precision. It also describes how to soundly prune the search space when decisions are made in the abstract interpretation.

Chapter 3 reviews an alternative formulation to performing a control-flow analysis using constraints. It shows the constraints that are generated, how to solve them, and outlines how the traditional abstract small-step interpreter when made flow insensitive produces the same results. The chapter continues by showing how to map these constraints to a pointer analysis, how to map these constraints to a SAT solver, and finally gives a linear-algebra encoding of these constraints.

Chapter 4 outlines a pushdown control-flow analysis, gives a linear-algebra encoding, and outlines how the results it produces are equivalent to the results of a linear encoding of the direct style constraints.

Chapter 5 reviews work in parallelizing static analyses, while Chapter 6 concludes the dissertation.

# CHAPTER 2

# IMPROVING SMALL-STEP ABSTRACT INTERPRETERS

One approach to determining the control-flow of a program is through abstract interpretation [9]. One well-known approach by Van Horn and Might is to take the semantics of a programming language and abstract them using a systematic approach [56]. However, this produces inefficient abstract machines. Some work has been done to improve these machines [21], but this chapter explains three more techniques that can also be used to improve the abstract interpreters that are built using this approach.

## 2.1 Concrete Semantics

Continuation-passing style (CPS) lambda calculus is the higher-order language over which the forthcoming developments operate. In order to understand the ideas presented in this chapter, an understanding of continuation-passing style lambda calculus and abstract interpretation is needed. Brief descriptions of both will be given. After presenting CPS, we will first quickly recall what a concrete small-step semantics looks like for lambda calculus in continuation-passing style. We will then proceed to demonstrate how this can easily be changed into an abstract interpreter with only a few small changes [56]. The original formulation of a popular abstract interpretation framework, $k$-CFA, operates on CPS lambda calculus and this work operates on the same language.

### 2.1.1 Continuation-Passing Style

In CPS, all expression are either call sites, variables, or lambda terms, like in the original lambda calculus, but with the additional restrictions that the body of a lambda terms must be a call site and that the function and arguments at a call site must be atomic, meaning that they can only be a variable or lambda term. Unlike the pure lambda calculus, we allow lambda terms to have multiple arguments. This language form has been shown to be a suitable intermediate representation for compilers of higher-order languages [3]. It

also has the benefit that its semantics can be described in a single transition relation. CPS is similar to the untyped lambda calculus but with additional constraints: functions never return, all calls are tail calls; where a function would normally return, the current continuation is invoked on the return value; and when calling a function, the caller must supply a continuation procedure.

The grammar for CPS lambda calculus is as follows.

$$call \in \mathsf{Call} ::= (f\ æ_1 \dots æ_n)$$
$$f, æ \in \mathsf{AExp} ::= v \mid lam$$
$$lam \in \mathsf{Lam} ::= (\lambda\ (v_1 \dots v_n)\ call)$$
$$v \in \mathsf{Var} \text{ is a set of identifiers}$$

We will now describe a small-step operational semantics using an abstract machine that can be used to evaluate a program. This machine will be very similar to the CESK machine of Felleisen [13]. However, it does not have a continuation component, because the continuations are explicit in the expressions. This is also slightly a nonstandard state space because we have environments mapping to addresses rather than values. This is to facilitate the abstraction of the machine using the Abstracting Abstract Machines approach [56].

We use a CES style abstract machine and provide a transition relation $(\Rightarrow) \subseteq \Sigma \times \Sigma$. The state space of this abstract machine is as follows. A control state contains a control expression, environment, and store.

$$\varsigma \in \Sigma = \mathsf{Call} \times \mathit{Env} \times \mathit{Store}$$
$$\rho \in \mathit{Env} = \mathsf{Var} \rightharpoonup \mathit{Addr}$$
$$\mathit{clo} \in \mathit{Clo} = \mathsf{Lam} \times \mathit{Env}$$
$$\sigma \in \mathit{Store} = \mathit{Addr} \rightharpoonup \mathit{Clo}$$
$$a \in \mathit{Addr} \text{ is an infinite set}$$

In order to evaluate atomic expressions, we introduce an auxiliary function, $\mathcal{A} : \mathsf{Atom} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Clo}$. In the case of a variable, it looks up the address of the variable in the environment and then looks up the value at that address in the store. In the case of a lambda term, it produces a closure by closing the lambda term with the current environment.

$$\mathcal{A}(v, \rho, \sigma) = \sigma(\rho(v))$$
$$\mathcal{A}(lam, \rho, \sigma) = (lam, \rho)$$

In defining the transition relation $(\Rightarrow)$, we find one of the main benefits of using CPS. The transition relation can be defined with a single rule. It starts by atomically evaluating

the function. It proceeds by allocating a new address for each argument. In the concrete semantics, a unique address is used that will never be used again. It then extends the environment of the closure and binds the values of the arguments to those addresses in the store.

$$\overbrace{(\llbracket (f\ æ_1 \ldots æ_n) \rrbracket, \rho, \sigma)}^{\varsigma} \Rightarrow (call, \rho'', \sigma'), \text{ where}$$
$$(\llbracket (\lambda\ (v_1 \ldots v_n)\ call) \rrbracket, \rho') = \mathcal{A}(f, \rho, \sigma)$$
$$\rho'' = \rho'[v_i \mapsto a_i]$$
$$\sigma' = \sigma[a_i \mapsto \mathcal{A}(æ_i, \rho, \sigma)]$$
$$a_i = alloc(v_i, \varsigma)$$

Given a program, we must be able to inject it into an initial state. Using $\mathcal{I} : \mathsf{Call} \to \Sigma$ we pair a program with an empty environment and empty store.

$$\mathcal{I}(call) = (call, [], [])$$

Given the initial state, the program is executed by generating successor states using the transition relation $(\Rightarrow) \subseteq \Sigma \times \Sigma$. The halt continuation can be simulated by having a free variable in the program. The transition relation will not have any closure bound to the free variable and thus cannot generate a successor state. Execution terminates when the halt continuation is applied. The meaning of the program is whatever value gets passed to the halt continuation.

## 2.2 Abstract Semantics

We will now explore how we can take this concrete semantics and make it abstract. Our abstract semantics will be guaranteed to terminate given any program. We begin by first abstracting the state space. Looking at the original concrete state space, the source of unboundedness is that addresses are unbounded. However, we can abstract the state space by making the number of addresses finite [56].

Besides this small change, the abstract state space looks very similar to the concrete state space, with the notable exception that the addresses are now finite and stores now map addresses to a set of abstract closures.

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \\
\hat{\rho} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \widehat{Addr} \\
\widehat{clo} \in \widehat{Clo} &= \mathsf{Lam} \times \widehat{Env} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \mathcal{P}\left(\widehat{Clo}\right) \\
\hat{a} \in \widehat{Addr} &\text{ is a finite set}
\end{aligned}$$

However, having a finite set of addresses means that in the abstract interpretation some addresses will be reused. This means that the store must be able to handle having more than one value, so we now map to a set of closures rather than a single closure. These sets cannot grow arbitrarily large because there are only a finite number of closures. This is why the indirection of the store was introduced. Having environments point to values rather than addresses would introduce structural recursion, because values contain environments. However, with the introduction of the store the cycle is broken [56].

The abstract transition relation $(\leadsto) \subseteq \hat{\Sigma} \times \hat{\Sigma}$ changes slightly from the concrete one in order to handle multiple closures. It now joins ($\sqcup$) values in the store. This means it takes the union of the set of closures that previously existed at that address and the set of closures that are being added and store the union at the address.

$$\begin{aligned}
\overbrace{(\llbracket (f\ æ_1 \ldots æ_n) \rrbracket, \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}} &\leadsto (call, \hat{\rho}'', \hat{\sigma}), \text{ where} \\
(\llbracket (\lambda\ (v_1 \ldots v_n)\ call) \rrbracket, \hat{\rho}') &\in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \\
\hat{\rho}'' &= \hat{\rho}'[v_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma})] \\
\hat{a}_i &= \widehat{alloc}(v_i, \hat{\varsigma})
\end{aligned}$$

We also have an abstract atomic evaluator that performs the same operations as its concrete counterpart $\hat{\mathcal{A}} : \mathsf{Atom} \times \widehat{Env} \times \widehat{Store} \rightharpoonup \mathcal{P}\left(\widehat{Clo}\right)$. The difference being that it now returns a set of abstract closures, rather than a single value.

$$\begin{aligned}
\hat{\mathcal{A}}(v, \hat{\rho}, \hat{\sigma}) &= \hat{\sigma}(\hat{\rho}(v)) \\
\hat{\mathcal{A}}(lam, \hat{\rho}, \hat{\sigma}) &= \{(lam, \hat{\rho})\}
\end{aligned}$$

The abstract transition relation is also very similar to its concrete counterpart. The main difference lies in that the atomic evaluator may return multiple values. This results in branching in the abstract transition graph.

To perform the analysis, we must then compute all the reachable states using the abstract transition relation $(\hat{\rightsquigarrow}) \subseteq \hat{\Sigma} \times \hat{\Sigma}$, generating successor states until a fixed point is reached.

$$\left\{\hat{\varsigma} : \hat{\mathcal{I}}(\mathit{call}) \hat{\rightsquigarrow}^* \hat{\varsigma}\right\}$$

### 2.2.1 Context Sensitivity

We can recover the original $k$-CFA analysis given these semantics by adding a time component to a state.

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Time}} \\
\hat{\rho} \in \widehat{\mathit{Env}} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\
\hat{\sigma} \in \widehat{\mathit{Store}} &= \widehat{\mathit{Addr}} \to \mathcal{P}\left(\widehat{\mathit{Clo}}\right) \\
\widehat{\mathit{clo}} \in \widehat{\mathit{Clo}} &= \mathsf{Lam} \times \widehat{\mathit{Env}} \\
\hat{a} \in \widehat{\mathit{Addr}} &= \mathsf{Var} \times \widehat{\mathit{Time}} \\
\hat{t} \in \widehat{\mathit{Time}} &= \mathsf{Call}^k
\end{aligned}$$

The abstract allocation function $\widehat{\mathit{alloc}} : \mathsf{Var} \times \widehat{\mathit{aState}} \to \widehat{\mathit{Addr}}$ is implemented as pairing the variable with the time component.

$$\widehat{\mathit{alloc}}(v, (\mathit{call}, \hat{\rho}, \hat{\sigma}, t)) = (v, t)$$

Also, on every transition, the function $\widehat{\mathit{tick}} : \Sigma \to \widehat{\mathit{Time}}$ is called and the time is advanced. However, it just takes the last $k$ call sites.

$$\widehat{\mathit{tick}}(\mathit{call}, \hat{\rho}, \hat{\sigma}, \hat{t}) = \overbrace{\mathit{call} : \hat{t}}^{\text{first } k \text{ values}}$$

We still need to inject the program into an initial abstract state $\hat{\mathcal{I}} : \mathsf{Call} \to \hat{\Sigma}$, but it is still paired with an empty environment, empty store, and empty time.

$$\hat{\mathcal{I}}(\mathit{call}) = (\mathit{call}, [], [], ())$$

## 2.3 Priority State Exploration

Choices exist when exploring states in a small-step abstract interpreter. When generating the abstract transition graph while computing $k$-CFA, the order in which we generate successor states is not important. However, if we are using store widening, the order in which we generate successor states matters because some states will help us jump to the minimum fixed point faster than others. The order in which states are explored is controlled by the

work list. The states can be explored in depth-first or breadth-first fashion. However, these are not the only options available. We can also use a priority queue to intelligently explore states which will help us reach a fixed point faster than either of these two approaches. In this section, we evaluate the different options that exist for a work list.

While control-flow analysis of higher-order languages is complex, the machinery underneath is actually quite simple. However, in these simple mathematics, there are several nuances that can affect the precision of the analysis. In the past, the primary focus has been the allocation function, which controls the address of the variables we are binding [17]. With this seemingly simple function, the polyvariance, complexity, and precision of the analysis is controlled. However, this is not the only source of nuance in a small-step abstract framework.

This section discusses the particulars of a possible implementation of $k$-CFA. The abstract interpreter from Section 2.1.1 can be made to run quickly by using global widening in an algorithm known as the time-stamp algorithm [49].

Once an understanding of the time-stamp algorithm is attained, we can dive into the meat of this section. It will be shown that it is important how exactly we handle the work list in the algorithm. The order in which we visit states and generate successor states matter.

The main contribution of this section is to point out and demonstrate the idea that the order of exploration matters when iterating over the work list.

Our second contribution is to demonstrate that using a priority queue for the work list can increase the speed of the analysis and also decrease the amount of memory required for the analysis. We demonstrate with empirical evidence the efficacy of this idea, even though the gains might not be substantial.

### 2.3.1 Implementing $k$-CFA

The simplest way to compute $k$-CFA is to construct the set of all reachable states over the transition relation, starting at the initial state. Any graph-searching algorithm is sufficient for finding this set. This will give us the desired result because every state in the concrete execution has an approximation in the set of abstract states generated by $k$-CFA. This means that any behavior that occurs in the concrete execution will be captured by the abstract execution.

Shivers devised two techniques for more quickly computing the set of reachable states: the aggressive-cutoff algorithm and the time-stamp algorithm [49].

We will give a short description of these two algorithms shortly and then will describe the algorithm in more detail.

### 2.3.2 The Aggressive-Cutoff Algorithm

While exploring the state space and generating the abstract transition relation, we only ever add information, we never take any away. We can exploit this monotonicity while exploring the state space. If a state we are about to explore is weaker than ($\sqsubseteq$) a state that we have already visited, we know that we have already captured the behavior of that state and do not need to generate its successor states again. This is the essence of the aggressive-cutoff algorithm.

### 2.3.3 The Time-Stamp Algorithm

The time-stamp algorithm is a form of the aggressive-cutoff algorithm. In the time-stamp algorithm, we modify the state-space search by joining the store of the state just pulled from the work list with the least upper bound of all the stores seen so far.

States contain a large environment and store that to compare requires a deep traversal. These states are sizable structures. To combat this issue, we perform the following steps.

We keep around a single-threaded store that we update after each transition. The store grows monotonically, so this is safe to do. We might add additional values that would not occur in the concrete execution, but this is always sound. Whenever we update the store with a new value, we increment a time-stamp. Then in our states, we no longer keep a reference to the store but to a time-stamp. A time-stamp with a lesser value is weaker than a time-stamp of a greater value. Thus, we can do subsumption testing based on the value of the time-stamp. The larger time-stamp approximates the smaller time-stamp.

This technique implements the aggressive cutoff algorithm while at the same time lowering the storage overhead. The original implementation of the time-stamp algorithm [49] showed that it did not cost too much precision.

### 2.3.4 Detailed Algorithm

Putting together the two above techniques, exploiting configuration monotonicity for early termination and configuration-widening, leads to an algorithm for computing Shivers' original $k$-CFA.

This algorithm in Figure 2.1 is taken directly from Might, but adapted slightly to fit the notation of this chapter [37].

| | |
|---|---|
| $\hat{S} \leftarrow \bot$ | Seen time-stamps, $\mathsf{Call} \times \widehat{Env} \times \widehat{Time} \to \mathbb{N}$. |
| $\hat{\Sigma}_{\text{todo}} \leftarrow \left\{\hat{\mathcal{I}}(pr)\right\}$ | The work list. |
| $\hat{\sigma}^* \leftarrow \bot$ | The global store. |
| $n^* = 1$ | The generation of the global store. |
| **procedure** SEARCH() | |
| **if** $\hat{\Sigma}_{\text{todo}} = \emptyset$ | |
| **return** | |
| **remove** $\hat{\varsigma} \in \hat{\Sigma}_{\text{todo}}$ | |
| $(call, \hat{\rho}, \hat{\sigma}, \hat{t}) \leftarrow \hat{\varsigma}$ | |
| $n \leftarrow \hat{S}[call, \hat{\rho}, \hat{t}]$ | The latest generation seen with this context. |
| **if** $n \geq n^*$ | |
| **return** SEARCH() | Done—by monotonicity of $\leadsto$. |
| $\hat{\varsigma} \leftarrow (call, \hat{\rho}, \hat{\sigma} \sqcup \hat{\sigma}^*, \hat{t})$ | Install the widened store. |
| $\hat{\Sigma}_{\text{next}} \leftarrow \{\hat{\varsigma}' : \hat{\varsigma} \leadsto \hat{\varsigma}'\}$ | Explore successors. |
| $\hat{S}[call, \hat{\rho}, \hat{t}] \leftarrow n^*$ | Mark the current generation of the store as *seen*. |
| $\hat{\sigma}_{\text{next}} \leftarrow \bigsqcup \left\{\hat{\sigma} : (call, \hat{\rho}, \hat{\sigma}, \hat{t}) \in \hat{\Sigma}_{\text{next}}\right\}$ | Check each successor for changes. |
| **if** $\hat{\sigma}_{\text{next}} \sqsupset \hat{\sigma}^*$ | |
| $n^* \leftarrow n^* + 1$ | Bump up the generation of the global store. |
| $\hat{\sigma}^* \leftarrow \hat{\sigma}_{\text{next}}$ | Widen the global store. |
| $\hat{\Sigma}_{\text{todo}} \leftarrow \hat{\Sigma}_{\text{todo}} \cup \hat{\Sigma}_{\text{next}}$ | |
| **return** SEARCH() | |

**Figure 2.1**: State-space search algorithm using the time-stamp algorithm for computing $k$-CFA: SEARCH

Using a side-effected global table, $\hat{S}$, we map the latest evaluation context $(call, \hat{\rho}, \hat{t})$ to the latest generation of the store that has been explored with that context:

$$\hat{S} : \mathsf{Call} \times \widehat{Env} \times \widehat{Time} \to \mathbb{N}$$

During the search, if the current state was explored with a generation of the store that is greater than or equal to the current generation of the global store, then that branch of the search has terminated. The monotonicity of the abstract transition relation guarantees that the behavior has already been approximated.

Otherwise, we widen the store of the state with the global store and generate successors, updating $\hat{S}$ to reflect that we have explored it with the current generation of the global store.

From the successor states, we see if they have contributed any changes to the global store. If they have, we widen the global store and bump its generation.

### 2.3.5 The Work List

Traditionally, when executing a work list algorithm, the order in which we explore states is not important. However, when we use the time-stamp algorithm, since each state can possibly contribute different values to the global store, the order does have an effect on the number of states that are explored. It has this effect because the quicker we can reach a fixed point of our global store, the quicker we can stop exploring states.

The work list is generally implemented using a list, with new states being appended to the front. This results in a depth-first search. However, we can explore these states in any order we wish. In the next section, we will discuss possible ordering schemes on this list, where we examine the contents of states in order try and guess which ones will help us reach the fixed point of the global store the quickest.

### 2.3.6 Priority Queue

There are four components to a state which we can use to guess if it will help us climb the lattice quicker: the expression, the environment, the store, and the time stamp.

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Time}$$

In addition to the properties of these components, we can also take advantage of temporal properties that arise during the execution of the abstract analysis.

We will now explore what properties of each of these components we could possibly use to help us order them in our work list. The abbreviations in the parenthesis are used in the evaluation section.

#### 2.3.6.1 Expression

These are possible priority schemes based on the expression component of a state.

- The type of the expression. If our language was richer and allowed for more language forms such as `if` or `set!`, we could prioritize a given form over another (CTP).
- The number of subexpressions. It might be the case that more subexpressions means that more values will be bound; thus, we should prioritize larger expressions over smaller ones (CSZ).
- Where the expression appears in the program. We could explore expressions that appear deeper in the program first (CDL) or we could take more of a breadth-first approach and try to visit expressions that appear higher in our program first (CBL).
- The number of times we have visited an expression. When we come across an expression in the course of the abstract interpretation, we might want to prioritize states with expressions that we have already seen or vice versa (CFQ).
- Top-level function or inner function. If the lambda term we are invoking originally was a top-level function in our program, it might be beneficial to explore inner functions before exploring other top-level functions.
- Prefer user lambdas over continuation lambdas. When converting to continuation-passing style, there are two types of lambda terms: user lambdas and continuation lambdas. Returns get converted into invocations of continuation lambdas (CCR).
- The size of the continuation. This might give a rough approximation of how much computation is left to do for a given state.

#### 2.3.6.2 Environment

These are possible priority schemes based on the environment component of a state.

- The environment size. This is another way to give a comparable value to an expression. A larger environment might signify that we will bind more values (ESZ).
- The flow set size of every address in the environment. How big the flow sets are determine partially how big the flow sets are that we will be binding to values. It stands to reason the larger these flow sets, the more values we will bind quickly (EFS).

#### 2.3.6.3 Store

These are possible priority schemes based on the store component of a state.

- The flow set size of the function we are applying. This determines how many successor states we will have. If we prefer states that will generate more states, we might be able to subsequently pick the best of those.
- The flow set size of the arguments. Given that we want to reach the fixed point as quick as possible and that adding entries in the store is what gets us there, the more values we bind the better (SAS).
- Number of successor states. If our language supported an if form, we could ask the question of whether we will be exploring one branch, both branches, or neither branch (SBF).
- The flow set size of the values we are binding. If our language supported `set!`, we might want to consider the flow set size of the variable we are binding or the flow set size of the value we are binding.
- Global store generation. The generation of the store is a metric of the size of the store. We might prefer to explore states that already have a larger store.

#### 2.3.6.4 Time

These are possible priority schemes based on the time component of a state.

- The number of times we have seen a given time. We will often see the same time stamp in the course of an abstract interpretation. We could prefer states with calling contexts that we have already seen or put a preference on new ones (TFQ).
- The value of the time. We could prefer longer contexts or shorter contexts. For contexts of the same length, we could prefer ones that appear earlier or later in the program we are analyzing (TVL).

### 2.3.7 Evaluation

To evaluate our idea, we took the implementation from Might et al. [40] which uses the time-stamp algorithm. We adapted it so it would use a priority queue for its work list,

Observing the run times from the original paper, you will note that the benchmarks run significantly faster. Updating the code to run on the latest version of Scala results in a 2x

speedup. We also identified a bug where successor states were being added multiple times to the work list. Removing these duplicate entries also resulted in a 2x speedup.

We are also running on better hardware, but given that we reran the original implementation on the newer hardware as a point of reference, this should not be a concern.

The abbreviations and descriptions for the priority schemes we evaluated in our implementation can be found in the previous section. We used the same benchmarks analyzed by the original implementation [40]. The first two benchmarks, eta and map, test common functional idioms; sat is a back-tracking SAT-solver; regex is a regular expression matcher based on derivatives; scm2java is a Scheme compiler that targets Java; interp is a meta-circular Scheme interpreter; scm2c is a Scheme compiler that targets C.

Tables 2.1 and 2.2 compare the number of states that were generated for each benchmark. In some cases, we can see that we generate only a fifth of the states as compared to the original implementation.

Tables 2.3 and 2.4 compare the runtimes of the varying strategies. In the best case, we were able to achieve a 1.5x speedup.

Although no specific strategy is best for all benchmarks, the strategy CFQ tends to do well both in terms of reducing the number of states and decreasing the runtime. For a control-flow analysis that needs to use low memory and run fast, using one of the strategies that performs better than the baseline BFS and DFS strategies is worth considering.

All benchmarks were run with a $k$ of zero or one. Every strategy produced the same store for its final result.

This work is similar to that of Bourdoncle [7]. It discusses choosing the optimal ordering of equations when using widening and narrowing. His work is different in that his abstract interpretation is set up on creating a system of equations and for each control point and finding the least fixed point for these system of equations. The abstract interpretation using operational semantics uses a single transfer function and we are more concerned with the implementation details of this transfer function. His work also works of an infinite lattice while our lattice is finite and we are using widening for the purposes of speeding up the analysis and making it tractable.

### 2.3.8 Conclusion

In this section, we have demonstrated that how states are processed is important when computing $k$-CFA. We have described that there is a difference between doing a depth-first vs. breadth-first search. We have also demonstrated that using a specific type of queue can play an important role in limiting the number of states explored.

**Table 2.1**: Number of states generated for $k = 0$.

| | eta | map | sat | regex | scm2java | interp | scm2c |
|---|---|---|---|---|---|---|---|
| BFS | 54 | 230 | 488 | 3692 | 7888 | 651 | 38899 |
| DFS | 66 | 186 | 293 | 2252 | 2595 | 657 | 21195 |
| CTP | 53 | 223 | 284 | 1831 | 2394 | 653 | 13663 |
| CSZ | 54 | 192 | 373 | 2063 | 2933 | 653 | 14510 |
| CDL | 54 | **141** | 343 | 2630 | 4110 | 653 | 19618 |
| CBL | 54 | 230 | 234 | 2205 | 2848 | **648** | 25808 |
| CFQ | 56 | 166 | **223** | **1271** | **1718** | 657 | **7660** |
| CCR | 53 | 272 | 520 | 2292 | 3557 | 656 | 14327 |
| ESZ | **48** | 178 | 296 | 2248 | 2966 | 649 | 25845 |
| EFS | 48 | 178 | 296 | 2248 | 2966 | 649 | 25845 |
| SAS | 53 | 247 | 373 | 1743 | 3414 | 655 | 13337 |
| SBF | 61 | 223 | 382 | 1645 | 3467 | 653 | 10288 |

**Table 2.2**: Number of states generated for $k = 1$.

| | eta | map | sat | regex | scm2java | interp | scm2c |
|---|---|---|---|---|---|---|---|
| BFS | 53 | 361 | 8696 | 12965 | 10001 | 635 | 157396 |
| DFS | **38** | 360 | 17216 | 11328 | 13397 | 635 | 130302 |
| CTP | 49 | 341 | 8831 | 6560 | 4080 | 635 | 94931 |
| CSZ | 53 | 344 | 6749 | 8467 | **3828** | 635 | 98366 |
| CDL | 53 | **260** | **4527** | 6039 | 4054 | 635 | 99471 |
| CBL | 44 | 343 | 7575 | **4369** | 4448 | 635 | 99333 |
| CFQ | 53 | 310 | 5819 | 7207 | 7651 | 635 | 96594 |
| CCR | 53 | 464 | 9855 | 8230 | 5444 | 635 | 98609 |
| ESZ | 51 | 342 | 6644 | 8932 | 4119 | 635 | 118390 |
| EFS | 51 | 342 | 6644 | 8932 | 4119 | 635 | 118390 |
| SAS | 49 | 442 | 8847 | 5661 | 5762 | 635 | **84808** |
| SBF | 47 | 456 | 9600 | 8283 | 6136 | 635 | 86390 |

**Table 2.3**: Time in milliseconds for each benchmark for $k = 0$.

| | eta | map | sat | regex | scm2java | interp | scm2c |
|---|---|---|---|---|---|---|---|
| BFS | 73 | 217 | 293 | 889 | 1214 | 828 | 3296 |
| DFS | 75 | 195 | 233 | 704 | 790 | 815 | 2617 |
| CTP | 66 | 217 | 230 | 622 | 777 | 818 | 2338 |
| CSZ | 69 | 202 | 264 | 692 | 850 | 832 | 2376 |
| CDL | 66 | **167** | 249 | 781 | 936 | 831 | 2344 |
| CBL | 80 | 229 | 216 | 691 | 831 | 867 | 2737 |
| CFQ | 68 | 182 | **194** | **536** | **708** | 828 | **1824** |
| CCR | 67 | 236 | 295 | 707 | 880 | **812** | 2366 |
| ESZ | **65** | 188 | 232 | 729 | 831 | 815 | 2754 |
| EFS | 71 | 200 | 238 | 729 | 881 | 878 | 3149 |
| SAS | 67 | 238 | 267 | 633 | 903 | 822 | 2319 |
| SBF | 74 | 221 | 267 | 618 | 901 | 826 | 2170 |

**Table 2.4**: Time in milliseconds for each benchmark for $k = 1$.

| | eta | map | sat | regex | scm2java | interp | scm2c |
|---|---|---|---|---|---|---|---|
| BFS | 65 | 274 | 1441 | 1587 | 1341 | 830 | 8878 |
| DFS | **56** | 273 | 1820 | 1478 | 1514 | **819** | 6413 |
| CTP | 61 | 272 | 1436 | 1122 | 940 | 825 | 6253 |
| CSZ | 65 | 271 | 1333 | 1354 | **921** | 827 | 6297 |
| CDL | 66 | **232** | **1139** | 1114 | 943 | 849 | 6247 |
| CBL | 65 | 274 | 1433 | **929** | 968 | 879 | 6603 |
| CFQ | 63 | 256 | 1184 | 1201 | 1227 | 834 | 5967 |
| CCR | 63 | 308 | 1533 | 1323 | 1048 | 821 | 6068 |
| ESZ | 62 | 269 | 1284 | 1348 | 931 | 831 | 7469 |
| EFS | 69 | 276 | 1355 | 1464 | 1006 | 895 | 9806 |
| SAS | 63 | 310 | 1537 | 1130 | 1127 | 834 | **5338** |
| SBF | 62 | 308 | 1528 | 1329 | 1116 | 822 | 5814 |

## 2.4 Environment Unrolling

We propose a new way of thinking about abstract interpretation with a method we term environment unrolling. Model checkers for imperative languages will often apply loop unrolling to make their state space finite in the presence of loops and recursion. We propose handling environments in a similar fashion by putting a bound on the number of environments to which a given closure can transitively refer. We present how this idea relates to a normal model of abstract interpretation, give a general overview of its soundness proof in regards to a concrete semantics, and show empirical results demonstrating the effectiveness of our approach.

### 2.4.1 Introduction

In general, the state space of an abstract machine for a lambda calculus is infinite because environments refer to closures and closures refer to environments [49]. In Abstracting Abstract Machines (AAM), a systematic recipe for breaking this cycle is given [56]. Bindings are store allocated and the number of addresses in the store is made finite.

We propose a slightly different perspective on how to make the state space finite. Model checkers for imperative language wills often apply loop unrolling to make their state space finite in the presence of loops and recursion. We propose handling environments in a fashion similar to how they handle loops, by putting a bound on the number of environments to which a given closure can transitively refer. As will be demonstrated, this provides a benefit in both speed and precision in an abstract interpretation.

To illustrate the basic idea, we provide a simple example using the following program.

```
(define (fact n)
  (if (zero? n) 1 (* n (fact (- n 1)))))
(fact 5)
```

In a traditional 0CFA, after the second recursive call, the environment would have the binding $[n \mapsto a]$, while the store would have the binding $[a \mapsto \{4, 5\}]$. However, if the environment were concrete, the environment would contain the binding $[n \mapsto 4]$ and the store would be empty.

### 2.4.2 Abstract Semantics

The abstract state space remains largely unchanged except for environments and closures.

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \\
\hat{\rho} \in \widehat{Env} &= \widehat{Env}_x + \widehat{Env}_0 \\
\widehat{Env}_x &= \mathsf{Var} \rightharpoonup \mathcal{P}\left(\widehat{Clo}\right) \\
\widehat{Env}_0 &= \mathsf{Var} \rightharpoonup \widehat{Addr} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \mathcal{P}\left(\widehat{Clo}\right) \\
\widehat{clo} \in \widehat{Clo} &= \mathsf{Lam} \times \widehat{Env} \\
\hat{a} \in \widehat{Addr} &\text{ is a finite set}
\end{aligned}$$

Environments are abstracted in a nonstandard way. There are now two types of environments. The first type of environment has a depth which places a bound on how many times we can extend the environment before using store allocation in the traditional fashion. We also have the environment from the original abstract semantics in Section 2.2, which maps variables to a set of addresses.

The subscript of the environment and the closure give its depth. The semantics maintain the invariant that the environment depth must be greater than the depth of any closure it binds.

$$\text{Given } \hat{\rho} \in \widehat{Env}_x \text{ then } \forall y \in \left\{ y : \hat{\rho}' \in \widehat{Env}_y, (lam, \hat{\rho}') \in \hat{d}, \hat{d} \in range(\hat{\rho}) \right\} . x > y$$

In fact, it will be one greater the maximum depth of a closure it binds.

$$\text{Given } \hat{\rho} \in \widehat{Env}_x \text{ then } x = 1 + \min \left\{ y : \hat{\rho}' \in \widehat{Env}_y, (lam, \hat{\rho}') \in \hat{d}, \hat{d} \in range(\hat{\rho}) \right\}$$

However, if the environment does not bind any values, it can be any depth.

The semantics also require a way to extract the depth from the environment. The environment is taken from the semantic category to which it belongs. In an implementation, this could be encoded in the type of environment and a new type of environment would be instantiated on environment extension.

We need to inject a program into an initial configuration. When this is done, the limit on the amount of unrolling $d$ is chosen.

$$\hat{c}_0 = \mathcal{I}(e) = (e, \hat{\rho}, [], \langle\rangle), \text{ where } \hat{\rho} = [] \in \widehat{Env}_d$$

The atomic evaluator is also slightly different than usual. It must take into account the depth of the environment. Lambda terms are handled in the traditional way. If we have a variable, we have to be cognizant of what type of environment we have. If we have an

$\widehat{Env}_0$, we evaluate in the standard way. If we have the other type of environment $\widehat{Env}_x$, we look up the value directly in the environment.

$$\hat{\mathcal{A}}(lam, \hat{\rho}, \hat{\sigma}) = \{(lam, \hat{\rho})\}$$
$$\hat{\mathcal{A}}(v, \hat{\rho}, \hat{\sigma}) = \begin{cases} \hat{\sigma}(\hat{\rho}(v)) & \hat{\rho} \in \widehat{Env}_0 \\ \hat{\rho}(v) & \hat{\rho} \in \widehat{Env}_x \end{cases}$$

The transition relation is also slightly different. It has three different scenarios it must take into account, depending on the values it is binding and the environment it is extending. The depth of the environment of the applied closure is $d_f$. The lowest depth of any abstract closure that is being applied is $d_y$. Each scenario updates generates $\hat{\rho}''$ and $\hat{\sigma}'$ slightly differently.

$$\overbrace{(\llbracket (f\ æ_1 \ldots æ_n) \rrbracket, \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}} \rightsquigarrow (call, \hat{\rho}'', \hat{\sigma}'), \text{ where}$$
$$(\llbracket (\lambda\ (v_1 \ldots v_n)\ call) \rrbracket, \hat{\rho}') \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$
$$d_f = x \text{ where } \hat{\rho}' \in \widehat{Env}_x$$
$$d_a = \min_i \left\{ y : \hat{\rho}''' \in \widehat{Env}_y, (lam, \hat{\rho}''') \in \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma}) \right\}$$
$$\hat{\rho}'' = \ldots$$
$$\hat{\sigma}' = \ldots$$

Each case is responsible for extending an environment and updating the store. The transition relation must take into account the depth of the current environment and what the depth will be after it is extended.

#### 2.4.2.1 Case 1

If the environment is at depth zero, it is updated in the standard way using the store.

$$\text{If } d_f = 0, \text{ then}$$
$$\hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{a}]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma})]$$
$$\hat{a}_i = \widehat{alloc}(v_i, \hat{\varsigma})$$

#### 2.4.2.2 Case 2

In the second case where we are still dealing with environments of depth greater than zero and the environment from the argument has depth greater than one, we do not need to allocate any addresses, but can simply update the environment.

$$\text{If } d_f > 0 \text{ and } d_a > 1, \text{ then}$$
$$\hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma})]]$$
$$\hat{\sigma}' = \hat{\sigma}$$

#### 2.4.2.3 Case 3

We must also be able to distinguish if extending the environment will drop the depth to zero. If that is the case, we need to allocate addresses for all the values in the environment and put them in the store.

$$\text{If } d_f > 0 \text{ and } d_a \leq 1, \text{ then}$$
$$\hat{\rho}'' = \widehat{alloc}_\rho(\hat{\varsigma})[v_i \mapsto \hat{a}]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma})]$$
$$\hat{a}_i = \widehat{alloc}(v_i, \hat{\varsigma})$$

As seen above, we need a function that can convert an environment of non-zero depth into an environment that is at depth zero.

$$\widehat{alloc}_\rho(\overbrace{call, \hat{\rho}, \hat{\sigma}}^{\hat{\varsigma}}) = (call, \hat{\rho}', \hat{\sigma}'), \text{ where}$$
$$v_i \in dom(\hat{\rho})$$
$$\hat{a}_i = \widehat{alloc}(v_i, \hat{\varsigma})$$
$$\hat{\rho}' = [][v_i \mapsto \hat{a}_i]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \{\hat{\rho}(v_i)\}]$$

### 2.4.3 Soundness

To prove the soundness, we must show that the abstract semantics simulate the concrete semantics following the standard proof found in [37]. The key insight for the proof is that we can provide an abstraction map that allocates a unique abstract address for every value in a non-zero depth environment.

### 2.4.4 Evaluation

We have implemented this analysis and show the results on the exact benchmarks presented in [12] in Table 2.5. Note that when the depth is zero, we get the same results as a traditional pushdown analysis. When using a depth of one, we see that precision is equal to or more precise than when $k = 1$, while at the same time generating a smaller abstract transition graph. With further increases in the depth, we see better precision and

**Table 2.5**: Environment unrolling benchmark results. The first three columns provide the name of the benchmark, the number of expressions, and variables in the program. The next seven columns show the results for running the original analysis with $k \in \{0, 1\}$ and with garbage collection off and on. Under each analysis, the first column is the number of *control states* and the second column is the number of *edges* computed during the analysis. The third column is the number of *singleton* variables. The last four columns show the results of our analysis. It gives the depth $d$ selected and the number of *control states*, *edges*, and *singleton* variables. The depth is not shown for larger values if the precision was not increased.

| Program | Exp | Var | $k$ | PDCFA | | | PDCFA+GC | | | $d$ | PDCFA+UE | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| mj09 | 19 | 10 | 0 | 38 | 38 | 4 | 33 | 32 | 4 | 0 | 38 | 38 | 4 |
| | | | 1 | 44 | 48 | 1 | 32 | 31 | 1 | 1 | 39 | 40 | 4 |
| | | | | | | | | | | 2 | 40 | 40 | 3 |
| | | | | | | | | | | 3 | 38 | 37 | 3 |
| | | | | | | | | | | 4 | 32 | 31 | 3 |
| eta | 21 | 13 | 0 | 32 | 32 | 6 | 30 | 29 | 8 | 0 | 32 | 32 | 6 |
| | | | 1 | 30 | 29 | 8 | 30 | 29 | 8 | 1 | 32 | 32 | 8 |
| | | | | | | | | | | 2 | 29 | 29 | 10 |
| kcfa2 | 20 | 10 | 0 | 36 | 35 | 4 | 35 | 34 | 4 | 0 | 36 | 35 | 4 |
| | | | 1 | 87 | 144 | 2 | 35 | 34 | 2 | 1 | 76 | 114 | 3 |
| | | | | | | | | | | 2 | 29 | 30 | 3 |
| kcfa3 | 25 | 13 | 0 | 50 | 51 | 5 | 53 | 52 | 5 | 0 | 50 | 51 | 5 |
| | | | 1 | 1761 | 4046 | 2 | 53 | 52 | 2 | 1 | 489 | 905 | 3 |
| | | | | | | | | | | 2 | 53 | 52 | 3 |
| blur | 40 | 20 | 0 | 523 | 813 | 3 | 299 | 335 | 9 | 0 | 523 | 813 | 3 |
| | | | 1 | 324 | 348 | 9 | 320 | 344 | 9 | 1 | 49 | 49 | 12 |
| | | | | | | | | | | 2 | 47 | 48 | 13 |
| loop2 | 41 | 16 | 0 | 108 | 117 | 4 | 67 | 71 | 4 | 0 | 108 | 117 | 4 |
| | | | 1 | 398 | 512 | 3 | 145 | 156 | 3 | 1 | 74 | 77 | 5 |
| sat | 51 | 23 | 0 | 545 | 773 | 4 | 254 | 317 | 4 | 0 | 545 | 773 | 4 |
| | | | 1 | 10872 | 14797 | 4 | 71 | 73 | 10 | 1 | 1400 | 1825 | 7 |
| | | | | | | | | | | 2 | 7625 | 9769 | 7 |
| | | | | | | | | | | 3 | 71 | 73 | 13 |

smaller transition graphs than even occurs with abstract garbage collection. The run times of the new analyses correlate to how many states and edges are in the graph. Environment unrolling gives improvements in both precision and speed.

Schmidt proposed a similar scheme [47]. He also indexes environments by numbers and if an environment refers to another environment, then its index is one less. Environments of index zero are simply joined. Sereni and Jones developed $k$-bounded CFA in their termination analysis which takes a similar approach of cutting off nested environments [48]. This work was not originally inspired by theirs but rather the optimization of CFA2 where variables that do not escape in a closure are represented exactly. Our approach is different in that when we reach index zero, we start abstracting through the store rather than joining the environment. This allows our approach to use global store widening techniques or abstract garbage collection.

### 2.4.5 Conclusion

We have shown a unique approach to abstract interpretation. We have described how our approach is similar to loop unrolling and given a general overview of its semantics. We have also provided results from an implementation of the analysis, which demonstrate its possible benefits.

While the presentation only demonstrated the abstract interpretation for ANF lambda calculus, it can immediately be applied to other language forms, including those involving mutation. In the case of mutation, we would need to always store allocate mutable variables.

## 2.5 Strong Function Call

This section presents an incremental improvement to abstract interpretation of higher-order languages, similar to strong update [8], which we term strong function call. In an abstract interpretation, after a function call, we know which abstract closure was called and can deduce that any other values found at the same abstract address in the abstract store could not possibly exist in the corresponding concrete state. Thus they can be removed from the abstract store without loss of soundness. We provide the intuition behind this analysis along with a general overview of its soundness proof.

### 2.5.1 Introduction

Because of the unmovable force of the halting problem, static analyses as a rule must be imprecise in some cases in order to remain inside the curtain of computability. Concrete values must be abstracted, but this abstraction leads to imprecision. Any techniques that

regain some if this lost precision are desirable. To this end, we propose strong function call. A small extension to a traditional $k$-CFA analysis [49].

In an abstract interpretation of a lambda calculus which uses store allocation, the source of nondeterminism is that an abstract address can point to several abstract closures. This results in a fork in the abstract transition graph. Any subsequent function calls that dereference that same address will also fork. However, the key insight used in this technique is that once we have made a function call, we know which procedure was called and can deduce that any other values in the store could not possibly exist in the corresponding concrete state. Thus, these extra values should be pruned from the store in order to improve the precision of subsequent dereferences of the address.

Strong function call is developed upon the abstract semantics presented in Section 2.2. However, the ideas presented in this section are not restricted to CPS, but can easily be extended to different language forms.

The approach taken in Section 2.2 to abstract the concrete interpreter was to make the address space finite as described in Abstracting Abstract Machines [56]. The consequence of this action is that the interpreter is now forced to have multiple values in the store at a single address. Note that these values cannot grow infinitely because the state space is finite. However, the issue arises that a single abstract address can represent multiple concrete addresses. This is where abstract counting comes into play [20]. The abstract count of an abstract address is the number of concrete addresses that abstract address represents.

### 2.5.2 Abstract Counting

A natural domain for abstract counting is the set $\hat{\mathbb{N}}$. The analysis cares if an abstract address represents a single concrete address or multiple concrete addresses. Our analysis will leverage the power of this information.

$$\hat{\mathbb{N}} = \{0, 1, \infty\}$$

Note that the abstract count cannot be gleaned simply from the store and the number of abstract closures at a particular address. It can be the case that a single abstract address representing multiple concrete addresses can only contain one value in the store. It can also be the case that an abstract address can represent a single concrete address but have multiple values in the store. Indeed, this is the case of which this analysis will take advantage.

Defining the operator $\oplus$ naturally extends over the addition operator. The operator will also lift point-wise over functions, allowing the transition function to update a store which maps abstract address to abstract counts. Here we see that we are retaining the vital information that we need, whether we have a single value or multiple values.

$$0 \oplus \hat{n} = \hat{n}$$
$$\hat{n} \oplus 0 = \hat{n}$$
$$1 \oplus 1 = \infty$$
$$\hat{n} \oplus \infty = \infty$$
$$\infty \oplus \hat{n} = \infty$$

One traditional use case for abstract counting in a higher-order setting is for strong update [8]. If it can be shown that an abstract address only represents one concrete address, the value does not need to be joined in the store, but can shadow the old value.

$$([\![\texttt{(set!-then } v \text{ æ } \mathit{call}\texttt{)}]\!], \hat{\rho}, \hat{\sigma}, \hat{\mu}) \rightsquigarrow (\mathit{call}, \hat{\rho}, \hat{\sigma}', \hat{\mu}), \text{ where}$$
$$a = \hat{\rho}(v)$$
$$\widehat{clo} = \hat{\mathcal{A}}(\text{æ}, \hat{\rho}, \hat{\sigma})$$
$$\hat{\sigma}' = \begin{cases} \hat{\sigma}[\hat{a} \mapsto \widehat{clo}] & \hat{\mu}(a) \leq 1 \\ \hat{\sigma} \sqcup [\hat{a} \mapsto \widehat{clo}] & \text{otherwise} \end{cases}$$

We now proceed by extending the abstract semantics for our analysis. We have added an additional store like entity, $\widehat{Count}$, which will keep track of abstract counts. The remaining components of the abstract state remain unchanged.

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \times \widehat{Count}$$
$$\hat{\mu} \in \widehat{Count} = \widehat{Addr} \rightharpoonup \hat{\mathbb{N}}$$

The function $\hat{\mathcal{G}}$ is the crux of our analysis.

$$\hat{\mathcal{G}} : \mathsf{Var} \times \widehat{Clo} \times \widehat{Env} \times \widehat{Store} \times \widehat{Count} \rightharpoonup \widehat{Store}$$

It updates the store by possibly pruning values that are no longer needed. It determines its action based on the syntactic type of the function we are applying. In the case of a lambda term, nothing is done to the store. If the case of a variable term, if the abstract address represents multiple concrete addresses, the function does nothing to the store. If the abstract address only represents a single concrete address, it shadows the value pointed

to by the address, restricting its value to only be the function that is being applied. In the case where there is only one closure at the address, this operation has no effect. However, in the case where there are multiple closures at that abstract address, it has the effect of restricting the store to only have a single value at that address.

$$\hat{\mathcal{G}}(lam, \widehat{clo}, \hat{\rho}, \hat{\sigma}, \hat{\mu}) = \hat{\sigma}$$

$$\hat{\mathcal{G}}(v, \widehat{clo}, \hat{\rho}, \hat{\sigma}, \hat{\mu}) = \begin{cases} \hat{\sigma}[\hat{\rho}(v) \mapsto \left\{\widehat{clo}\right\}] & \hat{\mu}(\hat{\rho}(v)) \leq 1 \\ \hat{\sigma} & \text{otherwise} \end{cases}$$

The newly defined augmented abstract transition relation is very similar to the original abstract transition relation. We have added the abstract counting map to maintain the information needed for our analysis. Also notice that we now use an auxiliary function to update the store before joining it with the values from the arguments of the call site. This restricts the store to only contain the closure that is being applied at the call site. This reduces the size of the abstract state and could result in increased precision and speed.

$$\overbrace{(\llbracket (f\ æ_1 \ldots æ_n) \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\mu})}^{\hat{\varsigma}} \rightsquigarrow (call, \hat{\rho}'', \hat{\sigma}'', \hat{\mu}'), \text{ where}$$

$$\widehat{clo} \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$

$$(\llbracket (\lambda\ (v_1 \ldots v_n)\ call) \rrbracket, \hat{\rho}') = \widehat{clo}$$

$$\hat{a}_i = \widehat{alloc}(v_i, \hat{\varsigma})$$

$$\hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{a}_i]$$

$$\hat{\sigma}' = \hat{\mathcal{G}}(f, \widehat{clo}, \hat{\rho}, \hat{\sigma}, \hat{\mu})$$

$$\hat{\sigma}'' = \hat{\sigma}' \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(æ_i, \hat{\rho}, \hat{\sigma})]$$

$$\hat{\mu}' = \hat{\mu} \oplus [\hat{a}_i \mapsto 1]$$

### 2.5.3 Soundness

To prove the soundness of this analysis, we provide an abstraction map that connects the concrete and abstract state spaces.

$$\alpha((call, \rho, \sigma)) = (call, \alpha(\rho), \alpha(\sigma), \alpha_\mu(\sigma))$$

$$\alpha(\rho) = \lambda v.\alpha(\rho(v))$$

$$\alpha(\sigma) = \lambda \hat{a}. \bigsqcup_{\alpha(a)=\hat{a}} \{\alpha(\sigma(a))\}$$

$$\alpha_\mu(\sigma) = \lambda \hat{a}. \bigoplus_{\alpha(a)=\hat{a}} 1$$

$$\alpha(lam, \rho) = \{(lam, \alpha(\rho)\}$$

$\alpha(a)$ is determined by the allocation function

From there, we prove that the abstract transition relation simulates the concrete transition relation.

**Theorem 1.** *If* $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$ *and* $\varsigma \Rightarrow \varsigma'$ *then there must exist* $\hat{\varsigma} \in \hat{\Sigma}$ *such that* $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$ *and* $\hat{\varsigma} \leadsto \hat{\varsigma}'$.

*Proof.* The proof follows in the same manner as presented in [37]. The only difference between this abstraction and the standard one can be found in the case where we restrict the size of the store. However, this is simple to account for, as the concrete semantics can only apply one function at a call site. Realizing that the concrete address can only hold one value and that the abstract address only represents one concrete address, it is sound to restrict the store to that one value. □

### 2.5.4 Conclusion

In this section, we have shown that strong update can be used to restrict the size of the store at application sites. By reducing the size of the store, it is highly likely we will gain both speed and precision. This has been shown to be the case with abstract garbage collection [39].

It might be easy to assume that abstract garbage collection would subsume this analysis. However, this is not the case. Abstract garbage collection filters out bindings that are not reachable from the root set. However, in this analysis, we are dealing with an address that is definitely live and will not be garbage collected. Even in the presence of abstract garbage collection, we still get flow sets that contain more than one value. Otherwise, the precision of an analysis with abstract garbage collection would be perfect.

This idea could also easily be extended to object-oriented languages. With its extensive use of polymorphism, one could imagine that the calling object of a method could easily have multiple flow sets. This analysis would allow us to soundly reduce the size of these flow sets.

## 2.6 Generalizing to Other Languages

The techniques developed in this chapter are applicable to any abstract machine created with the abstracting abstract machines approach. I want to demonstrate that the techniques developed in this chapter can be applied to other languages other than the simple lambda calculus. I would like to demonstrate that it is equally possible to apply these techniques

to an object oriented language. To this end, I will demonstrate that these techniques work on A-Normal Featherweight Java. Featherweight Java is a simple language that eliminates many features of the Java language to permit a small calculus. The original intent of creating this language was to provide a formal method to work on proofs. Like the lambda calculus relation to languages such as ML, Featherweight Java has a similar relation to Java. Featherweight Java programs are complete Java programs. They could be taken and compiled with a compiler.

I will first present the syntax of the language, a semantics to evaluate this language in a small-step setting, and then how to abstract these semantics to guarantee termination. Once this has been developed, I will present how the techniques presented in this chapter can be applied to these abstract semantics. This will be demonstrated by showing how the techniques apply to A-Normal Featherweight Java.

### 2.6.1 Syntax

A-Normal Featherweight Java differs from the original formulation of Featherweight Java. To aid in simplifying the semantics, arguments must be atomically evaluable. Statements are reintroduced into the language, but this does not result in a change in the expressive power of the language. The five statements allowed by the language are: field reference, method invocation, object allocation, casting, and return.

$$\begin{aligned}
\mathsf{Class} &::= \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overline{C''\ f\ ;}\ K\ \overline{M}\} \\
K \in \mathsf{Konst} &::= C\ (\overline{C\ f}\ )\ \{\texttt{super}(\overline{f'})\,;\ \overline{\texttt{this}.f'' = f'''\,;}\} \\
M \in \mathsf{Method} &::= C\ m(\overline{C\ v}\ )\ \{\ \overline{C\ v\ ;}\ \overline{s}\ \} \\
s \in \mathsf{Stmt} &::= v = e;^{\ell} \mid \texttt{return}\ v;^{\ell} \\
e \in \mathsf{Exp} &::= v.f \mid v.m(\overline{v}) \mid \texttt{new}\ C\ (\overline{v}) \mid (C)v \\
f \in \mathsf{FieldName} &= \mathsf{Var} \\
C \in \mathsf{ClassName} &\text{ is a set of class names} \\
m \in \mathsf{MethodCall} &\text{ is a set of method invocation sites} \\
\ell \in \mathsf{Lab} &\text{ is a set of labels}
\end{aligned}$$

### 2.6.2 Concrete State Space

The concrete state space used to evaluate the semantics appears very similar to the concrete state space used by continuation-passing style lambda calculus. However, there are no longer environments. There are now frame pointers which behave in a similar fashion

to the common frame pointer mechanism used by actual computers. To get an address for a local variable, the variable name is paired with the frame pointer. This could be thought of as pairing the base address of the frame pointer with the offset of the variable. Object addresses are created by pairing a base address with the field name, similar to how an object has a base address in physical memory and its fields are accessed as offsets.

$$\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Stmt} \times \mathit{FramePointer} \times \mathit{Store} \times \mathit{Time} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup D \\
a \in \mathit{Addr} &= \mathit{StackAddr} + \mathit{FieldAddr} \\
\mathit{StackAddr} &= \mathit{FramePointer} \times \mathsf{Var} \\
\mathit{FieldAddr} &= \mathit{ObjAddr} \times \mathsf{Field} \\
d \in D &= \mathit{Obj} + \mathit{Kont} \\
o \in \mathit{Obj} &= \mathsf{ClassName} \times \mathit{ObjAddr} \\
\kappa \in \mathit{Kont} &= \mathsf{Var} \times \mathsf{Stmt} \times \mathit{FramePointer} \\
\phi \in \mathit{FramePointer} &\text{ is a set of frame pointers} \\
\mathit{ObjAddr} &\text{ is a set of addresses} \\
t \in \mathit{Time} &\text{ is a set of timestamps}
\end{aligned}$$

### 2.6.3 Concrete Semantics

The concrete semantics are given by a transition relation and helper functions which are used by the transition relation.

#### 2.6.3.1 Helper Functions

The *tick* function simply prepends the current label to the existing labels. This guarantees a unique address at each allocation site.

There are two types of base address, those for stack frames and those for local variables. Pairing the stack frame with a variable name gives the address of local variables (including parameters for methods). There is also a special field reserved for the continuation. Pairing the base of an object address with the field name gives the address of fields for objects.

$$\begin{aligned}
\mathit{tick}(\ell, t) &= \ell : t \\
\mathit{alloc}(\varsigma, t) &= t
\end{aligned}$$

The function $\mathcal{M}$, given a method name and a class, looks up the method. If the method is not defined, then it moves up to the super class until one is found. Once the base class

Object is hit, it returns nothing. This means that methods can be overridden by having the same name. The first method found in the object hierarchy will be the one returned by this method.

The function $\mathcal{C}$ returns all the fields of the object. This includes all the field defined by its super classes as well.

$$\mathcal{M} : D \times \mathsf{MethodCall} \rightharpoonup \mathsf{Method}$$

$$\mathcal{C} : \mathsf{ClassName} \rightharpoonup \mathsf{FieldName}*$$

#### 2.6.3.2 Transition Relation

There are five rules in the transition relation, one for each type of statement: field reference, method invocation, object allocation, casting, and return.

**Field reference**

$$([\![v = v'\,.\,f\,;^{\ell}]\!], \phi, \sigma, t) \Rightarrow (succ(\ell), \phi, \sigma', t'), \text{ where}$$
$$t' = tick(\ell, t)$$
$$(C, a) = \sigma((\phi, v'))$$
$$\sigma' = \sigma[(\phi, v) \mapsto \sigma((a, f))]$$

**Method invocation**

$$([\![v = v_0\,.\,m(\overline{v'})\,;^{\ell}]\!], \phi, \sigma, t) \Rightarrow (s_0, \phi', \sigma', t'), \text{ where}$$
$$t' = tick(\ell, t)$$
$$[\![C\ m(\overline{C\ v''})\ \{\overline{C'\ v'''};\ \overline{s}\}]\!] = \mathcal{M}(d_0, m)$$
$$\phi' = alloc(\phi, t')$$
$$\kappa = (v, succ(\ell), \phi)$$
$$d_0 = \sigma((\phi, v_0))$$
$$\sigma' = \sigma[(\phi', \texttt{return}) \mapsto \kappa, (\phi', \texttt{this}) \mapsto d_0, (\phi', v_i'') \mapsto \sigma((\phi, v_i'))]$$

**Object allocation**

$$([\![v = \texttt{new}\ C\ (\overline{v'})\,;^{\ell}]\!], \phi, \sigma, t) \Rightarrow (succ(\ell), \phi, \sigma', t'), \text{ where}$$
$$t' = tick(\ell, t)$$
$$a = alloc(C, t')$$
$$\overline{f} = \mathcal{C}(C)$$
$$\sigma' = \sigma[(a, f_i) \mapsto \sigma((\phi, v_i')), (\phi, v) \mapsto (C, a)]$$

**Casting**

$$([\![v = (C')\ v']\!], \phi, \sigma, t) \Rightarrow (succ(\ell), \phi, \sigma', t') \text{ if } C <: C', \text{ where}$$
$$t' = tick(\ell, t)$$
$$(C, a) = \sigma((\phi, v'))$$
$$\sigma' = \sigma[(\phi, v) \mapsto \sigma((\phi, v'))]$$

**Return**

$$([\![\texttt{return}\ v\texttt{;}^{\ell}]\!], \phi, \sigma, t) \Rightarrow (s, \phi', \sigma', t'), \text{ where}$$
$$t' = tick(\ell, t)$$
$$(v', s, \phi') = \sigma((\phi, \texttt{return}))$$
$$d = \sigma((\phi, v))$$
$$\sigma' = \sigma[(\phi', v') \mapsto d].$$

### 2.6.4 Abstract State Space

The abstract state space differs from the concrete state space in that we now have a finite number of addresses and timestamps. Like before, this causes the store to be forced to store multiple values at a single address.

$$\varsigma \in \Sigma = \mathsf{Stmt} \times \widehat{FramePointer} \times \widehat{Store} \times \widehat{Time}$$
$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \rightharpoonup \hat{D}$$
$$\hat{a} \in \widehat{Addr} = \widehat{StackAddr} + \widehat{FieldAddr}$$
$$\widehat{StackAddr} = \widehat{FramePointer} \times \mathsf{Var}$$
$$\widehat{FieldAddr} = \widehat{ObjAddr} \times \mathsf{Field}$$
$$\hat{d} \in \hat{D} = \mathcal{P}\left(\widehat{Obj} + \widehat{Kont}\right)$$
$$\hat{o} \in \widehat{Obj} = \mathsf{ClassName} \times \widehat{ObjAddr}$$
$$\hat{\kappa} \in \widehat{Kont} = \mathsf{Var} \times \mathsf{Stmt} \times \widehat{FramePointer}$$
$$\hat{\phi} \in \widehat{FramePointer} \text{ is a finite set of frame pointers}$$
$$\widehat{ObjAddr} \text{ is a finite set of addresses}$$
$$\hat{t} \in \widehat{Time} \text{ is a finite set of timestamps}$$

### 2.6.5 Abstract Semantics

The abstract semantics are similar to their concrete counterparts but subtly different. They must take into account that when looking up values in the store, multiple abstract

values can be returned. This means that there could be multiple successor states when evaluating successors states using the abstract transition relation.

**Field reference**

$$
\begin{aligned}
([\![v = v'\,.\,f\,;^{\ell}]\!], \hat{\phi}, \hat{\sigma}, \hat{t}) \rightsquigarrow (succ(\ell), \hat{\phi}, \hat{\sigma}', \hat{t}'), \text{ where} \\
\hat{t}' &= \widehat{tick}(\ell, t) \\
(C, \hat{a}) &\in \hat{\sigma}((\hat{\phi}, v')) \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [(\hat{\phi}, v) \mapsto \hat{\sigma}((\hat{a}, f))]
\end{aligned}
$$

**Method invocation**

$$
\begin{aligned}
([\![v = v_0\,.\,m(\overline{v'})\,;^{\ell}]\!], \hat{\phi}, \hat{\sigma}, \hat{t}) \Rightarrow (s_0, \hat{\phi}', \hat{\sigma}', \hat{t}'), \text{ where} \\
\hat{t}' &= \widehat{tick}(\ell, \hat{t}) \\
[\![C\ m(\overline{C\ v''})\ \{\overline{C'\ v'''\,;}\ \overline{s}\}]\!] &= \hat{\mathcal{M}}(\hat{d}_0, m) \\
\hat{\phi}' &= \widehat{alloc}(\phi, t') \\
\hat{\kappa} &= (v, succ(\ell), \phi) \\
\hat{d}_0 &\in \hat{\sigma}((\hat{\phi}, v_0)) \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [(\hat{\phi}', \texttt{return}) \mapsto \hat{\kappa}] \sqcup [(\hat{\phi}', \texttt{this}) \mapsto \hat{d}_0] \\
&\quad \sqcup [(\hat{\phi}', v_i'') \mapsto \hat{\sigma}((\hat{\phi}, v_i'))]
\end{aligned}
$$

**Object allocation**

$$
\begin{aligned}
([\![v = \texttt{new}\ C\ (\overline{v'})\,;^{\ell}]\!], \hat{\phi}, \hat{\sigma}, \hat{t}) \rightsquigarrow (succ(\ell), \hat{\phi}, \hat{\sigma}', \hat{t}'), \text{ where} \\
\hat{t}' &= \widehat{tick}(\ell, \hat{t}) \\
\hat{a} &= \widehat{alloc}(C, \hat{t}') \\
\overline{f} &= \mathcal{C}(C) \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [(\hat{a}, f_i) \mapsto \hat{\sigma}((\hat{\phi}, v_i'))] \sqcup [(\hat{\phi}, v) \mapsto (C, \hat{a})]
\end{aligned}
$$

**Casting**

$$
\begin{aligned}
([\![v = (C')\ v']\!], \hat{\phi}, \hat{\sigma}, \hat{t}) \rightsquigarrow (succ(\ell), \hat{\phi}, \hat{\sigma}', \hat{t}') \text{ if } C <: C', \text{ where} \\
\hat{t}' &= \widehat{tick}(\ell, \hat{t}) \\
(C, \hat{a}) &\in \hat{\sigma}((\hat{\phi}, v')) \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [(\hat{\phi}, v) \mapsto \hat{\sigma}((\hat{\phi}, v'))]
\end{aligned}
$$

**Return**

$$([\![\texttt{return } v\texttt{;}^{\ell}]\!], \hat{\phi}, \hat{\sigma}, \hat{t}) \rightsquigarrow (s, \hat{\phi}', \hat{\sigma}', \hat{t}'), \text{ where}$$
$$\hat{t}' = \widehat{tick}(\ell, \hat{t})$$
$$(v', s, \hat{\phi}') \in \sigma((\hat{\phi}, \texttt{return}))$$
$$\hat{d} = \hat{\sigma}((\hat{\phi}, v))$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{\phi}', v') \mapsto \hat{d}]$$

### 2.6.6 Applying the Improvements to ANFJ

We now proceed to describe how each of the three techniques can be applied to these new abstract semantics, which are similar but slightly different, hopefully giving insight how these approaches could be applied to different languages.

#### 2.6.6.1 Priority Work Queue

In the instance of using a priority queue, as long as there is a transition relation and global store widening is used, the same techniques generally apply. There is still the decision to make of which state to visit when, and also the decision of when to join the values to the global store and when to widen the store of the individual states. The semantics for almost any language (including the object oriented language, A-Normal Featherweight Java) are still capable of using a global store.

We now show how global store widening works with the ANFJ abstract state space. We change how we do the abstract interpretation; instead of a function that computes a graph of the reachable states, we return a graph of partial states and a single global approximation of the store.

$$\widehat{System} = \mathcal{P}\left(\mathsf{Stmt} \times \widehat{FramePointer} \times \widehat{Time}\right) \times \widehat{Store}$$

However, the heuristics used would be different as the components of the state have changed.

#### 2.6.6.2 Environment Unrolling

The idea behind environment unrolling could be applied equally as well. In the semantics of A-Normal Featherweight Java, there are two transition rules when we allocate new addresses: on method invocation and on object allocation.

There are now two sources of infiniteness in the state space. New objects could be allocated ad infinitum as well as frame pointers. Therefore, objects could reference other objects infinitely and continuations could reference other continuations infinitely.

For object allocation, the class name is paired with a freshly allocated address. The behavior of environment unrolling for object allocation is mimicked by returning a fresh address depending on the depth of the object graph to which the fields are being initialized.

On method invocation, there is a single address that we need to allocate: a frame pointer to give a base address for the parameters of the method and the address for the return address.

In the case of continuation-passing style lambda calculus, these allocations are hidden in the same mechanism used everywhere for function call. However, they are represented more explicitly in this language form.

Once again, the behavior of environment unrolling can be mimicked by calculating the depth of the continuation and once it reaches a certain depth, start abstracting through the store.

Because of our choice of semantics using a frame pointer to generate addresses, rather than an environment, the unrolling cannot proceed in the same fashion. However, the same effect can be achieved through the allocation function.

Allocation happens at two locations, when allocating a frame pointer and when allocating an object. We need a function that calculates the depth of abstraction.

$$\widehat{alloc}_d((s,\hat{\phi},\hat{\sigma},\hat{t})) = \begin{cases} \text{fresh address} & \widehat{depth}(\hat{\phi},\hat{\sigma},\{\}) \leq d \\ \widehat{alloc}((s,\hat{\phi},\hat{\sigma},\hat{t})) & \text{otherwise} \end{cases}$$

The abstract depth of a frame pointer is computed with the following.

$$\widehat{depth}(\hat{\phi},\hat{\sigma},seen) = \begin{cases} \infty & \phi \in seen \\ 0 & \hat{\sigma}((\hat{\phi},\texttt{return})) = \texttt{halt} \\ 1 + \widehat{depth}(\hat{\phi}',seen \cup \{\hat{\phi}\}) & \hat{\sigma}((\hat{\phi},\texttt{return})) = (v,s,\phi') \end{cases}$$

The abstract depth of an object is computed with the following.

$$1 + \max\left\{\widehat{depth}((C,\hat{a}),\hat{\sigma}) : (C,\hat{a}) \in \hat{\sigma}((\hat{\phi},v_i))\right\}$$

$$\widehat{depth}((C,\hat{a}),\hat{\sigma}) = \max\left\{\widehat{depth}(\hat{o}_i,\hat{\sigma}) : \hat{o}_j \in \hat{\sigma}((\hat{a},f_i)),\ \overline{f} = \mathcal{C}(C)\right\}$$

When implementing this, it would make more sense to store the depth which each frame pointer and object base address, rather than recomputing it each time.

#### 2.6.6.3 Strong Function Call

Strong function call can be handled in a very similar fashion. In the case of a method invocation, there could be multiple receiver objects. The store can be sharpened once the method has been invoked.

Like before, a map is added to the abstract state space which maps how many concrete addresses an abstract address represents.

$$\varsigma \in \Sigma = \mathsf{Stmt} \times \widehat{\mathit{FramePointer}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Time}} \times \widehat{\mathit{Count}}$$
$$\hat{\mu} \in \widehat{\mathit{Count}} = \widehat{\mathit{Addr}} \rightharpoonup \hat{\mathbb{N}}$$

We change slightly the relation for method invocation. If it is the case that there are multiple receiver objects but the abstract count is one, the flow set size can be reduced to the receiver object whose method is being invoked.

The function $\hat{\mathcal{G}}$ is redefined slightly to take into account the different language forms it now handles.

$$\hat{\mathcal{G}} : \mathsf{Var} \times \hat{D} \times \times \widehat{\mathit{Store}} \times \widehat{\mathit{Count}} \rightharpoonup \widehat{\mathit{Store}}$$

Observe how a form of this is already happening in the abstract semantics. Only the receiver of the method invocation is bound to `this`, not the entire flow set of the variable.

$$\hat{\mathcal{G}}(v, \hat{d}, \hat{\phi}, \hat{\sigma}, \hat{\mu}) = \begin{cases} \hat{\sigma}[(\hat{\phi}, v) \mapsto \{\hat{d}\}] & \hat{\mu}((\hat{\phi}, v)) \leq 1 \\ \hat{\sigma} & \text{otherwise} \end{cases}$$

## 2.7 Advice for Implementors

This section discusses generally how these techniques could be implemented and how they interact with each other.

When implementing global store widening, the developer needs to be cognizant that the order in which states are visited could effect memory usage and the running time of the analysis. Depending on language features, using a global store might introduce precision issues. In the end, it might not be worth implementing a priority queue, but it is an option to be aware of, especially if stringent memory requirements are present.

Environment unrolling can be implemented fully in the allocation function for abstract addresses. This means that if the implementation has sufficient encapsulation, it would be easy to swap in a more precise allocation function. It would also be possible to have an adaptive allocation function where environment unrolling is used only for functions that need extra precision.

Strong function call is more of a cautionary tale. When developing an abstract interpreter, it is easy to convince yourself that you can make small changes that will optimize the abstract interpreter. However, these changes may make the analysis unsound. Going unsound is not always a bad thing, but the developer needs to be aware when making this decision. Unless some form of abstract counting is used, it is easy to make the analyzer unsound when sharpening.

### 2.7.1 Composing Optimizations

A natural question that arises is how these three features compose. The priority queue requires global store widening. Environment unrolling could be composed with global store widening, though the benchmarks evaluated used a state per store and were also in a pushdown setting. Strong function call could be used as well at the same time. In fact, all three of these cases could compose well, though you cannot use strong function call in a global store widened setting. However, it is possible to use it if widening is not quite as restrictive, for example in another setting where point-wise widening or some other form of widening that is not global is used.

# CHAPTER 3

# MAPPING CONTROL-FLOW CONSTRAINTS

An alternative approach to abstract interpretation for solving a control-flow analysis is using constraints. These constraints can be mapped to problems for which efficient solutions already exist. In this chapter, we will explore what these constraints look like and how we can map them to three separate problems: a pointer analysis, SAT, and linear algebra.

## 3.1 Constraint-Based Analysis

One way to perform a control-flow analysis is with constraints. In describing the constraints, we will operate over a simple language, the lambda calculus. We will first briefly describe the lambda calculus. Then, we will briefly describe how to generate the constraints.

### 3.1.1 Lambda Calculus

The lambda calculus only has three language forms: variable reference, lambda terms, and function application. This is the core of many functional programming languages, such as Racket, and thus also of higher-order languages.

$$e \in \mathsf{Exp} ::= v \mid (\lambda\ (v)\ e_b) \mid (e_1\ e_2)$$

$v \in \mathsf{Var}$ is a set of identifiers

### 3.1.2 Palsberg Constraints for Solving 0CFA

These are the constraints that are generated by Palsberg [42]. For expression $e \in \mathsf{Exp}$ under analysis, let $E$ be the set of all expressions in $e$.

$$\frac{(\lambda\ (v)\ e_b) \in E}{\{(\lambda\ (v)\ e_b)\} \subseteq \mathit{flows}[\![(\lambda\ (v)\ e_b)]\!]}$$

$$\frac{(e_1\ e_2) \in E \qquad (\lambda\ (v)\ e_b) \in \mathit{flows}[\![e_1]\!]}{\mathit{flows}[\![e_2]\!] \subseteq \mathit{flows}[\![v]\!]}$$

$$\frac{(e_1\ e_2) \in E \qquad (\lambda\ (v)\ e_b) \in \mathit{flows}[\![e_1]\!]}{\mathit{flows}[\![e_b]\!] \subseteq \mathit{flows}[\![(e_1\ e_2)]\!]}$$

The first rule states that a lambda term is in its own flow set. The second rules says that at a function application, for every lambda term that flows to the function being applied, the flow set of the formal parameter includes everything that is in the flow set of the argument. The third rule states that also at a function application, for every lambda term that flows to the function being applied, whatever is in the flow set of the body of the lambda term is also in the flow set of the function application.

We iterate over the expressions of the program, generating constraints for each expression. These constraints describe a set of lambda terms, $\mathit{flows}[\![e]\!]$, for each subexpression $e$ in our program.

### 3.1.3 The Lambda Calculus in Continuation-Passing Style

As mentioned in Chapter 2, this language differs from the lambda calculus in that lambda terms now have multiple arguments and the body of a lambda term is now restricted to be only a call site.

$$\begin{aligned} \mathit{call} \in \mathsf{Call} &::= (f\ \mathit{ae}_1 \ldots \mathit{ae}_n) \\ f, \mathit{ae} \in \mathsf{AExp} &::= \mathit{lam} \mid v \\ \mathit{lam} \in \mathsf{Lam} &::= (\lambda\ (v_1 \ldots v_n)\ \mathit{call}) \\ v \in \mathsf{Var} &\text{ is a set of identifiers} \end{aligned}$$

### 3.1.4 Palsberg Style Inference Rules for CPS

We will now explore how the constraints for a control-flow analysis of a continuation-passing style language change. The constraints for continuation-passing style are simpler because we no longer have to worry about the flow sets of the body of lambda terms. This is because we never return, but rather invoke the passed explicit continuation. This causes us to go from three inference rules to two.

The inference rules are as follows. For expression $e \in \mathsf{Exp}$ under analysis, let $E$ be the set of all expressions in $e$.

$$\frac{(\lambda\ (v_1 \ldots v_n)\ call) \in E}{\{(\lambda\ (v_1 \ldots v_n)\ call)\} \subseteq \mathit{flows}[\![(\lambda\ (v_1 \ldots v_n)\ call)]\!]}$$

$$\frac{(f\ æ_1 \ldots æ_n) \in E \qquad (\lambda\ (v_1 \ldots v_n)\ call) \in \mathit{flows}[\![f]\!]}{\mathit{flows}[\![æ_1]\!] \subseteq \mathit{flows}[\![v_1]\!] \ldots \mathit{flows}[\![æ_n]\!] \subseteq \mathit{flows}[\![v_n]\!]}$$

## 3.2 CFA via Pointer Analysis

Imagine we have an analysis or compiler optimization that needs a control-flow analysis, but we do not have a control-flow analysis immediately available to us. Our analysis might need to know the control flow of the program in order to prove some security properties or the absence of errors. Our compiler optimization might need to know the control flow of the program to speed up the program by inlining functions.

There are a large number of tools that can perform a related but different analysis: pointer analysis. A pointer analysis tells us which objects variables point to in a program. Wouldn't it be nice if we could just use these tools? We could spend a significant amount of time developing a new modern control-flow analysis tool, pulling in the latest features from these similar but different tools, or we could find ways to reuse these existing tools to solve the problem we have at hand. Using existing tools saves us the effort of developing our own tool and allows us to rely on mature and well tested tools.

### 3.2.1 Overview

Precise control-flow analysis is expensive. For example, 0CFA as described by Palsberg, which is the analysis we explore in this chapter, is cubic [42]. This work was initially motivated by trying to identify ways we could speed up control-flow analysis. With the advancement of GPUs being used for general process computing and more cores being available on commodity machines, one way to speed up the analysis is to parallelize it. The only work we know of that parallelizes control-flow analysis of higher-order languages is EigenCFA [46]. We also know of two recent tools that parallelize pointer analysis developed by Méndez-Lojo et al. [34, 33].

One option we considered to speed up control-flow analysis was to deeply understand these two tools for pointer analysis and port over any ideas that would improve EigenCFA. However, we took the option to use those tools directly. In order to do this, we needed to map a control-flow analysis into a pointer analysis. This chapter demonstrates how to

do this. To our knowledge, even though the similarities between these analyses have been known and there has been a general feeling in the community that they are the same, this mapping has not been explicitly laid out [40].

In this section, we plan to show that we can successfully take the state of the art in pointer analysis and improve upon the state of the art in control-flow analysis in terms of performance. Our goal is to leverage existing tools available to us from the pointer analysis community. We wish to exploit the efficiency of these static analysis tools for control-flow analysis of higher-order languages.

The contributions made in this section are as follows.

- We show that there exists a direct mapping between a control-flow analysis of higher-order languages and a pointer analysis for first-order languages. In fact, we show three different mappings, each serving a slightly different purpose. The first mapping, in Section 3.2.4, is to help us demonstrate the connection between a control-flow analysis and a pointer analysis. The second mapping, in Section 3.1.3, allows us to use the benchmarks that were used by the control-flow analysis tool EigenCFA. The final mapping, in Section 3.2.10, makes a different trade-off by containing more inference rules, but results in fewer variables. Because of this mapping, we end up with one of the fastest tools for 0CFA.

- We show in Section 3.2.7 that the constraints generated by a traditional control-flow analysis are equivalent to the constraints generated by a pointer analysis after going through our mapping. This is important because it means that the answer we get back from a pointer analysis tool will be the same answer we would get from a control-flow analysis tool.

- In Section 3.2.11, we then demonstrate the benefit of this mapping by comparing two recent parallel pointer analysis tools with a recent parallel control-flow analysis tool called EigenCFA [46]. We compare a control-flow analysis tool that runs on the GPU with a pointer analysis tool that also runs on the GPU [33]. We also compare these tools to one that runs on a single CPU with multiple cores [34]. Both of theses pointer analysis tools were written by Méndez-Lojo et al. For the benchmarks we used, this multithreaded implementation performs the best. We saw that the CPU multicore pointer analysis tool is able to run up to 35 times faster than the GPU control-flow analysis tool. With the mapping outlined in this chapter, we beat the fastest known GPU version of CFA with a pointer analysis tool that runs on the CPU.

The implementation details of a static analysis tool matter. With the wide range of work that has been done for pointer analysis, applying these techniques directly to control-flow analysis of higher-order languages is advantageous. Due to the mappings in this chapter, for the chosen benchmarks, we now have the fastest way we know of to do higher-order control-flow analysis that has the precision of a 0CFA as formulated by Palsberg [42].

### 3.2.2 Background

In this section, we give a brief description of a traditional pointer analysis and a traditional control-flow analysis. Control-flow analysis of higher-order programs and pointer analysis share much in common with each other and often, the pointer analysis and control-flow analysis communities use similar techniques [23]. However, the relationship between the techniques is often obscured by differing terminology and presentation styles.

The brief overview of the two analyses given here illustrates their differences. However, the mapping given in the next section illustrates their similarities. By showing that the gap between pointer analysis of first-order languages and control-flow analysis of higher-order languages is even narrower than once thought, this allows for further applications of the large amount of research that has gone into pointer analysis to be applied to control-flow analyses.

### 3.2.3 Pointer Analysis

Pointer analysis is one of the most fundamental static analyses with a broad range of applications. It is used by traditional optimizing compilers and by applications involved with program verification, bug finding, refactoring, and security.

Pointer analyses can change based on the desired precision. There always exists a trade-off between the speed, scalability, and precision of any analysis. Extensions also exist for handling specific language features. In this chapter, we will stick to using a very basic pointer analysis, though extended to handle very basic pointer arithmetic in order to handle fields of structures, whose importance will be demonstrated later.

We give a brief overview of a pointer analysis, describing the statements that are supported and the constraints that are generated from those statements.

#### 3.2.3.1 Pointer Statements

The pointer analysis tools we used were developed to analyze C programs. These tools only consider pointer statements, disregarding the other statements of the program. There are five pointer statements that are supported: assigning the address of a variable, copying

a pointer, dereferencing a pointer, assigning to dereferenced pointer, and simple pointer arithmetic.

$$x, y \in \mathsf{Var} ::= \text{a finite set of variables}$$
$$o \in \mathsf{Int} ::= \text{a finite set of integers}$$

$$x = \&y$$
$$x = y$$
$$x = *y$$
$$*x = y$$
$$x = y + o$$

The basic pointer arithmetic allows us to handle structures. Some pointer analysis algorithms "collapse" a structure into a single variable, but this comes at the cost of too much precision [58]. Other algorithms treat each field as a separate field based on offset and size. While this is not portable because the memory layout of structures is implementation dependent, the analysis is still correct as long as pointer arithmetic is used strictly for accessing fields of structures, and not used in other parts of the program to access arbitrary parts of memory. This is sufficient for our needs since the only pointer arithmetic used in our encoding is to dereference fields.

#### 3.2.3.2 Pointer Set Constraints

Now that we know what pointer statements we can handle, we will answer the basic question of how can we figure out which pointers point to what. A pointer analysis is usually formulated as a set-constraint problem. An analysis will iterate over the statements of the program, generating set constraints for each statement. These set constraints define the points-to sets $pts(x)$ for each variable $x$ in the program.

The following constraints are those generated by Andersen in his style of analysis [2]. In these constraints, $loc(v)$ represents the memory location denoted by $v$.

| | |
|---|---|
| $x = \&y$ | $loc(y) \in pts(x)$ |
| $x = y$ | $pts(y) \subseteq pts(x)$ |
| $x = *y$ | $\forall v \in pts(y) : pts(v) \subseteq pts(x)$ |
| $*x = y$ | $\forall v \in pts(x) : pts(y) \subseteq pts(v)$ |
| $x = y + o$ | $\{v + o : v \in pts(y)\} \subseteq pts(x)$ |

All the rules are generally straightforward. For a pointer dereference, we are stating that everything we could possibly point to is also pointed to by the variable we are assigning. For assigning to a pointer dereference, we are stating that everything that we could point to also points to what is pointed to on the right-hand side.

A pointer analysis can be flow-sensitive or flow-insensitive. A flow-sensitive analysis takes into account the order of statements in a program. A pointer analysis can also be context-sensitive or context-insensitive. A context-sensitive analysis takes into account the calling context (where the function was called) of the function that contains the statements we are analyzing. In practice, context-sensitivity and flow-sensitivity are too expensive and as such the tools we used in our evaluation are context-insensitive and flow-insensitive.

### 3.2.4 Encoding

Looking at the inference rules for pointer analysis and the inference rules for a control-flow analysis, we can already see the similarities between them. This section further explores these similarities.

This section describes how we can take a lambda calculus expression and encode it into pointer statements that can then be analyzed using a points-to analysis. Taking into account how we map lambda calculus expressions into pointer variables, and being able to reverse this mapping, allows us to use the results of the pointer analysis and convert them into a result for our control-flow analysis.

### 3.2.5 Analysis Compilation

Here we describe how to take a program written in lambda calculus and encode it into a C program which will compile and on which you could perform a points-to analysis, but which does not preserve the meaning of the program. The results of a points-to analysis run on this program, given sufficient support for structures, and the results of a control-flow analysis would be equivalent.

This conversion is more for illustrative purposes in order to get a better intuition on how the inference rules given afterwards work.

The ability of the analysis to handle pointer arithmetic, and thus structures precisely, allows us to create a relationship between variables. This allows the correlation that is needed between the variable that represents a given lambda term's parameter and the variable that represents its body.

Create a struct to enable deconstructing a lambda term.

```
struct lambda {
```

```
  struct lamdba *var;
  struct lambda *body;
};
```

Create a variable of type struct lambda for every lambda term that appears in the lambda calculus program, giving each lambda term a unique variable name.

```
struct lambda lam;
```

Create a variable of type struct lambda * for every function application and lambda term that appears in the lambda calculus program. Each expression needs a unique variable name.

```
struct lambda *exp;
```

We do not need to create a pointer for variables because they will be handled by the var pointer inside the structure for the lambda term that binds the variable. We will use this variable whenever we have a variable references that appears in another expression.

For every lambda term in the program, we need to assign the pointer for that lambda term to point to the structure for that same lambda term.

```
exp = &lam;
```

For every call site in the program there are three subexpressions and thus three pointer variables in our translated program: the pointer for the call itself, the pointer for the function, and the pointer for the argument. We need an assignment to state that the variable of the lambda term that is pointed to is bound to point to the same things as the argument. We also need an assignment to state that whatever the body of the lambda term being applied points to is also pointed to by the pointer representing the call site.

```
e1->var = e2;
exp = e1->body;
```

#### 3.2.5.1 Example

To make the previous transformations more explicit, we will convert the following program which is both a valid lambda calculus expression and valid Racket program.

$$((\lambda\ (x)\ (x\ x))\ (\lambda\ (y)\ (y\ y)))$$

The resulting C code would be the following. The above example has two lambda terms, so we create lamx and lamy and pointers lam1 and lam2. We assign these pointers to point to the structures. There are three call expressions, so we create pointers call1, call2, call3. We create the six assignment statements associated with these calls.

```
struct lambda lamx;
struct lambda lamy;

struct lambda *call1;
struct lambda *call2;
struct lambda *call3;

struct lambda *lam1;
struct lambda *lam2;

lam1 = &lamx;
lam2 = &lamy;

lam1->var = lam2;
call1 = lam1->body;

lamx.var->var = lamx.var;
call2 = lamx.var->body;

lamy.var->var = lamy.var;
call3 = lamy.var->body;
```

These statements can be converted into the simple pointer statements referenced earlier by using both intermediate variables and pointer arithmetic to access the fields of structures.

### 3.2.6 Inference Rules

We will now slightly simplify the above conversion by going directly to the simple pointer statements. The following rules describe how to encode a lambda calculus expression into pointer statements. The function $\mathcal{L} : \mathsf{Exp} \rightarrow \mathsf{Var}$ maps expressions to a unique variable. The variables need to be laid out in memory such that for a lambda term $(\lambda\ (v)\ e)$, where $a = \mathcal{L}(v)$ and $b = \mathcal{L}(e)$, $loc(b) = loc(a) + 1$. For references to the same variable in different parts of the program, $\mathcal{L}$ will map it to the same variable.

$$\frac{(\lambda\ (v)\ e) \in E \qquad x = \mathcal{L}((\lambda\ (v)\ e)) \qquad y = \mathcal{L}(v)}{x = \&y}$$

$$\frac{(e_1\ e_2) \in E \qquad x = \mathcal{L}(e_1) \qquad y = \mathcal{L}(e_2)}{*x = y}$$

$$\frac{(e_1\ e_2) \in E \qquad x = \mathcal{L}((e_1\ e_2)) \qquad y = \mathcal{L}(e_1)}{x = *y + 1}$$

Some of these forms are not one of the five pointer statements described as being supported by the tools we evaluated. However, it is easy to see how we can construct them

using intermediate variables. For example, we can change the single statement $x = *y + 1$ into the two statements $y_p = *y$ and $x = y_p + 1$.

Going back to our earlier example program.

$$((\lambda\ (x)\ (x\ x))\ (\lambda\ (y)\ (y\ y)))$$

Assume we have the following mapping from expressions to variable names.

$$\begin{aligned}
l_1 &= \mathcal{L}((\lambda\ (x)\ (x\ x))) \\
l_2 &= \mathcal{L}((\lambda\ (y)\ (y\ y))) \\
x &= \mathcal{L}(x) \\
y &= \mathcal{L}(y) \\
c_1 &= \mathcal{L}(((\lambda\ (x)\ (x\ x))\ (\lambda\ (y)\ (y\ y)))) \\
c_2 &= \mathcal{L}((x\ x)) \\
c_3 &= \mathcal{L}((y\ y))
\end{aligned}$$

We would then generate the following pointer statements.

$$\begin{aligned}
l_1 &= \&x \\
l_2 &= \&y \\
*l_1 &= l_2 \\
c_1 &= *l_1 + 1 \\
*x &= x \\
c_2 &= *x + 1 \\
*y &= y \\
c_3 &= *y + 1
\end{aligned}$$

### 3.2.7 Equivalence of Constraints

Recall that we are trying take a lambda calculus expression, convert it into pointer statements, run a pointer analysis on these statements, and then use those results as a solution to a control-flow analysis of our original lambda calculus expression. We will now go through these steps and demonstrate that the solution generated is equivalent if we were to use the original constraint-based formulation of Palsberg.

In the Palsberg constraints, there are three inference rules. We will examine each of these rules. We will take the lambda calculus expression and convert it into the equivalent pointer

statements. We will then generate the Andersen constraints from those pointer statements and show how those constraints are equivalent to the ones generated by Palsberg.

For the control-flow analysis, we generate constraints and find the least fixed point that satisfies the constraints, building up the set $\mathit{flows}[\![e]\!]$ for each expression $e$. For the points-to analysis, we also generate constraints and find the least fixed point that satisfies the constraints, building up the set $\mathit{pts}(x)$ for each variable $x$.

We need a way to deconstruct the results of the pointer analysis and convert them into useful results for our control-flow analysis. The key to this mapping is the labeling function $\mathcal{L} : \mathsf{Exp} \to \mathsf{Var}$ which we need to maintain certain properties in its mapping.

The labeling function assigns each expression a unique variable. In the mapping, given $x = \mathcal{L}(\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)})$, the result $\mathit{loc}(x)$ will represent the lambda term $\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)}$. This means that if $x$ is in the set of objects pointed to by a pointer, the lambda term is in the flow set of the expression that maps to that pointer.

**Theorem 2.** *Given* $x = \mathcal{L}(\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)})$ *we have*

$$\mathit{pts}(x) = \mathit{pts}(\mathcal{L}(\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)})) = \mathit{flows}[\![\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)}]\!]$$

We also require the property of the labeling function that if $x = \mathcal{L}(\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)})$ and $y = \mathcal{L}(e)$, that the address of $x$ be one greater than the address of $x$.

Case $\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)}$:

Given the labeling $x = \mathcal{L}(\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)})$ and $y = \mathcal{L}(v)$, we generate the pointer statement $x = \&y$. This generates the constraint $\mathit{loc}(y) \in \mathit{pts}(x)$. The location of $y$ is equal to the lambda term in our representation. The points-to set of $x$ is equal to the flow set of the lambda term.

$$\begin{aligned}
\mathit{loc}(y) &\in \mathit{pts}(x) \\
\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)} &\in \mathit{pts}(x) \\
\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)} &\in \mathit{pts}(\mathcal{L}(\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)})) \\
\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)} &\in \mathit{flows}[\![\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)}]\!]
\end{aligned}$$

This generates the desired constraint.

$$\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)} \in \mathit{flows}[\![\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)}]\!]$$

Case $\texttt{(}e_1\ e_2\texttt{)}$:

Given $x = \mathcal{L}(e_1)$ and $y = \mathcal{L}(e_2)$, we would generate the pointer statement $*x = y$. This generates the constraint $\forall v \in pts(x) : pts(y) \subseteq pts(v)$. Because $\mathit{flows}[\![e_1]\!] = pts(x)$, this generates the desired constraint.

$$\forall (\lambda\ (v)\ e) \in \mathit{flows}[\![e_1]\!] : \mathit{flows}[\![e_2]\!] \subseteq \mathit{flows}[\![v]\!]$$

Case $(e_1\ e_2)$:

Given $x = \mathcal{L}(e_1)$ and $y = \mathcal{L}(e_2)$, we would generate the pointer statement $x = *x + 1$, which we would split into the pointer statements $p = *y$ and $x = p + 1$. The first statement generates the following constraint.

$$\forall v \in pts(y) : pts(v) \subseteq pts(p)$$

This gives us all the lambda terms pointed to by $y$ because $\forall v \in pts(y)$. The points-to set of $p$ is going to contain at least the values pointed to by $y$ by this constraint, but since this is the only location where $p$ is assigned, $pts(v) = pts(p)$.

The second statement generates the following constraint.

$$\{v + 1 : v \in pts(p) \subseteq pts(x)\}$$

The expression $v + 1$ gives the body of a lambda term. Because $pts(x) = pts(\mathcal{L}(e_1\ e_2)) = \mathit{flows}[\![(e_1\ e_2)]\!]$, we generate the following original constraint

$$\forall (\lambda\ (v)\ e_b) \in \mathit{flows}[\![e_1]\!] : \mathit{flows}[\![e_b]\!] \subseteq \mathit{flows}[\![(e_1\ e_2)]\!]$$

### 3.2.8 EigenCFA: A Point for Comparison

One possible intermediate representation for compilers of functional languages is continuation-passing style [3]. Given a language in continuation-passing style we will demonstrate how the encoding changes. We do this because this is the language form that is accepted by EigenCFA, as used by our benchmarks in Section 3.2.11.

### 3.2.9 Pointer Statement Encoding of CPS

Encoding a program in continuation-passing style into pointer statements uses the following inference rules.

$$\frac{e = (\lambda\ (v_0 \ldots v_n)\ \mathit{call}) \in E \qquad x = \mathcal{L}(e) \qquad y = \mathcal{L}(v_0)}{x = \&y}$$

$$\frac{(f\ æ_0 \ldots æ_n) \in E \qquad x = \mathcal{L}(f) \qquad y = \mathcal{L}(æ_i)}{*x + i = y}$$

We will now demonstrate how this encoding results in fewer statements and variables. Luckily, our example program from before is already in continuation-passing style.

$$((\lambda\ (x)\ (x\ x))\ (\lambda\ (y)\ (y\ y)))$$

For this program, we generate the following pointer statements:

$$l_1 = \&x$$
$$l_2 = \&y$$
$$*l_1 + 0 = l_2$$
$$*x + 0 = x$$
$$*y + 0 = y$$

Generating the pointer statements is quite simple, and since we do not need to worry about the body of lambda terms, results in fewer statements.

### 3.2.10 Alternative Encoding

We can forgo creating a variable for each lambda term if we deconstruct directly when we have a lambda term in function position. A lambda term in function position is a let form and we know directly which variables we are binding, so the dereference to the lambda term is unnecessary. This results in more inference rules but results in fewer variables in the encoding.

$$\frac{((\lambda\ (x_0 \ldots x_n)\ call)\ æ_0 \ldots æ_n) \qquad æ_i = (\lambda\ (y \ldots)\ call_y)}{x_i = \&y}$$

$$\frac{((\lambda\ (x_0 \ldots x_n)\ call)\ æ_0 \ldots æ_n) \qquad æ_i = y}{x_i = y}$$

$$\frac{(f\ æ_0 \ldots æ_n) \qquad æ_i = (\lambda\ (y \ldots)\ call)}{*f + i = \&y}$$

$$\frac{(f\ æ_0 \ldots æ_n) \qquad æ_i = y}{*f + i = y}$$

### 3.2.11 Implementation

In this section, we explore how much we can improve upon the state of the art of higher-order control-flow analysis with this mapping. It turns out that a constraint based pointer analysis tool runs a lot faster than EigenCFA, a state of the art control-flow analysis tool. EigenCFA is a lot faster than traditional control-flow analysis tools, but even it is outperformed by an optimized pointer analysis tool.

We evaluate EigenCFA as well as two recent parallel pointer analysis tools for C. We compare the following three tools.

- **EigenCFA**

  A GPU implementation that accelerates a control-flow analysis for higher-order languages, operating on the simple binary CPS language [46]. It encodes the analysis as matrix operations on sparse matrices.

- **GPU Inclusion-based Points-to Analysis**

  A GPU implementation that accelerates an inclusion-based points-to analysis [33]. It is based on a graph algorithm that monotonically grows the graph based on the constraints generated by the pointer statements.

- **CPU Inclusion-based Points-to Analysis**

  An inclusion-based points-to analysis that runs in parallel using multiple threads on the CPU [34]. It uses the same graph algorithm as the previous tool.

For the GPU tools, we ran them under Ubuntu on a Nvidia GTX-480 "Fermi" GPU with 1.5 GB of memory and the latest Nvidia drivers. We ran the parallel CPU tool on an machine running Mac OS X 10.8 with two Intel Xeon 3.07 Ghz processors, each having 6 cores, and 64 GB of memory.

We ran each tool on the benchmarks from the EigenCFA paper. To run the pointer analysis tools, we first ran the programs through our encoding and changed the input to be compatible with their tools.

The running times of the tools can be found in Table 3.1. For EigenCFA the results are similar to those as from the original paper, though slightly slower. It is interesting to note that as the programs get large, the points-to analysis tools actually scale better than the original analysis.

To demonstrate how well the CPU scales with more threads, we ran a various number of threads. In Table 3.2, we see the running times for each of the values. The top row is

**Table 3.1**: The running time in milliseconds for each of the implementations explored. The first column is the number of terms found in the benchmark. We show the running times of running the CPU pointer analysis with one thread (CPU-1) and with 12 threads (CPU-12).

| terms | EigenCFA | Pointer GPU | CPU-1 | CPU-12 |
|---|---|---|---|---|
| 297 | 0.4 | 21 | 94 | 90 |
| 545 | 0.7 | 28 | 111 | 94 |
| 1,041 | 1.2 | 39 | 135 | 103 |
| 2,033 | 3 | 62 | 180 | 126 |
| 4,017 | 9 | 148 | 256 | 175 |
| 7,985 | 37 | 291 | 350 | 232 |
| 15,921 | 143 | 531 | 580 | 449 |
| 31,793 | 6367 | 1,317 | 784 | 836 |
| 63,537 | 3,709 | 4,030 | 1,578 | 1,452 |
| 127,025 | 31,228 | 12,175 | 3,819 | 2,557 |
| 190,513 | 142,162 | 40,881 | 13,615 | 4,349 |

**Table 3.2**: The running time in milliseconds for each of the benchmarks on the multi-threaded CPU implementation. This is to demonstrate how well the running time scales with the number of threads for the given benchmarks.

| | terms | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| threads | 297 | 545 | 1,041 | 2,033 | 4,017 | 7,985 | 15,921 | 31,793 | 63,537 | 127,025 | 190,513 |
| 1 | 94 | 111 | 135 | 180 | 256 | 350 | 580 | 784 | 1,578 | 3,819 | 13,615 |
| 2 | 91 | 107 | 129 | 167 | 247 | 322 | 495 | 767 | 1,796 | 3,812 | 12,216 |
| 4 | 89 | 99 | 115 | 140 | 192 | 280 | 475 | 692 | 1,315 | 2,501 | 6,997 |
| 8 | 86 | 93 | 107 | 127 | 180 | 230 | 449 | 820 | 927 | 2,848 | 5,293 |
| 10 | 85 | 93 | 103 | 129 | 170 | 229 | 467 | 823 | 1,176 | 2,765 | 4,833 |
| 12 | 90 | 94 | 103 | 126 | 175 | 232 | 449 | 836 | 1,452 | 2,557 | 4,349 |

the number of terms in the program. The columns are the run times in milliseconds. As we go down the column, the number of threads increase.

From this we observe that there is actually a large improvement in run time for running an analysis on the CPU rather than on the GPU. This likely means that for the given benchmarks, there is not parallelism that can be effectively exploited on a GPU. The GPU implementation visits every call site on every iteration, while the CPU implementation is able to more intelligently visit constraints. It will only visit constraints if they will add new values to the points-to sets of variables.

### 3.2.12 Future Work

There exist several avenues where this work could be extended.

#### 3.2.12.1 Pushdown Analysis

A major recent development in control-flow analysis has been pushdown analyses [57, 12]. This allows calls and returns to be precisely matched and has shown gains in precision. We believe it is possible to do a form of these analyses using our approach.

#### 3.2.12.2 Flow and Context Sensitivity

The pointer analysis tools we explored are flow- and context insensitive. Because of this, we do not preserve any flow or context information in our transformation. However, if we had a tool that took advantage of these features, it would be ideal if our transformation could preserve this information. It has been shown though that flow sensitivity does not add much precision for Racket and other Scheme-like languages because there is not much mutation [4]. However, if this approach was applied to languages that use mutation more commonly (such as Javascript), preserving flow sensitivity would likely be beneficial.

#### 3.2.12.3 Language Compilation

In is common to compile languages into other languages. If tools exists that analyze the target language but not the source language in which we are working, a flavor of this technique also applies. We could perform the desired analysis on the compiled language. Whether this would be useful or not would be highly dependent on our needs. If we need the analysis to provide information about our language in its original form, before we compiled it, we would need to ensure that the compilation process allows for decompilation and that the properties we are hoping to discover are not lost in the compilation process.

#### Additional Language Features

We have demonstrated how our technique works for the simple lambda calculus. However, mapping functional languages and specifically dynamic languages to the lambda calculus is nontrivial [45, 18]. In fact, we would recommend against performing this mapping solely to use our technique, as it is likely not to produce useful information. An alternative interesting approach worth investigating would be to see how this technique could be adapted in the presence of additional language features. How the additional language features are handled would be dependent on the information we wish our analysis to produce.

### 3.2.13 Related Work

One of the earliest works of a constraint-based formulation of control-flow analysis is Henglein's simple closure analysis [19]. It is based on unification and runs in almost linear time. Subsequent work applied a similar strategy to a points-to analysis, citing Henglein as an inspiration. This type of analysis is known as Steensgaard points-to analysis [50].

Control-flow analysis was also later developed in constraint form with the development of Palsberg [42]. This framework is based on subsets, rather than unification, and as such is more precise but is now cubic. At the same time, a similar formulation was developed for pointer analysis of C programs by Andersen [2].

### 3.2.14 Conclusion

This chapter demonstrates to static analysts, analyzing higher-order languages, a way to use existing tools by mapping their problem to ones that have already been solved with significant engineering effort behind them.

In this chapter, we have demonstrated that a pointer analysis of a first-order language can be used to solve a control-flow analysis of a higher-order language, leveraging the significant effort that has gone into the state of the art of pointer analysis. We provided the inference rules to do this and demonstrated how these result in the same constraints as the control-flow analysis. We then demonstrated that we can effectively take advantage of existing pointer analysis tools for significant speed-ups.

## 3.3 CFA via SAT Solvers

Control-flow analyses statically determine the control-flow of programs. This is a nontrivial problem for higher-order programming languages. This work attempts to leverage the power of SAT solvers to answer questions regarding control-flow. A brief overview

of a traditional control-flow analysis is presented. Then, an encoding is given which has the property that any satisfying assignment will give a conservative approximation of the true control-flow, along with additional ideas to improve the precision and efficiency of the encoding. The results of the encodings are then compared to those of a traditional implementation on several example programs. This approach is competitive in some instances with hand-optimized implementations. Finally, the chapter concludes with a discussion of the implications of these results and work that can build upon them.

### 3.3.1 Introduction

We present an alternative approach to the problem by encoding a control-flow analysis into SAT. The results are more similar to 0CFA than $k$-CFA as SAT is a NP-hard problem, while $k$-CFA is EXPTIME-hard. Similar work that took the idea of encoding $k$-CFA into another problem for performance reasons was done by Prabhu et al. [46]. They run the analysis on a GPU by encoding the problem into matrix operations. Another work that will feel similar to the work presented in this chapter is constraint-based 0CFA analysis as summarized by Nielson [41]. They formulate 0CFA using constraints on sets and then provide an algorithm for solving these constraints. This work differs in that the constraints are not encoded using matrices or sets, but propositional logic.

### 3.3.2 Motivation

Many problems are readily encoded into SAT and even though satisfiability is NP-complete, fast implementations are available. Every year, there is considerable work being done to create efficient SAT solvers. A CFA implementation based on satisfiability could benefit directly from that work.

### 3.3.3 Accomplishments

This work attempts to leverage the power of SAT solvers to answer questions regarding control-flow. It presents an encoding and compares its results to two traditional 0CFA implementations.

### 3.3.4 Encodings

This section describes the devised encoding scheme. Here is a simple program we will work with in describing the encodings. In the following explanation, each lambda term will be identified by its label.

```
((lambda¹ (x)
   ((lambda² (y)
```

```
      (y (lambda³ (z) (x z)))) x))
(lambda⁴ (a) (a a)))
```

For the encoding, we introduce a variable for every variable lambda pair in the program. The variable will be true if the lambda flows to the variable, and false if it does not. We will assume that the program has been alphatised, meaning that each variable is only bound by a single lambda. In the example, we have four variables and four lambda terms, resulting in 16 variables. Lambdas use their label as their subscript.

| | $\lambda_1$ | $\lambda_2$ | $\lambda_3$ | $\lambda_4$ |
|---|---|---|---|---|
| $a$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ |
| $x$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ |
| $y$ | $y_1$ | $y_2$ | $y_3$ | $y_4$ |
| $z$ | $z_1$ | $z_2$ | $z_3$ | $z_4$ |

To generate the clauses of our encoding, we look at each point where binding occurs in lambda calculus, at application sites. From the grammar of CPS lambda calculus, we can see that there are four cases which need to be considered. The function and the arguments at an application can either be a lambda term or a variable.

#### 3.3.4.1 Case 1: Lambda Lambda

The first case to consider is the simplest, when there is a lambda term in both function and argument position. The top level application of the sample program is an example of this.

```
((lambda¹ (x) call) (lambda⁴ (a) (a a)))
```

We know that the lambda in argument position flows to the parameter of the lambda in function position. For this call site, we would add the clause $x_4$.

#### 3.3.4.2 Case 2: Lambda Variable

The second case to consider is when there is still a lambda in function position but a variable in argument position. Observe the following call site from the example.

```
((lambda² (y) call) x)
```

If we know a lambda flows to $x$, then we know that it must flow to $y$. We must assume that any lambda can flow to x, so we must create a clause for each lambda. This results in the following clauses: $x_1 \rightarrow y_1$, $x_2 \rightarrow y_2$, $x_3 \rightarrow y_3$, $x_4 \rightarrow y_4$.

#### 3.3.4.3 Case 3: Variable Lambda

The third case to consider is having a variable in function position and a lambda term in argument position. Observe the following call site from the example.

```
(y (lambda^3 (z) call))
```

We must assume that any variable can flow to $y$. Thus, we need to create a clause for each lambda in the program. We infer that if a lambda term flows to $y$, then $\lambda_3$ will flow to the parameter of that lambda. This results in the following clauses: $y_1 \rightarrow x_3$, $y_2 \rightarrow y_3$, $y_3 \rightarrow z_3$, $y_4 \rightarrow a_3$.

#### 3.3.4.4 Case 4: Variable Variable

The most complicated case is when we have a variable in both function and argument position. Observe the following call site from the example program.

```
(x z)
```

We must assume that any lambda can flow to $x$ and any lambda can flow to $z$. If we know that two flows are true for $x$ and $z$, we can infer a third flow. For example, if we know $\lambda_2$ flows to $x$ and $\lambda_4$ flows to $z$, we can infer that $\lambda_4$ flows to $y$, the parameter of $\lambda_2$. Thus, we create the clause $x_2 \wedge z_4 \rightarrow y_4$. Since there are four lambda terms, there are 16 total such clauses that need to be generated.

### 3.3.5 Additional Encoding Details

The generated clauses described above are necessary but not sufficient. The problem is that every variable can be set to true and the formula is still satisfied. What we really want is the lowest possible number of flows set to true that still satisfy all the generated clauses. However, the SAT solver is free to give any satisfying solution. In the end, we have constraints that will never give us false negatives, but we need constraints that will ideally never give us false positives, or at least limit them. Note that in an analysis, having false positives is still sound; only in having false negatives does the analysis become unsound.

### 3.3.6 Additional Encodings

For each case, we will show additional clauses that can be added which will limit the number of false positives.

#### 3.3.6.1 Case 1: Lambda Lambda

Since the program is alphatised, we not only know that the given flow must be true, but we know that all other flows to that variable must be false. For the above example, we add the clauses: $\neg x_1$, $\neg x_2$, $\neg x_3$.

#### 3.3.6.2 Case 2: Lambda Variable

In the description found above, we said you could infer an additional flow if a given lambda flows to the variable in argument position. However, more can be inferred since the program is alphatised. The clauses are not just implications, because the call site is the only place where the binding of the variable can occur. Thus, we can change the clauses to equivalences: $x_1 \leftrightarrow y_1$, $x_2 \leftrightarrow y_2$, $x_3 \leftrightarrow y_3$, $x_4 \leftrightarrow y_4$.

#### 3.3.6.3 Case 3: Variable Lambda

Unlike the previous case, we cannot turn the inference described in the previous section for case 3 into an equivalence. The issue is that because the lambda which flows to the variable in function position can flow to other application sites where there is a variable in function position, this is not the only place where a binding can occur. However, we can infer the disjoin of all the call sites where the binding could occur. An example will be given below.

#### 3.3.6.4 Case 4: Variable Variable

Much like the previous case, we cannot infer equivalences because bindings can happen at any call site where there is variable in function position. However, like the above case, additional clauses can still be created; we can infer the disjoin of all the call sites where the binding could occur. For example, if $\lambda_3$ flows to $z$, it would mean that either $\lambda_3$ flows to $y$, $\lambda_3$ flows to $a$, or that $\lambda_3$ flows to $x$ and $\lambda_3$ flows to $z$. Thus, we would add the following clause: $z_3 \rightarrow y_3 \vee a_3 \vee (x_3 \wedge z_3)$.

### 3.3.7 Enhancements

The encodings presented above give way to some enhancements that can be used to make the encoding more efficient, by generating less clauses.

- Not all lambdas can flow. Lambdas that appear in function position cannot be bound to variables; thus, we do not need to create a variable for pairs involving lambdas in function position.

- Not all lambdas are compatible. Although the example shows lambda terms with only one parameter, the lambda terms can have any number of parameters. When there is a variable in function position, only lambdas with the same number of parameters as there are arguments at the application site need to be considered.

- Some clauses will be trivially true. While iterating through every lambda, when faced with a variable at an application site, some of the implications will involve the same pairs on both sides, thus they are trivially true and can be omitted.

In the implementation, the first two enhancements were used, but the third was omitted.

### 3.3.8 Complexity

In the described encoding, many clauses can be generated. However, it is bounded by a polynomial of the size of the program. The worst case to consider is when you have a variable in both function and argument position. You must consider each lambda flowing to each variable. If there are $n$ terms in the program, there are at most $n$ call sites and $n$ lambda terms. Thus, the number of generated clauses will be bound by $n^3$. This seems logical as one of the simplest formulations of 0CFA is "nearly" cubic: $O(n^3/\log n)$ [36].

### 3.3.9 Implementation and Evaluation

We implemented the encoding in Scala using the back end of the analyzer written by Might et al. for parsing and preprocess transformations [40]. We compared its runtimes to those of that same analyzer, which closely follows the formal semantics, as well as a fast Racket implementation, which employs abstract Church encodings and binary CPS lambda calculus [46]. MiniSat was used for solving the constructed encodings. All experiments were run on a 2.7 GHz Intel Core i7 on Ubuntu.

The first experiments were run on synthetic programs, which in a *constructive* complexity proof are shown to be the worst case for $k$-CFA when $k \geq 1$ and difficult for 0CFA [53, 54]. The results can be found in Table 3.3. The first column is the number of terms in the program. The second column is the runtime of the optimized Racket implementation. The Scala column is the runtime of the traditional Scala implementation. The SAT column is the time taken to encode and solve the problem using SAT. This column is broken down into its two components in the last two columns. The Encode column is the time taken to create the encoding. The Solve column is the time taken by MiniSat to solve the encoding.

We also looked into the sensitivity of the encoding to different SAT solvers, using SAT solvers that were some of the best performers from the 2011 international SAT competition. See Table 3.4. We report the time taken to solve the encoding, the number of flows that agree with the Scala implementation, and the number of flows that disagree. When there is a disagreement, the encoding says that the flow does occur but the traditional 0CFA reports that it does not.

**Table 3.3**: Runtime comparison of a control-flow analysis using a fast Racket implementation, a Scala implementation, and using MiniSAT.

| **Terms** | **Racket** | **Scala** | **SAT** | **Encode** | **Solve** |
|---|---|---|---|---|---|
| 37 | 0.008s | 1.059s | 0.730s | 0.725s | 0.005s |
| 63 | 0.016s | 1.056s | 0.796s | 0.792s | 0.004s |
| 115 | 0.046s | 1.454s | 1.025s | 1.017s | 0.008s |
| 219 | 0.222s | 2.338s | 1.418s | 1.387s | 0.031s |
| 427 | 1.374s | 5.337s | 2.759s | 2.642s | 0.117s |
| 843 | 8.396s | 44.873s | 11.337s | 10.481s | 0.856s |
| 1675 | 49.029s | 12m34.301s | 1m15.984s | 1m9.222s | 6.762s |
| 3339 | 4m46.726s | >6h | 8m50.671s | 8m43.103s | 7.568s |

**Table 3.4**: Runtime and precision results from some of the best performers from the 2011 international SAT competitions.

| Solver | Results | $n = 37$ | $n = 63$ | $n = 155$ | $n = 219$ | $n = 237$ | $n = 843$ | $n = 1675$ |
|---|---|---|---|---|---|---|---|---|
| minisat | Time | 0.005s | 0.007s | 0.012s | 0.039s | 0.133s | 0.848s | 6.714s |
| | Agree | 96 | 280 | 936 | 3400 | 12936 | 50440 | 199176 |
| | Disagree | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3S | Time | 2.548s | 2.570s | 2.554s | 2.777s | 5.952s | 1m15.335s | >2m |
| | Agree | 96 | 280 | 936 | 3400 | 12936 | 50440 | - |
| | Disagree | 0 | 0 | 0 | 0 | 0 | 0 | - |
| cirminisat | Time | 0.004s | 0.005s | 0.009s | 0.031s | 0.152s | 1.312s | 11.299s |
| | Agree | 96 | 280 | 936 | 3400 | 12936 | 50440 | 199176 |
| | Disagree | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| clasp | Time | 0.004s | 0.005s | 0.010s | 0.029s | 0.152s | 1.055s | 7.959s |
| | Agree | 54 | 150 | 486 | 1734 | 6534 | 25350 | 165378 |
| | Disagree | 42 | 130 | 450 | 1666 | 6402 | 25090 | 33798 |
| cryptominisat // | Time | 0.007s | 0.011s | 0.026s | 0.086s | 0.413s | 3.598s | >2m |
| | Agree | 96 | 280 | 936 | 3400 | 12936 | 50440 | - |
| | Disagree | 0 | 0 | 0 | 0 | 0 | 0 | - |
| csls // | Time | 0.006s | 0.006s | 0.034s | 0.579s | 32.656s | >2m | >2m |
| | Agree | 60 | 216 | 711 | 2754 | 12936 | - | - |
| | Disagree | 36 | 64 | 225 | 646 | 0 | - | - |
| eagleup | Time | 0.004s | 0.006s | 0.017s | 0.063s | 0.541s | 18.500s | >2m |
| | Agree | 70 | 192 | 674 | 2566 | 9479 | 36639 | - |
| | Disagree | 26 | 88 | 262 | 834 | 3457 | 13801 | - |
| glucose | Time | 0.011s | 0.012s | 0.020s | 0.050s | 0.198s | 1.415s | 4.805s |
| | Agree | 96 | 280 | 936 | 3400 | 12936 | 50440 | 199176 |
| | Disagree | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| glueminisat | Time | 0.004s | 0.005s | 0.011s | 0.036s | 0.164s | 1.363s | 11.640s |
| | Agree | 96 | 280 | 936 | 3400 | 12936 | 50440 | 199176 |
| | Disagree | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| lingeling | Time | 0.006s | 0.010s | 0.024s | 0.078s | 0.454s | 1.980s | 10.365s |
| | Agree | 96 | 280 | 936 | 3400 | 12936 | 50440 | 199176 |
| | Disagree | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| march_rw | Time | 0.007s | 0.010s | 0.033s | 0.466s | 18.936s | >2m | >2m |
| | Agree | 54 | 150 | 486 | 1734 | 6534 | - | - |
| | Disagree | 42 | 130 | 450 | 1666 | 6402 | - | - |
| plingeling | Time | 0.009s | 0.012s | 0.028s | 0.096s | 0.477s | 2.851s | 19.695s |
| | Agree | 96 | 280 | 936 | 3400 | 12936 | 50440 | 199176 |
| | Disagree | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| plingeling // | Time | 0.010s | 0.017s | 0.037s | 0.083s | 0.465s | 3.551s | 30.653s |
| | Agree | 96 | 280 | 574 | 1992 | 7813 | 30463 | 120112 |
| | Disagree | 0 | 0 | 362 | 1408 | 5123 | 19977 | 79064 |
| ppfolio | Time | 0.006s | 0.009s | 0.010s | 0.051s | 0.341s | 2.453s | 14.588s |
| | Agree | 78 | 280 | 936 | 3400 | 12936 | 50440 | 165378 |
| | Disagree | 18 | 0 | 0 | 0 | 0 | 0 | 33798 |
| ppfolio // | Time | 0.007s | 0.007s | 0.012s | 0.028s | 0.178s | 0.989s | 7.409s |
| | Agree | 92 | 280 | 936 | 3400 | 12936 | 50440 | 165378 |
| | Disagree | 4 | 0 | 0 | 0 | 0 | 0 | 33798 |
| qutersat | Time | 0.035s | 0.042s | 0.061s | 0.134s | 0.638s | 4.786s | 41.886s |
| | Agree | 96 | 280 | 936 | 3400 | 12936 | 50440 | 199176 |
| | Disagree | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| sattime2011 | Time | 0.005s | 0.008s | 0.021s | 0.093s | 0.796s | 10.857s | 0.020s |
| | Agree | 83 | 230 | 805 | 2919 | 11275 | 44001 | - |
| | Disagree | 13 | 50 | 131 | 481 | 1661 | 6439 | - |
| sparrow2011 | Time | 0.013s | 0.007s | 0.029s | 0.100s | 3.034s | >2m | >2m |
| | Agree | 72 | 266 | 936 | 3400 | 10846 | - | - |
| | Disagree | 24 | 14 | 0 | 0 | 2090 | - | - |

From the results in Table 3.3, we see encoding the problem and solving it with MiniSat takes about the same amount of time as the fast Racket implementation. However, this is not always the case. Experiments were also run on more traditional benchmarks. To run these, the language on which the encoding operates had to be enriched. Additional constructs were added (*e.g.*, `if` and `set!`) as well as support for Scheme primitives. The fast Racket implementation could not be run on these examples without using Church encodings, as it only supports pure binary CPS lambda calculus. See Table 3.5.

The first two benchmarks test common functional patterns; sat is a simple SAT solver; rsa is a RSA implementation; prime is a Solovay-Strassen primality tester; scm2java is a Scheme to Java compiler; interp is a Scheme interpreter.

These benchmarks provide a stark contrast to the previous examples in performance. Further investigation is needed to find the source of this large difference in performance. One possible explanation is that the Scheme primitives are not well modelled. Also, the traditional small-step abstract interpreter is able to use widening to converge to the minimum fixed point faster. In addition, since its analysis is directed by the syntax of the program more closely, it can explore less spurious flows.

For the first set of benchmarks, the results returned by the encoding are exactly the same as those provided by the traditional implementations. However, running #SAT on the encodings revealed that there are multiple valid interpretations. Thus, the encoding does not exactly encode traditional 0CFA, which has a unique minimum fixed point.

### 3.3.10 Alternative Approach Using BDDs

Another approach attempted was to use a binary decision diagram (BDD) instead of a SAT solver to solve the constraints. The constraints are encoded in the same way, but the approach has the benefit that the minimum prime implicant is readily available from the structure of the BDD. The minimum prime implicant provides an equivalent solution as 0CFA. However, in practice, using a BDD requires large amounts of memory and time for even simple examples.

### 3.3.11 Alternative Approach Using MaxSAT

Another approach that could be promising is to use a MaxSAT solver instead of a traditional SAT solver. The additional clauses from Section 3.3.5 could be elided and only the clauses from Section 3.3.4 would be needed. The partial maximum satisfiability problem has two types of clauses, hard and soft. The hard clauses must be satisfied, while the soft clauses can be relaxed. The solver finds the assignment with maximum number of soft

**Table 3.5**: Runtime comparison between a traditional abstract interpreter and determining the control-flow using MiniSAT.

| Program | Terms | Scala | SAT | Encode | Solve |
|---|---|---|---|---|---|
| eta | 79 | 0.879s | 0.805s | 0.801s | 0.004s |
| map | 182 | 0.879s | 0.805s | 0.801s | 0.004s |
| sat | 250 | 1.311s | 1.216s | 1.198s | 0.018s |
| rsa | 609 | 1.805s | 1.427s | 1.396s | 0.031s |
| prime | 891 | 2.258s | 4.584s | 4.269s | 0.315s |
| scm2java | 2505 | 3.845s | 1m6.550s | 1m0.090s | 6.460s |
| interp | 4484 | 6.314s | 5m6.519s | 4m26.078s | 40.441s |

clauses satisfied. All the clauses from Section 3.3.4 would be hard clauses and then for each variable, its negation would be added as a soft clause. A satisfying assignment from this formulation would be equivalent to 0CFA.

### 3.3.12 Conclusion

This work has presented an encoding for control-flow analysis of CPS lambda calculus. It has shown that in some cases, the approach can be as fast as a highly optimized solution. While the soundness of the encoding was not proven, empirical results showed it to be accurate.

This work also provides a solid basis for additional work. Many avenues exist which can build upon it. Better encoding schemes can be developed, which possibly could be even more precise than 0CFA, given the extra power provided by SAT solvers being able to solve NP-complete problems. Van Horn and Mairson give a reduction from SAT to $k$-CFA, effectively showing how to do SAT solving with $k > 1$ CFA, which merits further investigation [53]. Also, while this work operates on CPS lambda calculus, the encoding could easily be adapted to work on a more direct style language, such as ANF lambda calculus [15], as analyzed by Might and Prabhu [38].

## 3.4 CFA via Linear Algebra

This chapter provides the third mapping of 0CFA based on constraint solving in this dissertation. It provides a method to solve the constraints using linear-algebra operations.

There are two reasons for doing this. First, it gives one possible method on how to run the analysis on a GPU. GPUs excel at accelerating linear-algebra operations because they have low branching and are data parallel. By encoding the solving of the constraints to linear-algebra operations, it becomes easier to see how to run this on a GPU. Second, it provides a mechanism for something that we want to demonstrate in the following chapter, namely

that the solution generated by this approach is equivalent to the solution that is produced by a pushdown analysis that is monovariant, flow-insensitive, and context-insensitive.

This chapter proceeds by demonstrating how we can encode a lambda calculus term using vectors. In then demonstrates how to represent the abstract syntax tree of a lambda calculus program as matrices that can be indexed using these term vectors. Finally it shows how the constraints of Section 3.1 can be solved using linear algebra operations on these vectors and matrices. It uses vector and matrix multiplication ($\times$), matrix boolean or ($+$), and outer product ($\oplus$).

The lambda calculus expressions are encoded as vectors. There are three types of expressions (lambda terms, function application, and variable reference) and each one can also be represented by a vector.

The length of the vector is the number of lambda calculus expressions $|\mathsf{Exp}|$ in the program that is being analyzed. A unique index into this vector is assigned to each expression. For the vector representing an expression, the entry in the vector at the index of that expression is given a value of 1, while every other position in the vector is given a value of 0. Note that variables have two different roles in a lambda calculus expression. They can be a formal parameter of a lambda term or a variable reference. A variable no matter where it appears in the program, either as a parameter or a reference, is represented by the same vector.

$$\vec{e} \in \overrightarrow{\mathsf{Exp}} = \{0,1\}^{|\mathsf{Exp}|}$$
$$\vec{c} \in \overrightarrow{\mathsf{Call}} \subset \overrightarrow{\mathsf{Exp}}$$
$$\vec{l} \in \overrightarrow{\mathsf{Lam}} \subset \overrightarrow{\mathsf{Exp}}$$
$$\vec{v} \in \overrightarrow{\mathsf{Var}} \subset \overrightarrow{\mathsf{Exp}}$$

The abstract syntax tree is encoded as a matrices. It is encoded as selector matrices that allow for terms encoded as vectors to generate vectors for their subexpressions. **Fun** and **Arg** deconstruct a call site, allowing access to the function and argument. **Var** and **Body** deconstruct a lambda term, allowing access to the variable and body.

$$\mathbf{Var}\colon \overrightarrow{\mathsf{Lam}} \to \overrightarrow{\mathsf{Var}}$$
$$\mathbf{Body}\colon \overrightarrow{\mathsf{Lam}} \to \overrightarrow{\mathsf{Exp}}$$
$$\mathbf{Fun}\colon \overrightarrow{\mathsf{Call}} \to \overrightarrow{\mathsf{Exp}}$$
$$\mathbf{Arg}\colon \overrightarrow{\mathsf{Call}} \to \overrightarrow{\mathsf{Exp}}$$

Given a vector representing a lambda term $\vec{l}$, its argument is given by $\vec{l} \times \mathbf{Var}$ and its body is given by $\vec{l} \times \mathbf{Body}$. Given a vector representing a call site $\vec{c}$, the function is given by $\vec{c} \times \mathbf{Fun}$ and its argument is given by $\vec{c} \times \mathbf{Arg}$.

Solving the constraints builds up flow sets of each expression, as opposed to a store in the abstract interpreter approach which builds up flow sets for addresses. A flow set is also represented by a vector of type $\overrightarrow{\mathsf{Lam}}$. However, rather than representing a single lambda term by having a single bit set, the flow set vector can have multiple bits set. Encoding a set as a bit vector is a common practice where the vector represents a set that contains each element whose bit is set inside the vector.

The flows matrix encodes the flow sets which are the output of the analysis. It maps expressions to their flow sets, where the flow sets is encoded as a vector as described above.

$$\boldsymbol{\phi} \in \mathbf{Flows} \colon \overrightarrow{\mathsf{Exp}} \to \overrightarrow{\mathsf{Lam}}$$

Each expression has a row in the flow matrix that represents its flow set. To get the flows to set of expression $\vec{e}$, it is multiplied by the flow matrix $\vec{e} \times \boldsymbol{\phi}$.

We want to encode the three types of constraints into a function that we could implement in a GPU kernel call.

The first constraint states that lambda terms flow to themselves.

$$\frac{(\lambda\ (v)\ e_b) \in E}{(\lambda\ (v)\ e_b) \in \mathit{flows}[\![(\lambda\ (v)\ e_b)]\!]}$$

This constraint is handled simply be setting the bit for each lambda term in the flow matrix so that it flows to itself. If all the lambda terms were in adjacent rows in the flows to matrix and in the same order as the lambda terms in the columns, this would produce an identity matrix.

The second constraint states that the flow set of an argument of a call site is a subset of the formal parameter of the lambda term applied at that call site.

$$\frac{(e_1\ e_2) \in E \qquad (\lambda\ (v)\ e_b) \in \mathit{flows}[\![e_1]\!]}{\mathit{flows}[\![e_2]\!] \subseteq \mathit{flows}[\![v]\!]}$$

Given the vector for the call site $\langle\langle \mathit{call} \rangle\rangle = (e_1\ e_2)$, the call site is deconstructed through multiplication of the matrices encoding the AST, where $\langle\langle \mathit{call} \rangle\rangle \times \mathbf{Fun} = e_1$ and $\langle\langle \mathit{call} \rangle\rangle \times \mathbf{Arg} = e_2$. After determining the function, its flow set can be found through multiplying it by the flows matrix.

$$\mathit{flows}[\![e_1]\!] = \langle\langle \mathit{call} \rangle\rangle \times \mathbf{Fun} \times \boldsymbol{\phi} = \vec{f}$$

The flow set of the argument can also be found in a similar fashion.

$$\mathit{flows}[\![e_2]\!] = \langle\langle \mathit{call} \rangle\rangle \times \mathbf{Arg} \times \boldsymbol{\phi} = \vec{a}$$

The formal parameter of every lambda term that flows to $e_1$ can be found through the **Var** matrix.

$$\vec{v} = \vec{f} \times \mathbf{Var}$$

Taking the outer product of $\vec{v}$ and $\vec{a}$ creates a flow matrix where the flow set of the argument is now the flow set of every formal parameter of the lambda terms that flow to the function of the call site. The subset relation is enforced by applying boolean or to this and the original store.

$$\boldsymbol{\phi} + (\vec{v} \otimes \vec{a})$$

The third constraint states that the flow set of the body of any lambda term that flows to the function position is a subset of the flow set of the call site.

$$\frac{(e_1\ e_2) \in E \qquad (\lambda\ (v)\ e_b) \in \mathit{flows}[\![e_1]\!]}{\mathit{flows}[\![e_b]\!] \subseteq \mathit{flows}[\![(e_1\ e_2)]\!]}$$

The body of every lambda term that flows to the function position of the call site can be found through **Body** where $e_b = \vec{f} \times \mathbf{Body}$. Furthermore, the flow set of the body can be found by multiplying it by the flow matrix.

$$\mathit{flows}[\![e_b]\!] = \vec{f} \times \mathbf{Body} \times \boldsymbol{\phi} = \vec{b}$$

Taking the outer product of $\langle\langle \mathit{call} \rangle\rangle$ and $\vec{b}$ creates a flow matrix where the flow set of the call is now the flow set of the body. The subset relation can be enforced by taking applying boolean or to this and the original flow matrix.

$$\boldsymbol{\phi} + (\langle\langle \mathit{call} \rangle\rangle \otimes \vec{b})$$

Putting this all together the following function builds up the flow set.

$$\begin{aligned}
f_{\mathit{call}}(\boldsymbol{\phi}) = \boldsymbol{\phi}', \text{ where} \\
\vec{f} &= \langle\langle \mathit{call} \rangle\rangle \times \mathbf{Fun} \times \boldsymbol{\phi} \\
\vec{a} &= \langle\langle \mathit{call} \rangle\rangle \times \mathbf{Arg} \times \boldsymbol{\phi} \\
\vec{v} &= \vec{f} \times \mathbf{Var} \\
\vec{b} &= \vec{f} \times \mathbf{Body} \times \boldsymbol{\phi} \\
\boldsymbol{\phi}' &= \boldsymbol{\phi} + (\vec{v} \otimes \vec{a}) + (\langle\langle \mathit{call} \rangle\rangle \otimes \vec{b})
\end{aligned}$$

Initially setting all the flow sets for the lambda terms and iteratively applying this function until a fixed point is reached computes the minimum flow set.

# CHAPTER 4

# PUSHDOWN CONTROL-FLOW ANALYSIS

We describe a linear-algebraic encoding for pushdown control-flow analysis of higher-order programs. Pushdown control-flow analyses obtain a greater precision in matching calls with returns by encoding stack-actions on the edges of a Dyck state graph. This kind of analysis requires a number of distinct transitions and was not amenable to parallelization using the approach of EigenCFA. Recent work has extended EigenCFA, making it possible to encode more complex analyses as linear-algebra for efficient implementation on SIMD architectures. We apply this approach to an encoding of a monovariant pushdown control-flow analysis.

## 4.1 Introduction

The goal of static analysis is to produce a bound for program behavior before runtime. This is desirable for proving the soundness of code transformations, the absence or programming errors, or the absence of malware.

However, static analysis of higher-order languages such as Scheme is nontrivial. Due to the nature of first-class functions, data-flow affects control-flow and control-flow affects data-flow, resulting in the higher-order control-flow problem. This vicious cycle has resulted in even the simplest of formulations being nearly cubic [52, 54]. However, a trade-off exists in any analysis between precision and scalability, and finding the right balance for a particular application requires special attention and effort [37].

One way to increase the scalability of an analysis is to parallelize its execution. To this end, we provide a linear encoding of a pushdown control-flow analysis, giving potential speedups on many-core or SIMD architectures such as the GPU.

Prabhu et al. demonstrated the possibility of running a higher-order control flow analysis on the GPU [46]. However, their encoding has the major drawback that it only supports binary continuation-passing-style (CPS). It was restricted to a simple language which could be implemented as a single transition rule as not to introduce thread-divergence in SIMD

implementations. Currying all function calls and being forced to encode all language forms and program values in the lambda calculus is not ideal for real applications because it distorts the code under analysis.

Gilray et al. addressed this issue with a demonstration that richer language forms and values can be used within this style of encoding by *partitioning* transfer functions and more precisely encoding analysis components [16]. We build on this work, demonstrating that it is not only possible to encode richer language forms, but a fundamentally richer analysis. Specifically, we demonstrate that a pushdown analysis may also be encoded using this transfer-function partitioning. A pushdown analysis has the benefit that it precisely matches function calls with function returns [57].

In this chapter, we review the concrete semantics of ANF lambda calculus within a CESK machine. We then provide a direct abstraction of the pushdown-machine semantics to a monovariant pushdown control-flow analysis (0-PDCFA). We then partition the transfer function and show a linear encoding of that analysis which is faithful to its original precision.

## 4.2 Concrete Semantics

We give semantics for a pure lambda calculus in Administrative Normal Form (ANF). ANF is a core direct-style language which strictly let-binds all intermediate-expressions [15]. This structurally enforces an order of evaluation and greatly simplifies a formal semantics. ANF is at the heart of common intermediate-representations for Scheme and other higher-order programming languages.

For simplicity, we permit only call-sites, let-forms, and atomic-expressions (variables and $\lambda$-abstractions).

$$
\begin{aligned}
e \in \mathsf{E} &::= (\text{let } (x\ e)\ e)^l \\
&\quad \mid (ae\ ae\ \ldots)^l \\
&\quad \mid ae^l \\
ae \in \mathsf{AE} &::= x \mid lam \\
lam \in \mathsf{Lam} &::= (\lambda\ (x\ \ldots)\ e) \\
x \in \mathsf{Var} &::= \langle \textit{set of program variables} \rangle \\
l \in \mathsf{Label} &::= \langle \textit{set of unique labels} \rangle
\end{aligned}
$$

The concrete semantics for this machine will be given using a CESK machine [14], which has the following state space:

$$
\begin{aligned}
\varsigma \in \Sigma &= \mathsf{E} \times Env \times Store \times Time \times Kont \\
\rho \in Env &= \mathsf{Var} \rightharpoonup Addr \\
\sigma \in Store &= Addr \rightharpoonup Value \\
t \in Time &= \mathsf{Label}^* \\
\kappa \in Kont &= Frame^* \\
\phi \in Frame &= \mathsf{E} \times Env \times \mathsf{Var} \\
a \in Addr &= \mathsf{Var} \times Time \\
v \in Value &= \mathsf{Lam} \times Env
\end{aligned}
$$

Each state in the abstract-machine represents control at a particular expression-context $e$, with a binding environment $\rho$ encoding visible bindings of variables to addresses and a value-store (a model of the heap) mapping addresses to values. Each state is also specific to a timestamp $t$ encoding a perfect program-trace and a current continuation $\kappa$ encoding a stack of continuation frames.

The only values for this language are closures. To generate values given an atomic-expression, we will use an atomic-evaluator. Given a variable, it looks up the address of the value in the environment and then the value in the store. Given a $\lambda$-abstraction, we simply close it over the current environment.

$$
\begin{aligned}
\mathcal{A}\colon \mathsf{AE} \times \Sigma &\rightharpoonup Value \\
\mathcal{A}(x,\ (e,\ \rho,\ \sigma,\ t,\ \kappa)) &= \sigma(\rho(x)) \\
\mathcal{A}(lam,\ (e,\ \rho,\ \sigma,\ t,\ \kappa)) &= (lam,\ \rho)
\end{aligned}
$$

Looking at the grammar for our language, we can see that there are three expression forms: let bindings, applications, and atomic expressions. To fully present the semantics, we will provide a transition relation that has a rule for each form.

The first form we will describe is for let bindings. A let expression pushes a frame on the stack that captures the expression to evaluate when we return, the environment to be used, what variable we will bind, along with the stack as it exists when we push the new frame.

$$
\underbrace{((\text{let } (x\ e)\ e_\kappa)^l,\ \rho,\ \sigma,\ t,\ \kappa)}_{\varsigma} \Rightarrow (e,\ \rho,\ \sigma,\ t',\ \kappa')
$$

$$\begin{aligned} where \quad & \kappa' = (e_\kappa,\ \rho,\ x) : \kappa \\ & t' = l : t \end{aligned}$$

Function calls are a little bit more involved but not too complicated. We evaluate the function we are applying, as well as all the arguments. We create new address and set the values in the store. Note that since these are tail calls the stack is unchanged.

$$\frac{((\lambda\ (x_1\ \ldots\ x_j)\ e),\ \rho_\lambda) = \mathcal{A}(ae_f,\ \varsigma)}{\underbrace{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \rho,\ \sigma,\ t,\ \kappa)}_{\varsigma} \Rightarrow (e,\ \rho',\ \sigma',\ t',\ \kappa)}$$

$$\begin{aligned} where \quad & \rho' = \rho_\lambda[x_i \mapsto (x_i,\ t')] \\ & \sigma' = \sigma[(x_i,\ t') \mapsto \mathcal{A}(ae_i,\ \varsigma)] \\ & t' = l : t \end{aligned}$$

Finally, when we come across an atomic expression, we need to return. We do this by extracting the needed information from the top frame, extend and update the environment, and return to using the previous stack.

$$\frac{\kappa = (e,\ \rho_\kappa,\ x_\kappa) : \kappa'}{\underbrace{(ae^l,\ \rho,\ \sigma,\ t,\ \kappa)}_{\varsigma} \Rightarrow (e,\ \rho',\ \sigma',\ t',\ \kappa')}$$

$$\begin{aligned} where \quad & \rho' = \rho_\kappa[x_\kappa \mapsto (x_\kappa,\ t')] \\ & \sigma' = \sigma[(x_\kappa,\ t') \mapsto \mathcal{A}(ae,\ \varsigma)] \\ & t' = l : t \end{aligned}$$

These semantics may be used to evaluate a program $e$ by producing an initial state $\varsigma_0 = (e, \emptyset, \bot, (), ())$ and computing the transitive closure of $(\Rightarrow)$ from this state. Naturally, concrete executions may take an unbounded amount of time to compute in the general case. This manifests itself in the above semantics as an unbounded set of timestamps leading to an unbounded address-space, and as an unbounded stack used to represent the current continuation.

## 4.3 Abstract Semantics

We will now provide the abstract semantics of the analysis. Because our analysis is monovariant and only maintains one approximation for each variable, there is only one

environment for a given expression-context. Thus, it is elided from the state space. The stack is now the only source of unboundedness in these semantics.

$$\begin{aligned}
\hat{\varsigma} \in \widehat{\Sigma} &= \mathsf{E} \times \widehat{Store} \times \widehat{Kont} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Var} \to \widehat{Values} \\
\hat{\kappa} \in \widehat{Kont} &= \widehat{Frame}^* \\
\hat{\phi} \in \widehat{Frame} &= \mathsf{E} \times \mathsf{Var} \\
\hat{v} \in \widehat{Values} &= \mathcal{P}(\widehat{Value}) \\
\hat{d} \in \widehat{Value} &= \mathsf{Lam}
\end{aligned}$$

In providing the abstract semantics, we will once again need a way to evaluate atomic expressions. The atomic evaluator is very similar to its concrete counterpart. However, since there is only one environment, we look up the value of a variable using it directly. Also, we do not need to close lambdas over an environment as their expression-body is already specific to a particular monovariant environment.

$$\begin{aligned}
\hat{\mathcal{A}} &: \mathsf{AE} \times \widehat{\Sigma} \rightharpoonup \widehat{Values} \\
\hat{\mathcal{A}}(x,\ (e,\ \hat{\sigma},\ \hat{\kappa})) &= \hat{\sigma}(x) \\
\hat{\mathcal{A}}(lam,\ (e,\ \hat{\sigma},\ \hat{\kappa})) &= \{lam\}
\end{aligned}$$

The abstract transition relation is also very similar to its concrete counterpart. Note that the frames no longer store environments.

$$\underbrace{((\text{let } (x\ e)\ e_\kappa)^l,\ \hat{\sigma},\ \hat{\kappa})}_{\hat{\varsigma}} \twoheadrightarrow (e,\ \hat{\sigma},\ \hat{\kappa}')$$

$$where \quad \hat{\kappa}' = (e_\kappa,\ x) : \hat{\kappa}$$

Also note that when updating the store, we use the least-upper-bound to remain sound. This permits values to merge within flow-sets: $(\sigma_1 \sqcup \sigma_2)(\hat{a}) = \sigma_1(\hat{a}) \cup \sigma_2(\hat{a})$.

$$\frac{(\lambda\ (x_1\ \ldots\ x_j)\ e) \in \hat{\mathcal{A}}(ae_f,\ \hat{\varsigma})}{\underbrace{((ae_f\ ae_1\ \ldots\ ae_j),\ \hat{\sigma},\ \hat{\kappa})}_{\hat{\varsigma}} \twoheadrightarrow (e,\ \hat{\sigma}',\ \hat{\kappa})}$$

$$where \quad \hat{\sigma}' = \hat{\sigma} \sqcup [x_i \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma})]$$

Finally, when we return, we update the variable found in a stack-frame.

$$\frac{\hat{\kappa} = (e,\ x) : \hat{\kappa}'}{\underbrace{(ae,\ \hat{\sigma},\ \hat{\kappa})}_{\hat{\varsigma}} \twoheadrightarrow (e,\ \hat{\sigma}',\ \hat{\kappa}')}$$

$$where \quad \hat{\sigma}' = \hat{\sigma} \sqcup [x \mapsto \hat{\mathcal{A}}(ae,\ \hat{\varsigma})]$$

Simply enumerating all the states possible given this abstract transition relation is not guaranteed to terminate. However, there is a finite representation of the infinite state space of the stacks. If we use this transition relation to generate a Dyck state graph, our analysis will terminate. This is accomplished by taking the infinite stacks and encoding them into a finite graph, where the stack frames are labels on edges of that graph. Intuitively, we are making the explicit result of cycles in control-flow (unbounded stacks) implicit as cycles in a control-flow graph.

A Dyck state graph is a set of edges.

$$G \in \mathcal{P}(Q \times \Gamma \times Q)$$

The nodes in the graph $Q$ are the parts of an abstract state $\hat{\varsigma} \in \hat{\Sigma}$ sans the stack $\hat{\kappa} \in \widehat{Kont}$.

$$q \in Q = \mathsf{E} \times \widehat{Store}$$

The edges describe transition between nodes and contain the stack-action that exists between these nodes. There are three different stack actions: pushing a frame $\hat{\phi}^+$, leaving the stack unchanged $\epsilon$, and popping a frame $\hat{\phi}^-$.

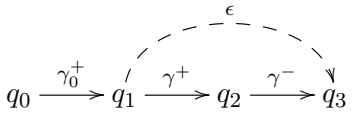
$$\gamma \in \Gamma = \hat{\phi}^+ \mid \epsilon \mid \hat{\phi}^-$$

Whether an edge exists in the graph can be taken directly from the abstract transition relation. We introduce the relation $(\xrightarrow{\gamma}) \subseteq Q \times \Gamma \times Q$ for edges in the Dyck state graph, defined in terms of the abstract transition relation.

$$q \xrightarrow{\hat{\phi}^+} q' \iff (q,\ \hat{\kappa}) \twoheadrightarrow (q',\ \hat{\phi} : \hat{\kappa})$$
$$q \xrightarrow{\epsilon} q' \iff (q,\ \hat{\kappa}) \twoheadrightarrow (q',\ \kappa)$$
$$q \xrightarrow{\hat{\phi}^-} q' \iff (q,\ \hat{\phi} : \hat{\kappa}) \twoheadrightarrow (q',\ \hat{\kappa})$$

To efficiently compute the Dyck state graph, an epsilon closure graph is needed. An epsilon closure graph has edges between all nodes that have no net stack change between

them. For instance, if we push a frame and then pop a frame, there should be an epsilon edge between the source node of the push edge and the target node of the pop edge. This is the epsilon edge between $q_1$ and $q_3$ below.




This allows us to immediately see that $\gamma_0$ is a possible top frame for $q_3$ when generating successor edges and nodes for $q_3$.

### 4.3.1 Transfer Function

When computing the analysis, we use a transfer function $\hat{f} : (Q \times \Gamma \times Q) \to (Q \times \Gamma \times Q)$ that takes a Dyck state graph and computes new edges at the frontier of the graph, generating a new Dyck state graph. We continually apply this transfer function until a fix-point is reached.

$$\begin{aligned}\hat{f}(G) &= G \cup \left\{(q, \gamma, q') : q \in Q,\ q \xrightarrow{\gamma} q'\right\}, \text{ where} \\ Q &= \left\{q' : (q, \gamma, q') \in G\right\} \cup \{q_0\}\end{aligned}$$

### 4.3.2 Global Store Widening

In the given abstract semantics, each state had its own store. However, to ensure the analysis will converge more quickly, global store-widening is usually employed. This form of widening is equivalent to using a global-store for all states which is computed as the least-upper-bound of all stores visited at any individual state. To accomplish this, we will remove the store from the nodes of the Dyck state graph and define the store-widened Dyck state graph as follows.

$$G_\triangledown \in \mathcal{P}(\mathsf{E} \times \Gamma \times \mathsf{E})$$

The globally store-widened transfer function then individually computes a new graph of expressions and stack actions, and a new global store.

$$\begin{aligned}\hat{f}_\triangledown(G_\triangledown, \hat{\sigma}) &= (G'_\triangledown, \hat{\sigma}'), \text{ where} \\ G'_\triangledown &= G_\triangledown \cup \left\{(e, \gamma, e') : e \in Q_e,\ (e, \hat{\sigma}) \xrightarrow{\gamma} (e', \_)\right\} \\ \hat{\sigma}' &= \bigsqcup \left\{\hat{\sigma}' : e \in Q_e,\ (e, \hat{\sigma}) \xrightarrow{\gamma} (\_, \hat{\sigma}')\right\} \\ Q_e &= \{e : (\_, \_, e) \in G_\triangledown\} \cup \{e_0\}\end{aligned}$$

An underscore represents a wildcard, i.e., any value.

### 4.3.3 Partitioning the Transfer Function

We can partition this monolithic transfer function, defining an individualized transfer function for each expression form in our language: $\hat{f}_{let}$, $\hat{f}_{call_i}$ and $\hat{f}_{ae}$. These functions are defined in precisely the same manner, but only use the rule applying to their specific language form. After each iteration, we merge the resulting Dyck state graphs and stores, taking their least-upper-bound. It has been shown that partitioning a system-space transfer function by rule in this manner is sound as the least-upper-bound of the system-spaces resulting from an application of each, is always equal to the system-space resulting from a single application of the combined $\hat{f}_{\triangledown}$ [16].

## 4.4 Linear Encoding

We will construct a transfer function for each abstract transition relation. This transfer function will update the store and will also be responsible for creating a Dyck state graph. We will define these functions using matrix multiplication ($\times$), outer product ($\otimes$), and boolean or ($+$). The style of encoding we use is taken directly from the original approach of EigenCFA [46].

The abstract state space, because it is finite, is easy to represent in vector and matrix form. If the elements in the domain are given a canonical order, we can represent a set of those elements using a bit vector. If an element from the domain is present in the set, the vector representing that set should have its bit set at the index corresponding to the offset of that element in the ordering. In our encoding, we will represent the set of states using a vector $\vec{s} \in \vec{S}$. We will represent atomic expressions, either variables or values, with $\vec{a} \in \vec{A}$, and we will use $\vec{v} \in \vec{V}$ to represent flow sets of abstract values.

$$\vec{s} \in \vec{S} = \{0,1\}^{|\mathsf{E}|}$$
$$\vec{a} \in \vec{A} = \{0,1\}^{|\widehat{Var}|+|\widehat{Value}|}$$
$$\vec{v} \in \vec{V} = \{0,1\}^{|\widehat{Value}|}$$

We can also encode the abstract syntax tree as matrices. We can extract the body of a closure using **Body** or the variables it binds using $\mathbf{Var_i}$. We can also deconstruct the components of a let expression using $\mathbf{Arg_1}$, **LetCall**, and **LetBody**.

$$\mathbf{Body}\colon \vec{V} \to \vec{S} \qquad \mathbf{Arg_i}\colon \vec{S} \to \vec{A}$$
$$\mathbf{Fun}\colon \vec{S} \to \vec{A} \qquad \mathbf{LetCall}\colon \vec{S} \to \vec{S}$$
$$\mathbf{Var_i}\colon \vec{V} \to \vec{A} \qquad \mathbf{LetBody}\colon \vec{S} \to \vec{S}$$

The store is a matrix that maps atomic expressions to abstract values.

$$\boldsymbol{\sigma}\colon \vec{A} \to \vec{V}$$

We also represent the Dyck state graph using three matrices. These three matrices map states to states, which in the case of our linear encoding, are expressions in our program. We use three different matrices to represent the three types of edges that can be found in the Dyck state graph.

$$\boldsymbol{\gamma_+}\colon \vec{S} \to \vec{S}$$

$$\boldsymbol{\gamma_\epsilon}\colon \vec{S} \to \vec{S}$$

$$\boldsymbol{\gamma_-}\colon \vec{S} \to \vec{S}$$

We also use a matrix to represent the epsilon closure graph which aids in the construction of the matrices encoding the Dyck state graph.

$$\boldsymbol{\epsilon}\colon \vec{S} \to \vec{S}$$

We now define the transfer function for the three types of expressions our language supports, let bindings, applications, and atomic expressions.

For let expressions, we first extract the sub-expression whose value will be bound to the variable of the let expression, $\vec{s}_{let} \times \mathbf{LetCall}$. We then record the push edge in the Dyck state graph, $\boldsymbol{\gamma_+} + (\vec{s}_{let} \otimes \vec{s}_{next})$.

$$\begin{aligned} f_{\vec{s}_{let}}(\boldsymbol{\gamma_+}) &= (\boldsymbol{\gamma_+}') \\ \textit{where} \quad & \vec{s}_{next} = \vec{s}_{let} \times \mathbf{LetCall} \\ & \boldsymbol{\gamma_+}' = \boldsymbol{\gamma_+} + (\vec{s}_{let} \otimes \vec{s}_{next}) \end{aligned}$$

Applications are somewhat more involved. We first pull out of the store the abstract values that we are applying for the given call site. We then extract the values of the arguments. We then get variables that we are binding from the closures we are applying. We then record the updated values in the store. We must also record that we made a tail-call in the Dyck state graph. We do this by updating $\boldsymbol{\gamma_\epsilon}$. We then must also update any epsilon edges.

$$
\begin{aligned}
f_{\vec{s}_{call_j}}(\boldsymbol{\sigma}, \boldsymbol{\gamma_\epsilon}, \boldsymbol{\epsilon}) &= (\boldsymbol{\sigma}', \boldsymbol{\gamma_\epsilon}', \boldsymbol{\epsilon}') \\
\textit{where} \quad & \vec{v}_f = \vec{s}_{call_j} \times \mathbf{Fun} \times \boldsymbol{\sigma} \\
& \vec{v}_i = \vec{s}_{call_j} \times \mathbf{Arg_i} \times \boldsymbol{\sigma} \\
& \vec{a}_i = \vec{v}_f \times \mathbf{Var_i} \\
& \boldsymbol{\sigma}' = \boldsymbol{\sigma} + (\vec{a}_1 \otimes \vec{v}_1) + \ldots + (\vec{a}_j \otimes \vec{v}_j) \\
& \vec{s}_{next} = \vec{v}_f \times \mathbf{Body} \\
& \boldsymbol{\gamma_\epsilon}' = \boldsymbol{\gamma_\epsilon} + (\vec{s}_{call_j} \otimes \vec{s}_{next}) \\
& \boldsymbol{\epsilon}' = f_\epsilon(\boldsymbol{\epsilon},\ \vec{s}_{call_j},\ \vec{s}_{next})
\end{aligned}
$$

Finally, we come to the last case where we have an atomic expression and must return. We first must compute the flow set of the atomic expression. We then look up the top frames of our stack. We then update the environment by binding the variable found at the top stack frame. We also extract the expression that we will be executing next. Finally, we record the pop edge and update the epsilon closure graph accordingly.

$$
\begin{aligned}
f_{\vec{s}_{æ}}(\boldsymbol{\sigma},\ \boldsymbol{\gamma_+},\ \boldsymbol{\gamma_-},\ \boldsymbol{\epsilon}) &= (\boldsymbol{\sigma}',\ \boldsymbol{\gamma_+}',\ \boldsymbol{\gamma_-}',\ \boldsymbol{\epsilon}') \\
\textit{where} \quad & \vec{v} = \vec{s}_{æ} \times \mathbf{Arg_1} \times \boldsymbol{\sigma} \\
& \vec{s}_{push} = \vec{s}_{æ} \times \boldsymbol{\epsilon}^\top \times \boldsymbol{\gamma_+}^\top \\
& \vec{a} = \vec{s}_{push} \times \mathbf{Arg_1} \\
& \boldsymbol{\sigma}' = \boldsymbol{\sigma} + (\vec{a} \otimes \vec{v}) \\
& \vec{s}_{next} = \vec{s}_{push} \times \mathbf{LetBody} \\
& \boldsymbol{\gamma_-}' = \boldsymbol{\gamma_-} + (\vec{s}_{æ} \otimes \vec{s}_{next}) \\
& \boldsymbol{\epsilon}' = f_\epsilon(\boldsymbol{\epsilon},\ \vec{s}_{æ},\ \vec{s}_{push})
\end{aligned}
$$

The epsilon closure graph aids in the construction of the Dyck state graph. It contains edges between states that have no net stack change. This allows us to quickly find the top frames when we need to return. When updating the epsilon closure graph, we not only need to record the new edges, but take all existing predecessors and successors into account.

$$
\begin{aligned}
f_\epsilon(\boldsymbol{\epsilon},\ \vec{s}_s,\ \vec{s}_t) &= \boldsymbol{\epsilon}' \\
\textit{where} \quad & \vec{s}_n = \vec{s}_t \times \boldsymbol{\epsilon} \\
& \vec{s}_p = \vec{s}_s \times \boldsymbol{\epsilon}^\top \\
& \boldsymbol{\epsilon}' = \boldsymbol{\epsilon} + (\vec{s}_s \otimes \vec{s}_t) \\
& \qquad + (\vec{s}_s \otimes \vec{s}_n) \\
& \qquad + (\vec{s}_p \otimes \vec{s}_t) \\
& \qquad + (\vec{s}_p \otimes \vec{s}_n)
\end{aligned}
$$

## 4.5 Example

To help give a better understanding of how the encoding works, we provide a short example.

```
(let (id^x0 (lambda (v^x1) v^l2)^l1 d̂0)
  (id (lambda (w^x2)
        (let (a^x3 (w id)^l5)
          (a a)^l6)^l4)^d̂1)^l3)^l0
```

For this program, there are only two denotable values, the two lambda terms. There are two let expressions, three call sites, and one atomic reference as the body of a lambda. There are also four variables in this program.

We will first discuss how you would encode the abstract syntax tree using matrices. Recall that there are six matrices that are needed.

First, given a flow set, we want to be able to extract which expressions are the body of a lambda term. Below we can see that $l_2$ is the body of the first lambda and $l_4$ is the body of the second lambda.

$$
\mathbf{Body} = \begin{array}{c} \\ d_0 \\ d_1 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\
\left[ \begin{array}{ccccccc}
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array} \right]
\end{array}
$$

We also need a way to extract the function being applied at a call site, whether it be a lambda term or a variable reference. Because there are only three call sites in the program, only three rows in the matrix have entries with non-zero values. In our example, every call site has a variable reference in function position.

$$\mathbf{Fun} = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{cccccc} x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 \end{array} \\ \left[\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right] \end{array}$$

There must also be a way to extract the arguments of a call site. This matrix can also be used to determine what atomic expression we are evaluating when our control state is at an atomic expression.

$$\mathbf{Arg_1} = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{cccccc} x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 \end{array} \\ \left[\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right] \end{array}$$

Once we have a flow set, we want to be able to extract the variable that we are binding when we apply the functions in our flow set.

$$\mathbf{Var_1} = \begin{array}{c} \\ \hat{d}_0 \\ \hat{d}_1 \end{array} \begin{array}{c} \begin{array}{cccccc} x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 \end{array} \\ \left[\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \end{array}$$

We also need to be able to know what the expression is whose value we will bind to a variable when we have a let expression. This lets us know what our successor state will be. This is used when we push a frame onto our stack.

$$\mathbf{LetCall} = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\ \left[\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

Given a let expression, we also need to know where we should return to once we have evaluated the expression which will provide the value we are binding.

$$\mathbf{LetBody} = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\ \left[\begin{array}{ccccccc} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

The store for this program is actually rather small. We are interested in finding out which lambda terms flow to which variables. With four variables and two lambda terms, there are only eight entries that can be set. Note that we have an identity matrix at the bottom of the store.

$$\boldsymbol{\sigma} = \begin{array}{c} \\ x_0 \\ x_1 \\ x_2 \\ x_3 \\ \\ \hat{d}_0 \\ \hat{d}_1 \end{array} \begin{array}{c} \begin{array}{cc} \hat{d}_0 & \hat{d}_1 \end{array} \\ \left[\begin{array}{cc} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \end{array}$$

To encode a Dyck state graph, we actually need three separate matrices. A value of one represents that there exists an edge between two states. The contents of the frame (the variable to bind and the expression to execute next) are both available using $\mathbf{Arg_1}$ and **LetBody**. After running the analysis on the above program, the results of the three matrices would be as follows.

$$\boldsymbol{\gamma}_+ = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\ \left[\begin{array}{ccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

$$\boldsymbol{\gamma}_- = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\ \left[\begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

$$\boldsymbol{\gamma_\epsilon} = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\ \left[\begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

We also need the epsilon closure graph. Initially it is an identity matrix because every state has an implicit epsilon edge to itself.

$$\boldsymbol{\epsilon} = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\ \left[\begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

## 4.6 Conclusion

We have described a linear encoding for a pushdown control-flow analysis as originally formulated by Earl et al. [12] building upon the general framework of abstract interpretation [9]. By precisely matching calls and returns, a pushdown control-flow analysis gives even more precision than a traditional finite state control-flow analysis. By demonstrating the feasibility of a linear encoding, we have demonstrated that it is at least possible to run a pushdown control-flow analysis on a SIMD architecture. However, a direct translation would likely be inefficient as the matrices are very sparse. Novel techniques such as those used in EigenCFA make the analysis run quickly [46].

## 4.7 Equivalence to Direct Style Constraints

In this section, we outline how the linear-algebra described in this chapter produces equivalent results to those described in Chapter 3.

Intuitively, pushdown control-flow analysis does not buy us anything when we are flow insensitive and monovariant because even though we can precisely match calls with returns, because of the monovariance of the analysis, we must return any value that could ever possibly get returned by that function. This happens with any caller. So even though we precisely match the call and return, it does not matter in terms of precision.

Stated differently, the monovariance causes all arguments that are ever passed to a function to be joined. In general, pushdown control-flow analysis is more precise, because it

allows us to match the parameters to a more precise return value. However, a flow-insensitive analysis must assume that any expression can be called after any other; we must revisit states whenever a new value is added to the store.

Given the flow matrix, we can construct the matrices representing store and the Dyck state graph, as well as the epsilon closure graph. We can also go the other way. Given the matrices representing the store and Dyck state graph, we can construct the flow matrix.

We can also do a case-wise analysis for each of the supported lambda calculus terms and show that they will contain the same values.

# CHAPTER 5

# RELATED WORK

Several works exists that explore parallelizing static analyses both on the GPU and on multiple cores, whether they be on the same chip or in a distributed environment. In this chapter, we explore prominent works in parallelizing static analysis.

Pingali et al. in their work discuss why parallelizing algorithms such as those used by a pushdown control-flow analysis is difficult [43].

Despite the difficulty in parallelizing static analyses, the GPU has proven itself to be capable of accelerating nontrivial static analyses. EigenCFA is a monovariant control-flow analysis for higher-order languages [46]. Gilray et al. further advance the ideas of EigenCFA, allowing for richer languages to be analyzed [16]. Méndez-Lojo et al. take a similar approach in writing an inclusion-based points-to analysis for the GPU [33]. They improve upon the ideas of EigenCFA by requiring less memory and locking, as well as removing redundant work and hiding latencies.

Parallelizing static analyses is definitely not limited to the GPU. Many analysis have been developed for parallel and distributed systems. Méndez-Lojo et al. also produced a parallel implementation of their points-to analysis for running on multiple cores [34]. Albarghouthi et al. have also demonstrated the feasibility of running top-down interprocedural analyses on multiple cores [1].

Distributed systems have also been used to tackle finite state model checking. Model checking suffers from the *state space explosion problem*. As we try to verify complex models, the state space that must be checked becomes too large to do in a reasonable amount of time. To combat this, we can try and take advantage of all the computing resources that are available to us by scaling our solutions to multiple machines in a clustered environment. Static analysis can also suffer from the *state space explosion problem*. Lopes and Rybalchenko describe a distributed system for predicate abstract refinement [26]. While Bingham et al. have developed a distributed system for finite state model checking that can handle billions of states [6].

## 5.1 The Essence of Parallelism

One of the prominent abstractions for reasoning about and exploiting parallelism is the dependence graph. However, the dependence graph is suitable only for "regular" algorithms that use dense arrays. The dependence graph is not a suitable abstraction for "irregular" algorithms where the key data structures are graphs, trees, or sets (prominent data structures in abstract interpretation). One prominent use of "irregular" algorithms is in optimizing compilers which perform iterative elimination-based data-flow analysis on structures for interprocedural control-flow graphs. Pushdown control-flow analysis also fits this definition of an "irregular" algorithm.

However, if we shift to a data-centric formulation of algorithms in which algorithms are expressed in terms of their actions on data structures, we can distill out algorithmic properties important for parallelization [43]. Tao analysis performs a structural analysis of an algorithm using three axes: topology, active nodes, and the operator. For the topology, it studies the data structure on which computation occurs. For active nodes, it studies how nodes become active and how active nodes should be ordered. The operator is the operation that is performed on the active node.

Amorphous data-parallelism is ubiquitous in algorithms. Given a set of active nodes and an ordering on active nodes, amorphous data-parallelism is the parallelism that arises from simultaneously processing active nodes. These active nodes are the parts of the data structures of our algorithms that need to be operated on next. In other words, this parallelism comes from the computations that are not ordered by the transitive closure of a dependence graph.

Static dependence graphs are inadequate abstractions for irregular algorithms because of the following reasons.

1. Dependencies between activities in irregular algorithms are most likely complex functions of runtime data values. They cannot be captured by a traditional static dependence graph.

2. Many irregular algorithms exhibit don't-care nondeterminism. This is a property that is hard to model with dependence graphs.

3. Whether or not it is safe to execute two activities in parallel at a given point in time may depend on activities created later in time. It is unclear how to model this with a dependence graph.

One measure of amorphous parallelism in irregular algorithms is the number of active nodes that can be processed in parallel at each step of the algorithm for a given input. This means that once we have computed an abstract interpretation of a program, we could know optimally how to parallelize the program. However, as we are running the abstract interpretation, we cannot be sure.

## 5.2 Control-Flow Analysis with GPUs

Prabhu et al. created EigenCFA, a 0CFA implementation suitable for the GPU [46]. GPUs are typically used for operating over continuous domains with little branching in the instruction set, but a control-flow analysis uses a discrete domain that often involves branching. EigenCFA successfully overcomes these issues.

The main contribution of EigenCFA is reducing the transfer function of an abstract semantics to a single kernel call. To achieve this goal, the analysis only operates on binary CPS.

### 5.2.1 Linear Encodings

Another major contribution of EigenCFA is the use of abstract Church encodings. There are many language features that at first glance would be difficult to abstractly interpret on a GPU. However, the authors observe that the abstract behavior of many disparate language features are identical in the abstract semantics. For example, termination can be modelled by a nonterminating loop, which will terminate in our abstract semantics. Mutating a variable is the same as binding a variable. Recursion can be modelled with mutation which in turn is modelled again as binding. Basic values can me modelled as nontermination which will terminate in our abstract semantics. Also, conditionals can be encoded without branching since we usually take both branches in an abstract interpretation; these can abstractly be represented as two subsequent calls.

Before we can encode the transfer function as a sequence of matrix operations, we must encode the abstract syntax tree and the abstract domain as matrices. Some of these encodings can also be done implicitly. If we were to assign a label to a lambda term, we could use the corresponding label for the variable that the lambda binds. Also, we could use the subsequent label for the body of the lambda. It is also important to note that though we encode it as a matrix, in reality, since only one value is set per row, we can encode it as a vector and use the index as an implicit value. A pushdown analysis can also take advantage of these implicit representations.

### 5.2.2 Sparseness

A key insight of EigenCFA is that the abstract store resulting from a 0CFA analysis is sparse. It is unlikely that a lambda will flow to many places in a program. As such, it is advantageous to encode the store as a sparse matrix. The initial implementation of 0CFA attempted to use dense matrices but was unable to achieve speedups until sparse matrices were employed.

A pushdown analysis would also be able to take advantage of sparse matrices for the store. However, the component that exists in a pushdown system and not in a finite state representation is the Dyck state graph. The Dyck state graph will also be sparse, but the epsilon closure graph contains some entries that contain many values, while other parts of the matrix remain sparse.

In 0CFA, the store grows monotonically. This allows for the exploitation of benign race conditions. The order that we go up the lattice is unimportant. Adding the same value multiple times to the store will not change the result of the analysis.

This is also something that is exploitable in a pushdown analysis because the Dyck state graph and epsilon closure graph also grow monotonically. We can add the same value to a binding multiple times without changing the final answer. We can add the same edge to the Dyck state graph, without changing the final result of the analysis.

## 5.3 Partitioning the Transfer Function

Gilray et al. build upon the foundational work of EigenCFA, while at the same time addressing some of its issues [16].

EigenCFA encodes the analysis in a single kernel. This forces all the language features to be desugared into a single form, a call site with two arguments. This design choice disallows an analysis of simple language features such as primitive operations and conditionals.

However, by partitioning the transfer function, we can allow for more language forms. A separate kernel can be created for every language form of interest. Then all language forms of the same type in the program under analysis can be grouped together. This allows for more complicated kernels depending on the analysis, which will have less thread divergence. We can create a separate kernel for call sites with a different number of arguments. We can create a kernel for conditionals, primitive operations, and mutation. Each of these kernels will be able to take advantage of the SIMD architecture because there will be lesser thread divergence than if we had a single massive kernel which handled each language form.

This partitioning of the transfer function is sound. Joining the resulting store after applying each individual transfer function is equivalent to the store that would be created

by a single transfer function.

For a pushdown analysis, we can partition this monolithic transfer function, defining an individualized transfer function for each expression form in our language: $\hat{f}_{let}$, $\hat{f}_{call_i}$ and $\hat{f}_{ae}$. These transfer functions are defined in precisely the same manner as the single transfer function, but only use the rule applying to their specific language form. After each iteration, we merge the resulting Dyck state graphs, epsilon closure graphs, and stores, taking their least-upper-bound. A pushdown analysis must also be able to acquire the information for top stack frames, required when we return from a function call.

### 5.3.1 Global Store Widening

To ensure a static analysis will converge more quickly, global store-widening is usually employed. This form of widening is equivalent to using a global-store for all states. This global store the least-upper-bound of all stores visited at any individual state. To accomplish this in a pushdown analysis, the store is removed from the nodes of the Dyck state graph and the store-widened Dyck state graph is defined as follows:

$$G_\triangledown \in \mathcal{P}(\mathsf{Exp} \times \Gamma \times \mathsf{Exp})$$

The globally store-widened transfer function then individually computes a new graph of expressions and stack actions, and a new global store.

$$\begin{aligned}
\hat{f}_\triangledown(G_\triangledown, \hat{\sigma}) &= (G'_\triangledown, \hat{\sigma}'), \text{ where} \\
G'_\triangledown &= G_\triangledown \cup \left\{(e, \gamma, e') : e \in Q_e,\ (e, \hat{\sigma}) \xrightarrow{\gamma} (e', \hat{\sigma}')\right\} \\
\hat{\sigma}' &= \bigsqcup \left\{\hat{\sigma}' : e \in Q_e,\ (e, \hat{\sigma}) \xrightarrow{\gamma} (e', \hat{\sigma}')\right\} \\
Q_e &= \left\{e' : (e, \gamma, e') \in G_\triangledown\right\} \cup \{e_0\}
\end{aligned}$$

### 5.3.2 Higher Precision and Richer Domains

Another shortcoming of EigenCFA addressed by Gilray et al. is that EigenCFA propagates flows caused by dead call sites. It might be the case in a program that a call site will never be invoked. An abstract interpretation could also recognize that a call site is dead and not propagate flows from the given call site. EigenCFA applies all call sites and has no notion of whether a call site is dead or not. This issue is resolvable by maintaining a vector of active call sites and only invoking them. However, this comes at the cost of some parallelism.

Another issue of EigenCFA is its abstract encodings. It only allows for lambda terms to be values within a program. However, this need not be the case. As long as we can represent

an abstract value as an entry in a matrix, a GPU implementation of a static analysis could handle it. In using the abstract Church encodings, it becomes very difficult to reason about the results of EigenCFA. For example, if we were interested in type recovery, rather than represent numbers and symbols as lambda terms for each unique value in the program, we could represent them as a single column within the matrix. The use of encoding richer abstract domains in matrix form was used by Banterle and Giacobazzi [5], where they were able to represent Octagon Abstract Domains.

## 5.4 Avoiding Locks and Reducing Memory Requirements

Another recent work bringing flow analysis to the GPU implements an inclusion-based points-to analysis [33]. This analysis is very similar to a control-flow analysis of higher-order languages. Instead of determining which lambda terms flow to which expressions, it attempts to answer the question of which pointer variables can point to which other variables. Their implementation scales to handling hundreds of thousands of variables.

This work contains many ideas that improve upon the ones presented in EigenCFA. They reduce memory usage, eliminate the need for locks, remove redundant work, and hide memory transfer latency.

Like pushdown control-flow analysis, this algorithm is difficult to parallelize effectively on the GPU, because it performs extensive modifications to the underlying data structure and performs relatively little computation.

The authors make the observation that the constraints of an inclusion-based points-to analysis can be reformulated as graph rewrite rules. Every variable becomes a node in the graph and the constraints between them are represented as edges. Graph rewrite rules then are applied based on the edges of a given node. They rewrite rules are applied until a fixed point is reached. These rules do not need to be fired in any particular order, they can be interleaved in any fashion. A quality shared by a control-flow analysis of higher-order languages.

When adding a new edge to the graph, locking is only required if multiple nodes are attempting to add an edge to the same node. However, if edges are only updated by a single node, then no synchronization is required. This idea is immediately transferable to a control-flow analysis, because we have separated work based on call sites and separate call sites might in fact update the same variable.

A commonality between control-flow analysis of higher-order language and pointer analysis is that the final result is sparse. Just like it is unlikely that a variable will point to many lambdas, it is unlikely that a pointer variable will point to many variables or

address locations. Creating an efficient data structure to represent the constraints is a difficult problem. During the course of the analysis, million of edges may need to be added. An analysis of the Linux kernel results in 1.498 billion edges. The memory layout needs to address minimizing memory transactions, maximizing coalescing, and avoid thread divergence within warps. All difficult problems on the GPU. To address this issue, the authors introduce a sparse representation of adjacency matrices that can grow as needed but still retains small amounts of memory. They use a linked list of bit vectors for this representation. This also takes advantage of the fact that variables which point to each other generally appear close together in the source code. This is a major advantage over the matrix representation used by EigenCFA.

Another tactic used by their implementation is to keep track of which nodes are active. For each iteration, a node generates new edges, and then no longer becomes active. If a new edge is added to a node, then it becomes active. By keeping track of which nodes are active, we can reduce large amounts of duplicated work but still fully exploit all the available parallelism.

The authors also hide the latency of transferring data between the CPU and GPU. Data are copied in the background between iterations. Thus, when calculating the new edges for the current generation, the edges from the old generation are copied to the CPU. This has the effect of hiding the latency.

Both of these techniques are adaptable to a control-flow analysis of higher-order languages.

## 5.5 Parallel Inclusion-based Points-to Analysis

Before working on their GPU implementation, the previous authors worked on parallelizing their algorithm on multiple CPU cores [34].

They recognize that in a constraint graph for points-to inclusion-based analysis, based on graph rewriting, there are many active nodes in the constraint graph. They make the observation that if the rewrites at two active nodes do not interfere with each other, they can be performed in parallel. Parallelizing then becomes the activity of finding noninterfering active nodes.

Parallelizing this form of static analysis is much harder than parallelizing regular applications like dense matrix multiplication and stencil operations. These type of applications are known as *regular* applications. They are easier to parallelize because the dependencies between computations are known before runtime. On the other hand, with *irregular* com-

putations, dependencies of computation are known at only at runtime. The dependencies between these computations are functions of runtime data.

### 5.5.1 Building upon an Existing Parallel Framework

Their implementation builds upon the Galois system framework. This is a framework intended to support parallel execution of irregular applications and is derived from an operator formulation. An operator is concerned with three elements: the active element, the activity, and the neighborhood. The active element is the node or edge on which a computation is centered. The activity is the computation itself, derived from an operator. The neighborhood is the set of nodes and edges read or written by an activity.

Recall that opportunities for exploiting parallelism exists in graph algorithms where there are multiple active nodes. However, by choosing which active nodes to operate one, we must take into consideration neighborhood constraints.

In the Galois system, activities are executed by running speculatively and committing the computation once gaining an abstract lock for a neighborhood. This leads to four sources of overhead which can limit scalability: enforcing neighborhood constraints, copying data for rollbacks, aborted activities, and dynamic assignment of work.

Their implementation overcomes these overhead challenges. To eliminate abstract locks, they note that if the graph were read only, no locks would be needed. The graph rewrite rules never remove nodes or edges from the graph, but only add them. There is also no need to copy data because nodes and edges are never removed but only added. We can eliminate copying data because no conflicting data will ever be generated, just possibly redundant data. We also never need to abort an activity because the added edges are always part of the final solution. In regards to assignment of work, nodes actually perform little work. To overcome this challenge, they employ iteration coalescing. When an activity adds an edge to the graph, it checks to see if any constraints are violated; if so, it puts work on a local work queue, rather than the global work queue.

## 5.6 Parallelizing Interprocedural Analyses

Similar work, at least on the surface, is that of Albarghouthi et al. [1]. They develop a framework for performing a modular analysis of top-down interprocedural analysis. There are two ways to perform modular program analysis: top-down and bottom-up. We can start at the leaves and work our way up function calls, which would be easier to scale. Or we can start at the top node and work our way down to function calls. This work is significant in that it is top-down.

### 5.6.1 MapReduce Style Parallelism

They create a generic framework, named BOLT, which uses MapReduce style parallelism. It generates a query Q over procedure P which results in sub queries for calls made to other procedures by procedure P. In the map stage, we run multiple queries in parallel. In the reduce state, interdependencies are managed between queries.

A similar approach can be taken for an abstract interpretation of higher-order languages. As we generate successor states and explore the abstract transition graph, it can be the case that there are many states for which we need to generate successors states. These can in fact be done in parallel. Then in the reduce stage, we can determine if future successor states need to be generated. In the course of generating successor states, it is possible that we are generating a state that has already been seen before. In this case, we no longer need to generate successors. In the reduce step, we then would need to coordinate between states to determine if it the state has been visited before.

In the past, I have experimented with implementing an existing analysis on top of the MapReduce framework [10]. The major difficulty in scaling is that for the analysis to scale without changing the underlying framework, the abstract transition graph must have a large dominating frontier when performing the breadth-first search.

### 5.6.2 Must-Analysis vs May-Analysis

One strength of the BOLT system is that it can handle may-analyses and must-analyses. In a may-analysis, we determine behaviors that may occur during the actual runtime of the program, but do not necessarily occur. This comes from a conservative analysis. This is what is generated from a control-flow analysis. The analysis itself is conservative and thus produces a may happen result. We do not perform an under-approximation of the runtime of the program. However, if an abstract transition relation were developed that could create a under-approximation of the runtime behavior of the program, any tool we develop would be able to handle must-analyses.

### 5.6.3 Parameterization

BOLT is parameterized by an intraprocedural analysis algorithm used to analyze a single procedure. Our analysis is parameterized by an abstract transition relation that generates successors states for a given abstract state. This means that our framework will be able to handle any advancements that can be contained within the abstract transition relation, such as abstract garbage collection [39].

Bolt employs a pluggable architecture. It assumes that its underlying intraprocedural analysis is a pure function. It takes a query as input and returns a set of queries. No resources are shared between threads. This is similar to the abstract transition relation. It is a pure function that takes abstract states and returns abstract states.

They were able to successfully run their analysis on 45 Microsoft Windows device drivers.

If a function makes calls to other functions, Bolt exploits this opportunity for parallelism by exploring the called functions in parallel. This source of parallelism might not be exploitable if our internal representation of the language only allows for one function call in the body of a lambda. However, an object oriented framework can take advantage of multiple entry point saturation [24].

## 5.7 Distributed Model Checking

Lopes and Rybalchenko develop a distributed model checker by taking an existing model checker ARMC [44] and an existing distributive framework DAHL [26]. Their tool is based on predicate abstraction and refinement-based algorithm for software verification. Their distributed workers communicate via message passing.

One of the main concerns they address is the inherent nondeterminism present in distributed computing. In the case of counterexample refinement, it is not the case that all counterexamples are equal. We need to be very deliberate in which ones we explore. We do not want to be at the mercy of the response times and processing power of our workers.

They discovered that when a naive distribution scheme is used, an order of magnitude difference can arise. In a control-flow analysis, how successor states are generated is an important problem. If we naively distribute the owners of abstract states, it might be the case that the overhead introduced by the distribution might result in longer running times than if we just ran the analyses serially.

In their tool, they generate all states a program can reach and check if error states are included. Our tool will do the same thing. However, once they have generated an error state, they begin to run counter examples to refine the abstraction to possibly remove the error state. This is similar to Shivers' reflow analysis [49] where we could improve the allocation function in order to gain more precision to eliminate spurious flows in the abstract transition graph. These refinements could be run in parallel.

### 5.7.1 Architecture

In their architecture, they use a single master node and a set of worker nodes. The worker nodes request work from the master nodes. It was shown that having a master node

was not a bottleneck for the program. In their scheme, the master node was needed because they wanted to give a deterministic exploration of counter examples.

The success of predicate abstraction-based verifiers depends on the choice of counterexamples. However, in an abstract interpreter, generally a single allocation function is used that will cause the same coarseness to appear throughout the entire abstract transition graph. Because we must explore all states in order to remain sound, it doesn't matter what order we explore the states, as long as we are not performing any widening.

Even in the presence of widening, the final result would still be deterministic, even though a different number of nodes might be generated. The final result will still be the same. This is due to the associativity and commutativity of the join operator on stores. At the end of the analysis, if we were to join every store, we would get the same results. However, we might explore a different amount of states. Depending on our search strategy, it is possible that we can explore a state earlier that subsumes other states that need to be explored. There is no need to explore the weaker state if we have already explored the stronger state.

## 5.8 Industrial Strength Explicit State Model Checking

Bingham et al. present the implementation details of an industrial strength explicit state model checker they named PReach (Parallel REACHability) [6]. Their tool is able to explore 30 billion states of a cache coherence protocol model. To their knowledge, this is the largest state space ever to be explored in a publication. They are able to successfully overcome the problem that the number of states grows exponentially with the number of state variables.

Explicit state model checkers are generally limited by the amount of memory the machine has available. Disk-based approaches exist but are inherently slower than checking a table in RAM, as the model checker can be slowed down by a factor of 30. However, by running in a distributed manner, it is possible to increase the amount of available memory and scale to very large models. A parallel explicit state model checker also allows us to model check faster by speeding up the exploration of the state space. This allows us to check models that would be too large to explore on a single processor in a realistic amount of time.

Increases in memory and speed are important resources for analyses of higher-order programs as well.

### 5.8.1 Implementation

The PReach tool has two implementation levels. An Erlang program handles the distributed aspects of the tool and contains less than 1000 lines of code. This layer calls into the second layer, a tool named Muf$\varphi$, for parsing, state expansion, hash table look-ups and insertion, and for invariant and assertion violation detection.

PReach is based on the DEMC algorithm of Stern and Dill [51]. It performs a distributed breadth-first search of the state space by using a uniform random hash function that associates an owner node with each state. The owner of a state is for responsible for generating its successors. Whenever a state is generated, it is sent to its owner node who is responsible for checking if the state has already been explored and generating successor states if necessary.

We could take almost any finite state abstract interpreter and easily parallelize it with the PReach framework. We would just need to implement the interface it requires. PReach makes four basic calls to Mur$\varphi$: *Initial*, *Successors*, *Visited*, and *Insert*. These features can be extracted directly from the state space search algorithm of $k$-CFA [49]. The *Initial* function is equivalent to the injection function. The *Successors* function is equivalent to the abstract small-step transition relation. The *Visited* function is equivalent to the set membership test of visited states. Finally, the *Insert* function is equivalent to the union of a state to the visited set. However, additional requirements remain for a pushdown analysis.

The PReach framework can even handle advanced abstract interpretation features such as abstract garbage collection, abstract counting, and taint analysis [39, 25]. This is achievable because these features are contained within the transition relation. Because the framework is agnostic to the what occurs inside the *Successors* function, all these features could be supported.

PReach has several features that contribute to its scalability and robustness, including message backoff schemes, load balancing, and batch messaging.

### 5.8.2 Message Backoff Schemes.

An initial implementation of PReach would immediately send all successors to their owner. However, nodes would grind to a halt or crash. The problem was that these nodes were accumulating a disproportionate number of messages in their mailboxes. They could not be moved to their work queue fast enough and paging resulted.

To address this issue, they introduced a crediting mechanism. Each node has a given number of credits for sending messages to each other node. This will likely be an important consideration.

### 5.8.3 Load Balancing.

Another innovation of PReach is its load balancing. The slowest node determines the total runtime of the system. In performing an abstract interpretation, it is likely that at first there will not be much potential for parallel exploration. This potential will also be limited as we approached the fixed point of the computation.

To make sure each node is given approximately the same amount of work, load balancing is needed. Kumar and Mercer proposed an aggressive rebalancing technique which compares work queue sizes of adjacent nodes and has nodes pass states to neighbors with smaller work queues [22].

### 5.8.4 Batch Messaging.

The final innovation of PReach is that states are sent in batches. This can result in speedups of 10 to 20 times (using batches of size 100 to 1000 as opposed to sending the states individually).

### 5.8.5 Improvements

The PReach framework could also be improved upon with techniques from abstract interpretation, such as Shivers' aggressive-cutoff algorithm and widening [49]. When testing for set membership in the visited set, it does not have to be implemented as a strict membership test, but can be improved upon with a subsumption test. Because the framework is unaware of the implementation details of *Visited*, this feature could easily be added.

However, to employ widening in our abstract interpretation, the framework would need to be changed slightly. This could be achieved by having each node have a single store that it uses for the nodes it is responsible for. Then whenever it receives new nodes, it would widen them with its node-specific store.

# CHAPTER 6

# CONCLUSION

This dissertation presented several techniques towards improving control-flow analysis of higher languages, both in terms of precision and the actual running time of the analysis. Three techniques were presented that improve the small-step abstract interpreter approach to static analysis.

Chapter 2 demonstrated that the order in which states are explored is important when computing $k$-CFA using global store widening. Counter to intuition, differences in the number of states explored exist when exploring with the traditional graph exploration strategies, depth first, and breadth first. This observation gave rise to idea that deliberate choices can be made in exploring the abstract state space in a more efficient manner. By using the features of the states to guide the exploration, it is possible to achieve reduction in the number of states by a factor of five and attain a 1.5x speedup of the analysis.

It also demonstrated a unique approach to abstracting environments in an abstract interpretation. It described how the approach is similar to loop unrolling. It demonstrated that holding off abstracting addresses for a while can result in smaller abstract state spaces and increased precision. It could even be more precise than abstract garbage collection in some instances.

It also demonstrated a technique, similar to strong update, which restricts the size of the store at application sites. Reduction in the store size is likely to lead to increases in both speed and precision, as has been show to be the case with abstract garbage collection. This improvement to the analysis deals with addresses that are alive, but have flow set sizes that are greater than one. Three mappings of the constraint-based formulation of control-flow analysis were also presented.

Chapter 3 demonstrated how to use existing tools for pointer analysis to solve control-flow analysis, which are able to run both on the GPU and in parallel on multicore CPU. This allows us to leverage the significant effort that has gone into the state of the art of

pointer analysis. The chapter provided the inference rules that allowed us to run an analysis in parallel.

It also demonstrated how to map the constraints into SAT. It demonstrated that in some cases, the approach could be as fast as a highly optimized solution. However, this chapter could potentially lead to much more work. There may exist encodings that can take advantage of the extra powers afforded by SAT solvers, which could lead to even more precision.

It also demonstrated that these constraints could be encoded in linear-algebra operations. This was done, having two goals in mind. First, it makes the problem suitable for running the analysis on the GPU. Second, it provides additional intuition on how a monovariant and flow-insensitive control-flow analysis is in fact equivalent to solving the direct-style constraints.

Chapter 4 described a linear encoding for a pushdown control-flow analysis. By precisely matching calls and returns, a pushdown control-flow analysis gives even more precision than a traditional finite state control-flow analysis. By demonstrating the feasibility of a linear encoding, this chapter demonstrated that it is possible to run a pushdown control-flow analysis on a GPU. It also provides some intuition on how the analysis described in Chapter 3 is the same.

# REFERENCES


[1] Albarghouthi, A., Kumar, R., Nori, A. V., and Rajamani, S. K. Parallelizing top-down interprocedural analyses. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2012), PLDI '12, ACM, pp. 217–228.

[2] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[3] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[4] Ashley, J. M., and Dybvig, R. K. A practical and flexible flow analysis for Higher-Order languages. *ACM Transactions on Programming Languages and Systems 20*, 4 (1998), 845–868.

[5] Banterle, F., and Giacobazzi, R. A fast implementation of the octagon abstract domain on graphics hardware. In *Static Analysis*, H. Nielson and G. Filé, Eds., vol. 4634 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, ch. 20, pp. 315–332.

[6] Bingham, B., Bingham, J., de Paula, F. M., Erickson, J., Singh, G., and Reitblatt, M. Industrial strength distributed explicit state model checking. In *Proceedings of the 2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology* (Washington, DC, USA, 2010), PDMC-HIBI '10, IEEE Computer Society, pp. 28–36.

[7] Bourdoncle, F. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications*, D. Bjørner, M. Broy, and I. Pottosin, Eds., vol. 735 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin/Heidelberg, 1993, ch. 9, pp. 128–141.

[8] Chase, D. R., Wegman, M., and Zadeck, F. K. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1990), PLDI '90, ACM, pp. 296–310.

[9] Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1979), POPL '79, ACM, pp. 269–282.

[10] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[11] Earl, C., Might, M., and Horn, D. V. Pushdown Control-Flow analysis of Higher-Order programs. In *Proceedings of the 2010 Workshop on Scheme and Functional Programming (Scheme 2010)* (Montreal, Quebec, Canada, Aug. 2010).

[12] Earl, C., Sergey, I., Might, M., and Van Horn, D. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2012), ICFP '12, ACM, pp. 177–188.

[13] Felleisen, M. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Aug. 1987.

[14] Felleisen, M., and Friedman, D. P. A calculus for assignments in Higher-Order languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1987), POPL '87, ACM, pp. 314+.

[15] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (New York, NY, USA, June 1993), vol. 28 of *PLDI '93*, ACM, pp. 237–247.

[16] Gilray, T., King, J., and Might, M. Partitioning 0-CFA for the GPU. In *28th Workshop on (Constraint) Logic Programming (WLP 2014)* (Sept. 2014).

[17] Gilray, T., and Might, M. A survey of polyvariance in abstract interpretations. In *Trends in Functional Programming*, J. McCarthy, Ed., vol. 8322 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 134–148.

[18] Guha, A., Saftoiu, C., and Krishnamurthi, S. The essence of JavaScript. In *ECOOP 2010 Object-Oriented Programming*, T. D'Hondt, Ed., vol. 6183 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 126–150.

[19] Henglein, F. Simple closure analysis. Tech. rep., Department of Computer Science, University of Copenhagen (DIKU), Mar. 1992.

[20] Jagannathan, S., Thiemann, P., Weeks, S., and Wright, A. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1998), POPL '98, ACM, pp. 329–341.

[21] Johnson, J. I., Labich, N., Might, M., and Van Horn, D. Optimizing abstract abstract machines. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2013), ICFP '13, ACM, pp. 443–454.

[22] Kumar, R., and Mercer, E. G. Load balancing parallel explicit state model checking. *Electronic Notes in Theoretical Computer Science 128*, 3 (Apr. 2005), 19–34.

[23] Lhotak, O., Smaragdakis, Y., and Sridharan, M. Pointer analysis (dagstuhl seminar 13162). *Dagstuhl Reports 3*, 4 (2013), 91–113.

[24] Liang, S., Keep, A. W., Might, M., Lyde, S., Gilray, T., Aldous, P., and Van Horn, D. Sound and precise malware analysis for android via pushdown reachability and Entry-Point saturation. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices* (New York, NY, USA, 2013), SPSM '13, ACM, pp. 21–32.

[25] Liang, S., and Might, M. Hash-Flow taint analysis of Higher-Order programs. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2012), PLAS '12, ACM.

[26] Lopes, N. P., and Rybalchenko, A. Distributed and predictable software model checking. In *Verification, Model Checking, and Abstract Interpretation*, R. Jhala and D. Schmidt, Eds., vol. 6538 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 340–355.

[27] Lyde, S., Byrd, W. E., and Might, M. Control-Flow analysis of dynamic languages via pointer analysis. In *Proceedings of the 11th Symposium on Dynamic Languages* (New York, NY, USA, 2015), DLS '15, ACM.

[28] Lyde, S., Gilray, T., and Might, M. A linear encoding of pushdown Control-Flow analysis. In *Proceedings of the 2014 Workshop on Scheme and Functional Programming* (2014).

[29] Lyde, S., and Might, M. Control-Flow analysis with SAT solvers. In *Trends in Functional Programming*, J. McCarthy, Ed., vol. 8322 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 125–133.

[30] Lyde, S., and Might, M. Environment unrolling. In *Workshop on Higher-Order Program Analysis 2014 (HOPA 2014)* (2014).

[31] Lyde, S., and Might, M. Strong function call. In *Workshop on Higher-Order Program Analysis 2014 (HOPA 2014)* (2014).

[32] Lyde, S., and Might, M. State exploration choices in a Small-Step abstract interpreter. In *Proceedings of the 2015 Workshop on Scheme and Functional Programming* (2015).

[33] Méndez-Lojo, M., Burtscher, M., and Pingali, K. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2012), PPoPP '12, ACM, pp. 107–116.

[34] Méndez-Lojo, M., Mathew, A., and Pingali, K. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2010), OOPSLA '10, ACM, pp. 428–443.

[35] Midtgaard, J. Control-flow analysis of functional programs. *ACM Comput. Surv. 44*, 3 (June 2012).

[36] Midtgaard, J., and Van Horn, D. Subcubic control flow analysis algorithms. *Computer Science Research Report*, 125 (2010), 1–35.

[37] Might, M. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, June 2007.

[38] Might, M., and Prabhu, T. Interprocedural dependence analysis of Higher-Order programs via stack reachability. In *Proceedings of the 2009 Workshop on Scheme and Functional Programming (Scheme 2009)* (Boston, Massachussetts, USA, Aug. 2009).

[39] Might, M., and Shivers, O. Improving flow analyses via CFA: Abstract garbage collection and counting. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2006), ICFP '06, ACM, pp. 13–25.

[40] Might, M., Smaragdakis, Y., and Van Horn, D. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. Object-Oriented program analysis. In *Proceedings of the 31st Conference on Programming Language Design and Implementation (PLDI 2006)* (Toronto, Canada, June 2010), pp. 305–315.

[41] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of Program Analysis*, corrected ed. Springer, Dec. 1999.

[42] Palsberg, J. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems 17*, 1 (Jan. 1995), 47–62.

[43] Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M. A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., and Sui, X. The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 12–25.

[44] Podelski, A., and Rybalchenko, A. ARMC: The logical choice for software model checking with abstraction refinement. In *Practical Aspects of Declarative Languages*, M. Hanus, Ed., vol. 4354 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, ch. 16, pp. 245–259.

[45] Politz, J. G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., and Krishnamurthi, S. Python: The full monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2013), OOPSLA '13, ACM, pp. 217–232.

[46] Prabhu, T., Ramalingam, S., Might, M., and Hall, M. EigenCFA: Accelerating flow analysis with GPUs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, ACM, pp. 511–522.

[47] Schmidt, D. Trace-Based abstract interpretation of operational semantics. 237–271.

[48] Sereni, D., and Jones, N. D. Termination analysis of Higher-Order functional programs. In *Programming Languages and Systems*, K. Yi, Ed., vol. 3780 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, ch. 19, pp. 281–297.

[49] Shivers, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[50] Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1996), POPL '96, ACM, pp. 32–41.

[51] Stern, U., and Dill, D. L. Parallelizing the murphi verifier. In *Proceedings of the 9th International Conference on Computer Aided Verification* (London, UK, UK, 1997), CAV '97, Springer-Verlag, pp. 256–278.

[52] Van Horn, D. *The Complexity of Flow Analysis in Higher-Order Languages*. PhD thesis, Brandeis University, Boston, MA, Aug. 2009.

[53] Van Horn, D., and Mairson, H. G. Relating complexity and precision in control flow analysis. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2007), ICFP '07, ACM, pp. 85–96.

[54] Van Horn, D., and Mairson, H. G. Deciding kCFA is complete for EXPTIME. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2008), ICFP '08, ACM, pp. 275–282.

[55] Van Horn, D., and Mairson, H. G. Flow analysis, linearity, and PTIME. In *Static Analysis*, M. Alpuente and G. Vidal, Eds., vol. 5079 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 255–269.

[56] Van Horn, D., and Might, M. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2010), ICFP '10, ACM, pp. 51–62.

[57] Vardoulakis, D., and Shivers, O. CFA2: a Context-Free Approach to Control-Flow Analysis. In *European Symposium on Programming* (2010), pp. 570–589.

[58] Yong, S. H., Horwitz, S., and Reps, T. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1999), PLDI '99, ACM, pp. 91–103.