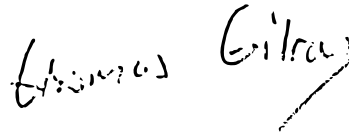


# INTROSPECTIVE POLYVARIANCE FOR CONTROL-FLOW ANALYSES

by

Thomas Gilray

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 2016

# ABSTRACT

Static analysis aims to approximate dynamic program behaviors statically. This task brings one up against the fundamental limits of computability. As identifying any nontrivial program property is undecidable in the general case, we must compromise by allowing for imprecision in order to ensure computability—with yet further compromises permitting us to place strict upper bounds on analysis complexity. Deciding how this compromise should be struck is the central challenge of static analysis.

This dissertation extends the methodology of abstracting abstract machines (AAM), a systematic approach to the abstract interpretation of abstract-machine semantics, by connecting it to a well-defined notion of polyvariance. The polyvariance of a static analysis, broadly construed, is the degree to which variables or syntactic program points are differentiated into a multiplicity of static approximations of their dynamic behavior. Increasing the polyvariance of an analysis adjusts its compromise in favor of precision, although particular flavors of polyvariance may vary widely in their efficiency (and efficacy) for particular targets. We propose that, in an AAM setting, allocation characterizes polyvariance; that is, the polyvariance of an analysis can be entirely determined by the allocation of abstract addresses. Some forms of polyvariance cannot be implemented without necessary dynamic information; however, if the instrumentation of an analysis is left open as a parameter, we can show both that every form of polyvariance has an abstract allocator and that every tuning of abstract allocation represents a sound form of polyvariance. Polyvariance as a design space is not all-encompassing; however, we observe it to be surprisingly broad, and include techniques from type systems, abstract interpretations, and other methods, along with fundamental variations on the structure of analyses.

Our investigation culminates in a novel form of introspective polyvariance that yields a guarantee of perfect precision in its modeling of the call stack and incurs no asymptotic complexity overhead. We show how this form of polyvariance may be implemented in AAM-style analyses with only a trivial change to existing code and present a machine-verified proof of its bisimulation with an incomputable model of perfect stack precision.

# CONTENTS

# LIST OF FIGURES

# ACKNOWLEDGMENTS

I couldn't have written this without the lifelong nurture of my parents, who gave me a love for truth and the tools to begin looking for it. My mother, Barbara, modeled intuition, empathy, love for language, and a deep appreciation for authenticity; I never once felt that any truthful choice in my life could possibly disappoint her. I also remember, gratefully now, how many times she would have me rework something I had purportedly finished, teaching me to look for the ways it could be made better before calling it complete and to take pride in *how* and not simply *that* something is done. My father, Joe, modeled rational analytical thinking, scrupulous honesty, and a passion for understanding; from him, I learned to think math, and later programming, were thrilling puzzles and a medium in which I could be creative. I remember the many times he would work something out for me on paper at our kitchen table and his clear excitement when he would succeed in teaching me something new—his passion for reaching an *"ah hah!"* moment of clarity became mine. I am also grateful to my sister, Annika, for always believing in me and for modeling a work ethic I can still only aspire toward, and to my aunt, Audrey, for encouraging me to stay in grad school, for modeling thoughtfulness and open-mindedness, and teaching me to always look for another perspective. I have been gratuitously fortunate to also have grandparents, aunts and uncles, and good friends, who support me and continue to make life a Joy.

I want to thank all those I collaborated with and learned from while at the University of Utah, including James King, Sidharth Kumar, Steven Lyde, Michael D. Adams, William Byrd, Shuying Liang, Michael Ballantyne, Peter Aldous, David Darais, Leif Andersen, Maria Jenkins, Suresh Venkatasubramanian, Robert Kirby, and David Van Horn. I am also grateful to my committee, Mary Hall, Matthew Flatt, Zvonimir Rakamaric, and Andrew Keep, for all I learned in their classes, their lab meetings, and our discussions. I fondly remember many late nights trying to complete experiments with James King, compiler discussions with Andrew Keep, discussions on logic programming and enourmously helpful feedback on early drafts from William Byrd, and my close collaboration with Michael D. Adams. Michael was a second mentor to me in the last couple of years and our collaboration on a paper (and the content of Chapter 5), especially working out a careful proof on the whiteboard which

took a good portion of our summer, stands out as a highlight of my time in grad school. He taught me to think and speak more carefully and I am a better writer from his persistent admonitions to express myself in a more precise and thoughtfully structured way.

Finally, I want to thank my advisor, Matthew Might, for giving me this opportunity to learn from him and work in his lab. Matt values each student's love for learning and its development as much as its short-term produce, and he appreciates the importance of a student's internal motivations. Matt has his students pursue what interests them most and gives consistent support but also a high degree of independence, requiring them to develop into self-reliant researchers of their own. I recall many times in my first year at Utah, learning from Matt at his whiteboard and being not just inspired by the material he was teaching me, but by his passion for it and ease with it. I continued in graduate school in no small part due to my hope that the experience could give me qualities I admired in my advisor. Both Matt's courage in his personal life and his earnest as a scientist are an inspiration to me and his example has acted as a polestar for my own jouney as a student. I couldn't have wanted a better mentor than Matt and I hope grad school will prove to be the first of many opportunities to learn and collaborate with him.

# CHAPTER 0

# THESIS

*Polyvariance may be soundly, efficiently, and arbitrarily refined, permitting static analyses to adapt a style more efficient for their target than existing nonintrospective heuristics.*

To demonstrate this thesis, we produce a parametric static analysis that precisely ranges over all conceivable styles of polyvariance. We explore adaptive approaches which introspect on the behavior of the analysis as it runs to determine their polyvariance, and we produce one such form of introspective polyvariance that hits the perfect sweet spot in terms of efficiency. This technique guarantees a certain form of perfect precision "for free"—specifically, at no asymptotic complexity overhead, no development overhead, and lower average complexity over a set of benchmarks.

**Chapter 1** introduces central concepts from formal semantics, static analysis, and abstract interpretation using a toy calculator language. *Chapter 1 should be skipped or skimmed by readers familiar with these topics.*

**Chapter 2** introduces operational semantics for the $\lambda$-calculus in several different styles, explains their relation to familiar definitional interpreters, and primes the reader with design trade-offs relevant to approximation. *Chapter 2 should also be skimmed by readers especially familiar.*

**Chapter 3** reviews the abstracting abstract machines (AAM) methodology, polyvariance, store widening, traditional approaches to soundness, and the *a posteriori* soundness theorem which guarantees the soundness of all abstract allocators.

**Chapter 4** produces a parametric semantics freely tunable by both an instrumentation and an abstract allocator. It then surveys a variety of existing styles of polyvariance (instantiating each within our framework), novel strategies and introspective approaches, explores the design space in a more principled way, and provides tools and theory for navigating this space before or during an analysis.

**Chapter 5** introduces the problem of return-flow conflation in higher-order flow analysis, provides an incomputable model of perfect stack precision, and discusses three existing computable approaches to obtaining perfect stack precision (highlighting drawbacks and lessons learned). It then provides an unexpected form of introspective polyvariance which adapts the allocation of continuations to the behavior of an analysis ensuring perfect stack precision ("for free"). Chapter 5 also presents a machine-verified proof of this technique's equivalence with the incomputable model used to define perfect stack precision.

**Chapter 6** discusses high-performance implementation techniques. This includes extending an approach to encoding these analyses as linear algebra which may be solved efficiently on a graphics processing unit (GPU) and a novel sparse-matrix format which permits efficient dynamic matrix updates within this approach.

# CHAPTER 1

# STATIC ANALYSIS

The problem of program analysis (learning how a program behaves) may be broken into two very different approaches. *Dynamic analysis* (or *profiling*) is the task of understanding how a program behaves when it is executed. Dynamic analyses gather information from actual runs of a program which aim at being representative of that program's likely behaviors. Such information is useful for optimizing the layout of a program in memory to improve locality, for determining which portions of a program dominate a typical runtime and are most important to optimize, for debugging, and for many other applications.

*Static analysis*, by contrast, is the task of understanding how a program may behave (or must behave) given only its source text. Static analyses attempt to learn the same kind of information a programmer uses when writing code (*e.g.*, abstract data types, temporal properties, and invariants), but do so with perfect mathematical rigor to yield true guarantees regarding program behavior. While rigor is as natural for a computer as it is unnatural for a programmer, it may also be that creativity is as unnatural for a computer as it is natural for a programmer. Suitably, a part of the challenge in designing a static analysis is the distance between the enormous space of sound variations in exactly how a computer might go about this task and the much smaller range of strategies which would model any particular program particularly well. This is a challenge of balancing guarantees of analysis efficiency with guarantees of analysis precision and the challenge of being creative in how this balance is structured for a particular target of analysis.

Static analysis brings one up against the fundamental limits of computability, the halting problem, Rice's theorem (Rice, 1953), and related limitations. Any static analysis (of a Turing-equivalent language) which guarantees a precise result for a nontrival property of its target program will also be incomputable in the general case. Moreover, this same trade-off exists for computable analyses as well. An analysis which can guarantee a better complexity class for its runtime must also give up the ability to guarentee a degree of precision (*e.g.*, those properties of programs which would have required enumerating a data structure whose

size is in $\Theta(f(n))$ to prove). On the other hand, in the purely practical case, there is a well-known paradox where more precision can also tend to be less expensive. For example, Wright and Jagannathan (1998) discuss an exponential-time analysis which in many typical cases improves on both the precision and performance of a classic polynomial-time analysis.

The next chapters discuss a principled approach to constructing formal semantics of programming languages, and functional (higher-order) languages in particular, and to abstracting those semantics to obtain an approximation in a variety of styles. The remainder of this chapter introduces formal semantics using a toy calculator language, applies the methodology of abstract interpretation to obtain an approximation of these semantics, and discusses some of the central challenges to scaling these ideas up to production.

## 1.1 Formal Semantics

Consider a language of basic arithmetic where we can add, subtract, and multiply integers. We can define the language inductively using a context-free grammar.

$$
\begin{array}{rll}
a \in \mathsf{ArithExp} ::= & \texttt{(+}\ a\ a\texttt{)} & \text{[addition]} \\
\mid & \texttt{(-}\ a\ a\texttt{)} & \text{[subtraction]} \\
\mid & \texttt{(*}\ a\ a\texttt{)} & \text{[multiplication]} \\
\mid & z & \text{[a constant]} \\
z \in \mathbb{Z} \triangleq & \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\} & \text{[integers]}
\end{array}
$$

Each $a$ is either an arithmetic expression of the form `(+` $a_1$ $a_2$`)` or `(-` $a_1$ $a_2$`)` or `(*` $a_1$ $a_2$`)` or it is simply a particular integer z. Each $a_1$ and $a_2$ inside $a$ are themselves expressions, allowing us to build up arbitrarily complex arithmetic (using our three operations), and making ArithExp an infinite set.

With only the set ArithExp defined, its elements $a$ are meaningless pieces of syntax. A value like `(+ 1 1)` does not mean 2, is not equivalent to 2, and cannot be reduced to 2, unless we formally define an *interpretation* for it which connects the dots with mathematical rigor. There are a many good systems for doing this, and for defining computer languages, there are at least three fundamental approaches which are common: axiomatic semantics, denotational semantics, and operational semantics (Winskel, 1993). An *axiomatic semantics* defines a language in terms of logical constraints—"this is what is true about an expression". A *denotational semantics* defines a language in terms of otherwise established mathematical entities—"this is what an expression denotes within an understood setting". An *operational semantics* defines a language in terms of its operational behavior—"this is how an expression

is reduced and evaluated step by step". Each of these approaches has its advantages and applications which suit it well. We will use operational semantics because they are related to practical program interpreters and are most similar to language implementations and the operation of actual computing machinery. An operational semantics is also convenient for bisimulation proofs which require us to take subtle differences in program evaluation into account.

We can define an operational semantics of our arithmetic language using a rule of inference for each syntactic form in the language. We define an *evaluation function* which reduces a given expression to its integer value.

$$(\rightarrow_{\text{calc}}) : \mathsf{ArithExp} \rightarrow \mathbb{Z}$$

In each rule, the statements above the line are called the antecedents, and the statement under the line is called their consequent; one logical statement proceeds from a conjunction of the others.

$$\frac{z \in \mathbb{Z}}{z \rightarrow_{\text{calc}} z} \qquad \frac{a_1 \rightarrow_{\text{calc}} z_1 \qquad a_2 \rightarrow_{\text{calc}} z_2}{\texttt{(+}\ a_1\ a_2\texttt{)} \rightarrow_{\text{calc}} z_1 + z_2}$$

$$\frac{a_1 \rightarrow_{\text{calc}} z_1 \qquad a_2 \rightarrow_{\text{calc}} z_2}{\texttt{(-}\ a_1\ a_2\texttt{)} \rightarrow_{\text{calc}} z_1 - z_2} \qquad \frac{a_1 \rightarrow_{\text{calc}} z_1 \qquad a_2 \rightarrow_{\text{calc}} z_2}{\texttt{(*}\ a_1\ a_2\texttt{)} \rightarrow_{\text{calc}} z_1 \cdot z_2}$$

The first rule is the axiom of our formal system. It says, if $z$ is an integer, then $z$ evaluates to itself. The second rule defines the meaning of `(+` $a_1$ $a_2$`)`, and so all addition in our language. It says, if $a_1$ evaluates to an integer $z_1$ and $a_2$ evaluates to an integer $z_2$, then `(+` $a_1$ $a_2$`)` evaluates to an integer $z_1 + z_2$. We can visualize the full derivation tree for an expression `(+ 1 2)` like so:

$$\frac{\dfrac{1 \in \mathbb{Z}}{1 \rightarrow_{\text{calc}} 1} \qquad \dfrac{2 \in \mathbb{Z}}{2 \rightarrow_{\text{calc}} 2}}{\texttt{(+ 1 2)} \rightarrow_{\text{calc}} 3}$$

Because 1 is an integer, the syntax 1 evalutes to the integer 1. Because 2 is an integer, the syntax 2 evalutes to the integer 2. Because 1 evaluates to 1 and 2 evaluates to 2, `(+ 1 2)` evaluates to $1 + 2$ and thus 3.

Now consider the expression:

```
(+ (+ 1 4) (* (- 5 8) 2))
```

A full derivation tree for this expression follows from the same principals as before:

$$\dfrac{\dfrac{\dfrac{1 \in \mathbb{Z}}{1 \to_{\text{calc}} 1} \quad \dfrac{4 \in \mathbb{Z}}{4 \to_{\text{calc}} 4}}{\texttt{(+ 1 4)} \to_{\text{calc}} 5} \quad \dfrac{\dfrac{\dfrac{5 \in \mathbb{Z}}{5 \to_{\text{calc}} 5} \quad \dfrac{8 \in \mathbb{Z}}{8 \to_{\text{calc}} 8}}{\texttt{(- 5 8)} \to_{\text{calc}} -3} \quad \dfrac{2 \in \mathbb{Z}}{2 \to_{\text{calc}} 2}}{\texttt{(* (- 5 8) 2)} \to_{\text{calc}} -6}}{\texttt{(+ (+ 1 4) (* (- 5 8) 2))} \to_{\text{calc}} -1}$$

## 1.2 Abstract Interpretation

Abstract interpretation is a particular theory for the static analysis of programs, put forth in the course of a series of papers: Cousot and Cousot (1976, 1977a,b,c). The central idea is to statically determine approximate information about a program from a concrete or exact semantics as automatically as possible. Instead of performing an interpretation of a language which precisely determines the meaning of each expression, we derive an *abstract interpretation* which approximately determines the meaning of each expression. To produce an abstract semantics given our concrete (exact) semantics, we first define abstraction and concretization functions which connect our concrete domains (*e.g.*, $\mathbb{Z}$) to abstract ones (*e.g.*, $\hat{\mathbb{Z}}$). Typographically, we give abstract entities a hat to easily differentiate between concrete and abstract without losing track of basic underlying roles. In order to perform a *sign analysis* which differentiates between positive and negative integers where possible (but does not differentiate between two positive integers), we could define abstract integers as a set of signs:

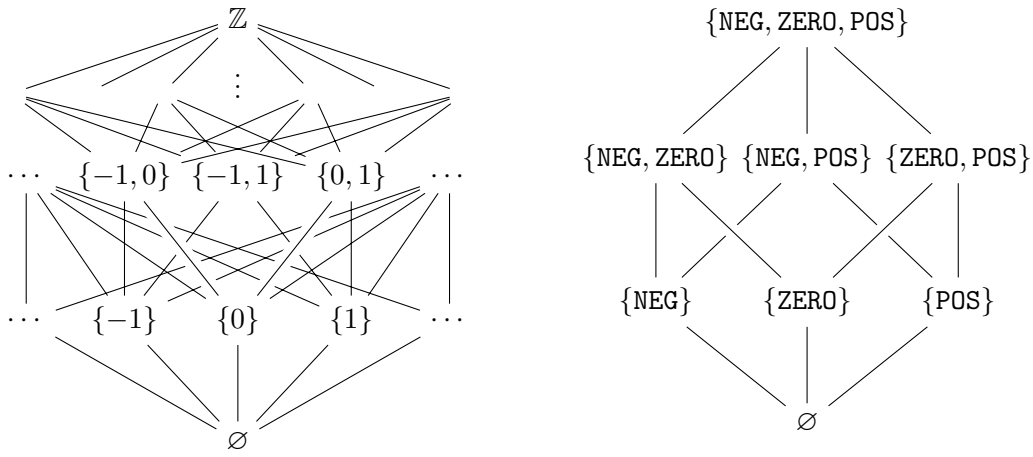
$$\hat{z} \in \hat{\mathbb{Z}} \triangleq \{\texttt{NEG}, \texttt{ZERO}, \texttt{POS}\} \qquad \text{[signs]}$$

From here, we would like to produce composable abstraction and concretization functions which describe the relationship between our concrete interpretation ($\to_{\text{calc}}$) and our forthcoming sign analysis ($\rightsquigarrow_{\text{sign}}$). As our approximation stands, however, we would need to allow imprecision within our concrete integers domain so a value like `POS` has a particular value it concretizes to. We would like to say it stands in for $\{1, 2, 3, \ldots\}$ (*i.e.*, all values that abstract to `POS`), so we must use sets of exact integers as our concrete values domain. This, in turn, necessitates we allow further imprecision within our abstract integers domain so a range of concrete values like $\{0, 1, 2\}$ can be abstracted to $\{\texttt{ZERO}, \texttt{POS}\}$ (which would stand in for values that are either zero, or a positive integer). For this reason, we lift both integers and signs to respective lattices (*e.g.*, their power sets).

$$zs \in ZS \triangleq \mathcal{P}(\mathbb{Z}) \qquad \text{[concrete values]}$$

$$\widehat{zs} \in \widehat{ZS} \triangleq \mathcal{P}(\hat{\mathbb{Z}}) \qquad \text{[abstract values]}$$

Every granularity of precision in both domains now has a nearest correspondent in the other.




To connect our concrete domains to our abstract domains, we produce an abstraction function $\alpha_{ZS} : ZS \to \widehat{ZS}$ and a concretization function $\gamma_{ZS} : \widehat{ZS} \to ZS$. For our simple sign analysis, these functions could be:

$$\begin{aligned}
\alpha_{ZS}(zs) \triangleq\ & \{\texttt{NEG} \mid z \in zs \wedge z < 0\} \\
& \cup \{\texttt{ZERO} \mid z \in zs \wedge z = 0\} \\
& \cup \{\texttt{POS} \mid z \in zs \wedge z > 0\} \\
\gamma_{ZS}(\widehat{zs}) \triangleq\ & \{z \mid z < 0 \wedge z \in \mathbb{Z} \wedge \texttt{NEG} \in \widehat{zs}\} \\
& \cup \{z \mid z = 0 \wedge \texttt{ZERO} \in \widehat{zs}\} \\
& \cup \{z \mid z > 0 \wedge z \in \mathbb{Z} \wedge \texttt{POS} \in \widehat{zs}\}
\end{aligned}$$

To construct different approximations, and thereby different kinds of analyses, we would vary the abstract domains and Galois connections used. We must respect two important properties, however. First, both functions (abstraction and concretization) must be monotonic (*i.e.*, $zs_1 \subseteq zs_2 \implies \alpha_{ZS}(zs_1) \subseteq \alpha_{ZS}(zs_2)$). Second, both functions are constrained by the other: $\alpha_{ZS}(zs) \subseteq \widehat{zs} \iff zs \subseteq \gamma_{ZS}(\widehat{zs})$. These two properties cause our defined notions of abstraction and concretization to form a monotone *Galois connection*:

$$ZS \underset{\alpha_{ZS}}{\overset{\gamma_{ZS}}{\leftrightarrows}} \widehat{ZS}$$

Each of these functions uniquely determines its adjoint. Given a monotonic abstraction function only (*e.g.*, $\alpha_{ZS}$), its corresponding concretization function can be calculated:

$$\gamma_{ZS}(\widehat{zs}) = \bigcup \{zs \in ZS \mid \alpha_{ZS}(zs) \subseteq \widehat{zs}\}$$

That some $\alpha$ and $\gamma$ form a monotone Galois connection further implies that $\gamma \circ \alpha$ is expansive and that $\alpha \circ \gamma$ is reductive:

$$Id \sqsubseteq \gamma \circ \alpha \qquad \textit{i.e.,} \quad \forall zs \in ZS.\ zs \subseteq \gamma_{ZS}(\alpha_{ZS}(zs))$$
$$\alpha \circ \gamma \sqsubseteq Id \qquad \textit{i.e.,} \quad \forall \widehat{zs} \in \widehat{ZS}.\ \alpha_{ZS}(\gamma_{ZS}(\widehat{zs})) \subseteq \widehat{zs}$$

With a Galois connection for integers, we may produce a derived Galois connection for arithmetic expressions. We start by generalizing exact arithmetic expressions to a concrete lattice and producing a corresponding abstract lattice.

$$\begin{aligned} as \in AS ::= {} & \texttt{(+}\ as\ as\texttt{)} \\ \mid {} & \texttt{(-}\ as\ as\texttt{)} \\ \mid {} & \texttt{(*}\ as\ as\texttt{)} \\ \mid {} & zs \end{aligned} \qquad\qquad \begin{aligned} \widehat{as} \in \widehat{AS} ::= {} & \texttt{(+}\ \widehat{as}\ \widehat{as}\texttt{)} \\ \mid {} & \texttt{(-}\ \widehat{as}\ \widehat{as}\texttt{)} \\ \mid {} & \texttt{(*}\ \widehat{as}\ \widehat{as}\texttt{)} \\ \mid {} & \widehat{zs} \end{aligned}$$

Although not set inclusion ($\subseteq$) as with our power-set domains, the orders for these lattices ($\sqsubseteq$) are straightforward and distribute over each of the arithmetic operations. For example, $\widehat{as}_1 \sqsubseteq \widehat{as}'_1 \wedge \widehat{as}_2 \sqsubseteq \widehat{as}'_2 \iff \texttt{(+}\ \widehat{as}_1\ \widehat{as}_2\texttt{)} \sqsubseteq \texttt{(+}\ \widehat{as}'_1\ \widehat{as}'_2\texttt{)}$.

To connect our domains of concrete syntax with their respective concrete lattice domains, we define a pair of injection functions $\mathcal{I}_{ZS} : \mathbb{Z} \to ZS$ and $\mathcal{I}_{AS} : \mathsf{ArithExp} \to AS$ which place exact syntactic values within their concrete lattices.

$$\mathcal{I}_{ZS}(z) \triangleq \{z\}$$
$$\mathcal{I}_{AS}(a) \triangleq \begin{cases} \texttt{(+}\ \mathcal{I}_{AS}(a_1)\ \mathcal{I}_{AS}(a_2)\texttt{)} & a = \texttt{(+}\ a_1\ a_2\texttt{)} \\ \texttt{(-}\ \mathcal{I}_{AS}(a_1)\ \mathcal{I}_{AS}(a_2)\texttt{)} & a = \texttt{(-}\ a_1\ a_2\texttt{)} \\ \texttt{(*}\ \mathcal{I}_{AS}(a_1)\ \mathcal{I}_{AS}(a_2)\texttt{)} & a = \texttt{(*}\ a_1\ a_2\texttt{)} \\ I_{ZS}(z) & a = z \end{cases}$$

Then, as before, we may define a Galois connection between $AS$ and $\widehat{AS}$.

$$\alpha_{AS}(as) \triangleq \begin{cases} \texttt{(+}\ \alpha_{AS}(as_1)\ \alpha_{AS}(as_2)\texttt{)} & as = \texttt{(+}\ as_1\ as_2\texttt{)} \\ \texttt{(-}\ \alpha_{AS}(as_1)\ \alpha_{AS}(as_2)\texttt{)} & as = \texttt{(-}\ as_1\ as_2\texttt{)} \\ \texttt{(*}\ \alpha_{AS}(as_1)\ \alpha_{AS}(as_2)\texttt{)} & as = \texttt{(*}\ as_1\ as_2\texttt{)} \\ \alpha_{ZS}(zs) & as = zs \end{cases}$$

We may now consider abstracting an expression, and then evaluating it approximately, using some abstract evaluation function ($\rightsquigarrow_{\text{sign}}$), to determine which sign or signs its resulting value could be.

$$\begin{array}{ccc}
\texttt{(+ (+ 1 4) (* (- 5 8) 2))} & \xrightarrow{\rightarrow_{\text{calc}}} & -1 \\
\Big\downarrow{\scriptstyle \mathcal{I}_{AS}} & & \\
\texttt{(+ (+ \{1\} \{4\}) (* (- \{5\} \{8\}) \{2\}))} & & \\
\Big\downarrow{\scriptstyle \alpha_{AS}} & & \\
\texttt{(+ (+ \{POS\} \{POS\}) (* (- \{POS\} \{POS\}) \{POS\}))} & \xrightarrow{\rightsquigarrow_{\text{sign}}} & \texttt{?}
\end{array}$$

As we have not defined ($\rightsquigarrow_{\text{sign}}$), we are not sure what value it would produce (hence a "**?**"). We know that a concrete evaluation of the example expression yields -1 and that -1 injected and abstracted is {`NEG`}.

$$\begin{array}{ccc}
\texttt{(+ (+ 1 4) (* (- 5 8) 2))} & \xrightarrow{\rightarrow_{\text{calc}}} & -1 \\
\Big\downarrow{\scriptstyle \mathcal{I}_{AS}} & & \Big\downarrow{\scriptstyle \mathcal{I}_{ZS}} \\
\texttt{(+ (+ \{1\} \{4\}) (* (- \{5\} \{8\}) \{2\}))} & & \{-1\} \\
\Big\downarrow{\scriptstyle \alpha_{AS}} & & \Big\downarrow{\scriptstyle \alpha_{ZS}} \\
\texttt{(+ (+ \{POS\} \{POS\}) (* (- \{POS\} \{POS\}) \{POS\}))} & & \{\texttt{NEG}\}
\end{array}$$

If the abstract evaluation function ($\rightsquigarrow_{\text{sign}}$) must produce values that soundly simulate those computed by ($\rightarrow_{\text{calc}}$), then we know whatever approximate value it yields should be at least a superset of {`NEG`}, with {`NEG`} itself representing the best possible precision. If the value given by ($\rightsquigarrow_{\text{sign}}$) does not include the possibility of negative numbers, then our approximate evaluation of the expression $\widehat{as}$ is unsound (incorrect).

In the general case, we must show that:

$$a \rightarrow_{\text{calc}} z \wedge \alpha_{AS}(\mathcal{I}_{AS}(a)) \sqsubseteq \widehat{as} \implies \widehat{as} \rightsquigarrow_{\text{sign}} \widehat{zs} \wedge \alpha_{ZS}(\mathcal{I}_{ZS}(z)) \subseteq \widehat{zs}$$

Or diagrammatically:

$$\begin{array}{ccc}
a & \xrightarrow{\rightarrow_{\text{calc}}} & z \\
\Big\downarrow{\scriptstyle \mathcal{I}_{AS}} & & \Big\downarrow{\scriptstyle \mathcal{I}_{ZS}} \\
as & & zs \\
\Big\downarrow{\scriptstyle \alpha_{AS}} \sqsubseteq & & \Big\downarrow{\scriptstyle \alpha_{ZS}} \subseteq \\
\widehat{as} & \xrightarrow{\rightsquigarrow_{\text{sign}}} & \widehat{zs}
\end{array}$$

This *soundness* property can form a basis for either justifying an abstract interpretation as sound *a posteriori*, or for constructing one such that it is sound (Cousot and Cousot, 1979).

$$\frac{\widehat{zs} \in \widehat{ZS}}{\widehat{zs} \rightsquigarrow_{\text{sign}} \widehat{zs}} \qquad \frac{\widehat{as}_1 \rightsquigarrow_{\text{sign}} \widehat{zs}_1 \qquad \widehat{as}_2 \rightsquigarrow_{\text{sign}} \widehat{zs}_2}{(\texttt{+}\ \widehat{as}_1\ \widehat{as}_2) \rightsquigarrow_{\text{sign}} \widehat{zs}_1 \oplus \widehat{zs}_2}$$

$$\frac{\widehat{as}_1 \rightsquigarrow_{\text{sign}} \widehat{zs}_1 \qquad \widehat{as}_2 \rightsquigarrow_{\text{sign}} \widehat{zs}_2}{(\texttt{-}\ \widehat{as}_1\ \widehat{as}_2) \rightsquigarrow_{\text{sign}} \widehat{zs}_1 \ominus \widehat{zs}_2} \qquad \frac{\widehat{as}_1 \rightsquigarrow_{\text{sign}} \widehat{zs}_1 \qquad \widehat{as}_2 \rightsquigarrow_{\text{sign}} \widehat{zs}_2}{(\texttt{*}\ \widehat{as}_1\ \widehat{as}_2) \rightsquigarrow_{\text{sign}} \widehat{zs}_1 \odot \widehat{zs}_2}$$

Each rule of inference in $(\rightarrow_{\text{calc}})$ has a corresponding rule in $(\rightsquigarrow_{\text{sign}})$. We can directly calculate the most precise possible operations $(\oplus)$, $(\ominus)$, $(\odot)$ on sets of signs $(\widehat{zs})$ such that the abstract evaluation function $(\rightsquigarrow_{\text{sign}})$ is sound with respect to $(\rightarrow_{\text{calc}})$ and $\alpha_{ZS}$. The most precise sound abstract interpretation is the one which behaves the same as concretizing, concretely interpreting, and then abstracting again.

$$\widehat{zs}_1 \oplus \widehat{zs}_2 = \alpha_{ZS}(\{z_1 + z_2 \mid z_1 \in \gamma_{ZS}(\widehat{zs}_1) \wedge z_2 \in \gamma_{ZS}(\widehat{zs}_2)\})$$
$$\widehat{zs}_1 \ominus \widehat{zs}_2 = \alpha_{ZS}(\{z_1 - z_2 \mid z_1 \in \gamma_{ZS}(\widehat{zs}_1) \wedge z_2 \in \gamma_{ZS}(\widehat{zs}_2)\})$$
$$\widehat{zs}_1 \odot \widehat{zs}_2 = \alpha_{ZS}(\{z_1 \cdot z_2 \mid z_1 \in \gamma_{ZS}(\widehat{zs}_1) \wedge z_2 \in \gamma_{ZS}(\widehat{zs}_2)\})$$

Given these tunings, for example, the operation $\{\texttt{POS}\} \odot \{\texttt{NEG}\}$ yields $\{\texttt{NEG}\}$ and the operation $\{\texttt{NEG}\} \ominus \{\texttt{NEG}\}$ yields $\{\texttt{POS}, \texttt{ZERO}, \texttt{NEG}\}$. If any of these operations were defined to always return $\{\texttt{POS}, \texttt{ZERO}, \texttt{NEG}\}$, only precision would be lost and not soundness.

The proof tree for $\alpha_{AS}(\mathcal{I}_{AS}(\texttt{(+ 1 2)})) \rightsquigarrow_{\text{sign}} \{\texttt{POS}\}$ is:

$$\frac{\dfrac{\{\texttt{POS}\} \in \widehat{ZS}}{\{\texttt{POS}\} \rightsquigarrow_{\text{sign}} \{\texttt{POS}\}} \qquad \dfrac{\{\texttt{POS}\} \in \widehat{ZS}}{\{\texttt{POS}\} \rightsquigarrow_{\text{sign}} \{\texttt{POS}\}}}{(\texttt{+}\ \{\texttt{POS}\}\ \{\texttt{POS}\}) \rightsquigarrow_{\text{sign}} \{\texttt{POS}\}}$$

The proof tree for our larger example would look like:

$$\frac{\dfrac{\dfrac{\{\texttt{POS}\} \in \widehat{ZS}}{\{\texttt{POS}\} \rightsquigarrow_{\text{sign}} \{\texttt{POS}\}} \qquad \dfrac{\{\texttt{POS}\} \in \widehat{ZS}}{\{\texttt{POS}\} \rightsquigarrow_{\text{sign}} \{\texttt{POS}\}}}{(\texttt{+}\ \{\texttt{POS}\}\ \{\texttt{POS}\}) \rightsquigarrow_{\text{sign}} \{\texttt{POS}\}} \qquad \dfrac{\dfrac{\{\texttt{POS}\} \in \widehat{ZS}}{\{\texttt{POS}\} \rightsquigarrow_{\text{sign}} \{\texttt{POS}\}} \qquad \dfrac{\{\texttt{POS}\} \in \widehat{ZS}}{\{\texttt{POS}\} \rightsquigarrow_{\text{sign}} \{\texttt{POS}\}}}{(\texttt{-}\ \{\texttt{POS}\}\ \{\texttt{POS}\}) \rightsquigarrow_{\text{sign}} \{\texttt{POS}, \texttt{ZERO}, \texttt{NEG}\}} \qquad \dfrac{\{\texttt{POS}\} \in \widehat{ZS}}{\{\texttt{POS}\} \rightsquigarrow_{\text{sign}} \{\texttt{POS}\}}}{(\texttt{*}\ (\texttt{-}\ \{\texttt{POS}\}\ \{\texttt{POS}\})\ \{\texttt{POS}\}) \rightsquigarrow_{\text{sign}} \{\texttt{POS}, \texttt{ZERO}, \texttt{NEG}\}}}{(\texttt{+}\ (\texttt{+}\ \{\texttt{POS}\}\ \{\texttt{POS}\})\ (\texttt{*}\ (\texttt{-}\ \{\texttt{POS}\}\ \{\texttt{POS}\})\ \{\texttt{POS}\})) \rightsquigarrow_{\text{sign}} \{\texttt{POS}, \texttt{ZERO}, \texttt{NEG}\}}$$

Unfortunately, this abstract interpretation has given us no information at all. It is a good example of how even the most precise abstract interpretation which respects a given Galois connection can easily lose precision due to the fundamental loss of structure in the abstract domains. Our style of abstraction is simply unable to determine anything in general about a positive number subtracted from another positive number. Because imprecisions become compounded as evaluation progresses, this top value $(\top)$, which says nothing about the number it represents, propagates all the way to our final result.

## 1.3 Central Challenges

Dynamic languages, and dynamic program behaviors generally, are notoriously difficult to model precisely. Much work has gone into simply defining a correct concrete semantics of Python and JavaScript (Guth, 2013; Ranson et al., 2008; Smeding, 2009). Perhaps the most compelling effort to date at giving a complete and tractable semantics for JavaScript has been Guha et al. (2010) with $\lambda_{JS}$ and its successor, Politz et al. (2012) with $\lambda_{S5}$. This approach reduces programs to a simple core language consisting of fewer than 35 syntactic forms, reifying the hidden and implicit complexity of full JavaScript as explicit complexity written in the core language.

Desugaring is appealing for analysis designers as it gives a simple and precise semantics to abstract; however, it also presents one of the major obstacles to precise analysis as it adds a significant runtime environment and layers of indirection through it. Consider an example the authors of $\lambda_{S5}$ use to motivate the need for their carefully constructed semantics: `[] + {}` yields the string `"[object Object]"`. Strangely enough, this behavior is correct as defined by the ECMAScript specification for addition—a complex algorithm encompassing a number of special cases which can interact in unexpected ways (ECMA, 2011). The desugaring process for $\lambda_{S5}$ replaces addition with a function call to `%PrimAdd` from the runtime environment. `%PrimAdd` in-turn calls `%ToPrimitive` on both its arguments before breaking into cases. This means that for any uses of addition to return precise results, or likely anything other than $\top$ (a total loss of precision), an analysis is needed which can distinguish between different uses of `%ToPrimitive` through multiple layers of indirection. An analysis of `%ToPrimitive` which is unique both to its call site in `%PrimAdd` and to `%PrimAdd`'s call site in turn could be capable of producing a precise value. *Context sensitive* techniques like this are forms of polyvariance, our broader subject, and increase the expense of an analysis but also its potential for higher precision.

Unlike the toy language of this chapter, a concrete evaluation of a program written in a Turing-equivalent language may not terminate even when the program is itself finite. Finitely modeling programs with this capacity for arbitrary complexity requires an analysis designer to select an appropriate compromise between computational expense and precision. In the case of call sensitivity (context sensitivity using calling context) and many other forms of polyvariance, the compromise is an unpleasant one leading to analyses with worst-case run times that are exponential in the size of the program (Van Horn and Mairson, 2008). As the desugarer for $\lambda_{S5}$ bloats even small programs to more than ten thousand lines, applying context-sensitivity to every function in this case is sure to result in an absurdly

time-consuming analysis. Devising strategies for performing a compromise which is more efficient in general, or which is better adapted to a particular target of analysis, amounts to one of the central challenges of static analysis.

Better understanding the design space of these trade-offs, allowing for introspection on the ongoing behavior of a particular analysis, and adapting the compromise being struck to suite the target of this analysis and its goals will allow us to take important steps in tackling the central challenges of static analysis in practice.

# CHAPTER 2

# OPERATIONAL SEMANTICS

This chapter explores a variety of operational abstract-machine semantics for the untyped $\lambda$-calculus, introduces major components of the machines we will be using, and addresses some basic trade-offs in designing a semantics for a functional (higher-order) language. As we progress through this chapter, each *formal semantics* (*i.e.*, a formal system defining a language) will be accompanied by an implementation written in Racket (a dialect of Lisp based on Scheme) to help the intuition for each development (Racket Community, 2015). A basic familiarity with the untyped $\lambda$-calculus and Scheme-like programming languages will be assumed.

Note, regarding scope, that some symbols will be redefined for each new machine introduced; these symbols are scoped within their local sections.

## 2.1 Big-Step Semantics

Plotkin (1981) describes a general approach to specifying an evaluation-oriented (*i.e.*, operational) semantics called *structural operational semantics* (SOS). The main idea of SOS is to break a language into managable components and, using simple manipulations of these, to define a syntax-directed transition function which can step program evaluation incrementally from start to finish. Plotkin points out that a transition between machine states can often be alternately viewed as a single step or viewed as multiple steps and adds "part of the spirit of our method is to choose steps of the appropriate size". Kahn (1987) proposes that a natural and concise approach to SOS is to define inductive transitions which only take "big" steps directly to a result value. A specification like this is called a *big-step* operational semantics or a *natural* semantics. A big-step semantics can frequently be defined using fewer rules than other SOS approaches; they tend to be simple and intuitive. Big-step semantics are the closest operational semantics to denotational semantics in the sense that they interpret programs directly into values. A denotational semantics connects syntax to entities in an understood setting and so relies on having established meanings for its

```
(define (eval e [rho (hash)])
        (match e
               [`(,efun ,earg)
                (define vfun (eval efun rho)†)
                (define varg (eval earg rho)†)
                (match vfun
                       [`((lambda (,x) ,ebody) ,rho_lam)
                        (eval ebody (hash-set rho_lam x varg))‡])]
               [`(lambda (,x) ,ebody)
                `((lambda (,x) ,ebody) ,rho)]
               [(? symbol? x)
                (hash-ref rho x)]))
```

**Figure 2.1**. An interpreter for the direct-style, untyped $\lambda$-calculus, written in Racket. †Direct-style calls. ‡A tail-recursive call.

denotations. In a big-step operational semantics, the transition function does all the work of evaluating a program internally and so encapsulates the meaning of program syntax.

We may define the untyped $\lambda$-calculus inductively using a context-free grammar:

| | |
|---|---|
| $f, e \in \mathsf{LamExp} ::= \texttt{(}f\ e\texttt{)}$ | [application] |
| $\mid lam$ | [$\lambda$-abstraction] |
| $\mid x$ | [variable reference] |
| $lam \in \mathsf{Lam} ::= \texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e\texttt{)}$ | [$\lambda$-abstractions] |
| $x, y \in \mathsf{Var}$ is a set of identifiers | [variables] |

Figure 2.1 shows an interpreter for this language. Expressions are evaluated in the context of an environment `rho`. Variable references are determined by the current environment. Closure creation pairs a syntactic lambda with the current environment. Call sites require a recursive call to `eval` for each subexpression before making a tail-recursive call for evaluating the body of the function being applied. We allow the stack (*i.e.*, the continuation or evaluation context) to be modeled implicitly, in meta-circular fashion, by the stack in the host language.

Formally, program states ($\varsigma$) range over pairs of a *control expression* ($e$) and an *environment* ($\rho$). Environments in turn are partial functions from variables to values. As our language is the pure $\lambda$-calculus with no primitive values or operations added, every value is a closure *clo* which pairs a syntactic lambda with an environment over which it is closed.

$$\varsigma \in \Sigma \triangleq \mathsf{LamExp} \times \mathit{Env}$$

$$\rho \in \mathit{Env} \triangleq \mathsf{Var} \rightharpoonup \mathit{Clo}$$

$$\mathit{clo} \in \mathit{Clo} \triangleq \mathsf{Lam} \times \mathit{Env}$$

We can produce an evaluation function $(\rightarrow_\lambda) : \Sigma \rightharpoonup \mathit{Clo}$ which reduces a machine state to its value, a closure. This function can be defined formally using a rule of inference for each case in our grammar. First, a lambda in the context of an environment reduces to a closure which pairs that syntactic lambda with the current environment.

$$\frac{}{((\lambda\ (x)\ e), \rho) \rightarrow_\lambda ((\lambda\ (x)\ e), \rho)}\ [\text{clo}]$$

Second, a variable reference in the context of an environment reduces to the closure bound to that variable within the current environment.

$$\frac{}{(x, \rho) \rightarrow_\lambda \rho(x)}\ [\text{var}]$$

Third, an application in the context of an environment reduces both the expression in call position and the expression in argument position, in the context of the current environment, and then yields whatever value the body of the function invoked reduces to when evaluated in the context of its closure's environment ($\rho_\lambda$) extended with a binding for its parameter.

$$\frac{(e_1, \rho) \rightarrow_\lambda ((\lambda\ (x)\ e_3), \rho_\lambda) \qquad (e_2, \rho) \rightarrow_\lambda \mathit{clo} \qquad (e_3, \rho_\lambda[x \mapsto \mathit{clo}]) \rightarrow_\lambda \mathit{clo}'}{((e_1\ e_2), \rho) \rightarrow_\lambda \mathit{clo}'}\ [\text{app}]$$

Each of these formal rules corresponds to one of the three match clauses in the Racket interpreter from Figure 2.1. The first two antecedents in the deduction for a call site are effectively direct-style recursive uses of $(\rightarrow_\lambda)$, while the third antecedent is effectively tail recursion.

This reveals several possible disadvantages to using a big-step semantics, particularly for the purposes of approximation to a static analysis. To begin with, nontermination is possible within a $(\rightarrow_\lambda)$-step at any point during evaluation and where a step does not terminate, neither does it have a proof tree. In addition, there are multiple ways nonermination is possible, due to an infinite number of tail recursions, due to an infinite number of direct style recursions, or some combination of the two (visually, proof trees are unbounded both vertically and horizontally).

Furthermore, a big-step semantics gains some of its concision by allowing the call stack to be represented implicitly instead of explicitly; just as our big-step Racket interpreter was able to model the stack in a meta-circular way, so is the stack represented implicitly in the

structure of proof trees for $(\rightarrow_\lambda)$ deductions. If some $(e, \rho) \rightarrow_\lambda clo$ (read "e, rho reduces to clo") in the context of further inferences, then it is effectively being reduced on the top of a stack; if not, then the stack is empty. For the purposes of deriving a static analysis from our semantics, it may not be desirable to hide a program's behavior in this way.

In Figure 2.2, some examples of proof trees for $(\rightarrow_\lambda)$ are worked out.

## 2.2 Small-Step Semantics

A *small-step* operational semantics is one where each transition relation is computable, even if the overall evaluation induced by these individual steps may not terminate (Hennessy, 1990). The $(\rightarrow_\lambda)$-machine could nonterminate in multiple ways within every reduction step. The abstract machines of this section can only nonterminate because an infinite number of steps is required to evaluate a program, never because any specific step nonterminates. In a Racket interpreter implementing a small-step semantics, every recursive call the interpreter makes to itself must be a tail call. Unlike a big-step semantics, a small-step semantics is able to give an accurate account of even a nonterminating program as an infinite sequence of computable steps. This means that a small-step semantics provides a result we can finitely approximate, even when it cannot be finitely computed. Transforming our semantics to use small, computable steps will make it easier to produce a finite approximation of these semantics in the next chapter.

This section presents two alternative approaches to transforming our semantics to a small-step SOS. One approach is to explicitly model the stack, giving us finer control over its behavior and representation and making it easier to explicitly approximate in the next chapter. Another approach is to entirely elide the problem of managing a stack by using an intermediate representation (IR) in continuation passing style.

### 2.2.1 Continuation-Passing-Style Semantics

Continuation passing style (CPS) constrains call sites to tail position so functions may never return; instead, callers must explicitly pass a continuation forward to be invoked on the result (Plotkin, 1975). This makes our semantics tail recursive (small-step) and more convenient to approximate while skipping over the challenges of manually managing a stack and its abstraction.

CPS is a broadly applicable transformation for compiler optimization and program analysis (Appel, 2007). If the transformation to CPS records which lambdas correspond to continuations, a program may again, along with any optimizations and analysis results, be precisely reconstituted in direct-style form. This means the advantages of CPS can

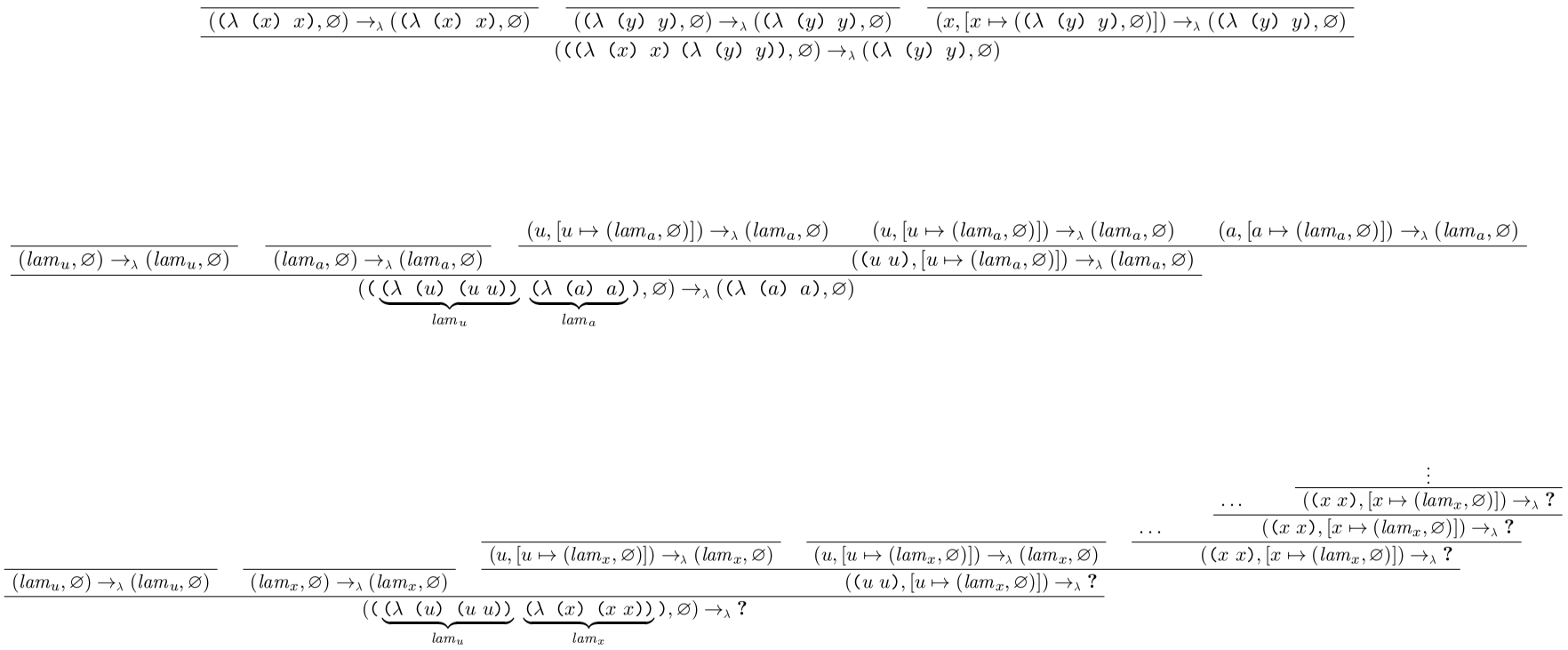



**Figure 2.2**. Example proof trees for big-step $\lambda$-calculus. The last program (a.k.a. Omega—the u-combinator applied to itself) nonterminates and thus has no valid proof tree. A "**?**" is used to indicate nontermination before a value could be reached.

be utilized without compromise or loss of information (Kennedy, 2007). The grammar structurally distinguishes between call-sites *call* and atomic expressions *ae*:

$$\begin{aligned} \mathit{call} \in \mathsf{Call} &::= \texttt{(}\mathit{ae}\ \mathit{ae}\ \texttt{...)} \mid \texttt{(halt}\ \mathit{ae}\texttt{)} \\ \mathit{lam} \in \mathsf{Lam} &::= \texttt{(}\lambda\ \texttt{(}x\ \texttt{...)}\ \mathit{call}\texttt{)} \\ \mathit{ae} \in \mathsf{AExp} &::= \mathit{lam} \mid x \\ x &\in \mathsf{Var}\ \textit{is a set of program variables} \end{aligned}$$

We define the evaluation of programs in this language using a relation ($\rightarrow_{\text{CE}}$), over states of an abstract-machine, which determines how the machine transitions from one state to another. Instead of an evaluation function which interprets machine states directly into values as in our big-step semantics, we use a transition function which produces the "next" state of the machine, having performed one small step of computation. States ($\varsigma$) range over control expression (a call site), and binding environment components:

$$\begin{aligned} \varsigma \in \Sigma &\triangleq \mathsf{Call} \times \mathit{Env} \\ \rho \in \mathit{Env} &\triangleq \mathsf{Var} \rightharpoonup \mathit{Clo} \\ \mathit{clo} \in \mathit{Clo} &\triangleq \mathsf{Lam} \times \mathit{Env} \end{aligned}$$

The only difference between the domains of our big-step machine and those of this one is that every expression is a call site which may not return and whose arguments may be atomically evaluated. Otherwise, environments and closures function as before.

Evaluation of atomic expressions is handled by an auxiliary function ($\mathcal{A}$) which produces a value (*clo*) for an atomic expression in the context of a state ($\varsigma$). This is done by a lookup in the environment and store for variable references ($x$), and by closure creation for $\lambda$-abstractions (*lam*). In a language containing syntactic literals, these would be translated into equivalent semantic values here.

$$\begin{aligned} \mathcal{A} &: \mathsf{AExp} \times \Sigma \rightharpoonup \mathit{Clo} \\ \mathcal{A}(x,\ (\mathit{call},\ \rho)) &\triangleq \rho(x) \\ \mathcal{A}(\mathit{lam},\ (\mathit{call},\ \rho)) &\triangleq (\mathit{lam},\ \rho) \end{aligned}$$

The transition function $(\rightarrow_{\text{CE}}) : \Sigma \rightharpoonup \Sigma$ yields at most one successor for a given predecessor in the state space $\Sigma$. This is defined:

```
(define (eval call [rho (hash)])
        (define (atomic-eval ae)
                (match ae
                       [`(lambda ,args ,call) `((lambda ,args ,call) ,rho)]
                       [(? symbol? x) (hash-ref rho x)]))
        (match call
               [`(halt ,ae)
                (atomic-eval ae)]
               [`(,aefun ,aeargs ...)
                (define vfun (atomic-eval aefun))
                (define vargs (map (lambda (ae) (atomic-eval ae)) aeargs))
                (match vfun
                       [`((lambda ,xs ,callbody) ,rho_lam)
                        (define rho+ (foldl (lambda (x v rho+)
                                              (hash-set rho+ x v))
                                            rho_lam xs vargs))
                        (eval callbody rho+)])]))
```

**Figure 2.3**. An interpreter for the untyped CPS $\lambda$-calculus, written in Racket.

$$\overbrace{((ae_f\ ae_1\ \ldots\ ae_j), \rho)}^{\varsigma} \rightarrow_\Sigma (call', \rho'), \text{where}$$
$$((\lambda\ (x_0 \ldots x_j)\ call'), \rho_\lambda) = \mathcal{A}(ae_f,\ \varsigma)$$
$$clo_i = \mathcal{A}(ae_i,\ \varsigma)$$
$$\rho' = \rho_\lambda[x_i \mapsto clo_i]$$

Execution steps to the call-site of the lambda invoked (as given by the atomic evaluation of $ae_f$). This closure's environment ($\rho_\lambda$) is extended with a binding for each variable $x_i$ to the atomic evaluation of $ae_i$. A state becomes stuck if a `halt`-form is reached or if the program is malformed (*e.g.*, a free variable is encountered).

To fully evaluate a program $call_0$ using these transition rules, we *inject* it into our state space using a helper $\mathcal{I} : \mathsf{Call} \rightarrow \Sigma$:

$$\mathcal{I}(call) \triangleq (call, \varnothing)$$

A small-step interpreter for CPS in Racket is shown in Figure 2.3. Notice that every recursive call is now in tail position and no stack is required.

We may now perform a standard lifting of ($\rightarrow_{\text{CE}}$) to a collecting semantics defined over sets of states.

$$s \in S \triangleq \mathcal{P}(\Sigma)$$

Such a semantics strictly accumulates reachable states, instead of progressing from one state to another, collecting the information created over a program's evaluation. Our collecting relation ($\rightarrow_{s_{\text{CE}}}$) is a monotonic, total function that gives a set including the trivially reachable state $\mathcal{I}(call_0)$ plus the set of all states immediately succeeding those in its input.

$$s \rightarrow_{s_{\text{CE}}} s' \triangleq s' = \{\varsigma' \mid \varsigma \in s \wedge \varsigma \rightarrow_{\text{CE}} \varsigma'\} \cup \{\mathcal{I}(call_0)\}$$

If the program $call_0$ terminates, iteration of ($\rightarrow_{s_{\text{CE}}}$) from $\bot$ (*i.e.*, the empty set $\varnothing$) does as well. That is, $(\rightarrow_{s_{\text{CE}}})^n(\bot)$ is a fixed point containing $call_0$'s full program trace for some $n \in \mathbb{N}$ whenever $call_0$ is a terminating program. No such $n$ is guaranteed to exist in the general case (when $call_0$ is a nonterminating program) as our language (the untyped CPS $\lambda$-calculus) is Turing-equivalent, our semantics is fully precise, and the state space we defined is infinite.

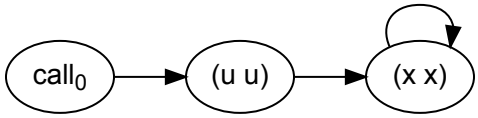
Unlike with our big-step semantics (where a nonterminating program has no valid proof tree or value), even nonterminating programs have specific (though infinite and sometimes incomputable) values. Consider, for example, a small-step reduction of the program Omega (the u-combinator applied to itself):

$$\begin{aligned}
((\lambda\ (u)\ (u\ u))\ (\lambda\ (x)\ (x\ x))), \varnothing) &\rightarrow_{\text{CE}} ((u\ u), [u \mapsto ((\lambda\ (x)\ (x\ x))), \varnothing)]) \\
&\rightarrow_{\text{CE}} ((x\ x), [x \mapsto ((\lambda\ (x)\ (x\ x))), \varnothing)]) \\
&\rightarrow_{\text{CE}} ((x\ x), [x \mapsto ((\lambda\ (x)\ (x\ x))), \varnothing)]) \\
&\rightarrow_{\text{CE}} \ldots
\end{aligned}$$

Although the program nonterminates, a finite fixed point still exists for ($\rightarrow_{s_{\text{CE}}}$). Specifically:

$$\begin{aligned}
\{&((\lambda\ (u)\ (u\ u))\ (\lambda\ (x)\ (x\ x))), \varnothing), \\
&((u\ u), [u \mapsto ((\lambda\ (x)\ (x\ x))), \varnothing)]), \\
&((x\ x), [x \mapsto ((\lambda\ (x)\ (x\ x))), \varnothing)])\ \}
\end{aligned}$$

The infinite concrete program trace is a closed graph which loops back on itself:



In most cases, however, nonterminating programs will have concrete evaluations with an infinite number of distinct states. For example, the following nonterminating sequence of small-step reductions extends the environment without bound.

$$
\begin{aligned}
&(((\lambda\ (w)\ (w\ w\ w))\ (\lambda\ (x\ y)\ (x\ x\ (\lambda\ (z)\ z))))), \varnothing) \\
&\quad \to_{\text{CE}} ((w\ w\ w), [w \mapsto ((\lambda\ (x\ y)\ (x\ x\ (\lambda\ (z)\ z))), \varnothing)]) \\
&\quad \to_{\text{CE}} ((x\ x\ (\lambda\ (z)\ z)), \overbrace{[x \mapsto ((\lambda\ (x\ y)\ (x\ x\ (\lambda\ (z)\ z))), \varnothing), y \mapsto \ldots]}^{\rho_0}) \\
&\quad \to_{\text{CE}} ((x\ x\ (\lambda\ (z)\ z)), \overbrace{[x \mapsto ((\lambda\ (x\ y)\ (x\ x\ (\lambda\ (z)\ z))), \varnothing), y \mapsto ((\lambda\ (z)\ z), \rho_0)]}^{\rho_1}) \\
&\quad \to_{\text{CE}} ((x\ x\ (\lambda\ (z)\ z)), \overbrace{[x \mapsto ((\lambda\ (x\ y)\ (x\ x\ (\lambda\ (z)\ z))), \varnothing), y \mapsto ((\lambda\ (z)\ z), \rho_1)]}^{\rho_2}) \\
&\quad \to_{\text{CE}} \ldots
\end{aligned}
$$

Although no finite fixed point for $(\to_{S_{\text{CE}}})$ exists in this case, an infinite fixed point does exist because it ranges over a complete lattice (Tarski, 1955) and in the next chapter, we will be able to compute an approximation of otherwise incomputable fixed points like this one.

### 2.2.2 Direct-Style Semantics

We can model a direct-style language with a small-step semantics by passing a stack along explicitly. For simplicity, we will assume an input converted to monadic normal form or administrative normal form (ANF) (Danvy, 2003; Flanagan et al., 1993; Kennedy, 2007).

| | |
|---|---|
| $e \in \mathsf{ANFExp} ::= \texttt{(let ([}x\ (f\ ae)\texttt{])}\ e\texttt{)}$ | [call] |
| $\mid ae$ | [return] |
| $f, ae \in \mathsf{AExp} ::= x \mid lam$ | [atomic expressions] |
| $lam \in \mathsf{Lam} ::= \texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e\texttt{)}$ | [lambda abstractions] |
| $x, y \in \mathsf{Var}$ is a set of identifiers | [variables] |

All intermediate expressions are administratively `let`-bound, and the order of operations is made explicit as a stack of such `let`s. This not only simplifies our semantics, but is convenient for analysis as every intermediate expression can naturally be given a unique identifier. A variety of differently formed continuations are also unified by a single `let`-continuation. In section 2.1, LamExp had two different continuations (one for a subexpression in call position, another for one in argument position), though we did not have to manage them explicitly. In the case of a richer intermediate language, there may be a dozen or more

flavors of continuations (for conditionals, letrec, *etc.*), so implementing all of these with a single `let`-continuation may be to our advantage.

We define the evaluation of programs in this language using a relation ($\rightarrow_{\text{CEK}}$), over states ($\varsigma$) of an abstract machine. These states contain a new component ($\kappa$) that encodes the current list of stack frames.

$$\begin{aligned}
\varsigma \in \Sigma &\triangleq \mathsf{ANFExp} \times \mathit{Env} \times \mathit{Kont} \\
\rho \in \mathit{Env} &\triangleq \mathsf{Var} \rightharpoonup \mathit{Clo} \\
\mathit{clo} \in \mathit{Clo} &\triangleq \mathsf{Lam} \times \mathit{Env} \\
\kappa \in \mathit{Kont} &\triangleq \mathit{Frame}^* \\
\phi \in \mathit{Frame} &\triangleq \mathsf{Var} \times \mathsf{ANFExp} \times \mathit{Env}
\end{aligned}$$

Stack frames contain a variable to bind upon returning, an expression to return to, and an environment to reinstate and extend with a return value.

Because each step of this machine must terminate but may rely on evaluating atomic expressions $ae$, we again produce a computable atomic-expression evaluator $\mathcal{A}$ which encapsulates the same two cases as before: closure creation and variable reference.

$$\begin{aligned}
\mathcal{A} &: \mathsf{AExp} \times \Sigma \rightharpoonup \mathit{Clo} \\
\mathcal{A}(x,\ (e,\ \rho,\ \kappa)) &= \rho(x) \\
\mathcal{A}(\mathit{lam},\ (e,\ \rho,\ \kappa)) &= (\mathit{lam},\ \rho)
\end{aligned}$$

The transition function $(\rightarrow_{\text{CEK}}) : \Sigma \rightharpoonup \Sigma$ produces at most one successor for a given predecessor in the state space. For call sites, this is defined:

$$\begin{aligned}
\overbrace{(\texttt{(let ([}y\ (ae_f\ ae_v)\texttt{])}\ e),\ \rho,\ \kappa)}^{\varsigma} &\rightarrow_{\text{CEK}} (e',\ \rho',\ \kappa'), \text{where} \\
(\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e'\texttt{)},\ \rho_\lambda) &= \mathcal{A}(ae_f,\ \varsigma) \\
\mathit{clo}_v &= \mathcal{A}(ae_v,\ \varsigma) \\
\rho' &= \rho_\lambda[x \mapsto \mathit{clo}_v] \\
\kappa' &= (y, e, \rho) : \kappa
\end{aligned}$$

Control moves into the body of the function invoked ($ae_f$) just as it did in section 2.2.1, except a new continuation $\kappa'$ is produced which extends the current continuation $\kappa$ with a stack frame for the `let`-form binding $y$.

For return points, $(\rightarrow_{\text{CEK}})$ is defined:

$$\overbrace{(ae,\ \rho,\ (x, e, \rho_\kappa) : \kappa)}^{\varsigma} \rightarrow_{\text{CEK}} (e,\ \rho',\ \kappa), \text{where}$$
$$clo_v = \mathcal{A}(ae,\ \varsigma)$$
$$\rho' = \rho_\kappa[x \mapsto clo_v]$$

The top stack frame is opened and popped, the frame's expression $e$ is reinstated, and the frame's environment $\rho_\kappa$ is also reinstated after being extended with a binding for $x$ to the return value $clo_v$. If a return point is reached on an empty stack, no successor is produced and evaluation halts.

Figure 2.4 shows a small-step interpreter for ANF written in Racket. Instead of explicitly passing a continuation along as a user lambda due to CPS conversion, this interpreter passes a continuation along as a distinguished top-level component of machine states. This requires us to do the work of specifically managing stack pushes and stack pops, however it also gives

```
(define (eval e [rho (hash)] [kappa '()])
        (define (atomic-eval ae)
                (match ae
                       [`(lambda (,x) ,e) `((lambda (,x) ,e) ,rho)]
                       [(? symbol? x) (hash-ref rho x)]))
        (match e
               [`(let ([,xlet (,aefun ,aearg)]) ,elet)
                (define vfun (atomic-eval aefun))
                (define varg (atomic-eval aearg))
                (match vfun
                       [`((lambda (,x) ,ebody) ,rho_lam)
                        (eval ebody
                              (hash-set rho_lam x varg)
                              (cons `(letk ,xlet ,elet ,rho) kappa))])]
               [(or (? symbol?) `(lambda (,x) ,ebody))
                (define retv (atomic-eval e))
                (if (null? kappa)
                    retv
                    (match (car kappa)
                           [`(letk ,xlet ,elet ,rho_let)
                            (eval elet
                                  (hash-set rho_let xlet retv)
                                  (cdr kappa))]))])))
```

**Figure 2.4**. A *small-step* interpreter for the direct-style, untyped $\lambda$-calculus, written in Racket. Because an explicit stack is passed along, every call to `eval` is tail recursive.

us full control over how the stack is represented and manipulated.

Before formulating program evaluation as a fixed point problem, we produce a helper for injecting a program into an initial state with an empty stack and environment:

$$\mathcal{I} : \mathsf{ANFExp} \to \Sigma$$
$$\mathcal{I}(e) \triangleq (e, \varnothing, \epsilon)$$

We perform the standard lifting of $(\to_{\text{CEK}})$ to a collecting semantics defined over sets of states.

$$s \in S \triangleq \mathcal{P}(\Sigma)$$

As before, our collecting relation $(\to_{S_{\text{CEK}}})$ is a monotonic, total function that gives a set including the trivially reachable state $\mathcal{I}(e_0)$ plus the set of all states immediately succeeding those in its input.

$$s \to_{S_{\text{CEK}}} s' \triangleq s' = \{\varsigma' \mid \varsigma \in s \wedge \varsigma \to_{\text{CEK}} \varsigma'\} \cup \{\mathcal{I}(e_0)\}$$

A fixed point for $(\to_{S_{\text{CEK}}})$ is a (possibly infinite) set of states in $\Sigma$ which contains every state that execution passes through when evaluating $e_0$.

# CHAPTER 3

# ABSTRACTING ABSTRACT MACHINES

This chapter introduces a general approach to static analysis via abstract interpretation of small-step abstract-machine semantics. As we saw in the previous chapters, abstract machines are a versatile, concise, and familiar way to construct formal semantics for programming languages that are very much like normal definitional interpreters in how they define a process of evaluating programs step by step.

The methodology of *abstracting abstract machines* (AAM) encompases a variety of tools and alternatives for turning an underlying concrete (*i.e.*, precise) small-step abstract machine into a static analysis by approximating or further *abstracting* it (Johnson et al., 2013; Might, 2010; Van Horn and Might, 2010). AAM aims to be as systematic as possible in extending the approach of abstract interpretation to the unique challenges and opportunities of abstract machines and functional (higher-order) languages in particular. The AAM methodology is flexible in allowing a high degree of control over how program states are represented and is easy to instrument and extend.

AAM provides us with the tools to abstract an abstract machine and obtain an approximation of its operational behavior in a variety of styles. Importantly, one such style aims to focus all unboundedness in a semantics on the machine's address space. Over the next chapters, we will extend this methodology to producing an analysis framework parameterized by an allocation function mapping variables to abstract addresses as they are bound.

## 3.1 Finitizing an Abstract Machine

This section takes up the CPS abstract machine from Section 2.2.1 and discusses the problem of finitizing its state space to obtain a computable approximation. Recall that in CPS, every call site is in tail position and all return flow has been encoded as a call site invoking a continuation.

$$call \in \mathsf{Call} ::= \texttt{(}ae\ ae\ \texttt{...)} \mid \texttt{(halt}\ ae\texttt{)}$$
$$lam \in \mathsf{Lam} ::= \texttt{(}\lambda\ \texttt{(}x\ \texttt{...)}\ call\texttt{)}$$
$$ae \in \mathsf{AExp} ::= lam \mid x$$
$$x \in \mathsf{Var}\ \textit{is a set of program variables}$$

Our small-step CE-machine is defined over control expression (in pure CPS, a call site) and environment components.

$$\varsigma \in \Sigma \triangleq \mathsf{Call} \times \mathit{Env}$$
$$\rho \in \mathit{Env} \triangleq \mathsf{Var} \rightharpoonup \mathit{Clo}$$
$$\mathit{clo} \in \mathit{Clo} \triangleq \mathsf{Lam} \times \mathit{Env}$$

For an abstract machine to implement a Turing-complete language, its state space must be infinite. In this case, the state space $\Sigma$ is infinite because closures and environments are mutually recursive: closures contain an environment and environments map variables to closures. In Section 2.2.1, we observed how a program like

$$\texttt{((}\lambda\ \texttt{(}w\texttt{)}\ \texttt{(}w\ w\ w\texttt{))}\ \texttt{(}\lambda\ \texttt{(}x\ y\texttt{)}\ \texttt{(}x\ x\ \texttt{(}\lambda\ \texttt{(}z\texttt{)}\ z\texttt{))))}$$

can repeatedly extend its current environment without bound. The direct-style CEK-machine of Section 2.2.2 exhibits both recursion between environments and closures and recursion in its stack (a stack $\kappa$ is either $\epsilon$ or some $\phi{:}\kappa'$).

### 3.1.1 A Three-Step Approach

The fundamental tool AAM proposes for producing a finite abstract semantics (one we can connect to its concrete semantics and use to approximate program behavior as illustrated in Chapter 1) is to "*cut*" each source of recursion in an abstract machine using a store-passing-style transformation. Deploying this technique in a variety of different ways yields a wide design space of reasonable approximations.

The process for producing an approximate abstract-machine semantics given a concrete small-step abstract machine (such as $\Sigma$) can be systematized as three steps:

1. Each source of recursion in the machine is *cut* by indirecting it through a set of addresses $a \in \mathit{Addr}$. In the CPS machine $\Sigma$, this gives us just two options. We can either define closures as pairs of a syntactic lambda and a pointer to its environment, $\mathit{clo} \in \mathsf{Lam} \times \mathit{Addr}$, or we can define environments as functions from variables to

addresses for their respective closures. We select this latter design as it coincides nicely with the standard store-passing style used to define stateful language features.

$$\varsigma \in \Sigma \triangleq \mathsf{Call} \times \mathit{Env} \times \mathit{Store}$$
$$\rho \in \mathit{Env} \triangleq \mathsf{Var} \rightharpoonup \mathit{Addr}$$
$$\sigma \in \mathit{Store} \triangleq \mathit{Addr} \rightharpoonup \mathit{Clo}$$
$$\mathit{clo} \in \mathit{Clo} \triangleq \mathsf{Lam} \times \mathit{Env}$$
$$a \in \mathit{Addr} \triangleq \mathsf{Var} \times \mathbb{N}$$

A store $\sigma$ gets passed along in each state accumulating addresses bound to closures. Addresses can be defined in various ways, so long as we can produce an infinite number of them and each is fresh when allocated.

$$\overbrace{((ae_f\ ae_1\ \ldots\ ae_j), \rho, \sigma)}^{\varsigma} \rightarrow_\Sigma (\mathit{call}', \rho', \sigma'), \text{where}$$
$$((\lambda\ (x_0 \ldots x_j)\ \ \mathit{call}'), \rho_\lambda) = \mathcal{A}(ae_f,\ \varsigma)$$
$$\mathit{clo}_i = \mathcal{A}(ae_i,\ \varsigma)$$
$$\rho' = \rho_\lambda[x_i \mapsto a_i]$$
$$\sigma' = \sigma[a_i \mapsto \mathit{clo}_i]$$
$$a_i = (x_i, |\mathit{dom}(\sigma)|)$$

Formally, each address $a_i$ is *fresh* iff $a_i \notin \mathit{dom}(\sigma) \wedge (a_i = a_k \implies i = k)$. A particular strategy for allocating a fresh address is to pair the variable being allocated for with the current number of points in the store. This both keeps addresses allocated at the same time distinct, and addresses allocated at different times distinct.

Atomic evaluation needs the store so it can read addresses, but does not need to produce a modified store because atomic evaluation is stateless (even in languages supporting mutation).

$$\mathcal{A} : \mathsf{AE} \times \Sigma \rightharpoonup \mathit{Clo}$$
$$\mathcal{A}(x,\ (\mathit{call},\ \rho,\ \sigma)) \triangleq \sigma(\rho(x))$$
$$\mathcal{A}(\mathit{lam},\ (\mathit{call},\ \rho,\ \sigma)) \triangleq (\mathit{lam},\ \rho)$$

The updated store-passing-style state space $\Sigma$ contains no direct recursion and is infinite only because we allow the set of addresses to be infinite. (We are only interested in analyses of finite programs, so while our syntactic domains are infinite

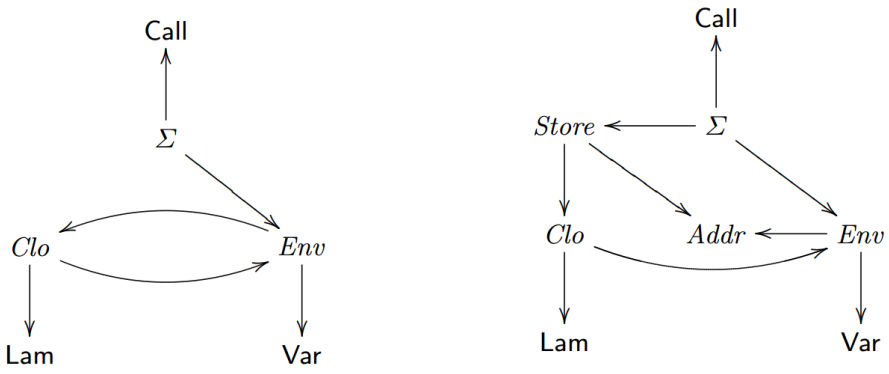



**Figure 3.1**. A dependence graph for the components of a CPS abstract machine before and after cutting the dependency of environments on closures.

in the general case, for a particular target program, they are each finite.) Figure 3.1 shows a dependence graph for $\Sigma$ both before and after the store-passing transformation of this step (Might, 2010).

2. The next step is to abstract our set of addresses to a finite set of *abstract* addresses $\hat{a} \in \widehat{Addr} = \mathsf{Var}$ with a Galois connection $(\alpha_{Addr}, \gamma_{Addr})$ and propagate this change up to $\Sigma$. At each step, moving from $\widehat{Addr}$ up to $\Sigma$, we compose an existing Galois connection with a syntactic domain or another Galois connection to derive a new Galois connection automatically (Might, 2010). In most cases, these inferred Galois connections are quite simple. For example, we can visualize a rule for Cartesian product:

$$\frac{A \underset{\alpha_A}{\overset{\gamma_A}{\leftrightarrows}} \hat{A} \qquad B \underset{\alpha_B}{\overset{\gamma_B}{\leftrightarrows}} \hat{B}}{A \times B \underset{\alpha_{AB}}{\overset{\gamma_{AB}}{\leftrightarrows}} \hat{A} \times \hat{B}}$$

And in this case, $\alpha_{AB}(a, b) = (\alpha_A(a), \alpha_B(b))$ (Nielson et al., 2004). Lifting our Galois connection for addresses to one for environments yields a set of abstract environments $\mathsf{Var} \rightharpoonup \widehat{Addr}$. Abstract environments remain functions because only the images of these functions need a Galois connection. In the case of the store, however, because the domain of stores is abstract, abstract stores become relations. In the end, we obtain a Galois connection for closures and machine states through the simple inference rule for Cartesian products.

$$\hat{\varsigma} \in \hat{\Sigma} \triangleq \mathsf{Call} \times \widehat{Env} \times \widehat{Store}$$
$$\hat{\rho} \in \widehat{Env} \triangleq \mathsf{Var} \rightharpoonup \widehat{Addr}$$
$$\hat{\sigma} \in \widehat{Store} \triangleq \widehat{Addr} \to \mathcal{P}(\widehat{Clo})$$
$$\widehat{clo} \in \widehat{Clo} \triangleq \mathsf{Lam} \times \widehat{Env}$$
$$\hat{a} \in \widehat{Addr} \triangleq \mathsf{Var}$$

This compaction of the address set to ensure the finiteness of our analysis has led to nondeterminism (*i.e.*, relationality) in the store. Each abstract address can become bound to more than one abstract closure (an abstract store yields *flow sets*) because it must over-approximate the behavior of multiple concrete addresses (which may, during a concrete evaluation, become bound to multiple distinct closures). Nondeterminism in the store, in turn, leads to a further change: nondeterminism in the abstract transition *relation*.

The total Galois connection produced by this process may be defined:

$$\alpha_\Sigma((e,\ \rho,\ \sigma)) \triangleq (e,\ \alpha_{Env}(\rho),\ \alpha_{Store}(\sigma))$$
$$\alpha_{Env}(\rho) \triangleq \{(x,\ \alpha_{Addr}(a)) \mid (x,\ a) \in \rho\}$$
$$\alpha_{Store}(\sigma) \triangleq \bigsqcup_{(a,clo)\in\sigma} [\alpha_{Addr}(a) \mapsto \{\alpha_{Clo}(clo)\}]$$
$$\alpha_{Clo}((lam,\ \rho)) \triangleq (lam,\ \alpha_{Env}(\rho))$$
$$\alpha_{Addr}((x,\ n)) \triangleq x$$

3. So far, this systematic process of abstracting an abstract machine yields abstract domains, but not an abstract transition relation. To produce a relation $(\hat{\rightsquigarrow}_\Sigma) \subseteq \hat{\Sigma} \times \hat{\Sigma}$ between abstract states, we have two options: calculate it or justify it.

   Our Galois connection for states represents a notion of simulation which any acceptable $(\hat{\rightsquigarrow}_\Sigma)$ must respect to soundly approximate $(\rightarrow_\Sigma)$. It also implies many different analyses which are sound with respect to it (precise or not). For example, an abstract transition relation which yields the entire abstract state space at every step is trivial and fully imprecise, but still sound with respect to $(\rightarrow_\Sigma)$ and $\alpha_\Sigma$. We may produce a definition for $(\hat{\rightsquigarrow}_\Sigma)$ that we would prefer to implement, and then justify it as sound by showing that (simulation is preserved across every transition)

$$\varsigma \rightarrow_\Sigma \varsigma' \wedge \alpha_\Sigma(\varsigma) \sqsubseteq \hat{\varsigma} \implies \exists \hat{\varsigma}'.\ \hat{\varsigma} \hat{\rightsquigarrow}_\Sigma \hat{\varsigma}' \wedge \alpha_\Sigma(\varsigma') \sqsubseteq \hat{\varsigma}',$$

   or we may directly calculate the most precise abstract transition relation that is sound. Cousot and Cousot (1979) gives a method for directly calculating the optimal analysis

provided a Galois connection. The most precise analysis is one which behaves the same as concretizing, then applying the concrete semantics, and then abstracting again.

In the case of our CPS analysis, the abstract transition relation may be defined:

$$\overbrace{((ae_f\ ae_1\ \ldots\ ae_j), \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}} \rightsquigarrow_{\hat{\Sigma}} (call', \hat{\rho}', \hat{\sigma}'), \text{where}$$

$$\begin{aligned}((\lambda\ (x_0 \ldots x_j)\ call'), \hat{\rho}_\lambda) &\in \hat{\mathcal{A}}(ae_f,\ \hat{\varsigma}) \\ \widehat{clo}_i &= \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma}) \\ \hat{\rho}' &= \hat{\rho}_\lambda[x_i \mapsto \hat{a}_i] \\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \widehat{clo}_i] \\ \hat{a}_i &= x_i\end{aligned}$$

And its abstract atomic evaluator is defined:

$$\hat{\mathcal{A}} : \mathsf{AE} \times \hat{\Sigma} \rightharpoonup \mathcal{P}(\widehat{Clo})$$

$$\hat{\mathcal{A}}(x,\ (call,\ \hat{\rho},\ \hat{\sigma})) \triangleq \hat{\sigma}(\hat{\rho}(x))$$

$$\hat{\mathcal{A}}(lam,\ (call,\ \hat{\rho},\ \hat{\sigma})) \triangleq \{(lam,\ \hat{\rho})\}$$

A weak update is performed on the store instead which results in the least upper bound of the existing store and each new binding. Join on abstract stores distributes point-wise:

$$\hat{\sigma} \sqcup \hat{\sigma}' \triangleq \lambda \hat{a}.\ \hat{\sigma}(\hat{a}) \cup \hat{\sigma}'(\hat{a})$$

Unless it is desirable, and provably safe to do so (Might and Shivers, 2006), we never remove closures already seen. Instead, we strictly accumulate every closure bound to each $\hat{a}$ (*i.e.*, abstract closures which simulate closures bound to addresses which $\hat{a}$ simulates) over the lifetime of the program. A flow set for an address $\hat{a}$ indicates a range of values which over-approximates all possible concrete values that can flow to any concrete address approximated by $\hat{a}$. For example, if a concrete machine binds $(y, 345) \mapsto clo_1$ and $(y, 903) \mapsto clo_2$, its approximation might bind $y \mapsto \{\widehat{clo}_1, \widehat{clo}_2\}$. Precision is lost for $(y, 345)$ both because its value has been merged with $\widehat{clo}_2$, and because the environments for $\widehat{clo}_1$ and $\widehat{clo}_2$ in-turn generalize over many possible addresses for their free variables (the environment in $\widehat{clo}_1$ is less precise than the environment in $clo_1$).

### 3.1.2 Approximation of Fixed Points

To fully evaluate a program $call_0$ using these concrete transition rules, we *inject* it into our state space using a helper $\mathcal{I} : \mathsf{Call} \to \Sigma$:

$$\mathcal{I}(call) \triangleq (call, \varnothing, \varnothing)$$

Likewise, to perform a full analysis of a program $call_0$ using the abstract transition rules, we inject it into our state space using a helper $\hat{\mathcal{I}} : \mathsf{Call} \to \hat{\Sigma}$:

$$\hat{\mathcal{I}}(call) \triangleq (call, \varnothing, \bot)$$

As abstraction has turned the store into a total function that yields sets (*i.e.*, a relation), its initial value is the function which yields an empty set for every input, $\bot$.

We may now perform a lifting of $(\rightarrow_\Sigma)$ and $(\rightsquigarrow^\wedge_\Sigma)$ to their collecting semantics defined over sets of states: $(\rightarrow_s)$ and $(\rightsquigarrow^\wedge_s)$.

$$s \in S \triangleq \mathcal{P}(\Sigma) \qquad\qquad \hat{s} \in \hat{S} \triangleq \mathcal{P}(\hat{\Sigma})$$

$$\alpha_S(s) \triangleq \{\alpha_\Sigma(\varsigma) \mid \varsigma \in s\}$$

Such semantics strictly accumulates reachable states, instead of progressing from one state to another, collecting the information created over a program's evaluation. By phrasing concrete evaluation as a fixed-point problem, we can identify a specific incomputable value to approximate for even a nonterminating program (Tarski, 1955). Each collecting relation is a monotonic, total function that gives a set including the trivially reachable state ($\mathcal{I}(call_0)$ or $\hat{\mathcal{I}}(call_0)$) plus the set of all states immediately succeeding those in its input.

$$s \rightarrow_s s' \triangleq s' = \{\varsigma' \mid \varsigma \in s \wedge \varsigma \rightarrow_\Sigma \varsigma'\} \cup \{\mathcal{I}(call_0)\}$$
$$\hat{s} \rightsquigarrow^\wedge_s \hat{s}' \triangleq \hat{s}' = \{\hat{\varsigma}' \mid \hat{\varsigma} \in \hat{s} \wedge \hat{\varsigma} \rightsquigarrow^\wedge_\Sigma \hat{\varsigma}'\} \cup \{\hat{\mathcal{I}}(call_0)\}$$

If the program $call_0$ terminates, iteration of $(\rightarrow_s)$ from $\bot$ (*i.e.*, the empty set $\varnothing$) does as well. That is, $(\rightarrow_s)^n(\bot)$ is a fixed point containing $call_0$'s full program trace for some $n \in \mathbb{N}$ whenever $call_0$ is a terminating program. No such $n$ is guaranteed to exist in the general case (when $call_0$ is a nonterminating program) as our language is Turing-equivalent; however, an abstract interpretation will allow us to approximate even an incomputable infinite set of states. Because $(\rightsquigarrow^\wedge_\Sigma)$ soundly simulates $(\rightarrow_\Sigma)$, with respect to $\alpha_\Sigma$, and $(\hat{\Sigma})$ is finite, we know that for some $n \in \mathbb{N}$ the value, $(\rightsquigarrow^\wedge_s)^n(\bot)$ is an approximation of the least fixed point of $(\rightarrow_s)$, though itself incomputable.

## 3.2 *A Priori* Soundness

This section gives a proof that $(\hat{\rightsquigarrow}_\Sigma)$ soundly simulates $(\rightarrow_\Sigma)$ with respect to $\alpha_\Sigma$. Normally, this is called an *a posteriori* justification of an abstract interpretation to contrast it with the calculational approach where the most precise interpretation which respects the given Galois connection is directly calculated. Although this justification of soundness takes place *after* construction of both an abstraction map and abstract interpretation, it takes place *before* any analysis is actually run. We will refer to the present justification process as *a priori*, in this sense, to contrast it with Section 4.3 where we will look at a method for proving whole classes of analyses sound by using an abstraction map which cannot be fully constructed until after the analysis is actually run.

**Theorem 1.** *(Simulation across transition) If a transition $\varsigma \rightarrow_\Sigma \varsigma'$ is legal and $\hat{\varsigma}$ simulates $\varsigma$, then there must exist a legal transition $\hat{\varsigma} \hat{\rightsquigarrow}_\Sigma \hat{\varsigma}'$ where $\hat{\varsigma}'$ simulates $\varsigma'$.*

$$\varsigma \rightarrow_\Sigma \varsigma' \wedge \alpha_\Sigma(\varsigma) \sqsubseteq \hat{\varsigma} \implies \exists \hat{\varsigma}'.\ \hat{\varsigma} \hat{\rightsquigarrow}_\Sigma \hat{\varsigma}' \wedge \alpha_\Sigma(\varsigma') \sqsubseteq \hat{\varsigma}',$$

*Proof.* We expand the definition of our assumed antecedent transition:

$$\overbrace{((ae_f\ ae_1\ \ldots\ ae_j), \rho, \sigma)}^{\varsigma} \rightarrow_\Sigma \overbrace{(call', \rho', \sigma')}^{\varsigma'}, \text{where}$$

$$\begin{aligned}
((\lambda\ (x_0 \ldots x_j)\ \ call'), \rho_\lambda) &= \mathcal{A}(ae_f,\ \varsigma) \\
clo_i &= \mathcal{A}(ae_i,\ \varsigma) \\
\rho' &= \rho_\lambda[x_i \mapsto a_i] \\
\sigma' &= \sigma[a_i \mapsto clo_i] \\
a_i &= (x_i, |dom(\sigma)|)
\end{aligned}$$

and we assume $\hat{\varsigma}$ simulates $\varsigma$:

$$\begin{aligned}
\alpha_\Sigma \overbrace{((ae_f\ ae_1\ \ldots\ ae_j),\ \rho,\ \sigma)}^{\varsigma} &\sqsubseteq \overbrace{((ae_f\ ae_1\ \ldots\ ae_j),\ \hat{\rho},\ \hat{\sigma})}^{\hat{\varsigma}} \\
\alpha_{Env}(\rho) &\sqsubseteq \hat{\rho} \\
\alpha_{Store}(\sigma) &\sqsubseteq \hat{\sigma}
\end{aligned}$$

We will show that a transition $\hat{\varsigma} \hat{\rightsquigarrow}_\Sigma \hat{\varsigma}'$ is legal such that $\alpha_\Sigma(\varsigma') \sqsubseteq \hat{\varsigma}'$.

$$\overbrace{((ae_f\ ae_1\ \ldots\ ae_j), \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}} \rightsquigarrow_{\hat{\Sigma}} (call', \hat{\rho}', \hat{\sigma}'), \text{where}$$

$$\begin{aligned}
((\lambda\ (x_0 \ldots x_j)\ \ call'), \hat{\rho}_\lambda) &\in \hat{\mathcal{A}}(ae_f,\ \hat{\varsigma}) \\
\widehat{clo}_i &= \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma}) \\
\hat{\rho}' &= \hat{\rho}_\lambda[x_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \widehat{clo}_i] \\
\hat{a}_i &= x_i
\end{aligned}$$

We can break this down into showing several simpler properties:

$$\{\alpha_{Clo}(\mathcal{A}(ae,\ \varsigma))\} \sqsubseteq \hat{\mathcal{A}}(ae,\ \hat{\varsigma}) \qquad \text{(Property 1)}$$

$$\alpha_{Addr}(a_i) = \hat{a}_i \qquad \text{(Property 2)}$$

$$\alpha_{Env}(\rho') \sqsubseteq \hat{\rho}' \qquad \text{(Property 3)}$$

$$\alpha_{Store}(\sigma') \sqsubseteq \hat{\sigma}' \qquad \text{(Property 4)}$$

Property (1) shows the abstract atomic-expression evaluator to be a sound simulation of the concrete atomic-expression evaluator. From this, it follows that a transition $\hat{\varsigma} \rightsquigarrow_{\hat{\Sigma}} \hat{\varsigma}'$ is legal such that $\alpha_{Env}(\rho_\lambda) \sqsubseteq \hat{\rho}_\lambda$. Properties (3) and (4) rely on (1,2) and complete the task of showing that $\alpha_\Sigma(\varsigma') \sqsubseteq \hat{\varsigma}'$.

**Property 1.** For all $ae$, we assume $ae \in dom(\rho) \vee ae \in \mathsf{Lam}$ by the *lexical safety* of the concrete machine (not shown; see Might, 2007) and break into these two cases for showing that $\{\alpha_{Clo}(\mathcal{A}(ae,\ \varsigma))\} \sqsubseteq \hat{\mathcal{A}}(ae,\ \hat{\varsigma})$:

$$\begin{aligned}
\{\alpha_{Clo}(\mathcal{A}(ae,\ \overbrace{(e,\ \rho,\ \sigma)}^{\varsigma}))\} &= \{\alpha_{Clo}((ae,\ \rho))\} && \text{(Assume } ae \in \mathsf{Lam}\text{)} \\
&\sqsubseteq \{(ae,\ \hat{\rho})\} && (\alpha_{Clo} \text{ and } \alpha_{Env}(\rho) \sqsubseteq \hat{\rho}) \\
&= \hat{\mathcal{A}}(ae,\ \hat{\varsigma}) \\
\{\alpha_{Clo}(\mathcal{A}(ae,\ \overbrace{(e,\ \rho,\ \sigma)}^{\varsigma}))\} &= \{\alpha_{Clo}(\sigma(\rho(ae)))\} && \text{(Assume } ae \in dom(\rho)\text{)} \\
&\sqsubseteq \alpha_{Store}(\sigma)(\alpha_{Addr}(\rho(ae))) && (\alpha_{Store}) \\
&\sqsubseteq \hat{\sigma}(\alpha_{Addr}(\rho(ae))) && (\alpha_{Store}(\sigma) \sqsubseteq \hat{\sigma}) \\
&= \hat{\sigma}(\alpha_{Env}(\rho)(ae)) && (\alpha_{Env}) \\
&\sqsubseteq \hat{\sigma}(\hat{\rho}(ae)) && (\alpha_{Env}(\rho) \sqsubseteq \hat{\rho}) \\
&= \hat{\mathcal{A}}(ae,\ \hat{\varsigma})
\end{aligned}$$

**Property 2.** For all $a_i$, we show that $\alpha_{Addr}(a_i) = \hat{a}_i$:

$$\begin{aligned}
\alpha_{Addr}(a_i) &= \alpha_{Addr}(x_i,\ \varsigma) && (a_i = (x_i, |dom(\sigma)|)) \\
&= x_i && (\alpha_{Addr}) \\
&= \hat{a}_i && (\hat{a}_i = x_i)
\end{aligned}$$

**Property 3.** We show that $\alpha_{Env}(\rho') \sqsubseteq \hat{\rho}'$:

$$\begin{aligned}
\alpha_{Env}(\rho') &= \alpha_{Env}(\rho_\lambda[x_i \mapsto a_i]) && (\rho' = \rho_\lambda[x_i \mapsto a_i]) \\
&\sqsubseteq \hat{\rho}_\lambda[x_i \mapsto \alpha_{Addr}(a_i)] && (\alpha_{Env} \text{ and } \alpha_{Env}(\rho_\lambda) \sqsubseteq \hat{\rho}_\lambda) \\
&= \hat{\rho}_\lambda[x_i \mapsto \hat{a}_i] && (\textit{Property 2}) \\
&= \hat{\rho}'
\end{aligned}$$

**Property 4.** We show that $\alpha_{Store}(\sigma') \sqsubseteq \hat{\sigma}'$:

$$\begin{aligned}
\alpha_{Store}(\sigma') &= \alpha_{Store}(\sigma[a_i \mapsto \mathcal{A}(ae_i,\ \varsigma)]) && (\sigma' = \sigma[a_i \mapsto \mathcal{A}(ae_i,\ \varsigma)]) \\
&\sqsubseteq \alpha_{Store}(\sigma) \sqcup \alpha_{Store}([a_i \mapsto \mathcal{A}(ae_i,\ \varsigma)]) && (\alpha_{Store}) \\
&\sqsubseteq \hat{\sigma} \sqcup \alpha_{Store}([a_i \mapsto \mathcal{A}(ae_i,\ \varsigma)]) && (\alpha_{Store}(\sigma) \sqsubseteq \hat{\sigma}) \\
&= \hat{\sigma} \sqcup [\alpha_{Addr}(a_i) \mapsto \{\alpha_{Clo}(\mathcal{A}(ae_i,\ \varsigma))\}] && (\alpha_{Store}) \\
&\sqsubseteq \hat{\sigma} \sqcup [\alpha_{Addr}(a_i) \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma})] && (\textit{Property 1}) \\
&= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma})] && (\textit{Property 2}) \\
&= \hat{\sigma}'
\end{aligned}$$

The soundness of $(\rightsquigarrow_{\Sigma}^{\widehat{}})$, with respect to $(\rightarrow_{\Sigma})$ and $\alpha_{\Sigma}$, is preserved across transition. □

## 3.3 1-CFA

The analysis just produced is sound, but highly approximate. It is *monovariant* (a closely related term is *context insensitive*) which means each syntactic variable or intermediate expression we track during analysis recieves only a single flow set to over-approximate all its possible values. A more *polyvariant* analysis, by contrast, would differentiate a larger number of distinctly tracked approximations.

A simple example of a polyvariant strategy for analysis is *call sensitivity*, a form of context sensitivity which differentiates values based on a history of call sites that execution recently passed through. $k$-CFA, a classic framework for analysis of functional languages, uses callsensitivity to keep values separate and is parameterized by a number $k \in \mathbb{N}$ which specifies how many previous call sites to remember when producing new flow sets. We could

modify our analysis to instantiate 1-CFA by making two small changes. First, we modify the domain for abstract addresses so that each is the pair of a variable and the last call site execution passed through:

$$\hat{a} \in \widehat{Addr}_{\text{1CFA}} \triangleq \mathsf{Var} \times \mathsf{Call}$$

Second, we modify a single proposition defining our abstract transition relation to produce addresses unique to both the variable being allocated for and the most recent call site. Only our proposition constraining each $\hat{a}_i$ needs to be changed:

$$\hat{a}_i = (x_i, (ae_f\ ae_1\ \ldots\ ae_j))$$

These changes can cause the analysis to allocate a much larger number of abstract addresses and abstract environments, permitting both an increase in precision and analysis complexity.

An equivalence relation on these addresses may be lifted from a notion of equality for syntax; however, we must either affix unique labels to every program expression or assume that two identical pieces of syntax found in the same program are syntactically unequal. For simplicity, we assume the latter.

Tuning our abstract semantics to instantiate 2-CFA or another more context-sensitive (and more polyvariant) analysis style requires further instrumentation; unless we require our analysis to remember the penultimate call site as well, we cannot use it to further differentiate abstract addresses. Chapter 4 proposes a more principaled approach to tuning a CFA to *any conceivable* style of polyvariance.

## 3.4 *A Posteriori* Soundness

The usual process for proving the soundness of an abstract interpretation is *a priori* in the sense that it may be performed entirely before an analysis is executed. By contrast, Might and Manolios (2009) describes an *a posteriori* soundness proof where the abstraction map cannot be fully constructed until after analysis. This approach factors each $\alpha$ to separate the abstraction of addresses $\alpha_{Addr}$, producing a family of parametric maps $\beta$ such that $\beta(\alpha_{Addr}) = \alpha$. The authors show that in a "nondeterministic abstract interpretation", regardless of the allocation strategy taken during analysis, a consistent abstraction map may be constructed *a posteriori* which justifies each choice of abstract address whatever it may have been.

What is special about the allocation of abstract addresses which could make even a random number generator a sound choice of allocator? Clearly we could not define the operation of most other components of our abstract machine randomly and still guarentee

a sound analysis. Intuitively, it is because in a concrete evaluation of any program, we may select a fresh and unique address for every new allocation. (In fact, we might justify a garbage collection algorithm as safe by showing that when an address becomes unreachable, it may be reclaimed and the semantics are guarenteed to remain equivalent to allocating a fresh address.) Whatever the behavior of abstract address allocation, no inconsistency may arise in the $\alpha_{Addr}$ it induces because of just this property. No concrete address may become abstracted to two different abstract addresses because no concrete address is allocated more than once.

If a concrete machine and its abstract machine are simulated in lock-step, each concrete address $a$ and corresponding abstract address $\hat{a}$ represent a point $[a \mapsto \hat{a}]$ in the abstraction map $\alpha_{Addr}$. For the abstraction induced by the pairing of a concrete allocator and abstract allocator to be inconsistent, the same concrete address would need to be abstracted to two different abstract addresses. Because a concrete allocator must, by definition, produce a fresh address for every invocation, no such inconsistency is possible, regardless of the abstract allocator chosen. This makes abstract allocation a tunable analysis parameter with the unique property that every possible tuning results in a sound analysis.

### 3.4.1 Policy-Factored Semantics

Each transition accepts an allocator (used only for that transition) which maps variables to addresses.

$$
\begin{aligned}
& f_\Sigma : \Sigma \to (\mathsf{Var} \to Addr) \rightharpoonup \Sigma \\
& f_\Sigma \underbrace{((ae_f\ ae_1\ \ldots\ ae_j),\ \rho,\ \sigma)}_{\varsigma}(alloc_\pi) = (e,\ \rho',\ \sigma') \\
\textit{where} \quad & ((\lambda\ (x_1\ \ldots\ x_j)\ e)\ \rho_\lambda) = \mathcal{A}(ae_f,\ \varsigma) \\
& \rho' = \rho_\lambda[x_i \mapsto a_i] \\
& \sigma' = \sigma[a_i \mapsto \mathcal{A}(ae_i,\ \varsigma)] \\
& a_i = alloc_\pi(x_i)
\end{aligned}
$$

The concrete allocation policy $\pi$ produces a per-transition allocator given a concrete state.

$$
\begin{aligned}
\pi &: \Sigma \to \mathsf{Var} \to Addr \\
\pi(\varsigma)(x) &= (x,\ \varsigma)
\end{aligned}
$$

We should note the correspondence with our original (unfactored) semantics:

$$\varsigma \Rightarrow \varsigma' \iff f_\Sigma(\varsigma)(\pi(\varsigma)) = \varsigma'$$

Each transition accepts an abstract allocator mapping variables to abstract addresses. The local function $\widehat{fin}$ builds a state-finalizer for a given closure.

$$\hat{f}_\Sigma \colon \hat{\Sigma} \to \mathcal{P}((\mathsf{Var} \to \widehat{Addr}) \to \hat{\Sigma})$$

$$\hat{f}_\Sigma \underbrace{(\texttt{(}ae_f\ ae_1\ \ldots\ ae_j\texttt{)},\ \hat{\rho},\ \hat{\sigma})}_{\hat{\varsigma}} = \{\widehat{fin}(\widehat{clo}) \mid \underbrace{(\texttt{(}\lambda\ \texttt{(}x_1\ \ldots\ x_j\texttt{)}\ e\texttt{)},\ \hat{\rho}_\lambda)}_{\widehat{clo}} \in \hat{\mathcal{A}}(ae_f,\ \hat{\varsigma})\}$$

$$\widehat{fin} \colon \widehat{Clo} \to (\mathsf{Var} \to \widehat{Addr}) \to \hat{\Sigma}$$

$$\widehat{fin}(\texttt{(}\lambda\ \texttt{(}x_1\ \ldots\ x_j\texttt{)}\ e\texttt{)},\ \hat{\rho}_\lambda)(\widehat{alloc}_\pi) = (e,\ \hat{\rho}',\ \hat{\sigma}')$$

$$\begin{aligned} \textit{where} \qquad \hat{\rho}' &= \hat{\rho}_\lambda[x_i \mapsto \hat{a}_i] \\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma})] \\ \hat{a}_i &= \widehat{alloc}_\pi(x_i) \end{aligned}$$

An abstract allocation policy $\hat{\pi}$ for 1-CFA produces these per-transition allocators given an abstract state.

$$\hat{\pi} \colon \hat{\Sigma} \to (\mathsf{Var} \to \widehat{Addr})$$

$$\hat{\pi}(\texttt{(}ae\ \ldots\texttt{)},\ \hat{\rho},\ \hat{\sigma})(x) = (x,\ \texttt{(}ae\ \ldots\texttt{)})$$

We should again observe the correspondence with our original (unfactored) abstract semantics:

$$\hat{\varsigma} \rightsquigarrow_{\hat{\Sigma}} \hat{\varsigma}' \iff \exists \hat{h} \in \hat{f}_\Sigma(\hat{\varsigma}) \colon\ \hat{h}(\hat{\pi}(\hat{\varsigma})) = \hat{\varsigma}'$$

### 3.4.2 Policy-Factored Galois Connection

We produce a family of policy-factored abstraction maps $\beta$ which are identical to the original definition for $\alpha$ except parameterized by an abstraction map for addresses.

$$\begin{aligned} \beta_\Sigma(\alpha_{Addr})(e,\ \rho,\ \sigma) &= (e,\ \beta_{Env}(\alpha_{Addr})(\rho),\ \beta_{Store}(\alpha_{Addr})(\sigma)) \\ \beta_{Env}(\alpha_{Addr})(\rho) &= \{(x,\ \alpha_{Addr}(a)) \mid (x,\ a) \in \rho\} \\ \beta_{Store}(\alpha_{Addr})(\sigma) &= \bigsqcup_{(a,clo)\in\sigma} [\alpha_{Addr}(a) \mapsto \{\beta_{Clo}(\alpha_{Addr})(clo)\}] \\ \beta_{Clo}(\alpha_{Addr})(lam,\ \rho) &= (lam,\ \beta_{Env}(\alpha_{Addr})(\rho)) \end{aligned}$$

Yet again we should note the correspondence $\alpha_\Sigma = \beta_\Sigma(\alpha_{Addr})$, *i.e.*:

$$\alpha_\Sigma(\varsigma) = \hat{\varsigma} \iff \beta_\Sigma(\alpha_{Addr})(\varsigma) = \hat{\varsigma}$$

For a given $\alpha_{Addr}$, $\pi$, and $\hat{\pi}$, showing that $\hat{f}$ simulates $f$ under the abstraction $\beta(\alpha_{Addr})$ reduces to an inductive step for concrete transitions $f(\varsigma)(\pi(\varsigma))$:

$$\beta_\Sigma(\alpha_{Addr})(\varsigma) \sqsubseteq \hat{\varsigma} \implies \exists \hat{h} \in \hat{f}(\hat{\varsigma}) \colon\ \beta(\alpha_{Addr})(f(\varsigma)(\pi(\varsigma))) \sqsubseteq \hat{h}(\hat{\pi}(\hat{\varsigma}))$$

### 3.4.3 A Dependent Simulation Condition

A policy-factored abstract transfer function $\hat{f}$ is a *dependent simulation* of a factored concrete transfer function $f$ under a factored abstraction map $\beta$ iff:

For all abstraction maps $\alpha_{Addr} : Addr \to \widehat{Addr}$, concrete transition-allocators $alloc_\pi : \mathsf{Var} \to Addr$, abstract transition-allocators $\widehat{alloc_\pi} : \mathsf{Var} \to \widehat{Addr}$, and variables $x \in \mathsf{Var}$:

$$\beta_\Sigma(\alpha_{Addr})(\varsigma) \sqsubseteq \hat{\varsigma} \implies$$
$$\exists \hat{h} \in \hat{f}(\hat{\varsigma}) \colon \ \beta(\alpha_{Addr}[alloc_\pi(x) \mapsto \widehat{alloc_\pi}(x)])(f(\varsigma)(alloc_\pi)) \sqsubseteq \hat{h}(\widehat{alloc_\pi})$$

### 3.4.4 *A Posteriori* Construction of $\alpha_{Addr}$

Now we can show that even a nondeterministic choice of allocator is sound. We fix an arbitrary allocation policy $\hat{\pi}$ and run the analysis, producing a closed abstract transition graph $(\hat{S}, \rightsquigarrow)$. If the policy-factored analysis satisfies the dependent simulation condition, we can show after-the-fact that an $\alpha_{Addr}$ exists which justifies whatever $\hat{\pi}$ we had chosen.

**Theorem 2.** *(A posteriori soundness) Assuming:*

– $\boldsymbol{\varsigma} = \langle \varsigma_0,\ \varsigma_1 \ldots \rangle$ *and* $\boldsymbol{alloc} = \langle alloc_0,\ alloc_1,\ \ldots \rangle$ *is a concrete execution such that* $f_\Sigma(\varsigma_i)(alloc_i) = \varsigma_{i+1}$.

– $\hat{f}_\Sigma$ *is a dependent simulation of* $f_\Sigma$.

– $(\hat{S}, \rightsquigarrow)$ *is a closed abstract transition graph for* $\hat{f}_\Sigma$ *where* $\hat{\varsigma}_0 \in \hat{S}$.

– *For all* $\alpha_{Addr}$, $\beta(\alpha_{Addr})(\varsigma_0) \sqsubseteq \hat{\varsigma}_0$.

*Then, there exists a map* $\alpha_{Addr} : Addr \to \widehat{Addr}$ *such that the graph* $(\hat{S}, \twoheadrightarrow)$ *is a sound simulation of* $\boldsymbol{\varsigma}$ *under the abstraction map* $\beta(\alpha_{Addr})$.

*Proof.* We proceed by construction of $\alpha_{Addr}$. To do this, we define a sequence of abstract states $\hat{\boldsymbol{\varsigma}} = \langle \hat{\varsigma}_0,\ \hat{\varsigma}_1,\ \ldots \rangle$, abstract transition-alloctors $\widehat{\boldsymbol{alloc}} = \langle \widehat{alloc}_0,\ \widehat{alloc}_1,\ \ldots \rangle$ where $\hat{\varsigma}_i \in \hat{S}$ and $\hat{f}_\Sigma(\hat{\varsigma}_i)(\widehat{alloc_i}) = \hat{\varsigma_{i+1}}$. Along the way, we build a sequence of partial abstration maps $\boldsymbol{\alpha} = \langle \alpha_0,\ \alpha_1,\ \ldots \rangle$ and show by induction that $\beta(\alpha_i)(\varsigma_i) = \hat{\varsigma}_i$. We may start with $\alpha_0 = \lambda a.\bot$.

The dependent simulation condition directly states there must exist an $\widehat{alloc_i}$ and $\hat{\varsigma}_{i+1}$ such that $\hat{f}_\Sigma(\hat{\varsigma}_i)(\widehat{alloc_i}) = \hat{\varsigma}_{i+1}$ and $\beta(\alpha_{i+1})(\varsigma_{i+1}) \sqsubseteq \hat{\varsigma}_{i+1}$ where we construct the next abstraction map $\alpha_{i+1} = \alpha_i[alloc_i(x) \mapsto \widehat{alloc_i}(x)]$, updating it with a new binding for all addresses produced by $alloc_i$. □

## 3.5 Store Widening

Various forms of widening and further approximations may be layered on top of the naïve analysis ($\rightsquigarrow\hat{}_s$). One such approximation is store widening, which is necessary for our analysis to be polynomial-time in the size of the program. To see why store widening is so important, let us consider the complexity of an analysis using ($\rightsquigarrow\hat{}_s$). The height of the power-set lattice $(\hat{S}, \cup, \cap)$ is the number of elements in $\hat{\Sigma}$ which is the product of call sites, environments, and stores. A standard worklist algorithm at most does work proportional to the number of states it can discover (Might et al., 2010). Even in the monovariant case, analysis run-time is thus in:

$$O(\overbrace{n}^{|\mathsf{Call}|\times|\widehat{Env}|} \times \overbrace{2^{n^2}}^{|\widehat{Store}|})$$

The number of syntactic points in an input program is in $O(n)$. In the monovariant case, environments map variables to themselves and are isomorphic to the sets of free variables that may be determined for each syntactic point. The number of addresses produced by our monovariant analysis is in $O(n)$ as these are either syntactic variables or expressions. The number of value stores may be visualized as a table of possible mappings from every address to every abstract closure—each may be included in a given store or not as seen in Figure 3.2. The number of abstract closures is in $O(n)$ because lambdas uniquely determine a monovariant environment. (The same is true of call sites and their monovariant environments within states $\hat{\varsigma}$.) This times the number of addresses gives $O(n^2)$ possible additions to the value store.

The crux of the issue is that, in exploring a naïve state-space (where each state is specific to a whole store), we may explore both sides of every diamond in the store lattice. All combinations of possible bindings in a store may need to be explored, including every alternate path up the store lattice. For example, along one explored path, we might extend an address $\hat{a}_1$ with $\widehat{clo}_1$ before extending it with $\widehat{clo}_2$, and along another path, we might add these closures in the reverse order (*i.e.*, $\widehat{clo}_2$ before $\widehat{clo}_1$). We might also extend another address $\hat{a}_2$ with $\widehat{clo}_1$ either before or after either of these cases, and so forth. This potential for exponential blow-up is unavoidable without further widening or coarser structural abstraction.

Global-store widening is an essential technique for combating exponential blow up. This lifts the store alongside a set of reachable states instead of nesting them inside states $\hat{\varsigma}$. To formalize this, we define new *widened* state spaces that pair a set of reachable *configurations* (states *sans* stores) with a single, global value store we maintain as the least upper bound of

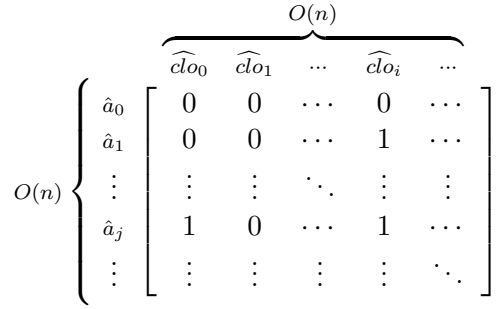



**Figure 3.2**. The value space of stores.

all stores we encounter during analysis. Instead of accumulating whole stores, and thereby all possible sequences of additions within such stores, the analysis strictly accumulates new values in the store in the same way ($\hat{\rightsquigarrow_S}$) accumulates reachable states in a collection $\hat{s}$:

$$\hat{\xi} \in \hat{\Xi} \triangleq \hat{R} \times \widehat{Store} \qquad \text{[state-spaces]}$$

$$\hat{r} \in \hat{R} \triangleq \mathcal{P}(\hat{C}) \qquad \text{[reachable configurations]}$$

$$\hat{c} \in \hat{C} \triangleq \mathsf{Call} \times \widehat{Env} \qquad \text{[configurations]}$$

A widened transfer function ($\hat{\rightsquigarrow_\Xi}$) may then be defined that, like ($\hat{\rightsquigarrow_S}$), is a monotonic, total function we may iterate to a fixed point.

$$(\hat{\rightsquigarrow_\Xi}) : \hat{\Xi} \to \hat{\Xi}$$

This may be defined in terms of ($\hat{\rightsquigarrow_\Sigma}$), as was ($\hat{\rightsquigarrow_S}$), by transitioning each reachable configuration using the global store to yield a new set of reachable configurations and a set of stores whose least upper bound is the new global store:

$$(\hat{r}, \hat{\sigma}) \mathrel{\hat{\rightsquigarrow_\Xi}} (\hat{r}', \hat{\sigma}'), \text{ where}$$

$$\hat{s} = \{\hat{\varsigma} \mid (call, \hat{\rho}) \in \hat{r} \wedge (call, \hat{\rho}, \hat{\sigma}) \mathrel{\hat{\rightsquigarrow_\Sigma}} \hat{\varsigma}\} \cup \{\hat{\mathcal{I}}(call_0)\}$$

$$\hat{r}' = \{(call, \hat{\rho}) \mid (call, \hat{\rho}, \hat{\sigma}) \in \hat{s}\}$$

$$\hat{\sigma}' = \bigsqcup_{(\_,\_,\hat{\sigma}'') \in \hat{s}} \hat{\sigma}''$$

In this definition, an underscore (wildcard) matches anything. The height of the $\hat{R}$ lattice is linear (as environments are monovariant) and the height of the store lattices are quadratic (as each global store is strictly extended). Each extension of the store may require $O(n)$ transitions because at any given store, we must transition every configuration to be sure

to obtain any changes to the store or otherwise reach a fixed point. A traditional worklist algorithm for computing a fixed point is thus cubic:

$$O(\overbrace{n}^{|\hat{C}|} \times \overbrace{n^2}^{|\widehat{Store}|})$$

Using advanced bit-packing techniques (Midtgaard, 2012), the best known algorithm for global-store-widened 0-CFA is in $O(\frac{n^3}{\log n})$.

# CHAPTER 4

# A PARAMETRIC SEMANTICS FOR POLYVARIANCE

In the past 35 years since call-sensitive data-flow analysis was introduced by Sharir and Pnueli Sharir and Pnueli (1981), a wide variety of both subtly and essentially distinct forms of polyvariant static analysis have been explored in the literature. The *polyvariance* of a static analysis, broadly construed, is the degree to which program values at runtime are broken into a multiplicity of distinct static approximations of their dynamic behavior. This is consistent with previous uses of the term, although the exact nature of its broad diversity of uses has not previously been well explored or formalized for any particular methodology.

For example, consider a function applied on different values across more than one call site—an identity function applied on both true and false in Racket (Racket Community, 2015):

```
... 0(let ([id (lambda (x) x)])
      1(id #f)
      2(id #t))
```

A *monovariant* analysis is one which maintains only a single structurally distinct approximation, a single variant, or a single flow set (in the nomenclature of flow analysis) that over-approximates the behavior of all possible values for each syntactic variable or intermediate expression. Although the variable `x` may become bound to `#f` when called from $e_1$ (*i.e.*, the first call site, `1(id #f)`) and `#t` when called from $e_2$ (*i.e.*, the second call site, `2(id #t)`), a monovariant analysis will merge these values and produce only a single flow set {`#t`, `#f`} for `x` (or perhaps {`bool`}, $\top$, etc., depending on the representation of abstract values and widening used).

A more *polyvariant* analysis, by contrast, would allow for a larger number of distinct flow sets, a choice which has the potential to increase analysis complexity, analysis precision, or both, depending on the target of analysis. The seminal and still most widely used form of

polyvariance, *k-call sensitivity*, distinguishes one context for each call trace or call history of length $k$ that precedes a syntactic binding site. In our example above, even a 1-call sensitive analysis (*e.g.*, Shivers' 1-CFA) would be enough to keep the values `#t` and `#f` from merging in a single flow set for `x`. A 1-call sensitive analysis produces two distinct flow sets for `x` in this program. One is unique to both `x` and the first call site $e_1$, the other is unique to both `x` and the second call site $e_2$.

A wide gamut of polyvariant techniques has been discussed in the literature (Agesen, 1995; Amtoft and Turbak, 2000; Banerjee, 1997; Bravenboer and Smaragdakis, 2009; Gilray and Might, 2013a,b; Gilray et al., 2016; Harrison, 1989; Holdermans and Hage, 2010; Jagannathan et al., 1997; Jones and Muchnick, 1982; Koot and Hage, 2015; Lhoták, 2006; Lhoták and Hendren, 2006, 2008; Liang et al., 2005; Milanova et al., 2005; Naik et al., 2006; Oxhøj et al., 1992; Palsberg and Pavlopoulou, 2001; Sharir and Pnueli, 1981; Shivers, 1991; Smaragdakis et al., 2011; Verstoep and Hage, 2015; Wright and Jagannathan, 1998), comprised of both subtle variations and completely disparate strategies from methods and applications like type systems, abstract interpretations, and constraint-based analyses. While many of these designs and presentations share elements in common, each was designed and implemented separately with rather little work focused on unifying or connecting distinct implementations and strategies (Amtoft and Turbak, 2000; Gilray and Might, 2013a; Smaragdakis et al., 2011).

We present a new methodology which both unifies and generalizes the myriad strategies for polyvariance as tunings of a single function, an abstract allocator. We show that the design space of polyvariance uniquely and exactly corresponds to the design space of tunings for this function and that no possible tuning can lead to an unsound analysis. All classic flavors of polyvariance can be easily recapitulated using our methodology and we are able to derive novel variations by generalizing each. By proving that no tuning of allocation is unsound, and by permitting arbitrary instrumentation of a core flow analysis to guide the behavior of this function, we are able to show that *all conceivable* sound strategies for polyvariance may be implemented in a parametric abstract semantics.

## 4.1 Myriad Styles of Polyvariance

Following Sharir and Pnueli Sharir and Pnueli (1981), call sensitivity was used by Jones, Muchnick, and Harrison in the '80s and then generalized to control-flow analysis of higher-order languages ($k$-CFA) by Shivers (Harrison, 1989; Jones and Muchnick, 1982; Shivers, 1991). The '90s saw a broader exploration of different strategies for polyvariance,

including a polynomial-time approximation for call-sensitive higher-order flow analysis by Jagannathan and Weeks (Jagannathan and Weeks, 1995) and Agesen's *Cartesian product algorithm* (CPA) (Agesen, 1995), an enhancement for type recovery algorithms. A variety of polyvariant type systems emerged, the preponderance of which are call sensitive (Amtoft and Turbak, 2000; Banerjee, 1997; Holdermans and Hage, 2010; Koot and Hage, 2015; Oxhøj et al., 1992; Palsberg and Pavlopoulou, 2001; Verstoep and Hage, 2015). Ideas from type systems also found their way back into flow analyses Amtoft and Turbak (2000); Cousot (1997); for example, inspired by `let`-polymorphism, Wright and Jagannathan (1998) presents *polymorphic splitting*, a style of call sensitivity that varies the degree of sensitivity on a per-function basis using the `let`-depth of each function as its heuristic. Milanova et al. (2005) introduces another very different style of polyvariance, *object sensitivity*, which uses a history of the allocation points of objects to differentiate program contexts. Like call sensitivity, object sensitivity forms a well-ordered design space of increasingly precise analyses that may reach concrete (precise) evaluation only in its limit. Growing evidence (particularly for points-to analysis of Java) supports the idea that object-sensitive analyses tend to be more effective and efficient than call-sensitive ones for object-oriented targets Bravenboer and Smaragdakis (2009); Lhoták (2006); Lhoták and Hendren (2006, 2008); Liang et al. (2005); Naik et al. (2006). Recently, Smaragdakis et al. (2011) have generalized object sensitivity to a wider range of variations and introduced a new approximation of these called *type sensitivity*.

Different styles of polyvariance may be viewed as different heuristics for managing the trade-off between complexity and precision in a static analysis. Call sensitivity supposes that program values will tend to correlate with recent call sites (or the surrounding few stack frames) and allows for more complexity in a way which is capable of expressing these correlations. For targets where this is a good heuristic, a greater number of more precise flow sets will result. For targets where it is not, a greater number of equally imprecise flow sets may result. Object sensitivity supposes that program values will tend to correlate with the allocation point of a function's receiving object (and the allocation point of its allocating object in turn, and so forth). The Cartesian product algorithm supposes that program values for one argument to a function will correlate with program values for other arguments to the same function. Polymorphic splitting supposes that more deeply nested function definitions will benefit from a greater degree of call history than less deeply nested definitions. Each strategy for polyvariance represents a gambit on the part of an analysis designer that targets of the analysis will tend to behave in a certain way.

### 4.1.1 Toward Better Trade-offs

To further illustrate this point, consider a `max` function:

```
... 0(let ([max (lambda (a b) (if (> a b) a b))])
      1(max 0 1)
      2(max ‘‘a’’ ‘‘at’’))
```

As before, a 1-call sensitive analysis will be precise enough to keep the values `0` and `‘‘a’’` from merging; however, if `max` is $\eta$-expanded $k$ times, a $k$-call sensitive analysis will not be enough to keep the approximation for `a`'s behavior from becoming $\{\texttt{int}, \texttt{string}\}$ (in the case of a type recovery, or $\{\texttt{0}, \texttt{‘‘a’’}\}$ for a constant propagation). In a sense, this is not imprecise because neither of these are spurious values for `a`. Even from this context-agnostic perspective, however, spurious interargument patterns are being implied between the approximations for `a` and `b`. It appears that `max` could be invoked on both an integer and a string at the same time. To eliminate this kind of imprecision, the Cartesian product algorithm builds up whole tuples of arguments for each function, preserving these interargument patterns and eliminating the possibility of calls like `(max ‘‘a’’ 1)`. For the function `max`, CPA has the same complexity as $k$-CFA but yields significantly greater precision. For a different function, one where all such interargument combinations are possible, CPA will exhaustively enumerate all combinations at great expense, while $k$-CFA implies them at no additional cost. For different targets of analysis, or even different portions of the same target of analysis, different styles of polyvariance can exhibit very different efficacies in yielding degrees of precision (or efficiencies at yielding the same degree of precision).

Similar trade-offs can be described for other forms of polyvariance and each further intersects with the well-known paradox of flow analysis that greater precision can, in practice, lead to smaller model sizes and faster runtimes (Wright and Jagannathan, 1998). While establishing better guarantees of analysis efficiency does correlate inversely with guarantees of analysis precision in absolute terms, analyses with more precise information for data flows will often have more precise control flows and explore a smaller overall model. Scaling polyvariant flow analysis to larger programs written in dynamic languages like Racket/Scheme, Python, or JavaScript, hinges on being able to reliably make good trade-offs and exploit this paradox; otherwise, the global use of polyvariance (for nearly all the varieties mentioned) yields an exponential-time analysis in the worst case due to the structure of environments in a higher-order setting (Might et al., 2010; Van Horn and Mairson, 2008). (The exception among the techniques mentioned is the poly-$k$-CFA of Jagannathan and Weeks (1995) which effectively uses flat environments.)

Much work has gone into simply defining a correct semantics for Python and JavaScript (Guth, 2013; Ranson et al., 2008; Smeding, 2009), with perhaps the most compelling effort (at least, for the purposes of constructing a static analysis) being that of the $\lambda_{JS}$ of Guha et al. (2010) and of its successor, the $\lambda_{S5}$ of Politz et al. (2012). This approach reduces programs to a simple core language consisting of fewer than 35 syntactic forms, reifying the hidden and implicit complexity of full JavaScript as explicit complexity written in the core language. Desugaring is appealing for analysis designers as it gives a simple and precise semantics to abstract; however, it also presents one of the major obstacles to precise analysis as it adds a significant runtime environment and layers of indirection through it. Consider an example the authors of $\lambda_{S5}$ use to motivate the need for their carefully constructed semantics: `[] + {}` yields the string "`[object Object]`". Strangely enough, this behavior is correct as defined by the ECMAScript specification for addition—a complex algorithm encompassing a number of special cases which can interact in unexpected ways ECMA (2011). The desugaring process for $\lambda_{S5}$ replaces addition with a function call to `%PrimAdd` from the runtime environment. `%PrimAdd` in-turn calls `%ToPrimitive` on both its arguments before breaking into cases. This means that for any uses of addition to return precise results, or likely anything other than $\top$, a $k$-call-sensitive analysis requires an intractable $k \geq 2$.

One potential solution might be to use the flat environments of poly-$k$-CFA or the mCFA of Might et al. (2010), however in the case of a language like Racket or Scheme, the frequent idiomatic use of higher-order functions could make this impractical for getting needed precision in the structure of environments. What seems to be needed are increasingly nuanced, introspective, and adaptive forms of polyvariance which better suit their targets and the properties we may wish to prove or discover for them. For example, a recent development shows that the polyvariance of continuations can be adapted in a way which guarantees perfect stack precision (*i.e.*, the perfect return flows of Vardoulakis and Shivers (2010), Earl et al. (2010), Earl et al. (2012), and Johnson and Van Horn (2014)) at no asymptotic complexity overhead Gilray et al. (2016), a quite ideal trade-off between complexity and precision obtained through a subtle refinement of the polyvariance used. The direction of research in this area and the challenges of precisely modeling dynamic higher-order programming languages suggests an important development would be an easy way to adjust the polyvariance of a flow analysis (in theory and in practical implementations) that is both always safe and fully general.

### 4.1.2 The Big Picture

We develop a unified approach to encoding *all* and *only* sound forms of polyvariance, as tunings of an allocation function. We show that the design space of polyvariance uniquely and exactly circumscribes the design space of tunings for this function and that no possible allocation strategy can lead to an unsound analysis. This leads us to the main idea of this paper: *allocation characterizes polyvariance*. All classic flavors of polyvariance can be easily recapitulated using our methodology and we are able to derive novel variations by generalizing each. Furthermore, all possible variations on allocation yield a sound polyvariant analysis.

There are thus two directions to consider: that every allocation strategy gives rise to a sound polyvariant analysis, and that every sound polyvariant analysis can be implemented by an allocation strategy. We employ the *a posteriori* soundness process of Might and Manolios (2009) to show that every allocator results in a sound analysis. This means we may instrument our core flow analysis arbitrarily to guide the allocator and so long as this extension to our analysis only impacts the store through the narrow interface of allocation, no instrumentation may lead to an unsound flow analysis. Furthermore, every form of polyvariance is expressible through this interface and we may express any allocation behavior by permitting any instrumentation. All forms of polyvariance are ways of merging and differentiating flow sets. In a store-passing-style interpreter, this is determined by the addresses we allocate.

In Figure 4.1, we summarize a selection of the styles of polyvariance we survey in Section 4.6. For each of these, classic styles of polyvariance and novel variations, there is a pair of an allocation function and an instrumentation that encodes it. For example, the instrumentation for $k$-call sensitivity adds tracking of $k$-length call histories to the analysis so that a call-sensitive allocator may produce addresses unique to both the syntactic variable being allocated for and the current approximate calling context.

## 4.2 Allocation as a Tunable Parameter

In the previous chapter, we systematically developed a global-store-widened analysis of CPS $\lambda$-calculus based on a concrete abstract-machine semantics. It is a *monovariant* analysis which means each syntactic variable or intermediate expression we track during analysis receives exactly one flow set to over-approximate all its possible values. A closely related term is *context insensitive*, which means insensitive to any form of context and is a broader term that may, for example, include analyses less precise than this as well. In Section 3.1.1,

| **Strategy** | **Allocator** | **Instrumentation** |
|---|---|---|
| Univariance | $\widetilde{alloc}_{\top}(x, \tilde{\varsigma}) \triangleq \top$ | None |
| Monovariance | $\widetilde{alloc}_{\text{0CFA}}(x, \tilde{\varsigma}) \triangleq x$ | None |
| 1-CFA | $\widetilde{alloc}_{\text{1CFA}}(x, (call, \_, \_)) \triangleq (x, call)$ | None |
| Call-Only Sensitivity | $\widetilde{alloc}_{\text{callonly}}(x, (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\iota})) \triangleq (x, \tilde{\iota})$ | Tracks a history of only call sites |
| Call+Return Sensitivity | $\widetilde{alloc}_{\text{calls}}(x, (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\iota})) \triangleq (x, \tilde{\iota})$ | Tracks a history of call and return points |
| Polymorphic Splitting | $\widetilde{alloc}_{\text{Lcalls}}(x, (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\iota})) \triangleq (x, \tilde{\iota})$ | Tracks a variable history of call sites |
| Object Sensitivity | $\widetilde{alloc}_{\text{obj}}(x, (call, \tilde{\rho}, \tilde{\sigma}, (\tilde{\sigma}_O, \tilde{O}))) \triangleq (x, \tilde{O})$ | Tracks a history of per-object allocation points |
| Closure Sensitivity | $\widetilde{alloc}_{\text{clo}}(x, (call, \tilde{\rho}, \tilde{\sigma}, (\tilde{\sigma}_O, \tilde{O}))) \triangleq (x, \tilde{O})$ | Tracks a history of per-closure creation points |
| Zeroth-Argument Sensitivity | $\widetilde{alloc}_{\text{arg}_0}(x, (\texttt{(}ae_f\, ae_0 \ldots\texttt{)}, \ldots)) \triangleq (x, \mathcal{T}(\tilde{\mathcal{A}}(ae_0, \tilde{\varsigma})))$ | None |
| Store Sensitivity | $\widetilde{alloc}_{\text{ss}}(x, (call, \ldots, (\tilde{\iota}, \tilde{\rho}_\Sigma, \tilde{\sigma}_\Sigma))) \triangleq (x, call, \tilde{\rho}_\Sigma, \tilde{\sigma}_\Sigma)$ | Rebuilds per-state stores lost by store widening |
| Concrete Evaluation | $\widetilde{alloc}_{\bot}(x, (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\iota}))) \triangleq (x, \lvert dom(\tilde{\sigma})\rvert)$ | None |

**Figure 4.1**. A selection of allocators.

the crucial propositional statement (among those defining our abstract transition relation ($\hat{\rightsquigarrow}_{\Sigma}$)) which made the analysis monovariant was this one:

$$\hat{a}_i = x_i$$

For each allocation, an address is produced which is unique only to the syntactic variable being allocated for.

The goal of this section will be to produce a parametric semantics which may be tuned by an allocator $\widetilde{alloc}$ that only varies this aspect of the analysis, but may do so without restriction. Although we will formalize this parametric semantics on its own, in the context of our ($\hat{\rightsquigarrow}_{S}$)-analysis, the primary change looks like:

$$\hat{a}_i = \widetilde{alloc}(x_i, \hat{\varsigma})$$

This would allow us to define monovariance as a tuning of this function:

$$\widetilde{alloc}_{\text{0CFA}}(x,\ \tilde{\varsigma}) \triangleq x$$

An equivalence relation on these addresses may be lifted from a notion of equality for syntax; however, we must either affix unique labels to every program expression or assume that two identical pieces of syntax found in the same program are syntactically unequal. For simplicity, we assume the latter.

The least polyvariant analysis has an allocator which produces even fewer distinct addresses—in fact, only a single address $\top$ which over-approximates all concrete addresses in any precise evaluation of the target:

$$\widetilde{alloc}_{\top}(x,\ \tilde{\varsigma}) \triangleq \top$$

We might call this the *univariant* allocation scheme because it produces only a single address and smashes all program values together. Even an analysis as imprecise as this could have a use. For example, univariant allocation would make for an exceptionally cheap analysis powering dead-code elimination.

Instead of defining a set of abstract addresses explicitly as done in Section 3.1.1, we can now allow this set to be defined implicitly by the image or codomain of the allocation function. This does mean that for an analysis to be computable, the allocator must only produce a finite number of abstract addresses. An allocator which does not produce a finite

number of addresses, essentially an *infinitely polyvariant* allocation strategy, may be used to tune our analysis to concrete evaluation:

$$\widetilde{alloc}_{\perp}(x,\ (call, \tilde{\rho}, \tilde{\sigma})) \triangleq (x, |dom(\tilde{\sigma})|)$$

This is also an example of a form of polyvariance which must introspect on the current program state in order to produce an address. Without looking at the current store (or using another method), a concrete allocator is unable to ensure it always produces a fresh address (and thus avoids all merging in the store). Being able to represent concrete evaluation as a choice of allocator is also useful because it allows us to write a precise interpreter and a static analysis simultaneously as a single body of code. Along with promoting code reuse and concision, this means testing either one also aids the robustness and stability of the other (Jenkins et al., 2015).

Now we have seen three simple points within the design space of allocation strategies and polyvariance. Univariant allocation and concrete allocation frame this design space and represent two top-most and bottom-most strategies; monovariance lies between. Two important questions are left to be answered about the correspondence between polyvariance and allocation, however. First, we must consider whether there is any tuning of allocation which is unsafe (*i.e.*, leads to an unsound analysis) or which is not polyvariant. Second, we must consider whether there are polyvariant strategies which may not be implemented as an allocator.

## 4.3 *A Posteriori* Soundness

The usual process for proving the soundness of an abstract interpretation is *a priori* in the sense that it may be performed entirely before an analysis is executed. This is the kind of soundness theorem we described in Section 3.2. By contrast, Might and Manolios Might and Manolios (2009) describes an *a posteriori* soundness process where the abstraction map cannot be fully constructed until after analysis. This approach factors each $\alpha$ to separate the abstraction of addresses $\alpha_{Addr}$, producing a family of parametric maps $\beta$ such that $\beta(\alpha_{Addr}) = \alpha$. A non-deterministic abstract interpretation is then constructed which simultaneously attempts all possible allocation strategies. (This could also be an arbitrary allocation function without loss of generality.) After the analysis is performed, regardless of the allocation strategy taken, a consistent abstraction map may be constructed *a posteriori* which justifies each choice of abstract address whatever it may have been. It is then always possible to plug this Galois connection for addresses into the parametric Galois connection defined by $\beta$ to obtain a complete connection and proof of soundness.

What is special about the allocation of abstract addresses which could make even a random number generator a sound choice of allocator? Clearly we could not define the operation of most other components of our abstract machine randomly and still guarantee a sound analysis. Intuitively, it is because in a concrete evaluation of any program, we may select a fresh and unique address for every new allocation. (In fact, we might justify a garbage collection scheme as safe by showing that when an address becomes unreachable it may be reclaimed and the semantics are guaranteed to remain equivalent to allocating a fresh address.) Allocating a sequence of fresh, unique addresses which are never duplicates of previous concrete addresses is thus a central characteristic of what it means to be a concrete allocator. Whatever the behavior of abstract address allocation then, no inconsistency may arise in the $\alpha_{Addr}$ it induces because of just this property. No concrete address may become abstracted to two different abstract addresses along the sound abstract program trace because no concrete address is allocated more than once.

To illustrate this point, consider Figure 4.2. It shows a program $call_0$ being injected into a starting state, $\varsigma_0$, and evaluated step by step. A static analysis performed by iterating an abstract transition relation will produce a transition graph, but for the analysis to be sound, the concrete program trace must abstract to some path through this graph. Such a path is illustrated below, spurious transitions dangling from it. Dotted lines are used to illustrate "abstracts to" relationships for states and addresses (points in $\alpha_\Sigma$ and $\alpha_{Addr}$). If a concrete machine and its abstract machine are simulated in lock-step, each abstract transition which allocates an address has two choices: either it can allocate an address $\hat{a}_i$ it has allocated before, or it may allocate a new address $\hat{a}_i$. In both cases, it is deciding what the corresponding (necessarily fresh) concrete address $a_i$ must abstract to in $\alpha_{Addr}$. In this way, a bisimulation incrementally builds up an abstraction map for addresses, incrementally adding each point $[a_i \mapsto \hat{a}_i]$, one at a time.

In the original presentation of the *a posteriori* soundness theorem, Might and Manolios state an assumption which says that each new abstraction map $\alpha_{Addr}[a_i \mapsto \hat{a}_i]$ must be consistent with whatever partial abstraction map $\alpha_{Addr}$ was built up previously. No further intuitions were given for this assumption, though it is actually the central property which allows the entire *a posteriori* soundness process to work. For the abstraction induced by the pairing of a concrete allocator and abstract allocator to be inconsistent, the same concrete address would need to be abstracted to two different abstract addresses. Because a concrete allocator must, by definition, produce a fresh address for every invocation, no such inconsistency is possible, regardless of the abstract allocator chosen. Each $\alpha_{Addr}[a_i \mapsto \hat{a}_i]$

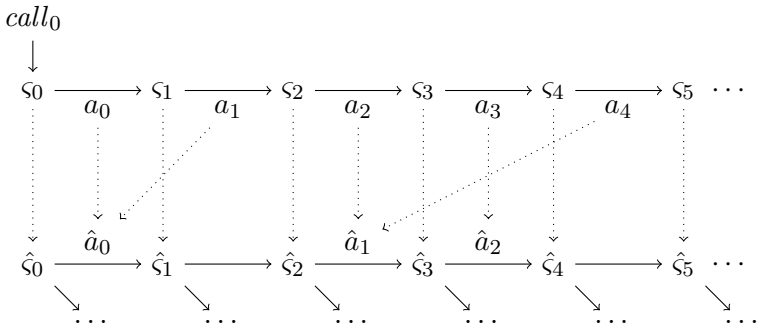



**Figure 4.2**. All strategies for allocation induce a consistent Galois connection for addresses.

must be consistent with $\alpha_{Addr}$ because the concrete address, $a_i$, cannot already be present in $\alpha_{Addr}$. This makes abstract allocation a tunable analysis parameter with the unique property that every possible tuning results in a sound analysis.

We may now review the *a posteriori* soundness theorem in the context of AAM.

**Theorem 3** (*A posteriori* soundness)**.** *If* $(\boldsymbol{\varsigma}, \mathbf{a})$ *is a concrete execution where* $\boldsymbol{\varsigma}$ *is a sequence of states and* $\mathbf{a} = \langle a_0, a_1, a_2, \ldots \rangle$ *is the sequence of concrete addresses allocated, and if some* $(\hat{\rightsquigarrow_s})$ *is a dependent simulation (Might and Manolios, 2009) of* $(\rightarrow_\Sigma)$ *under* $\beta$ *(dependent only on finding an abstraction map for addresses), and if* $\hat{s}$ *is a closed abstract transition graph over states, then there must exist an abstraction map for addresses,* $\alpha_{Addr}$*, such that* $\hat{s}$ *is a sound simulation of* $(\boldsymbol{\varsigma}, \mathbf{a})$ *under the abstraction map* $\beta(\alpha_{Addr})$*.*

*Proof.* (Summary of Might and Manolios (2009).) We proceed by performing an *a posteriori* construction of $\alpha_{Addr}$. This is done by building up a simulating sequence of abstract states $\hat{\boldsymbol{\varsigma}} = \langle \hat{\varsigma}_0, \hat{\varsigma}_1, \hat{\varsigma}_2, \ldots \rangle$, their respective abstract addresses $\tilde{\mathbf{a}} = \langle \tilde{a}_0, \tilde{a}_1, \tilde{a}_2, \ldots \rangle$, and a sequence of partial address abstraction maps $\boldsymbol{\alpha} = \langle \alpha_0, \alpha_1, \ldots \rangle$. Let $N$ be the length of $\boldsymbol{\varsigma}$ and the initial abstraction map $\alpha_0$ be $\perp$. At each step a next abstract address $\hat{a}_i$ (or several for multiple-argument lambdas) and abstract state $\hat{\varsigma}_i$ may be chosen simultaneously and nondeterministically from a nonempty set of candidate transitions guaranteed to exist by the dependent simulation condition (Might and Manolios, 2009). Each new point $[a_i \mapsto \hat{a}_i]$ is accumulated into an updated intermediate abstraction map $\alpha_i = \alpha_{i-1}[a_i \mapsto \hat{a}_i]$ which inductively builds up $\alpha_{Addr}$ in its limit:

$$\alpha_{Addr} = \lim_{i \to N} \alpha_i$$

Then $\hat{\boldsymbol{\varsigma}}$ in $\hat{s}$ is a simulation of $(\boldsymbol{\varsigma}, \mathbf{a})$ with respect to $\beta(\alpha_{Addr})$. □

## 4.4 Introspection and Instrumentation

Now we may consider whether or not any strategy for polyvariance can be implemented as a tuning of the allocation function. What about more precise forms of call sensitivity, 1-CFA or 2-CFA? A 1-call sensitive allocator can be defined by introspecting on the state being transitioned from and incorporating the most recent call site into the address being produced:

$$\widetilde{alloc}_{\text{1CFA}}(x,\ (call, \_, \_)) \triangleq (x_i, call)$$

This makes addresses (and their flow sets) unique to both the variable `x` and the call site which preceded the binding. If we were to attempt an implementation of a more precise variant of call sensitivity, however, like 2-CFA, we run into a problem because our analysis simply does not include the information necessary to guide this style of polyvariance. The current abstract state contains the most recent call site passed through, but it does not include the second most recent call site.

To permit a tuning for $\widetilde{alloc}_{\text{2CFA}}$, we could instrument our core flow analysis with a new fourth component of machine states that specifically tracks the second most recent call site. If we were to extend the analysis with such information, a 2-call sensitive allocator could be defined:

$$\widetilde{alloc}_{\text{2CFA}}(x,\ (call, \_, \_, call')) \triangleq (x_i, call, call')$$

In this case, $call'$ is a new component of machine states that represents the second most recent call site. Naturally, $(\rightsquigarrow^{\widehat{}}_{\Sigma})$ and $\hat{\mathcal{I}}$ would need to be extended to include this information.

Crucially, due to the *a posteriori* soundness theorem, we may add whatever instrumentation is needed to guide the behavior of an allocator. An analysis designer may wish to extend the core flow analysis in a way which is sound with respect to a dynamic analysis or instrumentation of the concrete semantics; however, even if the analysis is extended with unsound information about a program, this information can still be used to guide allocation behavior without any possibility of it causing unsoundness within the core flow analysis (*e.g.*, within the store). This means we may leave such instrumentation open as another parameter to a semantics and place no restrictions on its behavior. Because we lose no expressivity in this instrumentation, all conceivable allocation functions can be expressed as well. This means all strategies for merging and differentiation of abstract addresses (and their flow sets) are possible, and thus all forms of polyvariance may be expressed as a combination of some allocator and some instrumentation.

## 4.5 A Parametric Analysis

In this section, we present a parametric semantics which may be tuned by both an allocation function and an instrumentation (an arbitrary extension of the analysis). Typographically, we switch exclusively to using tildes to keep this machine distinct from the machine of Chapter 3.

Our parametric semantics is encoded in a function:

$$CFA : \overbrace{\tilde{\Sigma}}^{\text{start state}} \times \overbrace{(\tilde{\Sigma} \times \mathsf{Call} \times \widetilde{Env} \times \widetilde{Store} \to \mathcal{P}(\tilde{I}))}^{\text{instrumentation}} \times \overbrace{(\mathsf{Var} \times \tilde{\Sigma} \to \widetilde{Addr})}^{\text{allocator}} \to \overbrace{(\tilde{S} \to \tilde{S})}^{\text{analysis}}$$

*CFA* is a function of three arguments: a starting state, $\tilde{\varsigma}_0$, which specifies the program to interpret and its initial $\tilde{\imath}_0$, an instrumentation, $(\overset{\text{Inst}}{\rightsquigarrow})$, which may be used to extend the core analysis arbitrarily, and an allocator, $\widetilde{alloc}$. Given three such parameters, $CFA(\tilde{\varsigma}_0, \overset{\text{Inst}}{\rightsquigarrow}, \widetilde{alloc})$ yields a monotonic analysis function which may be iterated to a fixed point. If the image of $\widetilde{alloc}$ (the set $\widetilde{Addr}$ it can produce) and the image of $(\overset{\text{Inst}}{\rightsquigarrow})$ (the set of sets of $\tilde{I}$ it can produce) are finite sets, then there must exist an $n \in \mathbb{N}$ such that $(CFA(\tilde{\varsigma}_0, \overset{\text{Inst}}{\rightsquigarrow}, \widetilde{alloc}))^n(\bot)$ is a fixed point encoding a sound analysis of $\tilde{\varsigma}_0$ using the instrumentation and style of polyvariance specified.

Figure 4.3 shows the signatures of the three parameters to *CFA*. The allocator defines a set of addresses $\widetilde{Addr}$ for the analysis to use. The instrumentation relation defines a set of instrumentation data $\tilde{I}$ to extend the core flow analysis and enable a greater variety of allocators. An instrumentation is a function which, taking the underlying analysis transition into account, determines the instrumentation data to be included in successor states. Although this may not constrain the core flow analysis, to emphasize that it can encode an entire analysis of its own, we use the following syntactic sugar:

$$\tilde{\varsigma} \overset{\text{Inst}}{\rightsquigarrow} (call', \tilde{\rho}', \tilde{\sigma}', \tilde{\imath}') \iff \tilde{\imath}' \in (\overset{\text{Inst}}{\rightsquigarrow})(\tilde{\varsigma}, call', \tilde{\rho}', \tilde{\sigma}')$$

Figure 4.4 shows the remaining domains that the machine operates over. These are similar to the domains in Chapter 3 except that states and configurations contain instrumentation data ($\tilde{\imath}$) and addresses are specified implicitly by the allocator chosen.

Figure 4.5 defines the transition relation $(\rightsquigarrow_{\tilde{s}})$ yielded by *CFA* when supplied with all its arguments, as well as a store-widened version $(\rightsquigarrow_{\tilde{\Xi}})$ yielded by $CFA_{\triangledown}$. There are three meaningful changes from the semantics of Chapter 3, one for each parameter. First, the starting state $\tilde{\varsigma}_0$ is as provided and not produced by an injection function (this allows the user

$$\tilde{\varsigma}_0 \in \tilde{\Sigma}$$
$$\widetilde{alloc} \in \mathsf{Var} \times \tilde{\Sigma} \to \widetilde{Addr}$$
$$(\overset{\text{Inst}}{\rightsquigarrow}) \in \tilde{\Sigma} \times \mathsf{Call} \times \widetilde{Env} \times \widetilde{Store} \to \mathcal{P}(\tilde{I})$$

**Figure 4.3**. Parameters to our parametric semantics.

| | |
|---|---|
| $\tilde{s} \in \tilde{S} \triangleq \mathcal{P}(\tilde{\Sigma})$ | [analysis results] |
| $\tilde{\xi} \in \tilde{\Xi} \triangleq \tilde{R} \times \widetilde{Store}$ | [widened results] |
| $\tilde{r} \in \tilde{R} \triangleq \mathcal{P}(\tilde{C})$ | [reachable configs] |
| $\tilde{c} \in \tilde{C} \triangleq \mathsf{Call} \times \widetilde{Env} \times \tilde{I}$ | [configurations] |
| $\tilde{\varsigma} \in \tilde{\Sigma} \triangleq \mathsf{Call} \times \widetilde{Env} \times \widetilde{Store} \times \tilde{I}$ | [states] |
| $\tilde{\rho} \in \widetilde{Env} \triangleq \mathsf{Var} \rightharpoonup \widetilde{Addr}$ | [environments] |
| $\tilde{\sigma} \in \widetilde{Store} \triangleq \widetilde{Addr} \to \widetilde{Value}$ | [value stores] |
| $\tilde{\imath} \in \tilde{I}$ is defined by the parameter $(\overset{\text{Inst}}{\rightsquigarrow})$ | [inst. data] |
| $\tilde{a} \in \widetilde{Addr}$ is defined by the parameter $\widetilde{alloc}$ | [addresses] |
| $\tilde{v} \in \widetilde{Value} \triangleq \mathcal{P}(\widetilde{Clo})$ | [flow sets] |
| $\widetilde{clo} \in \widetilde{Clo} \triangleq \mathsf{Lam} \times \widetilde{Env}$ | [closures] |

**Figure 4.4**. Abstract domains for our parametric semantics.

to control the initial instrumentation data along with the program to be analyzed). Second, addresses $\tilde{a}_i$ are constrained only by the allocator $\widetilde{alloc}$ provided. Third, the instrumentation function $(\overset{\text{Inst}}{\rightsquigarrow})$ constrains the instrumentation data $\tilde{\imath}'$ based on all other components of a transition.

## 4.6 Allocation Characterizes Polyvariance

This section explores the design space opened up by a semantics parameterized over both an instrumentation and an abstract allocator, showing how it encompasses a variety of previously published polyvariant techniques, novel techniques, and variations on these.

### 4.6.1 Call Sensitivity ($k$-CFA)

Call-sensitive instrumentation tracks a history of up to $k$ call sites for use in differentiating addresses.

$$CFA(\tilde{\varsigma}_0, \overset{\text{Inst}}{\rightsquigarrow}, \widetilde{alloc}) \triangleq (\rightsquigarrow_{\tilde{S}})$$
$$CFA_{\triangledown}(\tilde{\varsigma}_0, \overset{\text{Inst}}{\rightsquigarrow}, \widetilde{alloc}) \triangleq (\rightsquigarrow_{\tilde{\Xi}})$$

---

$$(\rightsquigarrow_{\tilde{S}}) : \tilde{S} \to \tilde{S}$$
$$\tilde{s} \rightsquigarrow_{\tilde{S}} \tilde{s}', \text{ where}$$
$$\tilde{s}' = \{\tilde{\varsigma}' \mid (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\iota}) \in \tilde{s} \wedge (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\iota}) \rightsquigarrow_{\tilde{\Sigma}} \tilde{\varsigma}'\} \cup \{\tilde{\varsigma}_0\}$$

---

$$(\rightsquigarrow_{\tilde{\Xi}}) : \tilde{\Xi} \to \tilde{\Xi}$$
$$(\tilde{r}, \tilde{\sigma}) \rightsquigarrow_{\tilde{\Xi}} (\tilde{r}', \tilde{\sigma}''), \text{ where}$$
$$\tilde{s}' = \{\tilde{\varsigma}' \mid (call, \tilde{\rho}, \tilde{\iota}) \in \tilde{r} \wedge (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\iota}) \rightsquigarrow_{\tilde{\Sigma}} \tilde{\varsigma}'\} \cup \{\tilde{\varsigma}_0\}$$
$$\tilde{r}' = \{(call', \tilde{\rho}', \tilde{\iota}') \mid (call', \tilde{\rho}', \tilde{\sigma}', \tilde{\iota}') \in \tilde{s}'\}$$
$$\tilde{\sigma}'' = \bigsqcup_{(\_,\_,\tilde{\sigma}',\_) \in \tilde{s}'} \tilde{\sigma}'$$

---

$$\tilde{\mathcal{A}} : \mathsf{AE} \times \tilde{\Sigma} \rightharpoonup \widetilde{Value}$$
$$\tilde{\mathcal{A}}(x,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota})) = \tilde{\sigma}(\tilde{\rho}(x))$$
$$\tilde{\mathcal{A}}(lam,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota})) = \{(lam,\ \tilde{\rho})\}$$

---

$$(\rightsquigarrow_{\tilde{\Sigma}}) \subseteq \tilde{\Sigma} \times \tilde{\Sigma}$$
$$\overbrace{((ae_f\ ae_1\ \ldots\ ae_j), \tilde{\rho}, \tilde{\sigma}, \tilde{\iota})}^{\tilde{\varsigma}} \rightsquigarrow_{\tilde{\Sigma}} (call', \tilde{\rho}', \tilde{\sigma}', \tilde{\iota}')$$

where
$$((\lambda\ (x_0 \ldots x_j)\ call'), \tilde{\rho}_\lambda) \in \tilde{\mathcal{A}}(ae_f,\ \tilde{\varsigma})$$
$$\tilde{v}_i = \tilde{\mathcal{A}}(ae_i,\ \tilde{\varsigma})$$
$$\tilde{\rho}' = \tilde{\rho}_\lambda[x_i \mapsto \tilde{a}_i]$$
$$\tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a}_i \mapsto \tilde{v}_i]$$
$$\tilde{a}_i = \widetilde{alloc}(x_i, \tilde{\varsigma})$$
$$\tilde{\varsigma} \overset{\text{Inst}}{\rightsquigarrow} (call', \tilde{\rho}', \tilde{\sigma}', \tilde{\iota}') \qquad \text{(N.B. this syntactic sugar.)}$$

**Figure 4.5**. Transition rules for our parametric semantics.

$$(call,\ \_,\ \_,\ \tilde{\iota}) \overset{\text{Inst}}{\rightsquigarrow}_{call(k)} (\_,\ \_,\ \_,\ take_k(call:\tilde{\iota}))$$

The function $take_k$ returns the front at-most $k$ elements of its input as a new list. For this instrumentation to distinguish between two syntactically equivalent call sites located in different parts of a program, we assume two pieces of syntax are only equal when they are the same piece of syntax from the same part of the same program. This allows us to safely lift an equivalence relation on syntax to an equivalence relation for addresses.

Using $(\overset{\text{Inst}}{\rightsquigarrow}_{call(k)})$, we may tune our analysis to implement $k$-CFA using an allocator which incorporates these $k$-length call histories in the addresses it produces.

$$\widetilde{alloc}_{\text{call}}(x,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota})) \triangleq (x, \tilde{\iota})$$

The parametric semantics of Section 4.5 can be tuned to recapitulate the $k$-call sensitive style of polyvariance for a program $call_0$ using the parameterization:

$$CFA((call_0, \varnothing, \bot, \epsilon), \overset{\text{Inst}}{\rightsquigarrow}_{call(k)}, \widetilde{alloc}_{\text{call}})$$

#### 4.6.1.1 Ambiguity in $k$-CFA

The original formulation of $k$-CFA was described as tracking a history of the last $k$ call sites execution passed through; however, it was applied to a CPS intermediate representation. After a CPS transformation, every return point has been encoded as a call site. This means, as implemented, $k$-CFA was actually tracking a history of the first $k$ call sites *or* return points. A call sensitivity which only remembers call sites and not return points is a somewhat different form of polyvariance from what Shivers originally formalized Shivers (1991).

Using a direct-style language, this difference would be easy to formalize as a tuning of our parametric framework because the difference between calls and returns is syntactically evident. Using CPS, we must assume either partitioned CPS with both `lambda` and `cont` forms which behave identically but are kept separate, or we must assume this distinction is being encoded another way. To demonstrate a tuning for call-only sensitivity, we assume a predicate $Ret : \mathsf{Call} \to Bool$ which returns true if and only if the call site given was originally a return point (before CPS conversion). We are only required to change the behavior of our instrumentation:

$$(call, \_, \_, \tilde{\iota}) \overset{\text{Inst}}{\rightsquigarrow}_{callonly(k)} \begin{cases} (\_, \_, \_, take_k(call:\tilde{\iota})) & \neg Ret(call) \\ (\_, \_, \_, \tilde{\iota}) & Ret(call) \end{cases}$$

We can then instantiate our framework to a 2-call-only sensitive analysis as follows:

$$CFA((call_0, \varnothing, \bot, \epsilon))(\overset{\text{Inst}}{\rightsquigarrow}_{callonly(2)})(\widetilde{alloc}_{\text{call}})$$

We can also produce tunings which represent an analysis that remembers only return points or an analysis sensitive to the top $k$ stack frames. This means there are at least four reasonable interpretations of $k$-CFA which resolve the ambiguity between its original description and its original formalization. Each of these four styles of polyvariance are subtly different and may yield a different analysis result. Furthermore, none of these four styles of polyvariance strictly dominates the precision of any other. For each, we can find examples where that specific interpretation of $k$-CFA produces the best result. For example, the following snippet of Racket code (before CPS conversion) differentiates call+return sensitivity and call-only sensitivity.

```
(let ([id (lambda (x) x)]
      [f (lambda (g) (let ([v (g)]) v))])
     (f (lambda () (id #f)))
     (f (lambda () (id #t))))
```

The last call before binding `v` a first time is `(id #f)`, but the second time, it is `(id #t)`. This means a 1-call-only sensitive analysis will keep both addresses bound to `v` distinct. The last call *or* return before binding `v`, however, is the identity function's return point `x` in both cases. This means a 1-call+return style of polyvariance will merge both `#t` and `#f` at a single address for `v`.

Different styles of polyvariance represent different heuristics for the trade-off between precision and complexity and may strike a poor balance on one program while striking an excellent balance on another. Having a safe parametric framework which can so easily instantiate any conceivable heuristic could prove an important step in understanding which styles of polyvariance work best in what situations and thereby inform us how to better adapt the polyvariance used to suite a particular target of analysis.

#### 4.6.1.2 Variable Call Sensitivity

Wright and Jagannathan's polymorphic splitting is a form of adaptive call sensitivity inspired by `let`-polymorphism where the degree of polyvariance can vary between functions (Wright and Jagannathan, 1998). The number of `let`-form binding expressions (right-hand sides) within which a lambda was originally defined (in the case of our language, before CPS conversion) forms a simple heuristic for its call sensitivity when invoked. To implement $k$-call sensitivity with a per-function $k$, we assume a parameter function $L : \mathsf{Call} \to \mathbb{N}$ that takes the body of a lambda and gives back a $k$ for its `let`-depth (or any other heuristic for varying the maximum length call history).

$$(call,\ \_,\ \_,\ \tilde{\iota}) \overset{\text{Inst}}{\rightsquigarrow}_{Lcalls(L)} (call',\ \_,\ \_,\ take_{(L(call'))}(call : \tilde{\iota}))$$

This call history is then used for allocating addresses.

$$\widetilde{alloc}_{\text{Lcalls}}(x,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota})) \triangleq (x, \tilde{\iota})$$

Because all parameterizations of our semantics are sound, all possible heuristics $L$ are too. No tuning of $L$ can produce an infinite $k$, only arbitrarily large $k$. In the case of polymorphic-splitting, because no program can contain an infinite nesting of `let`-forms, every program has a `let`-polymorphic tuning of $L$.

This instrumentation and allocator generalize the behavior of polymorphic splitting and could be further generalized by adding a function like *Ret* from the previous subsection for selecting which call sites to include in the history to begin with. In this way, call sensitivity can be seen as a wide design space itself within the broader design space of polyvariant allocation strategies.

### 4.6.2 Object Sensitivity

Smaragdakis et al. (2011) distinguishes multiple variants of *object sensitivity*, first described by Milanova et al. (2005). This style of context sensitivity is entirely different from call sensitivity and uses a history of the allocation points for objects to guide polyvariance.

We temporarily extend our language with a `vector`-form to represent simple objects and present a faithful tuning for object sensitivity.

$$ae \in \mathsf{AE} ::= lam \mid x \mid vec$$

$$vec \in \mathsf{Vec} ::= \texttt{(vector}\ x\ \ldots\texttt{)}$$

We define abstract-object values permitted within flow sets (tuples of pointers):

$$\tilde{v} \in \widetilde{Value} \triangleq \mathcal{P}(\widetilde{Clo} + \widetilde{Obj})$$
$$\widetilde{obj} \in \widetilde{Obj} \triangleq \widetilde{Addr}^*$$

And give vector syntax an interpretation in the atomic-expression evaluator:

$$\tilde{\mathcal{A}}(x,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota})) = \tilde{\sigma}(\tilde{\rho}(x))$$
$$\tilde{\mathcal{A}}(lam,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota})) = \{(lam,\ \tilde{\rho})\}$$
$$\tilde{\mathcal{A}}(\texttt{(vector}\ x_0\ \ldots\ x_j\texttt{)},\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota})) = \{(\tilde{\rho}(x_0), \ldots \tilde{\rho}(x_j))\}$$

As the fields of our objects are effectively each vector's indices, and because these are strictly kept distinct instead of being merged, we can call this representation for vectors *field*

*sensitive* (Liang and Might, 2012). Flow sets for objects look like $\{(\tilde{a}_1, \tilde{a}_2, \tilde{a}_3), \ldots\}$ and not like $\{\tilde{a}_1, \tilde{a}_2, \tilde{a}_3, \ldots\}$, preserving the relationship between keys and values. Allowing these lists of addresses within flow sets is not a new source of unboundedness in the machine because the longest possible list is the length of the longest `vector` form in the finite program text.

In Smaragdakis' framework, $k$-full-object sensitivity tracks the allocation point of each object, the allocation point for the object which created it, and so forth. In our extended CPS language, the zeroth parameter to a function is its receiving object.

$$\tilde{O} \in \mathsf{Call}^*$$
$$\tilde{\sigma}_O \in \widetilde{Addr} \times \widetilde{Obj} \to \mathcal{P}(\mathsf{Call}^*)$$

Each state is extended with a current allocation history, $\tilde{O}$, and an object-sensitivities store, $\tilde{\sigma}_O$, which maps an abstract object at an address to a set of possible allocation histories for that object. Each transition extends $\tilde{\sigma}_O$ with a new allocation history (produced by extending the current allocation history $\tilde{O}$ with a new allocation point, the current call site) for each $ae_i$ that constructs a new object. Existing objects bound to a variable (some $y_i$) have their histories propagated along with the objects. Each transition then yields a successor for each possible allocation history associated with a receiving object. If global-store widening is used for an analysis, a similar form of widening might be used for object-sensitivities stores.

$$\overbrace{((ae_f\ \ldots\ ae_j),\ \tilde{\rho},\ \_,\ (\tilde{\sigma}_O, \tilde{O}))}^{\tilde{\varsigma}} \overset{\text{Inst}}{\rightsquigarrow}_{obj(k)} (call',\ \tilde{\rho}',\ \_,\ (\tilde{\sigma}'_O, \tilde{O}'))$$

$$\text{where} \quad ((\lambda\ (x_0 \ldots x_j)\ \ call'), \_) \in \tilde{\mathcal{A}}(ae_f, \tilde{\varsigma})$$
$$\widetilde{obj}_i \in \tilde{\mathcal{A}}(ae_i, \tilde{\varsigma})$$
$$\tilde{O}' \in \tilde{\sigma}'_O(\tilde{\rho}'(x_0), \widetilde{obj}_0)$$

$$\tilde{\sigma}'_O = \tilde{\sigma}_O \sqcup \bigsqcup_{ae_i = \texttt{(vector ...)}} [(\tilde{\rho}'(x_i), \widetilde{obj}_i) \mapsto \{take_k((ae_f\ \ldots\ ae_j) : \tilde{O})\}] \sqcup \bigsqcup_{ae_i = y_i} [(\tilde{\rho}'(x_i), \widetilde{obj}_i) \mapsto \tilde{\sigma}_O(\tilde{\rho}(y_i), \widetilde{obj}_i)]$$

An allocator for this style of polyvariance then pairs each variable with the current allocation history (ignoring the sensitivities store which only needs to be used internally).

$$\widetilde{alloc}_{\text{obj}}(x,\ (\_,\ \_,\ \_,\ (\tilde{\sigma}_O, \tilde{O}))) \triangleq (x, \tilde{O})$$

Smaragdakis et al. (2011) and Lhoták and Hendren (2006) find object sensitivity to be particularly efficient for object-oriented languages in their empirical investigations using the Java DaCapo and SpecJVM benchmarks. Kastrinis and Smaragdakis (2013) present combinations of object and call sensitivity. Combinations of styles of polyvariance can also be accomplished by a tuning of instrumentation and allocation. Section 4.6.5 presents a general method for combining multiple styles of polyvariance.

#### 4.6.2.1 Closure Sensitivity

Inspired by object sensitivity, we formalize a novel analogue for functional languages called *closure sensitivity*. In this style of polyvariance, we view closures as the fundamental objects of higher-order languages (in the terminology of object-oriented languages, they are their own receivers) and associate them with closure-creation histories directly. No changes need to be made to our CPS language and its semantics.

$$\overbrace{((ae_f\ \ldots\ ae_j),\ \tilde{\rho},\ \_,\ (\tilde{\sigma}_O, \tilde{O}))}^{\tilde{\varsigma}} \overset{\text{Inst}}{\rightsquigarrow}_{clo(k)} (call',\ \tilde{\rho}',\ \_,\ (\tilde{\sigma}'_O, \tilde{O}'))$$

$$\text{where} \quad ((\lambda\ (x_0 \ldots x_j)\ \ call'), \tilde{\rho}_\lambda) \in \tilde{\mathcal{A}}(ae_f, \tilde{\varsigma})$$
$$\widetilde{clo}_i \in \tilde{\mathcal{A}}(ae_i, \tilde{\varsigma})$$

$$\tilde{\sigma}'_O = \tilde{\sigma}_O \sqcup \bigsqcup_{ae_i=\texttt{(lambda ...)}} [(\tilde{\rho}'(x_i), \widetilde{clo}_i) \mapsto \{take_k((ae_f\ \ldots\ ae_j) : \tilde{O})\}]$$
$$\sqcup \bigsqcup_{ae_i=y_i} [(\tilde{\rho}'(x_i), \widetilde{clo}_i) \mapsto \tilde{\sigma}_O(\tilde{\rho}(y_i), \widetilde{clo}_i)]$$

$$\tilde{O}' \in \begin{cases} \{take_k((ae_f\ \ldots\ ae_j) : \tilde{O})\} & ae_f = \texttt{(lambda ...)} \\ \tilde{\sigma}_O(\tilde{\rho}(y_f), ((\lambda\ (x_0 \ldots)\ \ call'), \tilde{\rho}_\lambda)) & ae_f = y_f \end{cases}$$

Instead of vectors, closures created at the current call site become bound to the current allocation history (extended with the current call site) across each transition. Instead of the zeroth argument being used to determine successor-state allocation history, the value in call position is used.

$$\widetilde{alloc}_{\text{clo}}(x,\ (\_,\ \_,\ \_,\ (\tilde{\sigma}_O, \tilde{O}))) \triangleq (x, \tilde{O})$$

### 4.6.3 Argument Sensitivity

Agesen Agesen (1995) introduces a Cartesian product algorithm (CPA) as an enhancement to a type recovery algorithm (which can be viewed as an abstract interpretation

where dynamic program types are used as abstract values). We will consider the source of imprecision that the original formulation attempts to address, generalize its solution as a form of polyvariance in our approach, and discuss CPA's complexity and precision relative to call and object sensitivity.

The basic algorithm, that CPA extends, assigns a flow set of dynamic types for each variable in the program, it establishes constraints based on the program text, and it propagates values until all these constraints have been met. The primary method for overcoming this merging is introduced as the $p$-level expansion algorithm of Oxhøj et al. (1992)—a polyvariant type-inference algorithm and analogue to the call-string histories of Harrison and then Shivers, where the use of $p$ parallels that of $k$ in $k$-CFA. This is shown to be insufficient, however, as the authors of CPA give a case of merging which cannot be overcome by any value of $p$. Besson (2009) further illustrates this point in the context of Java, claiming "CPA beats $\infty$-CFA".

The original motivating example for CPA was a polymorphic `max` function:

```
... (let ([max (lambda (a b) (if (> a b) a b))])
         ...)
```

Here, the only constraint for an input to `max` is that it support comparison, so a call `(max` "a" "at"`)` makes as much sense as a call `(max 2 5)`. However, if both these calls are made with a sufficient amount of obfuscating call (or object) history behind them, merging will cause the flow sets for both $a$ and $b$ to each include both `string` and `int` (*i.e.*, abstract values for those types). This is imprecise as it implies that a call `(max` 2 "at"`)` is possible, even when it is not. The problem then, can be summarized as the existence of spurious interargument patterns which become inevitable when the flow sets for different syntactic arguments are entirely distinct.

The solution that CPA proposes is to replace flow sets of per-argument types, with flow sets of per-function tuples of types. In such an analysis, the function `max` itself would be typed $\{(\texttt{int}, \texttt{int}),\ (\texttt{string}, \texttt{string})\ \ldots\}$ preserving interargument patterns and eliminating spurious calls where the types do not match.

In essence, this change makes flow sets for each argument specific to the entire tuple of types received in a call. This suggests that, although no amount of call history will ensure the preservation of interargument correlations, a form of polyvariance which makes addresses specific to a tuple of abstract values for arguments can.

We must be careful here in extending this idea to an allocator for our CPS language. If a tuple of closures is included inside addresses, the mutual recursion of addresses, closures,

and environments makes the analysis unbounded. Instead, we assume a helper function $\mathcal{T}$ which further abstracts abstract values so they cannot contain addresses. For an approach especially similar to CPA itself, we might define $\mathcal{T}$ so it yields types. For a functional language, we can define $\mathcal{T}$ so that it strips environments out of closures and leaves just a set of syntactic lambdas. For example:

$$\mathcal{T}(\tilde{d}) \triangleq \{lam \mid (lam, \tilde{\rho}) \in \tilde{d}\}$$

In a sense, syntactic lambdas are at least as specific as a type (their type signature, whatever type system is used) whether or not that type is known a priori by an analysis (Gilray and Might, 2013a).

With this, we may define an *argument sensitive* style of polyvariance, like CPA, as an abstract allocator.

$$\widetilde{alloc}_{\text{CPA}}(x, \overbrace{(\texttt{(}ae_f\ ae_0\ \ldots\ ae_j\texttt{)},\ \_,\ \_,\ \_)}^{\tilde{\varsigma}}) \\ = (x, (\mathcal{T}(\tilde{\mathcal{A}}(ae_0, \tilde{\varsigma})),\ \ldots,\ \mathcal{T}(\tilde{\mathcal{A}}(ae_j, \tilde{\varsigma}))))$$

We can also observe how easy it would be to construct less precise variations of this allocator by including only some arguments within addresses. For example, including only the first argument might yield enough precision in many cases:

$$\widetilde{alloc}_{\text{arg}_0}(x, \overbrace{(\texttt{(}ae_f\ ae_0\ \ldots\ ae_j\texttt{)},\ \_,\ \_,\ \_)}^{\tilde{\varsigma}}) \\ = (x, \mathcal{T}(\tilde{\mathcal{A}}(ae_0, \tilde{\varsigma})))$$

We could even vary the arguments an analysis is sensitive to on a per-function basis like we did for polymorphic splitting in Section 4.6.1.2.

Like call sensitivity and object sensitivity, CPA can be of exponential complexity in the size of the program and is exceedingly impractical for use on sufficiently complex input programs. CPA is also, however, an excellent illustration of the principal that, in *practice*, more precision can also lead to smaller model sizes and faster analysis times. Where CPA improves precision, it is also fastest, and where CPA is unnecessary and delivers no improvement over $k$-CFA, it is enormously inefficient. For a function like `max`, one where the types of the arguments should match, CPA accumulates only a single value for each type that can flow to the function. For a function where all combinations of arguments are possible, CPA requires each combination to be enumerated explicitly. $k$-CFA *implies* all interargument combinations for equal precision at far greater efficiency. This would seem to support an effort to discover more adaptive variations on CPA.

### 4.6.4 Extreme-Precision Allocators

We can even further generalize the central idea of CPA to consider forms of polyvariance which preserves interaddress correlations in the store. What about an extreme case for the precision of an allocator where an analysis allocates addresses specific to entire stores (or portions of stores, or specific addresses in the store). As it turns out, we can even recover all the precision lost through structurally store widening as a form of store-sensitive polyvariance.

We assume the underlying allocator (in a store-sensitive setting) is $\widetilde{alloc}$ and its instrumentation is $\overset{\text{Inst}}{\rightsquigarrow}$. Using these, we may produce an instrumentation for recovering store sensitivity within a structurally store widened parametric semantics by rebuilding the state-specific environments and stores lost due to store widening.

$$\overbrace{((ae_f\ ae_0\ \ldots\ ae_j),\ \tilde{\rho},\ \tilde{\sigma},\ (\tilde{\iota}, \tilde{\rho}_\Sigma, \tilde{\sigma}_\Sigma))}^{\tilde{\varsigma}}$$
$$\overset{\text{Inst}}{\rightsquigarrow}_{ss(\widetilde{alloc},\ \overset{\text{Inst}}{\rightsquigarrow})} (call',\ \tilde{\rho}',\ \tilde{\sigma}',\ (\tilde{\iota}', \tilde{\rho}'_\Sigma, \tilde{\sigma}'_\Sigma))$$

$$\begin{aligned}
\text{where}\quad & ((\lambda\ (x_0 \ldots x_j)\ \ call'), \tilde{\rho}_\lambda) \in \tilde{\mathcal{A}}(ae_f,\ \tilde{\varsigma}) \\
& \tilde{v}_i = \tilde{\mathcal{A}}(ae_i,\ \tilde{\varsigma}) \\
& \tilde{\rho}'_\Sigma = \tilde{\rho}_\lambda[x_i \mapsto \tilde{a}_i] \\
& \tilde{\sigma}'_\Sigma = \tilde{\sigma}_\Sigma \sqcup [\tilde{a}_i \mapsto \tilde{v}_i] \\
& \tilde{a}_i = \widetilde{alloc}(x_i, \tilde{\varsigma}) \\
& ((ae_f\ ae_0\ \ldots\ ae_j),\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota}) \overset{\text{Inst}}{\rightsquigarrow} (call', \tilde{\rho}', \tilde{\sigma}', \tilde{\iota}')
\end{aligned}$$

We then use an allocator which embeds these recovered exact environments and stores to differentiate addresses.

$$\widetilde{alloc}_{\text{ss}}(x,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ (\tilde{\iota},\ \tilde{\rho}_\Sigma,\ \tilde{\sigma}_\Sigma))) \triangleq (x,\ call,\ \tilde{\rho}_\Sigma,\ \tilde{\sigma}_\Sigma)$$

Using a similar instrumentation which rebuilds exact environments, we can also recover the full environment sensitivity lost through closure conversion, or the use of mCFA or poly-$k$-CFA.

$$\widetilde{alloc}_{\text{es}}(x,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ (\tilde{\iota},\ \tilde{\rho}_\Sigma))) \triangleq (x,\ call,\ \tilde{\rho}_\Sigma)$$

In this way we can observe that some important forms of coarser structural abstraction (store widening and the use of flat environments) are encompassed by our design space for polyvariance.

### 4.6.5 Combining Forms of Polyvariance

For two forms of polyvariance, we may combine them by essentially taking the product of their instrumentations and the product of their allocators. Consider two forms of polyvariance characterized by $\widetilde{alloc}_0$ paired with $(\overset{\text{Inst}}{\rightsquigarrow}_0)$ and $\widetilde{alloc}_1$ paired with $(\overset{\text{Inst}}{\rightsquigarrow}_1)$, respectively.

We can produce a new instrumentation which compiles the information added by both $(\overset{\text{Inst}}{\rightsquigarrow}_0)$ and $(\overset{\text{Inst}}{\rightsquigarrow}_1)$:

$$(call,\ \tilde{\rho},\ \tilde{\sigma},\ (\tilde{\iota}_0, \tilde{\iota}_1)) \overset{\text{Inst}}{\rightsquigarrow}_{\times} (call',\ \tilde{\rho}',\ \tilde{\sigma}',\ (\tilde{\iota}'_0, \tilde{\iota}'_1))$$

$$\begin{aligned} \text{where} \quad & (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota}_0) \overset{\text{Inst}}{\rightsquigarrow}_0 (call',\ \tilde{\rho}',\ \tilde{\sigma}',\ \tilde{\iota}'_0) \\ & (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota}_1) \overset{\text{Inst}}{\rightsquigarrow}_1 (call',\ \tilde{\rho}',\ \tilde{\sigma}',\ \tilde{\iota}'_1) \end{aligned}$$

Likewise, we can produce a new allocator which returns an address specific to both addresses returned by $\widetilde{alloc}_0$ and $\widetilde{alloc}_1$:

$$\begin{aligned} & \widetilde{alloc}_{\times}(x,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ (\tilde{\iota}_0, \tilde{\iota}_1))) \triangleq \\ & \quad (\widetilde{alloc}_0(x, (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota}_0)), \widetilde{alloc}_1(x, (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota}_1))) \end{aligned}$$

# CHAPTER 5

# PERFECT STACK PRECISION

In this chapter, we will explore the issue of call-stack precision, the precision of matching calls with their respective returns, applying the approach of this dissertation.

In Section 5.1, we extend our intermediate language to include both explicit call sites and return points. This direct-style language, a $\lambda$-calculus in administrative normal form, requires us to give special consideration to modeling the call stack and is used for the remainder of Chapter 5. We will see specifically how the analyses discussed so far may lose precision upon function return, even where polyvariance initially keeps values separate across the corresponding call.

In Section 5.2, we formalize an idealized static analysis which provides us with a definition and model for perfect stack precision. This unbounded-stack machine may be tuned to any approximation of program values (as discussed in Chapter 3) and to any style of polyvariance for these values (as discussed in Chapter 4), but cannot lose precision in its modeling of the structure of the call stack (*i.e.*, which frames are on the stack at a given moment—where values can be returned to).

In Section 5.3, we discuss the three previous attempts to produce a computable static analysis with perfect stack precision. One of these approaches is necessarily exponential-time and requires additional engineering effort for its initial construction and continued maintainence (Vardoulakis and Shivers, 2010). Another requires only a quadratic-factor increase to the worst-case running time of the underlying analysis, but likewise incurs a similar penalty in terms of human labor (Earl et al., 2010). A third requires a worse-than-quadratic, polynomial-factor increase in complexity class, but may be considered free in terms of its development cost, assuming an analysis which uses the store-allocated-continuations approach (Johnson and Van Horn, 2014).

In Section 5.4, we apply these lessons learned and the techniques of the previous chapters to obtain a general method for perfect stack precision which is both computationally free (*i.e.*, no increase in worst-case complexity class or, as our benchmarks would suggest, average

states explored) and developmentally free (*i.e.*, no additional labor or engineering complexity for analyses using the store-allocated-continuations approach). This is accomplished using an introspective continuation allocator that adapts to the function entry points reached under whatever value allocator is used in the underlying analysis (*i.e.*, one using the CPS framework of previous chapters). We will compare this to the approaches of Section 5.3, discuss the intuition for why it works, analyze its complexity, and give a detailed proof of its precision with respect to the unbounded-stack machine of Section 5.2.

## 5.1 Administrative Normal Form

To address the return-flow merging problem, we replace our CPS intermediate language with a direct-style (call-by-value, untyped) $\lambda$-calculus in the administrative normal form (ANF) of Flanagan et al. (1993). ANF unifies all direct-style continuations (*e.g.*, argument and condition continuations) a richer language might support, providing only a single `let` continuation. This greatly simplifies the language and is convenient for analysis as every intermediate expression is administratively `let`-bound and may be assigned a unique variable name. ANF differs from the monadic normal form of Moggi (1991) only in that it is unable to `let`-bind a `let`-form—nested `let`s must form a sequence and not a tree (Kennedy, 2007).

$$
\begin{array}{rll}
e \in \mathsf{Exp} ::= & \texttt{(let ([}x\ \texttt{(}f\ ae\texttt{)])}\ e\texttt{)} & \text{[call]} \\
\mid & ae & \text{[return]} \\
f, ae \in \mathsf{AExp} ::= & x \mid lam & \text{[atomic expressions]} \\
lam \in \mathsf{Lam} ::= & \texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e\texttt{)} & \text{[lambda abstractions]} \\
x, y \in \mathsf{Var} & \text{is a set of identifiers} & \text{[variables]}
\end{array}
$$

Conversion to ANF depends on an evaluation strategy and makes the order of operations explicit as a stack of `let`-forms. Additional core forms permitting mutation, recursive binding, conditional branching, tail calls, and primitive operations add complexity, but do not complicate the technique we aim to discuss in this chapter and so are left out.

We may view the abstract machine formalized in this section as strictly an extension on the static analysis of previous chapters. For this reason, its components and their domains wear tildes and we will refer to it as the *finite-state machine* (to differentiate from the unbounded-stack machine of Section 5.2). For the remainder of this chapter, finite states ($\tilde{\varsigma}$) will be elements of $\tilde{\Sigma}$ defined:

$$\begin{aligned}
\tilde{\varsigma} \in \tilde{\Sigma} &\triangleq \mathsf{Exp} \times \widetilde{Env} \times \widetilde{Store} && \text{[states]} \\
&\quad \times \widetilde{KStore} \times \widetilde{Addr} \\
\tilde{\rho} \in \widetilde{Env} &\triangleq \mathsf{Var} \rightharpoonup \widetilde{Addr} && \text{[environments]} \\
\tilde{\sigma} \in \widetilde{Store} &\triangleq \widetilde{Addr} \to \tilde{D} && \text{[stores]} \\
\tilde{d} \in \tilde{D} &\triangleq \mathcal{P}(\widetilde{Clo}) && \text{[flow-sets]} \\
\widetilde{clo} \in \widetilde{Clo} &\triangleq \mathsf{Lam} \times \widetilde{Env} && \text{[closures]} \\
\tilde{\sigma}_\kappa \in \widetilde{KStore} &\triangleq \widetilde{Addr} \to \tilde{K} && \text{[continuation stores]} \\
\tilde{k} \in \tilde{K} &\triangleq \mathcal{P}(\widetilde{Kont}) && \text{[kont-sets]} \\
\tilde{\kappa} \in \widetilde{Kont} &\triangleq \widetilde{Frame} \times \widetilde{Addr} && \text{[continuations]} \\
\tilde{\phi} \in \widetilde{Frame} &\triangleq \mathsf{Var} \times \mathsf{Exp} \times \widetilde{Env} && \text{[stack frame]} \\
\tilde{a}, \tilde{a}_\kappa \in \widetilde{Addr} &\text{ is a finite set} && \text{[addresses]}
\end{aligned}$$

Continuations, which were previously encoded as user-space lambdas, are now distinguished as a machine component $\tilde{\kappa}$. These are formed by a stack frame ($\tilde{\phi}$) and an address for its sub-continuation in turn ($\tilde{a}_\kappa$). A stack frame is made of a variable $x$ to bind and an expression $e$ to reinstate (together making up the $\lambda$-abstraction used in CPS), along with an environment $\tilde{\rho}$ over which the expression $e$ is closed. In CPS, this environment would have mapped a variable for $e$'s continuation to an address. Instead, our ANF machine distinguishes an address ($\tilde{a}_\kappa$), for the sub-continuation or stack tail, alongside $\tilde{\phi}$. Likewise, each machine state $\tilde{\varsigma}$ is unique to an address for the current continuation, distinguished from the rest of the environment. To maintain simplicity as we progress, the store is factored at the top level into a value store ($\tilde{\sigma}$) and a continuation store ($\tilde{\sigma}_\kappa$).

There were two fundamental sources of unboundedness in the concrete machine: the value store (with an infinite domain of addresses), and the current continuation (modeled as an unbounded list of stack frames). We bound the value store $\tilde{\sigma}$ by restricting its domain to a finite set of addresses $\tilde{a}$, but we permit a *set* of abstract closures $\widetilde{clo}$ at each. We finitize the stack similarly by threading it through the store as a linked list. A continuation is thus represented by an address. This address points to a *set* of topmost frames, each paired with the address of its continuation in turn (i.e., that stack's tail). We separate the continuation store $\tilde{\sigma}_\kappa$ from the value store $\tilde{\sigma}$ to maintain simplicity as we progress.

Abstract environments $\tilde{\rho}$ change only because our address set is now finite. Abstract closures $\widetilde{clo}$ are approximate only by virtue of their environments using these abstract addresses. For each such $\tilde{a}$, the finite value store $\tilde{\sigma}$ denotes a *flow set* $\tilde{d}$ of closures. At each

point, a continuation store $\tilde{\sigma}_\kappa$ has a set of continuations $\tilde{k}$. Like closures, each abstract frame $\tilde{\phi}$ is only approximate by virtue of its abstracted environment. An abstract continuation $\tilde{\kappa}$ pairs a frame with an address $\tilde{a}_\kappa$ for the stack underneath.

As before, we define a helper for abstract atomic evaluation $\tilde{\mathcal{A}}$:

$$\tilde{\mathcal{A}} : \mathsf{AExp} \times \widetilde{Env} \times \widetilde{Store} \rightharpoonup \tilde{D}$$

$$\tilde{\mathcal{A}}(x, \tilde{\rho}, \tilde{\sigma}) \triangleq \tilde{\sigma}(\tilde{\rho}(x)) \qquad \text{[variable lookup]}$$

$$\tilde{\mathcal{A}}(lam, \tilde{\rho}, \tilde{\sigma}) \triangleq \{(lam, \tilde{\rho})\} \qquad \text{[closure creation]}$$

Note that atomic evaluation of a lambda expression new yields a set containing a single element for the closure of that lambda.

Because our address domain is now finite, multiple concrete allocations need to be represented by a single abstract address. There are a variety of sound strategies for doing this. Each strategy corresponds to a distinct style of analysis and is amenable to easy implementation by defining an auxiliary $\widetilde{alloc}$ helper to encapsulate these differences in behavior. Given the variable for which to allocate, the finite state performing the allocation, and the closures invoked, the abstract allocator returns an address:

$$\widetilde{alloc} : \mathsf{Var} \times \tilde{\Sigma} \to \widetilde{Addr}$$

One such behavior is to simply return the variable itself (as a 0-CFA would):

$$\widetilde{alloc}_0(x, \tilde{\varsigma}) \triangleq x$$

Using $\widetilde{alloc}_0$ would tune our finite-state semantics to the *monovariant* analysis style (also called zeroth-order CFA), a form of context-insensitive analysis. In a monovariant analysis, every closure that is bound to a variable $x$ at any point during a concrete execution ends up being represented in a single flow set when the analysis is complete.

Because we are also now store-allocating continuations and distinguishing a top-level continuation store, we likewise distinguish an abstract allocator specifically for addresses in this store:

$$\widetilde{alloc}_\kappa : \tilde{\Sigma} \times \mathsf{Exp} \times \widetilde{Env} \times \widetilde{Store} \to \widetilde{Addr}$$

A standard choice is to allocate based on the target expression:

$$\widetilde{alloc}_{\kappa\,0}((e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa), e', \tilde{\rho}', \tilde{\sigma}') \triangleq e'$$

We provide to this function all the information known about the transition being made. The value-store allocator is invoked before a successor $\tilde{\rho}'$ or $\tilde{\sigma}'$ is constructed. However, when

calling the continuation-store allocator, we provide information about the target state being transitioned to. The choice of $e'$ for allocating a continuation address makes sense considering the entry point of a function should know where it is returning. In fact, when performing an analysis of a continuation-passing-style (CPS) language, $e'$ also would naturally be the choice inherited from a monovariant value-store allocator (assuming an alpha-renaming such that every $x$ is unique to a single binding point).

We may now define a non-deterministic finite-state transition relation $(\rightsquigarrow_{\tilde{\Sigma}}) \subseteq \tilde{\Sigma} \times \tilde{\Sigma}$. Call sites transition as follows.

$$\overbrace{(\texttt{(let ([}y\ (f\ ae)\texttt{]) }e\texttt{)}, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa)}^{\tilde{\varsigma}} \rightsquigarrow_{\tilde{\Sigma}} (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa), \text{ where}$$

$$(\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e'\texttt{)}, \tilde{\rho}_\lambda) \in \tilde{\mathcal{A}}(f, \tilde{\rho}, \tilde{\sigma})$$

$$\tilde{\rho}' = \tilde{\rho}_\lambda[x \mapsto \tilde{a}]$$

$$\tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a} \mapsto \tilde{\mathcal{A}}(ae, \tilde{\rho}, \tilde{\sigma})]$$

$$\tilde{a} = \widetilde{alloc}(x, \tilde{\varsigma})$$

$$\tilde{\sigma}'_\kappa = \tilde{\sigma}_\kappa \sqcup [\tilde{a}'_\kappa \mapsto \{((x, e, \tilde{\rho}), \tilde{a}_\kappa)\}]$$

$$\tilde{a}'_\kappa = \widetilde{alloc}_\kappa(\tilde{\varsigma}, e', \tilde{\rho}', \tilde{\sigma}')$$

As $\tilde{\mathcal{A}}$ yields a set of abstract closures for $f$, a successor state is produced for each. Likewise, so each point in the store accumulates all closures bound at that abstract address $\tilde{a}$ and so we faithfully over-approximate all the addresses $a$ that $\tilde{a}$ simulates, we use a join operation when extending the store. The join of two stores distributes point-wise as follows.

$$\tilde{\sigma} \sqcup \tilde{\sigma}' \triangleq \lambda\tilde{a}.\ \tilde{\sigma}(\tilde{a}) \cup \tilde{\sigma}'(\tilde{a})$$

$$\tilde{\sigma}_\kappa \sqcup \tilde{\sigma}'_\kappa \triangleq \lambda\tilde{a}_\kappa.\ \tilde{\sigma}_\kappa(\tilde{a}_\kappa) \cup \tilde{\sigma}'_\kappa(\tilde{a}_\kappa)$$

Instead of generating a fresh address for $\tilde{a}$, we use our abstract allocation policy to select one. To instantiate a monovariant analysis like 0-CFA, this address is simply the syntactic variable $x$. Likewise, we generate an address for our continuation (a new stack frame atop the current continuation) and extend the continuation store.

The return transition is modified in the same way:

$$\overbrace{(ae, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa)}^{\tilde{\varsigma}} \rightsquigarrow_\Sigma^\sim (e, \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}_\kappa, \tilde{a}'_\kappa), \text{ where}$$
$$((x, e, \tilde{\rho}_\kappa), \tilde{a}'_\kappa) \in \tilde{\sigma}_\kappa(\tilde{a}_\kappa)$$
$$\tilde{\rho}' = \tilde{\rho}_\kappa[x \mapsto \tilde{a}]$$
$$\tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a} \mapsto \tilde{\mathcal{A}}(ae, \tilde{\rho}, \tilde{\sigma})]$$
$$\tilde{a} = \widetilde{alloc}(x, \tilde{\varsigma})$$

Where multiple topmost stack frames are pointed to by $\tilde{a}_\kappa$, this transition yields multiple successors. An updated environment and store are produced as before, but the continuation store remains as it was. The current continuation $\tilde{a}'_\kappa$ reinstated in each successor is the address associated with each topmost stack frame.

To approximately evaluate a program according to these abstract semantics, we first define an abstract injection function, $\tilde{\mathcal{I}}$, where the stores begin as functions, $\bot$, that map every abstract address to the empty set.

$$\tilde{\mathcal{I}} : \mathsf{Exp} \to \tilde{\Sigma}$$
$$\tilde{\mathcal{I}}(e) \triangleq (e, \varnothing, \bot, \bot, \tilde{a}_{\text{halt}})$$

The address $\tilde{a}_{\text{halt}}$ can be any otherwise unused address that is never returned by the allocation function. Our machine will eventually be unable to transition into this continuation and will then produce no successors, which simulates the behavior of our concrete machine upon reaching an empty stack ($\epsilon$).

We again lift $(\rightsquigarrow_\Sigma^\sim)$ to obtain a collecting semantics $(\rightsquigarrow_S^\sim)$ defined over sets of states:

$$\tilde{s} \in \tilde{S} \triangleq \mathcal{P}(\tilde{\Sigma})$$
$$\tilde{s} \rightsquigarrow_S^\sim \tilde{s}' \triangleq \tilde{s}' = \{\tilde{\varsigma}' \mid \tilde{\varsigma} \in \tilde{s} \wedge \tilde{\varsigma} \rightsquigarrow_\Sigma^\sim \tilde{\varsigma}'\} \cup \{\tilde{\mathcal{I}}(e_0)\}$$

Our collecting relation $(\rightsquigarrow_S^\sim)$ is a monotonic, total function that gives a set including the trivially reachable finite-state $\tilde{\mathcal{I}}(e_0)$ plus the set of all states immediately succeeding those in its input.

Because $\tilde{\Sigma}$ is now finite, we know the approximate evaluation of even a non-terminating $e_0$ will terminate. That is, for some $n \in \mathbb{N}$, the value $(\rightsquigarrow_S^\sim)^n(\bot)$ is guaranteed to be a fixed point containing an approximation of $e_0$'s full program trace.

### 5.1.1 Return-Flow Imprecision

To illustrate the effect of an imprecise stack on data-flow and control-flow precision, we first define a more precise 1-call-sensitive (first-order, 1-CFA) allocator. A $k$-call-sensitive

analysis style differentiates bindings to a variable so they are unique to a history of the last $k$ call sites reached before the binding. A history of length $k = 1$ then allocates an address unique to the call site immediately preceeding the binding by using the following allocator.

$$\widetilde{alloc}_1(x, (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{\kappa})) \triangleq (x, e)$$

Now, using $\widetilde{alloc}_1$, consider the following snippet of code where the variable `id` is already bound to `(λ (x) `$^0$`x)`:

```
... 1(let ([y (id #t)])
      2(let ([z (id #f)])
         3...))
```

We number these expressions for ease of reference. For example, $e_2$ refers to the `let`-form that binds `z`, and $e_0$ to the return point of `id`. We assume the starting configuration for this example is $(e_1, \tilde{\rho}, \tilde{a}_\kappa)$ where $\tilde{\rho}$ and $\tilde{a}_\kappa$ are the binding environment and continuation address at the start of this code. We likewise let $\tilde{\rho}_\lambda$ be the environment of `id`'s closure.

The first call to `id` transitions to evaluate $e_0$ with the continuation address $e_0$. This transition reaches the configuration $(e_0, \tilde{\rho}_\lambda[\mathtt{x} \mapsto (\mathtt{x}, e_1)], e_0)$ and binds $(\mathtt{x}, e_1)$ to `#t` and the continuation address $e_0$ to the continuation $((\mathtt{y}, e_2, \tilde{\rho}), \tilde{a}_\kappa)$, which gives us the following stores:

$$\begin{aligned}\tilde{\sigma} &= \{(\mathtt{x}, e_1) \mapsto \{\mathtt{\#t}\}\} \\ \tilde{\sigma}_\kappa &= \{e_0 \mapsto \{((\mathtt{y}, e_2, \tilde{\rho}), \tilde{a}_\kappa)\}\}\end{aligned}$$

Next, `id` returns and transitions from $e_0$ to $e_2$, extending the continuation's environment to $\tilde{\rho}[\mathtt{y} \mapsto (\mathtt{y}, e_0)]$ and reinstating the continuation address $\tilde{a}_\kappa$. This yields a configuration $(e_2, \tilde{\rho}[\mathtt{y} \mapsto (\mathtt{y}, e_0)], \tilde{a}_\kappa)$. This transition binds $(\mathtt{y}, e_0)$ to `#t`, giving us the following stores:

$$\begin{aligned}\tilde{\sigma} &= \{(\mathtt{x}, e_1) \mapsto \{\mathtt{\#t}\}, \\ &\qquad (\mathtt{y}, e_0) \mapsto \{\mathtt{\#t}\}\} \\ \tilde{\sigma}_\kappa &= \{e_0 \mapsto \{((\mathtt{y}, e_2, \tilde{\rho}), \tilde{a}_\kappa)\}\}\end{aligned}$$

Then the second call to `id` transitions to evaluate $e_0$ with the continuation address $e_0$ once again (recall the definition of $\widetilde{alloc}_{\kappa\,0}$). This transition reaches the configuration

$(e_0, \tilde{\rho}_\lambda[\mathtt{x} \mapsto (\mathtt{x}, e_2)], e_0)$, binding $(\mathtt{x}, e_2)$ to `#f` and the continuation address $e_0$ to the continuation $((\mathtt{z}, e_3, \tilde{\rho}[\mathtt{y} \mapsto (\mathtt{y}, e_0)]), \tilde{a}_\kappa)$, giving us the following stores:

$$\begin{aligned}
\tilde{\sigma} &= \{(\mathtt{x}, e_1) \mapsto \{\mathtt{\#t}\}, \\
&\qquad (\mathtt{y}, e_0) \mapsto \{\mathtt{\#t}\}, \\
&\qquad (\mathtt{x}, e_2) \mapsto \{\mathtt{\#f}\}\} \\
\tilde{\sigma}_\kappa &= \{e_0 \quad \mapsto \{((\mathtt{y}, e_2, \tilde{\rho}), \tilde{a}_\kappa), \\
&\qquad\qquad ((\mathtt{z}, e_3, \tilde{\rho}\,[\mathtt{y} \mapsto (\mathtt{y}, e_0)]), \tilde{a}_\kappa)\}\}
\end{aligned}$$

Next, `id` returns and transitions from $e_0$ to $e_3$, reinstating the continuation address $\tilde{a}_\kappa$ and extending the continuation's environment to $\tilde{\rho}[\mathtt{y} \mapsto (\mathtt{y}, e_0)][\mathtt{z} \mapsto (\mathtt{z}, e_0)]$. Because $e_0$ is bound to two continuations, this transition binds $(\mathtt{z}, e_0)$ to `#f` while another spuriously binds $(\mathtt{y}, e_0)$ to `#f`, causing return-flow imprecision in the following stores:

$$\begin{aligned}
\tilde{\sigma} &= \{(\mathtt{x}, e_1) \mapsto \{\mathtt{\#t}\}, \\
&\qquad (\mathtt{x}, e_2) \mapsto \{\mathtt{\#f}\}, \\
&\qquad (\mathtt{y}, e_0) \mapsto \{\mathtt{\#t}, \mathtt{\#f}\}, \\
&\qquad (\mathtt{z}, e_0) \mapsto \{\mathtt{\#f}\}\}
\end{aligned}$$

$$\begin{aligned}
\tilde{\sigma}_\kappa &= \{e_0 \quad \mapsto \{((\mathtt{y}, e_2, \tilde{\rho}), \tilde{a}_\kappa), \\
&\qquad\qquad ((\mathtt{z}, e_3, \tilde{\rho}\,[\mathtt{y} \mapsto (\mathtt{y}, e_0)]), \tilde{a}_\kappa)\}\}
\end{aligned}$$

The address $(\mathtt{y}, e_0)$, representing `y` within $e_3$, maps to both `#t` and `#f`, even though no concrete execution binds `y` to `#f`. A similar pair of transitions from $(e_0, \tilde{\rho}_\lambda[\mathtt{x} \mapsto (\mathtt{x}, e_1)], e_0)$ (the second of which is prompted by a change in the global continuation store at the address $e_0$) cause the same conflation for `z`.

Clearly, one solution is to increase the context sensitivity of our continuation allocator. Consider a continuation allocator $\widetilde{alloc}_{\kappa\,1}$ that like $\widetilde{alloc}_1$ uses a single call site of context and allocates a continuation address $(e', e)$ formed from both the expression being transitioned to, $e'$, and the expresson being transitioned from, $e$. This results in no spurious merging at return points because continuations are kept as distinct as the 1-call-sensitive value-store addresses we allocate.

It seems reasonable from here to suspect that perfect stack precision could always be obtained through a sufficiently precise strategy for polyvariant continuation allocation. The

difficulty is in knowing how to obtain this in the general case given an arbitrary value-store allocation strategy. Given that CFA2 and PDCFA promise a fixed method for implementing perfect stack precision, albeit at significant engineering and run-time costs, can perfect stack precision be implemented as a *fixed* adaptive continuation allocator? In this chapter, we both answer this question in the affirmative and show that this leads us not only to a trivial implementation but to only a constant-factor increase in run-time complexity.

## 5.2 An Unbounded-Stack Abstract Machine

In the same manner as previous work on this topic, we formalize perfect stack precision using a static analysis that leaves the structure of stacks fully unabstracted. Each frame of this unbounded stack is itself abstract because its environment is abstract and references the abstracted value store. States and configurations, however, directly contain lists of such frames that are unbounded in length. Environments, closures, stack frames, flow sets, and value stores are otherwise abstracted in the same manner as the finite machine of chapter 3. To differentiate this from the machines we have seen so far, we call this an unbounded-stack machine. In this chapter, components unique to this machine wear hats:

$$\hat{\varsigma} \in \hat{\Sigma} \triangleq \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Kont} \qquad \text{[states]}$$

$$\hat{\rho} \in \widehat{Env} \triangleq \mathsf{Var} \rightharpoonup \widehat{Addr} \qquad \text{[environments]}$$

$$\hat{\sigma} \in \widehat{Store} \triangleq \widehat{Addr} \to \hat{D} \qquad \text{[stores]}$$

$$\hat{d} \in \hat{D} \triangleq \mathcal{P}(\widehat{Clo}) \qquad \text{[flow-sets]}$$

$$\widehat{clo} \in \widehat{Clo} \triangleq \mathsf{Lam} \times \widehat{Env} \qquad \text{[closures]}$$

$$\hat{\kappa} \in \widehat{Kont} \triangleq \widehat{Frame}^* \qquad \text{[unbounded stacks]}$$

$$\hat{\phi} \in \widehat{Frame} \triangleq \mathsf{Var} \times \mathsf{Exp} \times \widehat{Env} \qquad \text{[stack frames]}$$

$$\hat{a} \in \widehat{Addr} \text{ is a finite set} \qquad \text{[addresses]}$$

Our atomic-expression evaluator works just as before:

$$\hat{\mathcal{A}} : \mathsf{AExp} \times \widehat{Env} \times \widehat{Store} \rightharpoonup \hat{D}$$

$$\hat{\mathcal{A}}(x, \hat{\rho}, \hat{\sigma}) \triangleq \hat{\sigma}(\hat{\rho}(x)) \qquad \text{[variable lookup]}$$

$$\hat{\mathcal{A}}(lam, \hat{\rho}, \hat{\sigma}) \triangleq \{(lam, \hat{\rho})\} \qquad \text{[closure creation]}$$

As does a monovariant allocator:

$$\widehat{alloc} : \mathsf{Var} \times \hat{\Sigma} \to \widehat{Addr}$$

$$\widehat{alloc_0}(x, \hat{\varsigma}) \triangleq x$$

This may be tuned to any other allocation strategy as easily as before.

We now define a non-deterministic unbounded-stack-machine transition relation $(\hat{\rightsquigarrow}_\Sigma) \subseteq \hat{\Sigma} \times \hat{\Sigma}$ and a rule for call-site transitions:

$$\overbrace{(\texttt{(let ([}y\ \texttt{(}f\ \texttt{)])}\ e), \hat{\rho}, \hat{\sigma}, \hat{\kappa})}^{\hat{\varsigma}} \hat{\rightsquigarrow}_\Sigma (e', \hat{\rho}', \hat{\sigma}', \hat{\phi}{:}\hat{\kappa}), \text{ where}$$
$$\hat{\phi} = (y, e, \hat{\rho})$$
$$(\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e'\texttt{)}, \hat{\rho}_\lambda) \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma})$$
$$\hat{\rho}' = \hat{\rho}_\lambda[x \mapsto \hat{a}]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(, \hat{\rho}, \hat{\sigma})]$$
$$\hat{a} = \widehat{alloc}(x, \hat{\varsigma})$$

This is slightly simplified from its analogue in $(\tilde{\rightsquigarrow}_\Sigma)$. The definitions of $e'$, $\hat{\rho}'$, and $\hat{\sigma}'$ are effectively identical, but the continuation store and continuation address have been replaced with an unbounded stack $\hat{\phi}{:}\hat{\kappa}$.

Likewise, the return transition also changes to the following.

$$\overbrace{(, \hat{\rho}, \hat{\sigma}, \hat{\phi}{:}\hat{\kappa})}^{\hat{\varsigma}} \hat{\rightsquigarrow}_\Sigma (e, \hat{\rho}', \hat{\sigma}', \hat{\kappa}), \text{ where}$$
$$\hat{\phi} = (x, e, \hat{\rho}_\kappa)$$
$$\hat{\rho}' = \hat{\rho}_\kappa[x \mapsto \hat{a}]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathcal{A}}(, \hat{\rho}, \hat{\sigma})]$$
$$\hat{a} = \widehat{alloc}(x, \hat{\varsigma})$$

To follow a return transition, the stack must contain at least one frame. Then the appropriate $e$ is reinstated with the environment $\hat{\rho}$ extended with an address for $x$. The store is extended and whatever stack tail existed after $\hat{\phi}$ is the successor's continuation $\hat{\kappa}$.

Unbounded-state injection is defined as we would expect:

$$\hat{\mathcal{I}} : \mathsf{Exp} \to \hat{\Sigma}$$

$$\hat{\mathcal{I}}(e) \triangleq (e, \varnothing, \bot, \epsilon)$$

As before, we lift $(\rightsquigarrow_{\hat{\Sigma}})$ to obtain a monotonic naïve collecting relation $(\rightsquigarrow_{\hat{s}})$ for a program $e_0$ that is defined over sets of unbounded-states:

$$\hat{s} \in \hat{S} \triangleq \mathcal{P}(\hat{\Sigma})$$

$$\hat{s} \rightsquigarrow_{\hat{s}} \hat{s}' \triangleq \hat{s}' = \{\hat{\varsigma}' \mid \hat{\varsigma} \in \hat{s} \wedge \hat{\varsigma} \rightsquigarrow_{\hat{\Sigma}} \hat{\varsigma}'\} \cup \{\hat{\mathcal{I}}(e_0)\}$$

This analysis is approximate but remains incomputable because the stack can grow without bound. Put another way, the height of the lattice $(\hat{S}, \cup, \cap)$ is infinite and so no finite number of $(\rightsquigarrow_{\hat{s}})$-iterations is guaranteed to obtain a fixed point.

### 5.2.1 Store-Widened Unbounded-Stack Analysis

As we will be comparing this unbounded-stack analysis to our new technique using precise store-allocated continuations, we derive a global-store-widened version as before:

$$\hat{\xi} \in \hat{\Xi} \triangleq \hat{R} \times \widehat{Store} \qquad \text{[state-spaces]}$$

$$\hat{r} \in \hat{R} \triangleq \mathcal{P}(\hat{C}) \qquad \text{[reachable configs.]}$$

$$\hat{c} \in \hat{C} \triangleq \mathsf{Exp} \times \widehat{Env} \times \widehat{Kont} \qquad \text{[configurations]}$$

A widened transfer function $(\rightsquigarrow_{\hat{\Xi}})$ is defined in terms of $(\rightsquigarrow_{\hat{\Sigma}})$ in exactly the same manner as $(\rightsquigarrow_{\tilde{\Xi}})$ was derived from $(\rightsquigarrow_{\tilde{\Sigma}})$ except that we now have only a single global value store and no continuation store:

$$(\rightsquigarrow_{\hat{\Xi}}) : \hat{\Xi} \to \hat{\Xi}$$

$$\begin{aligned}
&(\hat{r}, \hat{\sigma}) \rightsquigarrow_{\hat{\Xi}} (\hat{r}', \hat{\sigma}'), \text{ where} \\
&\quad \hat{s} = \{\hat{\varsigma} \mid (e, \hat{\rho}, \hat{\kappa}) \in \hat{r} \wedge (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \rightsquigarrow_{\hat{\Sigma}} \hat{\varsigma}\} \cup \{\hat{\mathcal{I}}(e_0)\} \\
&\quad \hat{r}' = \{(e, \hat{\rho}, \hat{\kappa}) \mid (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \in \hat{s}\} \\
&\quad \hat{\sigma}' = \bigsqcup_{(_,_,\hat{\sigma}'',_) \in \hat{s}} \hat{\sigma}''
\end{aligned}$$

## 5.3 Computable Approaches

### 5.3.1 Pushdown Control-Flow Analysis

Pushdown control-flow analysis (PDCFA) is a strategy for creating a computable equivalent to the precision of our unbounded-stack machine at a quadratic-factor increase to

the complexity class of the underlying finite analysis (*e.g.*, monovariant or 1-call-sensitive) Earl et al. (2010). This strategy tracks both reachable states (or in the store-widened case, configurations) as well as push or pop edges between them. A quadratic blow up comes from the fact that each pair of reachable states may have an explicitly-tracked edge between them. These edges implicitly represent, as possible paths through the graph, the stacks explicitly represented in the unbounded-stack machine. This graph precisely describes the regular expression of all stacks reachable in the pushdown states of the unbounded-stack analysis.

PDCFA formalizes a *Dyck state graph* for this. Where a sequence of pushes may be repeated *ad infinitum*, a Dyck state graph explicitly represents a cycle of push edges and a cycle of pop edges finitely. Broadly speaking, this is also how AAC and our adaptive continuation allocator works, except that such cycles are represented in the store instead of the state graph. A Dyck state graph is a state transition graph where each edge is annotated with either a frame push, a frame pop, or an epsilon. The set of continuations for a particular state in a Dyck state graph is determined by the pushes and pops along the paths that reach that state.

To formalize these Dyck state graphs, we reuse some components of our unbounded-stack machine, continuing to use hats as these machines are closely related:

$$\hat{g} \in \hat{G} \triangleq \hat{V} \times \hat{E} \times \widetilde{Store} \qquad \text{[Dyck graph]}$$

$$\hat{v} \in \hat{V} \triangleq \mathcal{P}(Q) \qquad \text{[Dyck vertices]}$$

$$\hat{q} \in \hat{Q} \triangleq \mathsf{Exp} \times \widetilde{Env} \qquad \text{[Dyck configs.]}$$

$$\hat{e} \in \hat{E} \triangleq \mathcal{P}(\hat{Q} \times \widetilde{Frame}_{\pm} \times \hat{Q}) \qquad \text{[Dyck edges]}$$

$$\hat{\phi}_{\pm} \in \widetilde{Frame}_{\pm} \triangleq \widetilde{Frame} \times \{\mathbf{push}, \mathbf{pop}\} \qquad \text{[edge actions]}$$

For readability, we style an edge $(\hat{q}, (\hat{\phi}, \mathbf{push}), \hat{q}') \in \hat{e}$ like so:

$$\hat{q} \xrightarrow{\hat{\phi}^+} \hat{q}' \in \hat{e}$$

It would be too verbose to formalize all the machinery required to compute a valid Dyck state graph. Instead, we define it from a completed unbounded-stack analysis $\hat{\xi}$. The function $\mathcal{DSG} : \hat{\Xi} \to \hat{G}$ produces a Dyck state graph from a fixed-point $\hat{\xi}$ for $(\rightsquigarrow_{\hat{\Xi}})$. The graph $\hat{g} = \mathcal{DSG}(\hat{\xi})$ is a valid Dyck state graph analysis for a program $e_0$ when $\hat{\xi}$ is the unbounded-stack analysis of $e_0$.

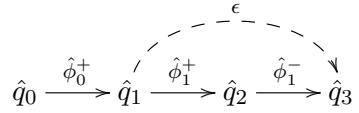
$$\mathcal{DSG}(\overbrace{(\hat{r}, \hat{\sigma})}^{\hat{\xi}}) \triangleq \overbrace{(\hat{v}, \hat{e}, \hat{\sigma})}^{\hat{g}}, \text{where}$$

$$
\begin{aligned}
\hat{v} &= \{(e, \hat{\rho}) \mid (e, \hat{\rho}, \hat{\kappa}) \in \hat{r}\} \\
\hat{e} &= \{(e, \hat{\rho}) \xrightarrow{\hat{\phi}_+} (e', \hat{\rho}') \;\mid\; (e, \hat{\rho}, \hat{\kappa}) \in \hat{r} \\
&\qquad\qquad \wedge\, (e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \rightsquigarrow_{\hat{\Sigma}} (e', \hat{\rho}', \hat{\sigma}, \hat{\phi} : \hat{\kappa})\} \\
&\quad \cup \{(e, \hat{\rho}) \xrightarrow{\hat{\phi}_-} (e', \hat{\rho}') \;\mid\; (e, \hat{\rho}, \hat{\kappa}) \in \hat{r} \\
&\qquad\qquad \wedge\, (e, \hat{\rho}, \hat{\sigma}, \hat{\phi} : \hat{\kappa}) \rightsquigarrow_{\hat{\Sigma}} (e', \hat{\rho}', \hat{\sigma}, \hat{\kappa})\}
\end{aligned}
$$

Although we do not formalize transition relations for Dyck state graphs themselves, it will be helpful for us to illustrate the major source of additional complexity in engineering a PDCFA directly. In the finite-state analysis, a transition is able to trivially compute a set of stacks by looking up the current continuation address in the continuation store. In the unbounded-stack analysis, a transition is able to trivially compute the stack by looking at the final component of the state or configuration being transitioned. In a Dyck state graph, canceling sequences of pushes and pops may place the set of topmost stack frames on edges arbitrarily distant from the configuration $\hat{q}$ being transitioned. In this way, the implicitness of stacks in a Dyck state graph obfuscates one of the most common operations needed to compute the analysis (*i.e.*, stack introspection). As an example, observe how the topmost stack frame $\hat{\phi}_0$ for $\hat{q}_3$ is located elsewhere in the graph:

$$\hat{q}_0 \xrightarrow{\hat{\phi}_0^+} \hat{q}_1 \xrightarrow{\hat{\phi}_1^+} \hat{q}_2 \xrightarrow{\hat{\phi}_1^-} \hat{q}_3$$

PDCFA therefore requires a non-trivial algorithm for stack introspection (Earl et al., 2012) and extra analysis machinery overall. Specifically, PDCFA requires the inductive maintenance of an *epsilon closure graph* in addition to the Dyck state graph as seen in the following.




This structure makes all sequences of canceling stack actions explicit as an epsilon edge. As we will see, this epsilon closure graph represents unnecessary additional complexity for both computer and analysis developer.

### 5.3.2 Abstracting Abstract Control

Abstracting abstract control (AAC) (Johnson and Van Horn, 2014) is another polynomial-time method for obtaining perfect stack precision. This technique works by store-allocating continuations using addresses unique enough to ensure no spurious merging and like PDCFA

does not require foreknowledge of the polyvariance or context sensitivity being used in the value store. The method is worse than PDCFA's quadratic-factor increase in run-time complexity. In the monovariant and store-widened case, its authors believe it to be in $O(n^8)$ (Johnson, 2015). However, AAC makes perfect stack precision available *for free* in terms of development cost (*i.e.*, labor).

Given the parametric semantics we built up in Chapter 4, we can define AAC's essential strategy in a single line:

$$\widetilde{alloc}_{\kappa\ \mathrm{AAC}}((e, \tilde{\rho}, \tilde{\sigma}, \tilde{\kappa}), e', \tilde{\rho}', \tilde{\sigma}') \triangleq (e', \tilde{\rho}', e, \tilde{\rho}, \tilde{\sigma})$$

That is, continuations are stored at an address unique to the target state's expression $e'$ and environment $\tilde{\rho}'$ as well as the source state's expression $e$, environment $\tilde{\rho}$, and store $\tilde{\sigma}$.

We have simplified AAC slightly and translated its notation to give this definition in the terms of our framework. A more faithful presentation of AAC shows fundamental differences between their framework and ours. AAC uses an eval-apply semantics and explodes each flow set into a set of distinct states across every application. The exact address AAC proposes using is $((\texttt{(}\lambda\ \texttt{(}y\texttt{)}\ e'\texttt{)}, \tilde{\rho}_\lambda), \widetilde{clo}, \tilde{\sigma})$ (Figure 7 in Johnson and Van Horn (2014)) where $(\texttt{(}\lambda\ \texttt{(}y\texttt{)}\ e'\texttt{)}, \tilde{\rho}_\lambda)$ is the target closure of an application, $\widetilde{clo}$ is one particular abstract closure flowing to $y$, and $\tilde{\sigma}$ is the value store in the source state. Our components $e'$ and $\tilde{\rho}'$ are isomorphic to the target closure because $e'$ is identical and $\tilde{\rho}'$ is produced from the combination of $\tilde{\rho}_\lambda$ and $y$. The source state's components ($e$, $\tilde{\rho}$, $\tilde{\sigma}$) are not as specific as $\widetilde{clo}$ and $\tilde{\sigma}$, but they do uniquely determine a flow set $\tilde{d}$ (the result of $\tilde{\mathcal{A}}$ invoked on $f$) that contains $\widetilde{clo}$. However, a semantics using an eval-apply factoring like AAC is needed to obtain a unique continuation address for every closure propagated across an application. This would have significantly complicated our presentation of the finite-state analysis, and in section 5.4 we will see that being specific to $\widetilde{clo}$ adds run-time complexity to an analysis without adding any precision.

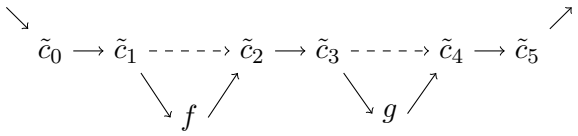
The intuition for AAC is that by allocating continuations specific to both the source state and target state of a call-site transition, no merging may occur when returning according to this (transition-specific) continuation-address. If we were to add some arbitrary additional context sensitivity (*e.g.*, 3-call-sensitivity), this information would be encoded in $\tilde{\rho}'$ and inherited by $\widetilde{alloc}_{\kappa\ \mathrm{AAC}}$ upon producing an address. Including this target-state binding environment in continuation addresses is the key reason why AAC allocates precise continuation addresses.

In Section 5.4, we will see that only the target state's expression $e'$ and environment $\tilde{\rho}'$ are truly necessary for obtaining the perfect stack precision of our unbounded-stack machine.

Including components of the transition's source state, its store, or its flow set only adds runtime complexity that is unnecessary for achieving perfect stack precision. This optimization extends AAC's core insight to be computationally *for free* while remaining precise and developmentally *for free*.

## 5.4 Perfect Stack Precision for Free

The primary intuition of our work can be illustrated by considering a set of intraprocedural configurations for some function invocation as in the following with $\tilde{c}_0$ through $\tilde{c}_5$.




The configuration $\tilde{c}_0$ represents the entry point to the function, and its incoming edge is a call-site transition. The configuration $\tilde{c}_5$ represents an exit point for the function, and its outgoing edge is a return-point transition. A transition where one intraprocedural configuration follows another, like $\tilde{c}_0 \to \tilde{c}_1$, is not technically possible in our restricted ANF language, but in more general languages would be. The function's body may call other functions $f$ and $g$ whose configurations are not a part of the same intraprocedural set of nodes. The primary insight behind our technique is that *a set of intraprocedural configurations (like $\tilde{c}_0$ through $\tilde{c}_5$) necessarily share the exact same set of genuine continuations* (in this example, the incoming call-sites for $\tilde{c}_0$).

### 5.4.1 An Introspective Entry-Point Allocator

We call the set of configurations $\tilde{c}_0$ through $\tilde{c}_5$ an *intraprocedural group* because they are those configurations that represent the body of a function for a single abstract invocation—defined by an entry point unique to some $e$ and $\tilde{\rho}$. Our central insight is to notice that this idea of an intraprocedural group also corresponds to those configurations that share a single set of continuations. Our finite-state machine represents this set of continuations with a continuation address, so if this continuation address is precise enough to uniquely determine an intraprocedural group's entry point ($e$ and $\tilde{\rho}$), then it can be used for all configurations in that same group. Thus, our allocator may be defined as simply:

$$\widetilde{alloc}_{\kappa\ \text{P4F}}((e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa), e', \tilde{\rho}', \tilde{\sigma}') = (e', \tilde{\rho}')$$

The impact of this change is easily missed, belied by its simplicity. We allocate a continuation based only on the expression and environment at the entry point of each intraprocedural

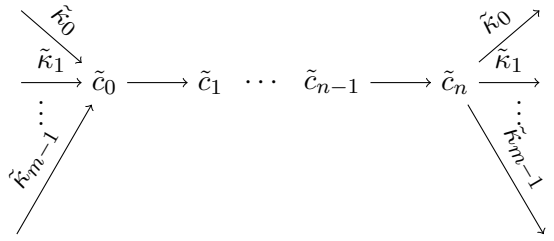
sequence of `let`-forms and it is precisely reinstated when each of the calls in these `let`-forms return.

Recall that the monovariant continuation allocator in our example from Section 5.1.1 resulted in return-flow merging because a single continuation address was being used for transitions to multiple entry points of different intraprocedural groups. More generally, return-flow merging occurs in a finite-state analysis when, at some return-point configuration $(ae, \tilde{\rho}_{ae}, \tilde{a}_\kappa)$, the set of continuations for $\tilde{a}_\kappa$ is less precise than the set of source configurations that transition to the entry point $(e, \tilde{\rho})$ of the same intraprocedural group. Because we allocate a continuation address specific to this exact entry point, and because that address is propagated by shallowly copying it to each return point for the same intraprocedural group, the set of continuations will be as precise as the set of source configurations transitioning to the same entry point in all cases. This means the return-flow merging problem cannot occur when using $\widetilde{alloc}_{\kappa\ \text{P4F}}$ and neither is there a run-time overhead for stack introspection.

In Section 5.4.5, we formalize these intuitions and provide a proof that our unbounded-stack analysis simulates (*i.e.*, is no more precise than) a finite-state analysis when using $\widetilde{alloc}_{\kappa\ \text{P4F}}$.

### 5.4.2 Analysis of Complexity

To see why this allocation scheme leads to only a constant-factor overhead, consider a set of configurations $\tilde{c}_0, \tilde{c}_1, \cdots, \tilde{c}_n$ that form an intraprocedural group and a set of call sites transitioning to $\tilde{c}_0$ with the continuations $\tilde{\kappa}_0, \tilde{\kappa}_1, \cdots, \tilde{\kappa}_{m-1}$. We can diagrammatically visualize this as the following.




Note that, for each call site, there is a corresponding return flow using the same continuation. Our allocation strategy means that all of the configurations $\tilde{c}_0, \tilde{c}_1, \cdots, \tilde{c}_n$ use the same continuation address $(e, \tilde{\rho})$. The global continuation store then maps this address to the set $\{\tilde{\kappa}_0, \tilde{\kappa}_1, \cdots, \tilde{\kappa}_{m-1}\}$.

Now consider what must be done if a new call site transitions to $c_0$. First, the continuation store must be extended to contain the continuation for this new call site, say $\tilde{\kappa}_m$, in the

continuation set at the address $(e, \tilde{\rho})$. Then the corresponding return edge transitions must be added. Note that none of $\tilde{c}_0, \tilde{c}_1, \cdots, \tilde{c}_{n-1}$ need to be modified or accessed. The only work done here beyond that of the underlying analysis is the extension of the continuation store by adding $\tilde{\kappa}_m$ at $(e, \tilde{\rho})$ and the addition of a corresponding return edge at return points. Thus, the additional work is a constant factor of the number of times a continuation is added to the continuation store.

A naïve analysis might lead us to conclude that this is bounded by the product of the number of continuation addresses and the number of continuations. However, there is a tighter bound. Each transition adds only one continuation to the continuation store. Thus, the work done is a constant factor of the number of transitions in the underlying analysis.

Note that this differs from AAC, which may make duplicate copies of the continuation set for an intraprocedural group as it produces one for each combination of components $e$, $\tilde{\rho}$, and $\tilde{\sigma}$ drawn from the source states transitioning to it. As a consequence, AAC allocates addresses strictly more unique than the target $(e', \tilde{\rho}')$ configuration. Two different source expressions $e_0$ and $e_1$ may both have transitions to $(e', \tilde{\rho}')$, but AAC will produce two different target configurations $\tilde{c}'_0$ and $\tilde{c}'_1$ because the continuation addresses they allocate will be distinct. This difference is maintained through the two variants of the function starting at $e'$ with environment $\tilde{\rho}'$, and when an exit point $ae$ is reached for each, the expression and its environment are the same and propagate the same values to two sets of continuations. Thus, these continuation addresses and the sets of stacks they represent are kept separate without any benefit.

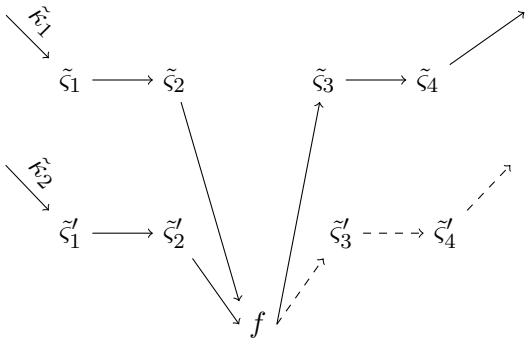
PDCFA, on the other hand, is more complex for an entirely different reason: the epsilon closure graph. Without the epsilon closure graph, PDCFA has no way to efficiently determine a topmost stack frame at each return transition. Both our method and AAC's method make this trivial by propagating an address explicitly to each state. While our method allows a continuation address to be shallowly propagated across each intraprocedural node in a function, the epsilon closure graph recomputes a separate set of incoming epsilon edges for every node. This means that the number of such edges for a given entry point $(e, \hat{\rho})$ is the number of callers times the number of intraprocedural nodes. This is a quadratic blow up from the number of nodes in a finite-state model. This is why monovariant store-widened PDCFA is in $O(n^6)$ instead of in $O(n^3)$ like traditional 0-CFA. We are able to naturally exploit our insight that each intraprocedural node following an entry point $(e, \hat{\rho})$ shares the same set of continuations (*i.e.*, the same epsilon edges) by propagating a pointer to this set instead of rebuilding it for each node. PDCFA is unable to exploit this insight

without adding machinery to propagate only a shallow copy of an incoming epsilon edge set intraprocedurally. It is likely that this insight could also be imported into the PDCFA style of analysis to yield a variant of PDCFA that incurs only a constant-factor overhead, but this would require additional machinery.

### 5.4.3 Constant Overhead Requires Store Widening

That no function can have two entry points that lead to the same exit point is a genuine restriction worth discussing further. If this were not true, our technique would be precise (assuming multiple entry points are not merged), but it would not necessarily be a constant-factor increase in complexity. The combination of no store widening (per-state value stores) and mutation is a good example of how this situation could arise.

To see how per-state stores can cause a further blow up in complexity, consider a function that is called with two different continuations and two different stores. Without store widening, each store causes a different state to be created for the entry point $(e, \tilde{\rho})$ of the function. In the following diagram, for example, $\tilde{\varsigma}_1$ is the state for the entry point with one store and $\tilde{\varsigma}'_1$ the state for the entry point with another store.




Now suppose that along both sequences of states, there is a call to some function $f$ and that $f$ contains a side effect that causes the previously different value stores to become equal. For example, in $\tilde{\varsigma}_1$ perhaps the address for $x$ maps to $\{\texttt{\#t}\}$ and in $\tilde{\varsigma}'_1$ it maps to $\{\texttt{\#f}\}$. If $x$ becomes bound to $\{\texttt{\#t}, \texttt{\#f}\}$ along both paths in the body of $f$, the stores along both paths would become identical.

A problem now arises. Should $f$ return only to one state using this common store such as $\tilde{\varsigma}_3$ or should it return to two different states (with identical stores) such as both $\tilde{\varsigma}_3$ and $\tilde{\varsigma}'_3$? Either choice has drawbacks. The semantics we have given would naturally yield the latter option, producing two distinct states that differ only by their continuation addresses (their original entry point). Because these states are otherwise identical, splitting $\tilde{\kappa}_1$ and $\tilde{\kappa}_2$

into sets represented by two different continuation addresses results in additional transitions and complexity without any benefit. Arguably, these continuation sets should be merged and represented by a single address. This corresponds to the former option and could save on run-time complexity but only at the cost of additional analysis machinery. This means per-state stores are incompatible with our goal of obtaining perfect stack precision *for free* in both senses (running time and human labor).

### 5.4.4 Implementation

We have implemented both our technique and AAC's technique for analysis of a simplified Scheme intermediate language. This language extends ANFExp with a variety of additional core forms including conditionals, mutation, recursive binding, tail calls, and a library of primitive operations. Our implementation was written in Scala and executed using Scala 2.11 for OSX on an Intel Core i5 (1.3 GHz) with 4GB of RAM. It is built upon the implementation of Earl et al. (2012), which implements both traditional $k$-CFA and PDCFA. The test cases we ran came from the Larceny R6RS benchmark suite (ack, cpstak, tak) and examples compiled from the previous literature on obtaining perfect stack precision (mj09, eta, kcfa2, kcfa3, blur, loop2, sat). As a sanity check, we have verified that both AAC and our method produce results of equivalent precision in every case. We ran each comparison using both a monovariant value-store allocator (Figures 5.1, 5.2, and 5.3), and a 1-call-sensitive polyvariant allocator (Figure 5.4, 5.5, and 5.6). Across the board, our method requires visiting strictly fewer machine configurations. In some of these cases, the difference is rather small, but in others, it is significant. We saw as much as a 16.0× improvement in the monovariant analysis and as much as a 10.4× improvement in the context-sensitive analysis. The mean speedup in terms of states visited was 5.4× and 4.9× in the monovariant and context-sensitive analyses, respectively.

### 5.4.5 Proof of Perfect Stack Precision

Proving soundness with respect to a concrete machine is fairly straightforward, as discussed in Chapter 3, but proving precision poses a greater challenge. To do this, we first define a simulation relation ($\sqsupseteq_\Xi$) where $\hat{\xi} \sqsupseteq_\Xi \tilde{\xi}$ (read as "$\hat{\xi}$ simulates $\tilde{\xi}$") if and only if all stored values and machine configurations in $\tilde{\xi}$ (including stacks implicit in this configuration) are accounted for in the unbounded-stack representation $\hat{\xi}$. Usually, the next step in such a proof would be to show that taking parallel steps preserves precision as in fallacy 4.

**Fallacy 4** (Steps preserve precision)**.** *If* $\hat{\xi} \rightsquigarrow_{\hat{\Sigma}} \hat{\xi}'$ *and* $\tilde{\xi} \rightsquigarrow_{\tilde{\Sigma}} \tilde{\xi}'$*, then* $\hat{\xi} \sqsupseteq_\Xi \tilde{\xi}$ *implies* $\hat{\xi}' \sqsupseteq_\Xi \tilde{\xi}'$*.*

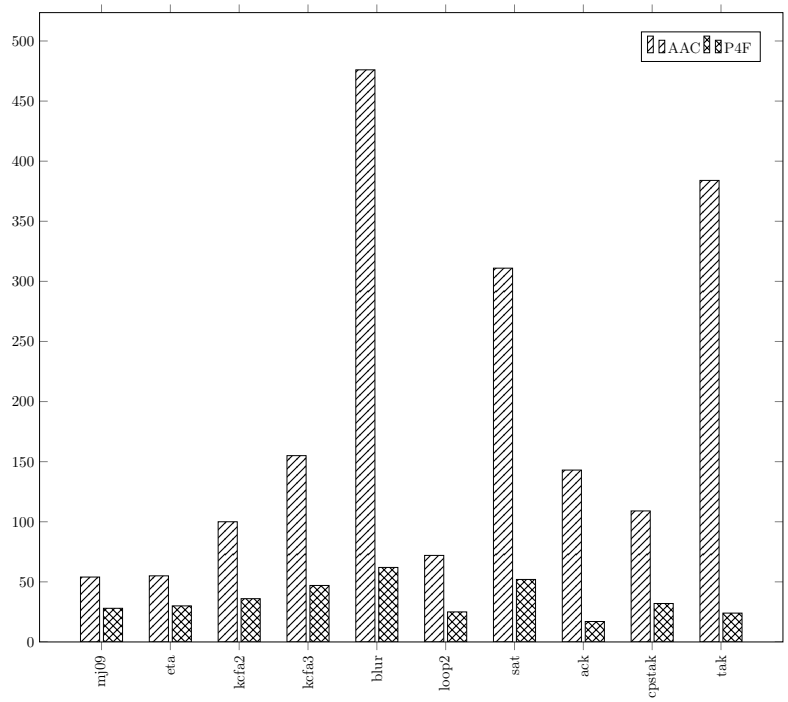



**Figure 5.1**. Benchmarks: A monovariant comparison with AAC in terms of states.

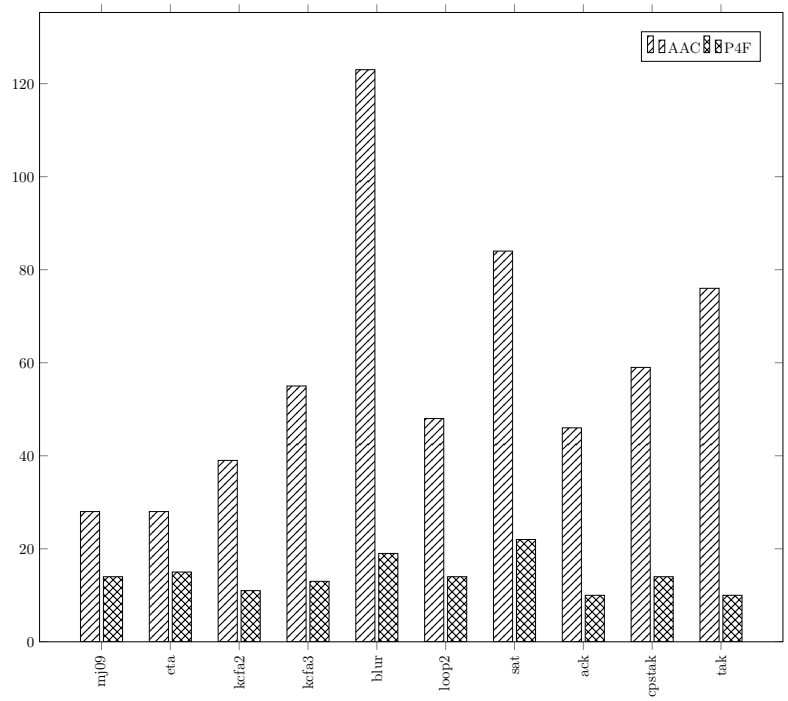



**Figure 5.2**. Benchmarks: A monovariant comparison with AAC in terms of configurations (states without stores).

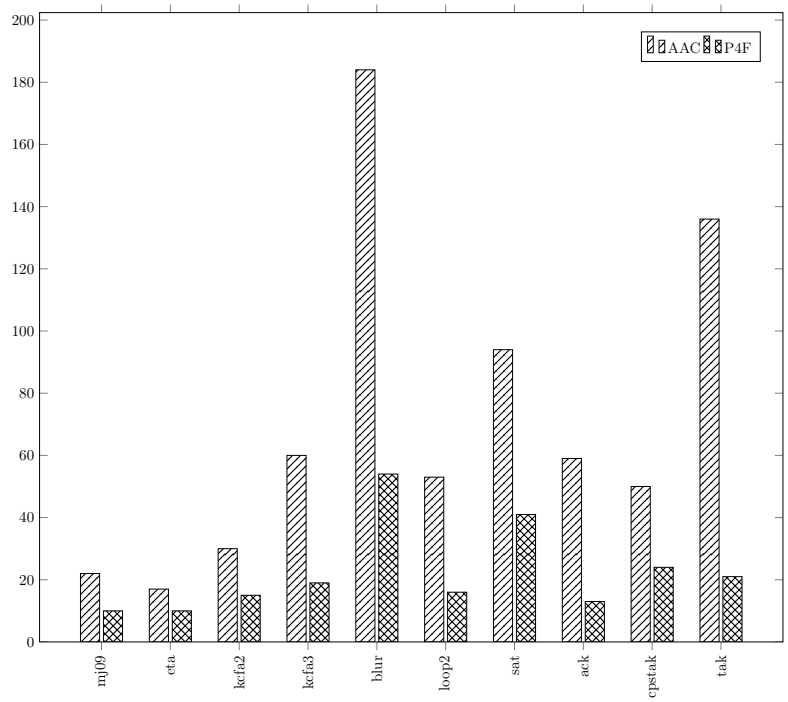



**Figure 5.3**. Benchmarks: A monovariant comparison with AAC in milliseconds of running time.

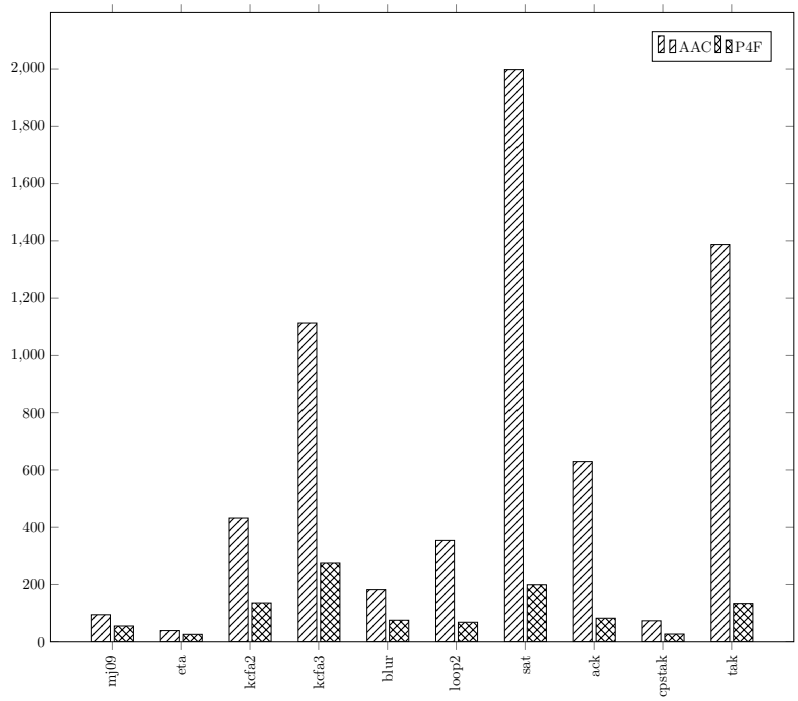



**Figure 5.4**. Benchmarks: A 1-call-sensitive comparison with AAC in terms of states.

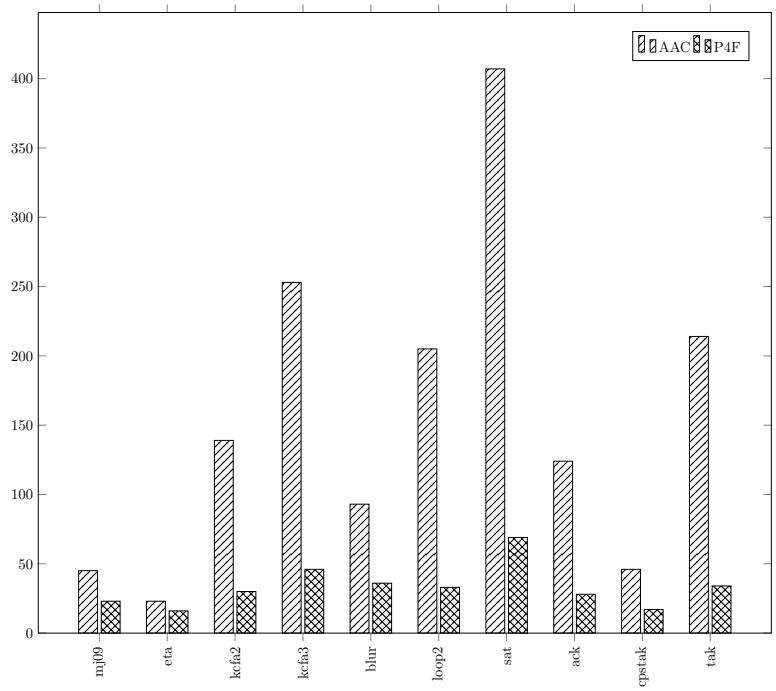



**Figure 5.5**. Benchmarks: A 1-call-sensitive comparison with AAC in terms of configurations (states without stores).

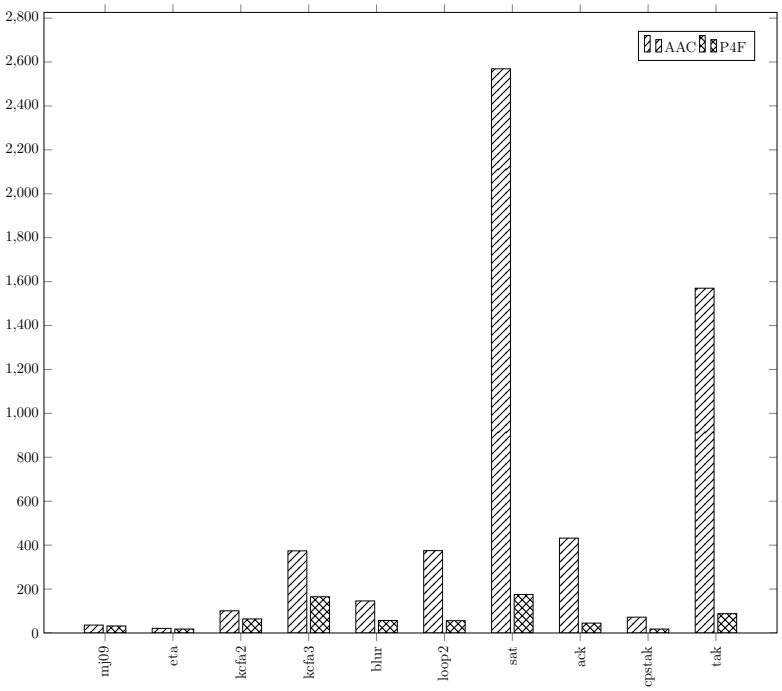



**Figure 5.6**. Benchmarks: A 1-call-sensitive comparison with AAC in milliseconds of running time.

However, Fallacy 4 is not true. This is because after some finite number of steps $\tilde{\xi}$ may contain a cycle in its continuation store. This means that an infinite family of successively longer stacks must also be in $\hat{\xi}$ for precision to hold. After a finite number of steps, however, all stacks in $\hat{\xi}$ are bounded by a finite length. Hence, there are stacks that precision says should be in $\hat{\xi}$ that are not.

We thus take a different approach to proving precision. Before going into the details, the high-level overview of this proof is as follows. Instead of stepping both $\hat{\xi}$ and $\tilde{\xi}$ in parallel, we show that successive steps of $\tilde{\xi}$ are all precise relative to any $\hat{\xi}$ that is already at a fixed point (*i.e.*, theorem 18 found at the end of this section). To show this, we need two inductions. One is over the steps taken by $\tilde{\xi}$, and the other is over the stacks implied by $\tilde{\xi}$. To separate these inductions, we define a well-formedness property (a binary predicate *wf*) that we can show is preserved by iterative steps from an initial $\tilde{\xi}_0$ (Lemma 12) and for which we can show that any well-formed $\tilde{\xi}$ is precise relative to any $\hat{\xi}$ that is at a fixed point (Lemmas 16 and 17).

The well-formedness property is defined in terms of two additional concepts. First, we formally define the stacks, $\tilde{\psi}$, implied by a continuation address, $\tilde{a}_\kappa$, and continuation store, $\tilde{\sigma}_\kappa$, in terms of a relation $\tilde{\psi} \in_\psi \tilde{a}_\kappa$ (via $\tilde{\sigma}_\kappa$) that we define. Then, we define paths, $(\hat{\hookrightarrow})$ and $(\tilde{\hookrightarrow})$, through $\hat{\xi}$ and $\tilde{\xi}$ in terms of a sequence of state steps, $(\hat{\rightsquigarrow}_\Sigma)$ and $(\tilde{\rightsquigarrow}_\Sigma)$, between states represented by configurations in $\hat{\xi}$ and $\tilde{\xi}$. This allows us to prove the precision of any well-formed $\tilde{\xi}$ (*i.e.*, Lemma 17) through a logical chain informally shown in Figure 5.7. In Lemma 13, we show that for any configuration $(e, \tilde{\rho}, \tilde{a}_\kappa)$ in the $\tilde{r}$ of a $\tilde{\xi}$ and any $\tilde{\psi}$ implied by $a_\kappa$ with the continuations store $\sigma_\kappa$ of $\tilde{\xi}$, there exists a path from the initial configuration $\tilde{c}_0$ with an empty stack $\epsilon$ to the configuration $(e, \tilde{\rho}, \tilde{a}_\kappa)$ with the implied stack $\tilde{\psi}$. In Lemma 14, we then show that there exists a corresponding path in $\hat{\xi}$ from $\hat{c}_0$ to $(e, H_{Env}(\tilde{\rho}), H_{Kont}(\tilde{\psi}))$. Finally, in Lemma 15, we show that the endpoint of that path is in $\hat{\xi}$ and thus $\tilde{\xi}$ is precise for $\hat{\xi}$ (Lemma 17).

$$
\begin{array}{ccc}
(\tilde{c}_0, \epsilon) \tilde{\hookrightarrow} ((e, \tilde{\rho}, \tilde{a}_\kappa), \tilde{\psi}) & \xleftarrow{\text{Lemma 13}} & \begin{array}{l}(e, \tilde{\rho}, \tilde{a}_\kappa) \in \tilde{r} \\ \tilde{\psi} \in_\psi \tilde{a}_\kappa \text{ (via } \tilde{\sigma}_\kappa)\end{array} \\
\Big\downarrow \text{Lemma 14} & & \Big\vdots \text{Lemma 17} \\
\hat{c}_0 \hat{\hookrightarrow} (e, H_{Env}(\tilde{\rho}), H_{Kont}(\tilde{\psi})) & \xrightarrow{\text{Lemma 15}} & (e, H_{Env}(\tilde{\rho}), H_{Kont}(\tilde{\psi})) \in \hat{r}
\end{array}
$$

**Figure 5.7**. The logical chain followed by our proof of perfect precision.

#### 5.4.5.1 Preliminaries

In order to prove precision, we first require that the address spaces for both $\hat{\sigma}$ and $\tilde{\sigma}$ correspond as follows.

**Assumption 5** (Address equivalence)**.** *There exists an equivalence* ($\equiv_{Addr}$) *between finite-state-machine addresses* ($\widehat{Addr}$) *and unbounded-stack-machine addresses* ($\widetilde{Addr}$) *that can be decomposed into a bijection* $\widehat{Addr} \underset{H_{Addr}}{\overset{T_{Addr}}{\rightleftharpoons}} \widetilde{Addr}$.

From this, we can define conversion bijections for most, both not all, components:

$$T_{Env}(\hat{\rho}) \triangleq \lambda x.T_{Addr}(\hat{\rho}(x)) \qquad H_{Env}(\tilde{\rho}) \triangleq \lambda x.H_{Addr}(\tilde{\rho}(x))$$

$$T_{Clo}((lam, \hat{\rho})) \triangleq (lam, T_{Env}(\hat{\rho})) \qquad H_{Clo}((lam, \tilde{\rho})) \triangleq (lam, H_{Env}(\tilde{\rho}))$$

$$T_{D}(\hat{d}) \triangleq \{T_{Clo}(\widehat{clo}) \mid \widehat{clo} \in \hat{d}\} \qquad H_{D}(\hat{d}) \triangleq \{H_{Clo}(\widetilde{clo}) \mid \widetilde{clo} \in \tilde{d}\}$$

$$T_{Store}(\hat{\sigma}) \triangleq \lambda \tilde{a}.T_{D}(\hat{\sigma}(H_{Addr}(\tilde{a}))) \qquad H_{Store}(\tilde{\sigma}) \triangleq \lambda \hat{a}.H_{D}(\tilde{\sigma}(T_{Addr}(\hat{a})))$$

$$T_{Frame}((x, e, \hat{\rho})) \triangleq (x, e, T_{Env}(\hat{\rho})) \qquad H_{Frame}((x, e, \tilde{\rho})) \triangleq (x, e, H_{Env}(\tilde{\rho}))$$

**Lemma 6** (Bijection)**.** *For all* $\hat{\tau}$ *and* $\tilde{\tau}$, $T_\tau$ *and* $H_\tau$ *form a bijection,* $\widehat{\tau} \underset{H_\tau}{\overset{T_\tau}{\rightleftharpoons}} \widetilde{\tau}$.

*Proof.* By unfolding and simplification. □

Likewise, we have equivalences ($\equiv_{Env}$, $\equiv_{Frame}$, $\equiv_{Clo}$) and precision relations ($\sqsupseteq_{\Xi}$, $\sqsupseteq_{Store}$, $\sqsupseteq_{R}$, $\sqsupseteq_{D}$, $\sqsupseteq_{Store}$) for all the components of our machine. These relations have the following signatures:

$$(\sqsupseteq_{\Xi}) \subseteq \hat{\Xi} \times \tilde{\Xi} \qquad \text{[state-space precision]}$$

$$(\sqsupseteq_{Store}) \subseteq \widehat{Store} \times \widetilde{Store} \qquad \text{[store precision]}$$

$$(\sqsupseteq_{R}) \subseteq \hat{R} \times \tilde{R} \times \widetilde{KStore} \qquad \text{[reachable configs. precision]}$$

$$(\equiv_{Env}) \subseteq \widehat{Env} \times \widetilde{Env} \qquad \text{[env. equivalence]}$$

$$(\equiv_{Frame}) \subseteq \widehat{Frame} \times \widetilde{Frame} \qquad \text{[frame equivalence]}$$

$$(\sqsupseteq_{D}) \subseteq \hat{D} \times \tilde{D} \qquad \text{[flow-set precision]}$$

$$(\equiv_{Clo}) \subseteq \widehat{Clo} \times \widetilde{Clo} \qquad \text{[closure equivalence]}$$

Note that ($\sqsupseteq_{R}$) is a trinary relation because finite-machine configurations are only meaningful in the context of a continuation store.

In addition, with the following assumption, we require that the value allocators respect the address correspondence.

**Assumption 7** (Allocation equivalence)**.** *If* $\hat{\rho} \equiv_{Env} \tilde{\rho}$*,* $\hat{\sigma} \sqsupseteq_{Store} \tilde{\sigma}$*, and* $\tilde{\psi} \in_{\psi} \tilde{a}_{\kappa}$ *(via* $\tilde{\sigma}_{\kappa}$*), then:*

$$\widehat{alloc}(x, (e, \hat{\rho}, \hat{\sigma}, H_{Kont}(\tilde{\psi}))) \equiv_{Addr} \widetilde{alloc}(x, (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_{\kappa}, \tilde{a}_{\kappa}))$$

This assumption uses $(\in_{\psi})$ and $H_{Kont}$ which deal with the stacks implied by an address and continuation store. We define an implicit stack as an unbounded list of finite-state continuations $\tilde{\kappa}$:

$$\tilde{\psi} \in \tilde{\Psi} = \widetilde{Kont}^{*} \qquad \text{[implicit stack]}$$

These $\tilde{\psi}$ are an intermediate representation in that, like $\hat{\kappa}$, their structure is unbounded, but the values in them are taken from the finite-state machine. We define a trinary relation $(\in_{\psi})$ that specifies which $\tilde{\psi}$ are implied by an $\tilde{a}_{\kappa}$ in $\tilde{\sigma}_{\kappa}$. This has the following base case:

$$\epsilon \in_{\psi} \tilde{a}_{\text{halt}} \text{ (via } \tilde{\sigma}_{\kappa})$$

and inductive case:

$$(\tilde{\phi}, \tilde{a}'_{\kappa}) \in \tilde{\sigma}_{\kappa}(\tilde{a}_{\kappa}) \wedge \tilde{\psi} \in_{\psi} \tilde{a}'_{\kappa} \text{ (via } \tilde{\sigma}_{\kappa}) \wedge \tilde{a}_{\kappa} \neq \tilde{a}_{\text{halt}} \implies ((\tilde{\phi}, \tilde{a}'_{\kappa}) : \tilde{\psi}) \in_{\psi} \tilde{a}_{\kappa} \text{ (via } \tilde{\sigma}_{\kappa})$$

Note that there were a few inversion properties here which needed to be formalized for mechanization. For example, the lefthand side of $(\in_{\psi})$ is $\epsilon$ if and only if the right-hand side is $\tilde{a}_{\text{halt}}$. Likewise, the converse of the inductive case also holds.

Then given such a $\tilde{\psi}$, we can directly construct its equivalent unbounded stack:

$$H_{Kont}(\epsilon) \triangleq \epsilon \qquad H_{Kont}((\tilde{\phi}, \tilde{a}_{\kappa}) : \tilde{\psi}) \triangleq H_{Frame}(\tilde{\phi}) : H_{Kont}(\tilde{\psi})$$

Also, given a finite-state configuration $\tilde{c}$ and an implicit stack $\tilde{\psi}$, we can construct an unbounded-stack configuration $\hat{c}$:

$$H_{C}((e, \tilde{\rho}, \tilde{a}_{\kappa}), \tilde{\psi}) \triangleq (e, H_{Env}(\tilde{\rho}), H_{Kont}(\tilde{\psi}))$$

Now we can define relations for comparing the precision of components across these machines:

$$\hat{a} \equiv_{Addr} \tilde{a} \triangleq \hat{a} = H_{Addr}(\tilde{a})$$

$$\hat{\rho} \equiv_{Env} \tilde{\rho} \triangleq dom(\hat{\rho}) = dom(\tilde{\rho}) \ \wedge \ \forall x \in dom(\tilde{\rho}).\ \hat{\rho}(x) \equiv_{Addr} \tilde{\rho}(x)$$

$$(lam, \hat{\rho}) \equiv_{Clo} (lam, \tilde{\rho}) \triangleq \hat{\rho} \equiv_{Env} \tilde{\rho}$$

$$(x, e, \hat{\rho}) \equiv_{Frame} (x, e, \tilde{\rho}) \triangleq \hat{\rho} \equiv_{Env} \tilde{\rho}$$

$$\hat{d} \sqsupseteq_{D} \tilde{d} \triangleq \forall \widehat{clo} \equiv_{Clo} \widetilde{clo}.\ \widetilde{clo} \in \tilde{d} \implies \widehat{clo} \in \hat{d}$$

$$\hat{\sigma} \sqsupseteq_{Store} \tilde{\sigma} \triangleq \forall \hat{a} \equiv_{Addr} \tilde{a}.\ \hat{\sigma}(\hat{a}) \sqsupseteq_{D} \tilde{\sigma}(\tilde{a})$$

As throughout, the use of a symbol twice in an above definition (*e.g.* $x$ in $\equiv_{Frame}$) indicates these components are exactly equal. The precision of analysis results and sets of reachable configurations are then defined like so:

$$\hat{r} \sqsupseteq_{R} \tilde{r} \text{ (via } \tilde{\sigma}_{\kappa}) \triangleq \forall (e, \tilde{\rho}, \tilde{a}_{\kappa}) \in \tilde{r}.\ \forall \tilde{\psi} \in_{\psi} \tilde{a}_{\kappa} \text{ (via } \tilde{\sigma}_{\kappa}).\ H_{C}((e, \tilde{\rho}, \tilde{a}_{\kappa}), \tilde{\psi}) \in \hat{r}$$

$$(\hat{r}, \hat{\sigma}) \sqsupseteq_{\Xi} (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_{\kappa}) \triangleq \hat{\sigma} \sqsupseteq_{Store} \tilde{\sigma} \ \wedge\ \hat{r} \sqsupseteq_{R} \tilde{r} \text{ (via } \tilde{\sigma}_{\kappa})$$

**Lemma 8** (Correspondence of bijection and precision)**.** *For addresses, environments, closures, frames, $\hat{\tau}$ and $\tilde{\tau}$, we have the following properties:*

$$\hat{\tau} \equiv_{\tau} \tilde{\tau} \iff \hat{\tau} = H_{\tau}(\tilde{\tau})$$

$$\hat{\tau} \equiv_{\tau} \tilde{\tau} \iff \tilde{\tau} = T_{\tau}(\hat{\tau})$$

*Proof.* By unfolding, simplification, the address bijection, and functional extensionality of environments. □

**Lemma 9** (Correspondence of conversion and precision for flow-sets)**.** *For all $\hat{d}$ and $\tilde{d}$, we have the following properties:*

$$H_{D}(\tilde{d}) \sqsupseteq_{D} \tilde{d}$$

$$\hat{d} \sqsupseteq_{D} T_{D}(\hat{d})$$

*Proof.* By unfolding, simplification, and Lemma 8. □

**Lemma 10** (Correspondence of conversion and precision for stores)**.** *For all $\hat{\sigma}$ and $\tilde{\sigma}$, we have the following properties:*

$$H_{Store}(\tilde{\sigma}) \sqsupseteq_{Store} \tilde{\sigma}$$

$$\hat{\sigma} \sqsupseteq_{Store} T_{Store}(\hat{\sigma})$$

*Proof.* By unfolding, simplification, Lemma 8, and Lemma 9. □

#### 5.4.5.2 Paths and Well-formedness

Now we define a few variants of the state step relations ($\rightsquigarrow_{\Sigma}^{\tilde{}}$) and ($\rightsquigarrow_{\Sigma}^{\hat{}}$). These sub-store step relations allow the actual store reached across the transition to be an intermediate

point, no greater than the store given. The unbounded-stack sub-store step relation ($\overset{\sqsubseteq}{\rightsquigarrow}^{\hat{}}_{\Sigma}$) is defined:

$$(e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \overset{\sqsubseteq}{\rightsquigarrow}^{\hat{}}_{\Sigma} (e', \hat{\rho}', \hat{\sigma}', \hat{\kappa}'), \text{where}$$
$$(e, \hat{\rho}, \hat{\sigma}, \hat{\kappa}) \rightsquigarrow^{\hat{}}_{\Sigma} (e', \hat{\rho}', \hat{\sigma}'', \hat{\kappa}')$$
$$\hat{\sigma}'' \sqsubseteq \hat{\sigma}'$$

A finite-state sub-store step relation ($\overset{\sqsubseteq}{\rightsquigarrow}^{\sim}_{\Sigma}$) is defined:

$$(e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa) \overset{\sqsubseteq}{\rightsquigarrow}^{\sim}_{\Sigma} (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa), \text{where}$$
$$(e, \tilde{\rho}, \tilde{\sigma}'', \tilde{\sigma}''_\kappa, \tilde{a}_\kappa) \rightsquigarrow^{\sim}_{\Sigma} (e', \tilde{\rho}', \tilde{\sigma}''', \tilde{\sigma}'''_\kappa, \tilde{a}'_\kappa)$$
$$(\tilde{\sigma}'' \sqsubseteq \tilde{\sigma}) \wedge (\tilde{\sigma}''' \sqsubseteq \tilde{\sigma}') \wedge (\tilde{\sigma}''_\kappa \sqsubseteq \tilde{\sigma}_\kappa) \wedge (\tilde{\sigma}'''_\kappa \sqsubseteq \tilde{\sigma}'_\kappa)$$

For ease in defining our forthcoming well-formedness properties, we also define two variants of this finite-state sub-store step relation which each additionally constrain the values added to the store. A variant requires a particular address ($\tilde{a}$) to be extended with a particular closure ($\widetilde{clo}$):

$$(e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa) \overset{\sqsubseteq}{\rightsquigarrow}^{\sim}_{\Sigma} (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa) \text{ (with } \tilde{a}, \widetilde{clo}), \text{where}$$
$$(e, \tilde{\rho}, \tilde{\sigma}'', \tilde{\sigma}''_\kappa, \tilde{a}_\kappa) \rightsquigarrow^{\sim}_{\Sigma} (e', \tilde{\rho}', \tilde{\sigma}''', \tilde{\sigma}'''_\kappa, \tilde{a}'_\kappa)$$
$$(\tilde{\sigma}'' \sqsubseteq \tilde{\sigma}) \wedge (\tilde{\sigma}''' \sqsubseteq \tilde{\sigma}') \wedge (\tilde{\sigma}''_\kappa \sqsubseteq \tilde{\sigma}_\kappa) \wedge (\tilde{\sigma}'''_\kappa \sqsubseteq \tilde{\sigma}'_\kappa) \wedge \widetilde{clo} \in \tilde{\sigma}'''(\tilde{a})$$

Another variant requires a particular address ($\tilde{a}''_\kappa$) to be extended with a particular continuation ($\widetilde{\kappa}$):

$$(e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa) \overset{\sqsubseteq}{\rightsquigarrow}^{\sim}_{\Sigma} (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa) \text{ (with } \widetilde{\kappa}, \tilde{a}''_\kappa), \text{where}$$
$$(e, \tilde{\rho}, \tilde{\sigma}'', \tilde{\sigma}''_\kappa, \tilde{a}_\kappa) \rightsquigarrow^{\sim}_{\Sigma} (e', \tilde{\rho}', \tilde{\sigma}''', \tilde{\sigma}'''_\kappa, \tilde{a}'_\kappa)$$
$$(\tilde{\sigma}'' \sqsubseteq \tilde{\sigma}) \wedge (\tilde{\sigma}''' \sqsubseteq \tilde{\sigma}') \wedge (\tilde{\sigma}''_\kappa \sqsubseteq \tilde{\sigma}_\kappa) \wedge (\tilde{\sigma}'''_\kappa \sqsubseteq \tilde{\sigma}'_\kappa) \wedge \widetilde{\kappa} \in \tilde{\sigma}'''_\kappa(\tilde{a}''_\kappa)$$

With these, we define a relation for paths between configurations within the unbounded-stack machine. Paths of length zero exist trivially:

$$(e, \hat{\rho}, \hat{\kappa}) \hookrightarrow^{\hat{}} (e, \hat{\rho}, \hat{\kappa}) \text{ (via } \hat{r}, \hat{\sigma}), \text{where}$$
$$(e, \hat{\rho}, \hat{\kappa}) \in \hat{r}$$

An existing path can be extended to a longer path where a further sub-store step exists:

$$(e, \hat{\rho}, \hat{\kappa}) \hookrightarrow^{\hat{}} (e', \hat{\rho}', \hat{\kappa}') \text{ (via } \hat{r}, \hat{\sigma}), \text{where}$$
$$(e, \hat{\rho}, \hat{\kappa}) \hookrightarrow^{\hat{}} (e'', \hat{\rho}'', \hat{\kappa}'') \text{ (via } \hat{r}, \hat{\sigma})$$
$$(e'', \hat{\rho}'', \hat{\sigma}, \hat{\kappa}'') \overset{\sqsubseteq}{\rightsquigarrow}^{\hat{}}_{\Sigma} (e', \hat{\rho}', \hat{\sigma}, \hat{\kappa}')$$

Similarly, we define a relation for paths between configurations within the finite-state machine. The end points of such paths are unique to both a configuration, and an implicit stack. Again, paths of length zero exist trivially:

$$\begin{aligned}((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \tilde{\hookrightarrow} ((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \;\; &(\text{via } \tilde{\xi},\, (\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)), \text{where} \\ &(e,\tilde{\rho},\tilde{a}_\kappa) \in \tilde{r}' \\ &\tilde{\psi} \in_\psi \tilde{a}_\kappa \text{ (via } \tilde{\sigma}'_\kappa)\end{aligned}$$

Paths may be extended with a sub-store return step to form a longer path:

$$\begin{aligned}((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \tilde{\hookrightarrow} ((e',\tilde{\rho}'_\kappa,\tilde{a}'_\kappa),\tilde{\psi}') \;\; &(\text{via } (\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),\, (\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)), \text{where} \\ \tilde{\rho}'_\kappa &= \tilde{\rho}_\kappa[x \mapsto \widetilde{alloc}(x,(ae,\tilde{\rho}'',\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}''_\kappa))] \\ (ae,\tilde{\rho}'',\tilde{a}''_\kappa) &\in \tilde{r} \\ (e',\tilde{\rho}'_\kappa,\tilde{a}'_\kappa) &\in \tilde{r}' \\ ((x,e',\tilde{\rho}_\kappa),\tilde{a}_\kappa) &\in \tilde{\sigma}_\kappa(\tilde{a}''_\kappa) \\ ((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \tilde{\hookrightarrow} ((ae,\tilde{\rho}'',\tilde{a}''_\kappa),((x,e',\tilde{\rho}_\kappa),\tilde{a}_\kappa):\tilde{\psi}') \;\; &(\text{via } (\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),\, (\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \\ (ae,\tilde{\rho}'',\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}''_\kappa) \overset{\sqsubseteq}{\tilde{\rightsquigarrow}}_{\tilde{\Sigma}} &(e',\tilde{\rho}'_\kappa,\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}'_\kappa)\end{aligned}$$

Paths may be extended with a sub-store call step to form a longer path:

$$\begin{aligned}((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \tilde{\hookrightarrow} ((e',\tilde{\rho}',\tilde{a}'_\kappa),((x,e'',\tilde{\rho}''),\tilde{a}''_\kappa):\tilde{\psi}') \;\; &(\text{via } (\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),\, (\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)), \text{where} \\ ((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) &\in \tilde{r} \\ ((x,e'',\tilde{\rho}''),\tilde{a}''_\kappa) &\in \tilde{\sigma}'_\kappa(\tilde{a}'_\kappa) \\ ((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \tilde{\hookrightarrow} ((\texttt{(let ([}x\ (f\ ae)\texttt{])}\ e''),\tilde{\rho}'',\tilde{a}''_\kappa),\tilde{\psi}') \;\; &(\text{via } (\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),\, (\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \\ (\texttt{(let ([}x\ (f\ ae)\texttt{])}\ e''),\tilde{\rho}'',\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}''_\kappa) \overset{\sqsubseteq}{\tilde{\rightsquigarrow}}_{\tilde{\Sigma}} &(e',\tilde{\rho}',\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}'_\kappa)\end{aligned}$$

This $(\tilde{\hookrightarrow})$ relation adds extra side conditions that ensure invariants needed in our proof.

Now we define well-formedness, a predicate defined in terms of several sub-properties. Well-formedness ($wf$) is a binary predicate with the first argument $\tilde{\xi}$ being the predecessor of the second argument $\tilde{\xi}'$, which is the result we say is well-formed:

$$\begin{aligned}wf(\tilde{\xi},\tilde{\xi}') \triangleq wf_{\tilde{\xi}}(\tilde{\xi},\tilde{\xi}') \wedge wf_{\sqsubseteq}(\tilde{\xi},\tilde{\xi}') \wedge wf_{\text{init}}(\tilde{\xi},\tilde{\xi}') \wedge wf_{\text{halt}}(\tilde{\xi},\tilde{\xi}') \\ \wedge wf_{\tilde{r}}(\tilde{\xi},\tilde{\xi}') \wedge wf_{\tilde{\sigma}}(\tilde{\xi},\tilde{\xi}') \wedge wf_{\tilde{\sigma}_\kappa}(\tilde{\xi},\tilde{\xi}')\end{aligned}$$

The $wf_{\tilde{\xi}}$ property requires that $\tilde{\xi}$ be well-formed and the predecessor of $\tilde{\xi}'$:

$$\begin{aligned}wf_{\tilde{\xi}}(\tilde{\xi},\tilde{\xi}') \triangleq (\tilde{\xi} = \tilde{\xi}' = (\{(e_0,\varnothing,\tilde{a}_{\text{halt}})\},\bot,\bot)) \\ \vee\ (\tilde{\xi} \tilde{\rightsquigarrow}_{\tilde{\Xi}} \tilde{\xi}' \wedge \exists\tilde{\xi}''.\ wf(\tilde{\xi}'',\tilde{\xi}))\end{aligned}$$

The $wf_{\sqsubseteq}$ property requires that $\tilde{\xi}'$ be component-wise greater than or equal to $\tilde{\xi}$:

$$wf_{\sqsubseteq}((\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \triangleq (\tilde{r} \subseteq \tilde{r}') \wedge (\tilde{\sigma} \sqsubseteq \tilde{\sigma}') \wedge (\tilde{\sigma}_\kappa \sqsubseteq \tilde{\sigma}'_\kappa)$$

The $wf_{\text{init}}$ property requires that the initial configuration be in $\tilde{\xi}'$:

$$wf_{\text{init}}(\tilde{\xi},(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \triangleq (e_0,\varnothing,\tilde{a}_{\text{halt}}) \in \tilde{r}'$$

The $wf_{\text{halt}}$ property requires that the halt-continuation address $a_{\text{halt}}$ not have any continuations associated with it:

$$wf_{\text{halt}}(\tilde{\xi},(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \triangleq \forall\tilde{\kappa}.\ \tilde{\kappa} \notin \tilde{\sigma}'_\kappa(\tilde{a}_{\text{halt}})$$

Finally, $wf_{\tilde{r}}$, $wf_{\tilde{\sigma}}$, and $wf_{\tilde{\sigma}_\kappa}$ ensure that everything in the $\tilde{r}$, $\tilde{\sigma}$, and $\tilde{\sigma}_\kappa$ for $\tilde{\xi}'$ has a reason to be there. For $wf_{\tilde{r}}$, this means that every element of $\tilde{r}$ has some path leading to it:

$$\begin{aligned}
&wf_{\tilde{r}}(\tilde{\xi},(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \triangleq \\
&\quad \forall(e,\tilde{\rho},\tilde{a}_\kappa) \in \tilde{r}'.\ \exists\tilde{\psi} \in_\psi \tilde{a}_\kappa \text{ (via } \tilde{\sigma}'_\kappa). \\
&\qquad ((e_0,\varnothing,\tilde{a}_{\text{halt}}),\epsilon) \tilde{\hookrightarrow} ((e,\tilde{\rho},\tilde{a}_\kappa),\tilde{\psi}) \ \text{ (via } \tilde{\xi},\ (\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa))
\end{aligned}$$

For $wf_{\tilde{\sigma}}$, this means that every value stored in $\tilde{\sigma}$ has some sub-store step ($\overset{\sqsubseteq}{\rightsquigarrow}_\Sigma^{\sim}$) that put it there:

$$\begin{aligned}
&wf_{\tilde{\sigma}}((\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \triangleq \\
&\quad \forall\tilde{a}.\ \forall\widetilde{clo} \in \tilde{\sigma}'(\tilde{a}).\ \exists(e,\tilde{\rho},\tilde{a}_\kappa) \in \tilde{r}.\ \exists(e',\tilde{\rho}',\tilde{a}'_\kappa) \in \tilde{r}'. \\
&\qquad (e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa) \overset{\sqsubseteq}{\rightsquigarrow}_\Sigma^{\sim} (e',\tilde{\rho}',\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}'_\kappa) \ \text{ (with } \tilde{a},\ \widetilde{clo})
\end{aligned}$$

For $wf_{\tilde{\sigma}_\kappa}$ this means that every value stored in $\tilde{\sigma}_\kappa$ has some sub-store step ($\overset{\sqsubseteq}{\rightsquigarrow}_\Sigma^{\sim}$) that put it there:

$$\begin{aligned}
&wf_{\tilde{\sigma}_\kappa}((\tilde{r},\tilde{\sigma},\tilde{\sigma}_\kappa),(\tilde{r}',\tilde{\sigma}',\tilde{\sigma}'_\kappa)) \triangleq \\
&\quad \forall\tilde{a}'_\kappa.\ \forall((x,e,\tilde{\rho}_\kappa),\tilde{a}_\kappa) \in \tilde{\sigma}'_\kappa(\tilde{a}'_\kappa).\ \overbrace{(e'_\kappa,\tilde{\rho}'_\kappa,(e'_\kappa,\tilde{\rho}'_\kappa))}^{Entry(\tilde{a}'_\kappa)} \in \tilde{r}' \\
&\qquad \wedge \exists f.\ \exists ae.\ (\texttt{(let ([}x\ \texttt{(}f\ ae\texttt{)])}\ e),\tilde{\rho}_\kappa,\tilde{a}_\kappa) \in \tilde{r} \\
&\qquad\quad \wedge (\texttt{(let ([}x\ \texttt{(}f\ ae\texttt{)])}\ e),\tilde{\rho}_\kappa,\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa) \\
&\qquad\qquad \overset{\sqsubseteq}{\rightsquigarrow}_\Sigma^{\sim} (e'_\kappa,\tilde{\rho}'_\kappa,\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}'_\kappa) \ \text{ (with } \tilde{a}'_\kappa,\ ((x,e,\tilde{\rho}_\kappa),\tilde{a}_\kappa))
\end{aligned}$$

For $wf_{\tilde{\sigma}_\kappa}$ and our lemmas below, we define a helper *Entry*, which maps a continuation address to the configuration that is the entry point for the function invocation that contains the configurations using that continuation address:

$$\widetilde{Entry} : \widetilde{Addr} \to \tilde{C}$$
$$\widetilde{Entry}(\tilde{a}_{\text{halt}}) \triangleq (e_0, \varnothing, \tilde{a}_{\text{halt}})$$
$$\widetilde{Entry}((e_\kappa, \tilde{\rho}_\kappa)) \triangleq (e_\kappa, \tilde{\rho}_\kappa, (e_\kappa, \tilde{\rho}_\kappa))$$

Finally, with the following assumption, we require that once allocation creates an address, it must always produce the same address for the same configuration even if the value or continuation stores have changed.

**Assumption 11** (Allocation consistency)**.** *If* $wf(\tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$*, and the state step* $(e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa) \rightsquigarrow_{\tilde{\Sigma}} (e', \tilde{\rho}'[x \mapsto \tilde{a}], \tilde{\sigma}'', \tilde{\sigma}''_\kappa, \tilde{a}'_\kappa)$ *holds where* $\tilde{a} = \widetilde{alloc}(x, (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa))$ *and there is a result step* $(\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa) \rightsquigarrow_{\tilde{\Xi}} (\tilde{r}', \tilde{\sigma}', \tilde{\sigma}'_\kappa)$*, then the corresponding allocation for* $e$*,* $\tilde{\rho}$*, and* $\tilde{a}_\kappa$*, but with* $\tilde{\sigma}'$ *and* $\tilde{\sigma}'_\kappa$*, is the same:*

$$\widetilde{alloc}(x, (e, \tilde{\rho}, \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}_\kappa)) = \tilde{a}$$

#### 5.4.5.3 Central Lemmas and Theorems

To start our proof, Lemma 12 shows that iterated steps produce well-formed $\tilde{\xi}$.

**Lemma 12** (Well-formedness of analysis results)**.** *If* $\tilde{\xi}'$ *is the result of taking zero or more steps of* $(\rightsquigarrow_{\tilde{\Xi}})$*, starting from the initial result,* $(\{(e_0, \varnothing, \tilde{a}_{halt})\}, \bot, \bot)$*, then* $wf(\tilde{\xi}, \tilde{\xi}')$*.*

*Proof.* We induct over steps. In the base case, we can easily show that the initial result is well-formed. We can also show that for any $\tilde{\xi}' \rightsquigarrow_{\tilde{\Xi}} \tilde{\xi}''$, if $wf(\tilde{\xi}, \tilde{\xi}')$ then $wf(\tilde{\xi}', \tilde{\xi}'')$. This is done using sublemmas for the components of well-formedness. We omit these for space. □

Next, with Lemma 13, we show that every configuration paired with one of its implied stacks has a path leading to it (*i.e.*, the top edge of Figure 5.7).

**Lemma 13** (Stacks have paths)**.** *If* $(e, \tilde{\rho}, \tilde{a}_\kappa) \in \tilde{r}$ *such that* $wf(\tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$*, then:*

$$\forall \tilde{\psi} \in_\psi \tilde{a}_\kappa \quad \textit{(via } \tilde{\sigma}_\kappa\textit{)}.$$
$$((e_0, \varnothing, \tilde{a}_{halt}), \epsilon) \tilde{\hookrightarrow} ((e, \tilde{\rho}, \tilde{a}_\kappa), \tilde{\psi}) \quad \textit{(via } \tilde{\xi}\textit{, } (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa)\textit{)}$$

*Proof.* By $wf_{\tilde{r}}(\tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$, there is some $\tilde{\psi}' \in_\psi \tilde{a}_\kappa$ (via $\tilde{\sigma}_\kappa$) for which $((e_0, \varnothing, \tilde{a}_{\text{halt}}), \epsilon) \tilde{\hookrightarrow} ((e, \tilde{\rho}, \tilde{a}_\kappa), \tilde{\psi}')$ (via $\tilde{\xi}$, $(\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa)$). However, this path uses $\tilde{\psi}'$ instead of our desired $\tilde{\psi}$. Thus,

we induct over $\tilde{\psi}$. If $\tilde{\psi}$ is the empty list, $\epsilon$, then $\tilde{a}_\kappa$ must be $\tilde{a}_{\text{halt}}$ and thus $\epsilon$ is the only $\tilde{\psi}$ for which $\tilde{\psi} \in_\psi \tilde{a}_\kappa$ (via $\tilde{\sigma}_\kappa$). So $\tilde{\psi}' = \tilde{\psi} = \epsilon$, and the path obtained from $wf_{\tilde{r}}(\tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$ equals our desired conclusion.

If $\tilde{\psi}$ is $((x, e_\kappa, \tilde{\rho}_\kappa), \tilde{a}'_\kappa) : \tilde{\psi}'$ for some $x$, $e_\kappa$, $\tilde{\rho}_\kappa$, $\tilde{a}'_\kappa$, $\tilde{\psi}'$, then by the path obtained from $wf_{\tilde{r}}(\tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$, there exists a path for $\tilde{\psi}$ from $\mathit{Entry}(\tilde{a}_\kappa)$ to $(e, \tilde{\rho}, \tilde{a}_\kappa)$:

$$(\mathit{Entry}(\tilde{a}_\kappa), \tilde{\psi}) \tilde{\hookrightarrow} ((e, \tilde{\rho}, \tilde{a}_\kappa), \tilde{\psi}) \quad (\text{via } \tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$$

By $wf_{\tilde{r}}(\tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$, there exist $f$ and $ae$ for a call site $(\texttt{(let ([x (f ae)]) } e_\kappa\texttt{)}, \tilde{\rho}_\kappa, \tilde{a}'_\kappa) \in \tilde{r}$ and a step from that call site to $\mathit{Entry}(\tilde{a}_\kappa)$:

$$(\texttt{(let ([x (f ae)]) } e_\kappa\texttt{)}, \tilde{\rho}_\kappa, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}'_\kappa) \overset{\sqsubseteq}{\tilde{\rightsquigarrow}}_{\tilde{\Sigma}} (e', \tilde{\rho}', \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa)$$

$$\text{where} \qquad (e', \tilde{\rho}', \tilde{a}_\kappa) = \mathit{Entry}(\tilde{a}_\kappa)$$

By the induction hypothesis, we have a path for $\tilde{\psi}'$ from $(e_0, \varnothing, \tilde{a}_{\text{halt}})$ to the call site:

$$((e_0, \varnothing, \tilde{a}_{\text{halt}}), \epsilon) \tilde{\hookrightarrow} ((\texttt{(let ([x (f ae)]) } e_\kappa\texttt{)}, \tilde{\rho}_\kappa, \tilde{a}'_\kappa), \tilde{\psi}')$$

We now have a path from $((e_0, \varnothing, \tilde{a}_{\text{halt}}), \epsilon)$ to the call site with $\tilde{\psi}'$, a step from the call site to $\mathit{Entry}(\tilde{a}_\kappa)$ that pushes $(x, e_\kappa, \tilde{\rho}_\kappa)$ onto the stack, and a path from $\mathit{Entry}(\tilde{a}_\kappa)$ to $(e, \tilde{\rho}, \tilde{a}_\kappa)$ with $\tilde{\psi}$. From these, we can then construct the path desired in our conclusion. □

Next, with Lemma 14, we show that every path in a well-formed $\tilde{\xi}$ has a corresponding path in any $\hat{\xi}$ that is at at fixed point (*i.e.*, the left edge of Figure 5.7).

**Lemma 14** (Path conversion)**.** *If* $(e, \tilde{\rho}, \tilde{a}_\kappa) \in \tilde{r}$ *such that* $wf(\tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$ *and* $(\hat{r}, \hat{\sigma}) \overset{\sqsubseteq}{\hat{\rightsquigarrow}}_{\hat{\Sigma}} (\hat{r}, \hat{\sigma})$*, then:*

$$\begin{aligned} &((e_0, \varnothing, \tilde{a}_{halt}), \epsilon) \tilde{\hookrightarrow} ((e, \tilde{\rho}, \tilde{a}_\kappa), \tilde{\psi}) \quad \textit{(via } \tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa)\textit{)} \\ &\quad \Longrightarrow (e_0, \varnothing, \epsilon) \hat{\hookrightarrow} H_C((e, \tilde{\rho}, \tilde{a}_\kappa), \tilde{\psi}) \quad \textit{(via } \hat{r}, \hat{\sigma}\textit{)} \end{aligned}$$

*Proof.* By induction over the finite-state path. We have three cases.

**Case:** The path is empty. Trivial.

**Case:** The last step of the path is a return. For some $ae$, $\tilde{\rho}'$, $\tilde{a}_\kappa$, and $\tilde{\rho}''$, there is a step $(ae, \tilde{\rho}', \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}'_\kappa) \overset{\sqsubseteq}{\tilde{\rightsquigarrow}}_{\tilde{\Sigma}} (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa)$ and, by the induction hypothesis, a path:

$$(e_0, \varnothing, \epsilon) \hat{\hookrightarrow} H_C((ae, \tilde{\rho}', \tilde{a}'_\kappa), ((x, e, \tilde{\rho}''), \tilde{a}_\kappa) : \tilde{\psi}) \quad (\text{via } \hat{r}, \hat{\sigma})$$

$$\text{where} \qquad \tilde{\rho} = \tilde{\rho}''[x \mapsto \widetilde{alloc}(x, (ae, \tilde{\rho}', \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}'_\kappa))]$$

We can then show that $(\hat{r}, \hat{\sigma})$ contains a step corresponding to the step in $(\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa)$:

$$(ae, H_{Env}(\tilde{\rho}'), \hat{\sigma}, (x, e, H_{Env}(\tilde{\rho}'')) : H_{Kont}(\tilde{\psi})) \\ \overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}} (e, H_{Env}(\tilde{\rho}), \hat{\sigma}, H_{Kont}(\tilde{\psi}))$$

Combining this with the path from the induction hypothesis, we can then construct the path in our conclusion.

**Case:** The last step of the path is a call. For some $ae$, $\tilde{\rho}'$, $\tilde{a}_\kappa$, and $\tilde{\rho}''$, we have $(y, e, \tilde{\rho}_\lambda) \in \tilde{\mathcal{A}}(f, \tilde{\rho}', \tilde{\sigma})$, and there is a step:

$$((\texttt{let ([}x\ (f\ ae)\texttt{])}\ e'), \tilde{\rho}', \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}'_\kappa) \overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}} (e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa), \text{ where}$$

$$\tilde{\rho} = \tilde{\rho}_\lambda[x \mapsto \widetilde{alloc}(x, ((\texttt{let ([}x\ (f\ ae)\texttt{])}\ e'), \tilde{\rho}', \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}'_\kappa))]$$

and, by the induction hypothesis, a path:

$$(e_0, \varnothing, \epsilon) \overset{\hat{}}{\hookrightarrow} H_C(((\texttt{let ([}x\ (f\ ae)\texttt{])}\ e'), \tilde{\rho}', \tilde{a}'_\kappa), \tilde{\psi}) \ \text{(via } \hat{r}, \hat{\sigma})$$

We can then show that $(\hat{r}, \hat{\sigma})$ contains a step corresponding to the step in $(\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa)$:

$$((\texttt{let ([}x\ (f\ ae)\texttt{])}\ e'), H_{Env}(\tilde{\rho}'), \hat{\sigma}, H_{Kont}(\tilde{\psi})) \\ \overset{\sqsubseteq}{\rightsquigarrow}_{\tilde{\Sigma}} (e, H_{Env}(\tilde{\rho}), \hat{\sigma}, (x, e', H_{Env}(\tilde{\rho}')) : H_{Kont}(\tilde{\psi}))$$

Combining this with the path from the induction hypothesis, we can then construct the path in our conclusion. □

Then, with Lemma 15, we show that the endpoint of any path in $\hat{\xi}$ is in $\hat{\xi}$ (*i.e.*, the bottom edge of Figure 5.7).

**Lemma 15** (Path endpoint)**.** *If* $(\hat{r}, \hat{\sigma}) \rightsquigarrow_{\hat{\Sigma}} (\hat{r}, \hat{\sigma})$*, then for any path* $\hat{c}_0 \overset{\hat{}}{\hookrightarrow} (e, \hat{\rho}, \hat{\kappa})$ *(via* $\hat{r}$*,* $\hat{\sigma}$*), we have:* $(e, \hat{\rho}, \hat{\kappa}) \in \hat{r}$*.*

*Proof.* Trivial. By induction. □

Finally, with Lemmas 16 and 17, we show that precision is preserved by the step relation $\rightsquigarrow_{\tilde{\Sigma}}$ (*i.e.*, the right edge of Figure 5.7). Then in theorem 18, we show that these are all precise, which is ultimately what we want to prove.

**Lemma 16** (Preservation of precision for value stores)**.** *If* $(\hat{r}, \hat{\sigma}) \rightsquigarrow_{\hat{\Xi}} (\hat{r}, \hat{\sigma})$*,* $wf(\tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$*,* $(\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa) \rightsquigarrow_{\tilde{\Xi}} (\tilde{r}', \tilde{\sigma}', \tilde{\sigma}'_\kappa)$*, and* $(\hat{r}, \hat{\sigma}) \sqsupseteq_\Xi (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa)$*, then* $\hat{\sigma} \sqsupseteq_{Store} \tilde{\sigma}'$*.*

*Proof.* Omitted for space. □

**Lemma 17** (Preservation of precision for reachable configurations)**.** *If* $(\hat{r}, \hat{\sigma}) \rightsquigarrow_{\hat{\Xi}} (\hat{r}, \hat{\sigma})$*,* $wf(\tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$*,* $(\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa) \rightsquigarrow_{\tilde{\Xi}} (\tilde{r}', \tilde{\sigma}', \tilde{\sigma}'_\kappa)$*, and* $(\hat{r}, \hat{\sigma}) \sqsupseteq_\Xi (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa)$*, then* $\hat{r} \sqsupseteq_{Store} \tilde{r}'$*.*

*Proof.* If we unfold the definition of $(\sqsupseteq_R)$, we must show that for all $(e, \hat{\rho}, \hat{\kappa}) \in \hat{r}'$ and $\tilde{\psi} \in_\psi \tilde{a}_\kappa$ (via $\tilde{\sigma}'_\kappa$), that $(e, H_{Env}(\tilde{\rho}), H_{Kont}(\tilde{\psi})) \in \hat{r}$. By Lemma 13, we have:

$$((e_0, \varnothing, \tilde{a}_{\text{halt}}), \epsilon) \tilde{\hookrightarrow} ((e, \tilde{\rho}, \tilde{a}_\kappa), \tilde{\psi}) \text{ (via } \tilde{\xi}, (\tilde{r}, \tilde{\sigma}, \tilde{\sigma}_\kappa))$$

From this, by Lemma 14, we have:

$$(e_0, \varnothing, \epsilon) \hat{\hookrightarrow} (e, H_{Env}(\tilde{\rho}), H_{Kont}(\tilde{\psi})) \text{ (via } \hat{r}, \hat{\sigma})$$

Finally, by Lemma 15, we have our conclusion. □

**Theorem 18** (Precision of analysis results)**.** *If* $\tilde{\xi}$ *is the result of taking zero or more steps of* $(\rightsquigarrow_{\tilde{\Xi}})$*, starting from* $(\{(e_0, \varnothing, \tilde{a}_{halt})\}, \bot, \bot)$*, and* $\hat{\xi} \rightsquigarrow_{\hat{\Xi}} \hat{\xi}$*, then* $\hat{\xi} \sqsupseteq_\Xi \tilde{\xi}$*.*

*Proof.* By induction over the number of steps, trivial simplifications, unfoldings, and Lemmas 12, 16, and 17. □

# CHAPTER 6

# IMPLEMENTATION STRATEGIES

This chapter explores strategies for more efficiently implementing the static analyses being discussed.

Some recent work has shown that very simple static analyses can be encoded as a series of fast matrix operations (Prabhu et al., 2010) and may be possible to extend to larger scale static analysis on the GPU. Other recent work shows how to implement an inclusion-based points-to analysis of C on the GPU by applying a set of semantic rules to the adjacency matrix of a sparse-graph (Mendez-Lojo et al., 2012). Both of these algorithms may be likened to finding the transitive closure of a graph encoded as an adjacency matrix. The matrix is repeatedly extended with new entries derived from sparse matrix-vector multiply (SpMV) until a fixed point is reached (no more edges need to be accumulated). These approaches to static analysis on the GPU are very different; both, however, require performant sparse-matrix operations and dynamic insertion of new entries.

## 6.1 General-Purpose GPU Programming

GPUs have become increasingly popular in recent years for solving computationally intense problems outside of graphics processing (Owens et al., 2007). The FLOP throughput on GPUs has continued to increase exponentially, significantly outpacing CPUs. Many high-performance computers now rely on the GPU as a work-horse for the computationally demanding linear-algebra which is often needed in large-scale scientific and engineering applications. For example, Cray's Titan supercomputer, the first machine to surpass 10 PFLOPS, uses 18,688 K20x Nvidia GPUs to accelerate its workload (Oak Ridge National Lab, 2016).

GPUs achieve their performance due to the use of a streaming SIMD (single-instruction, multiple-data) architecture which allows a group of small lightweight cores to perform the same operation on a vector of data and is ideal for certain tasks such as graphics and linear-algebra. This fine-grained parallelism within a streaming multiprocessor (SM) is constrained

to threads which operate in lock-step, but may be combined with coarser-grained parallelism from concurrent program execution across multiple SMs. Modern GPUs can have dozens of SMs and a combined total of several thousand concurrent threads. Within each SM, it is crucial that threads remain in lock-step, operating on the same instructions. In the CUDA programming model, *thread divergence* occurs when a branch instruction separates a group of threads into different control-flow paths. This causes threads on one path to continue while the others wait to run sequentially and is disastrous for performance.

With the addition of soldered memory, GPUs are also able to attain a significantly higher memory bandwidth than that of traditional CPUs. This is particularly important for our application as static analyses are typically memory-bound problems that involve largely data movement and manipulations of data structures rather than of data itself.

## 6.2 Extending EigenCFA

EigenCFA is a preliminary attempt at implementing static analysis as highly parallel matrix operations (Prabhu et al., 2010). It derives a rather clever encoding of 0-CFA as linear algebra which is efficient to perform on throughput-oriented hardware such as SIMD architectures like the GPU. Unfortunately, it is restricted by its need to encode the entire analysis as a single GPU kernel. This restriction has made it impossible to encode an analysis of simple language features like primitive operations and conditions. The presence of an addition operation or branch places even trivial programs outside the ability of EigenCFA to handle efficiently. In addition, EigenCFA is less precise than it should be as it uses the trivially sound approximation for control-flow behavior. Consider the following sample taken from a larger program where `f`, `g`, and `h` are unreachable functions.

```
(define (f x) (h x 0))
(define (g) (h 0 0))
```

Proving that these are in-fact unreachable requires a precise control-flow analysis. EigenCFA models only data-flow and assumes the reachability of all expressions in a program. Because of this, the call sites `(h 0 0)` and `(h x 0)` will be examined and their data-flows propagated. As `x` is unbound, even after handling propagations for `(h x 0)`, the corresponding formal parameter for h will remain unbound. An unbound variable is an implicit indication that a function (in this case `h`) is in-fact unreachable. In this case, an approximation of data-flow has resulted in a bound on control-flow, but this cannot be relied on to give the same precision as a traditional 0-CFA which models control explicitly. Because the callsite in g applies h on two constant values, once the analysis is complete, it will appear that h may

be reachable as possible values will have been found for both its parameters. Our linear encoding solves this problem by explicitly modeling both the control-flow and data-flow aspects of a program.

In this section, we review EigenCFA's approach to encoding static analyses as well as present a novel improvement which allows the essential strategy to be extended to real-world analyses with a variety of heterogenous semantic rules. We introduce a concrete and abstract semantics for a Scheme intermediate-representation, and derive a linear encoding for 0-CFA. Our encoding has been implemented both as a single-threaded CPU version and as single-stream and multistream GPU versions. All these have been tested for correctness against a standard worklist implementation and produce identical results for a suite of Scheme benchmarks. While the optimization effort is ongoing, preliminary results are promising, showing potential speedups of 20x or better over a single-threaded version of the encoding.

### 6.2.1 0-CFA of a Scheme-Like IR

We perform a structural abstraction bounding the machine's address-space to obtain a computable approximation of our concrete semantics. Notice that our abstract semantics contains several fundamental changes from its concrete counterpart. 0-CFA bounds the address-space to include exactly one address for each variable (monovariance). All values bound to a variable $x$ in any context therefore must be represented by a single address. This introduces merging between values in our store and nondeterminism in the transition relation.

To define an abstract operational semantics, we again need an abstract machine and a transition relation ($\approx\!\!>$) which matches up successors and predecessors within the machine's configuration-space. As we are effectively re-using an empty timestamp for every allocation, expressions will uniquely identify an environment mapping free-variables to themselves and the store may directly map variables $\hat{x}$ to sets of abstract values $\hat{v}$. Such a *flow-set* may indicate a range of possible concrete values for an address. Closures are now just lambdas.

$$\begin{aligned}
\hat{\varsigma} \in \widehat{\Sigma} &= \mathsf{E} \times \widehat{Store} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Var} \to \widehat{Values} \\
\hat{v} \in \widehat{Values} &= \mathcal{P}(\widehat{Value}) \\
\hat{d} \in \widehat{Value} &= \mathsf{Lam} + \widehat{Basic} \\
\widehat{Basic} &= \{\texttt{TRUE},\ \texttt{FALSE},\ \texttt{VOID},\ \texttt{INT},\ \ldots\}
\end{aligned}$$

Program constants map to their corresponding basic values. When performing a concrete

interpretation, these values are precise. When performing our abstract interpretation, there should only be a finite number of abstract basic values so they can be enumerated in our forthcoming encoding. For constant propagation, a set of program locations may be used. We use the notation $\alpha$ below to informally indicate the abstraction function a fully defined Galois-connection would employ to map a concrete machine component to its most precise abstract representative. For example, $\alpha(-3)$ could yield `NEG`.

The abstract atomic-expression evaluator returns flow-sets $\hat{v}$.

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

We also need an abstract prim-op evaluator $\hat{\delta}$ which maps a primitive operation *op* and list of flow-sets to a sound result. For example, $\hat{\delta}(\texttt{+},\ (\{\texttt{POS}\},\ \{\texttt{POS}\})) = \{\texttt{POS}\}$.

$$\hat{\delta}\colon \mathsf{OP} \times \widehat{Values}^{*} \rightharpoonup \widehat{Values}$$

A callsite has one successor for each closure that accepts a matching number of arguments indicated by the flow-set for $ae_f$ (the atomic-expression in call position).

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

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

Control moves inside the body of all invoked closures $e$. The updated store is now conservatively approximated by finding the least-upper-bound of the current store and each new binding. Stores are ordered point-wise by inclusion, *i.e.*, $(\hat{\sigma}_1 \sqcup \hat{\sigma}_2)(\hat{a}) = \hat{\sigma}_1(\hat{a}) \cup \hat{\sigma}_2(\hat{a})$.

Mutation is succeeded by a state for each possible continuation.

$$\frac{(\lambda\ (x_k)\ e) \in \hat{\mathcal{A}}(ae_k,\ \hat{\varsigma})}{\underbrace{((\text{set!}\ x\ ae_v\ ae_k),\ \hat{\sigma})}_{\hat{\varsigma}} \twoheadrightarrow (e,\ \hat{\sigma}')}$$

$$\begin{aligned} where \quad \hat{\sigma}' &= \hat{\sigma} \sqcup [x_k \mapsto \{\texttt{VOID}\}] \\ &\quad \sqcup [x \mapsto \hat{\mathcal{A}}(ae_v,\ \hat{\varsigma})] \end{aligned}$$

In addition, to conservatively simulate mutation of the variable $x$, all flows indicated for $ae_v$ are included along with all previous values.

$$\frac{(\lambda\ (x_k)\ e) \in \hat{\mathcal{A}}(ae_k,\ \hat{\varsigma})}{\underbrace{((\text{prim } op\ ae_1\ \ldots\ ae_j\ ae_k),\ \hat{\sigma})}_{\hat{\varsigma}} \twoheadrightarrow\ (e,\ \hat{\sigma}')}$$

$$\begin{aligned} where \quad \hat{\sigma}' &= \hat{\sigma} \sqcup [x_k \mapsto \hat{v}_k] \\ \hat{v}_k &= \delta(op,\ (\hat{\mathcal{A}}(ae_1,\ \hat{\varsigma})\ \ldots\ \hat{\mathcal{A}}(ae_j,\ \hat{\varsigma}))) \end{aligned}$$

Primitive operations use $\hat{\delta}$ to obtain an approximation of the return value and propagate this flow-set $\hat{v}_k$ to each continuation indicated for $ae_k$.

$$\frac{\hat{v} \in \hat{\mathcal{A}}(ae,\ \hat{\varsigma}) \qquad \hat{v} \neq \texttt{FALSE}}{\underbrace{((\text{if } ae\ e_t\ e_f),\ \hat{\sigma})}_{\hat{\varsigma}} \twoheadrightarrow\ (e_t,\ \hat{\sigma})}$$

$$\frac{\hat{v} \in \hat{\mathcal{A}}(ae,\ \hat{\varsigma}) \qquad \hat{v} = \texttt{FALSE}}{\underbrace{((\text{if } ae\ e_t\ e_f),\ \hat{\sigma})}_{\hat{\varsigma}} \twoheadrightarrow\ (e_f,\ \hat{\sigma})}$$

When a conditional is reached, both branches may be taken.

### 6.2.2 Naïvely Computing the Analysis

We first define an injection function $\hat{\mathcal{I}}$ which, given a program $e$, determines an initial state $\hat{\varsigma}_0 = \hat{\mathcal{I}}(e)$.

$$\begin{aligned} \hat{\mathcal{I}} &: \mathsf{E} \to \widehat{\Sigma} \\ \hat{\mathcal{I}}(e) &= (e,\ \bot) \end{aligned}$$

To compute our analysis, we can simply visit all states reachable from $\hat{\varsigma}_0$. We define a transfer function for the system-space of our program $\hat{f} : \mathcal{P}(\widehat{\Sigma}) \to \mathcal{P}(\widehat{\Sigma})$:

$$\hat{f}(\hat{S}) = \{\hat{\varsigma}' \ :\ \hat{\varsigma} \in \hat{S} \text{ and } \hat{\varsigma} \twoheadrightarrow \hat{\varsigma}'\} \cup \{\hat{\varsigma}_0\}$$

Unfortunately, this approach is impracticable as the total number of stores is exponential in the size of the program, even for this context-insensitive analysis.

### 6.2.3 Efficiently Computing the Analysis

A more efficient method uses a single store to replace the multitude of individual stores. This global store is maintained as the least-upper-bound of all stores seen so far. Global-store-widening is a sound, and in practice quite reasonable, approximation of the naïve calculation (Might, 2007; Shivers, 1991). With this form of widening applied, 0-CFA is in $O(\frac{n^3}{log(n)})$. We factor the store out of our state-space while retaining a set of reachable expressions denoted $\hat{r}$ as an explicit model of control-flow. EigenCFA compromises on precision by modeling only the store.

$$\hat{r} \in \widehat{Reach} = \mathcal{P}(\mathsf{E})$$
$$\hat{\xi} \in \hat{\Xi} = \widehat{Reach} \times \widehat{Store}$$

Over factored system-spaces $\hat{\Xi}$, the transfer function becomes:

$$\begin{aligned}
\hat{f} &: \hat{\Xi} \to \hat{\Xi} \\
\hat{f}(\hat{r},\ \hat{\sigma}) &= (\hat{r} \cup \hat{r}',\ \hat{\sigma}') \\
where \quad \hat{S} &= \{\hat{\varsigma}' : e \in \hat{r} \text{ and } (e,\ \hat{\sigma}) \approx\!\gg \hat{\varsigma}'\} \\
\hat{r}' &= \{e : (e,\ _) \in \hat{S}\} \\
\hat{\sigma}' &= \bigsqcup \{\hat{\sigma}'' : (_,\ \hat{\sigma}'') \in \hat{S}\}
\end{aligned}$$

The notation _ matches any value without binding it to a variable.

The store grows monotonically across transition, *i.e.*, $(\_,\ \hat{\sigma}) \approx\!\gg (\_,\ \hat{\sigma}')$ implies $\hat{\sigma} \sqsubseteq \hat{\sigma}'$, so $\hat{f}$ grows monotonically over $\hat{\Xi}$. Because $\hat{\Xi}$ is finite and $\hat{f}$ is continuous, we know that the least-fix-point of $\hat{f}$ is $\hat{f}^n(\bot, \bot)$ for some finite $n$.

## 6.3 Partitioning the Transfer Function

The central insight of our improvement to EigenCFA is that we can partition a transfer function by reachable state under evaluation. By grouping these individual transfer functions into GPU kernels according to like control-flow, we can minimize thread-divergence in a SIMD implementation. An individual transfer function $\hat{f}_e$ handles only the propagation of flows caused directly by $e$:

$$\begin{aligned}
\hat{f}_e &: \hat{\Xi} \to \hat{\Xi} \\
\hat{f}_e(\hat{r},\ \hat{\sigma}) &= (\hat{r} \cup \hat{r}'_e,\ \hat{\sigma}'_e) \\
\textit{where} \quad \hat{S}_e &= \{\hat{\varsigma}' : e \in \hat{r} \text{ and } (e,\ \hat{\sigma}) \twoheadrightarrow \hat{\varsigma}'\} \\
\hat{r}'_e &= \{e'' :\ (e'',\ _) \in \hat{S}_e\} \\
\hat{\sigma}'_e &= \bigsqcup\{\hat{\sigma}'' : (_,\ \hat{\sigma}'') \in \hat{S}_e\}
\end{aligned}$$

To determine the correctness and precision of this technique, we show its equivalence to an unpartitioned transfer function so we may exploit the corollary that a solution $\hat{\xi}$ which is simultaneously a fix-point for all $\hat{f}_e$ is guaranteed to be a fix-point for $\hat{f}$.

**Theorem 19** (Transfer Partitioning)**.**

$$\hat{f}(\hat{r},\ \hat{\sigma}) = \bigsqcup_{e \in \hat{r}} \hat{f}_e(\hat{r},\ \hat{\sigma})$$

*Proof. (Sketch)* Follows from the observation that $\hat{S}_e$ for all $e \in \hat{r}$ is a collection of covering subsets for $\hat{S}$.

$$\hat{S} = \bigcup_{e \in \hat{r}} \hat{S}_e$$

Therefore, $\hat{r}'$ is also the least-join of all $\hat{r}'_e$ as is $\hat{\sigma}'$ of all $\hat{\sigma}'_e$. □

## 6.4 A Linear Encoding of 0-CFA

Now that we may arbitrarily partition a transfer function to minimize thread-divergence, a linear encoding for handling a call site can be defined separately from a linear encoding that handles a conditional, a primitive operation, or another form. The goal now is to produce an implementation for each $f_e$ defined exclusively in terms of matrix multiplication ($\times$), outer product ($\otimes$), element-wise boolean-or ($+$), and dot product ($\cdot$).

For any finite domain, we can assign a canonical order to its contents and represent elements of its set or power-set as boolean vectors. Where vectors contain a single entry, they represent a single element in the set they encode, and where they contain more than one entry, the representation naturally extends to encoding more than one element at once. For example, as defined below, a value $\vec{v} \in \vec{V}$ is a vector representing a flow-set of abstract values. In the case of $\vec{S}$, we use $\vec{r}$ in all cases to denote a set of states and $\vec{s}$ to denote a particular state (*i.e.*, a vector with a single entry).

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

Vectors $\vec{a}$ represent atomic-expressions, either variables or values. This is a design choice taken directly from EigenCFA which allows the various cases required for $\hat{\mathcal{A}}$ to be implemented as a single multiplication.

A function $g$ over these vectors can be encoded as multiplication with a matrix, and may handle inputs which encode a set so long as the property $g(x \cup y) = g(x) \cup g(y)$ holds for all $x$ and $y$. The store is such a function, one which maps variables to a flow-set of values, and values to themselves:

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

If values are ordered after variables in $\vec{A}$, the bottom of the store will always be an identity matrix. Below is an example of a lookup showing how the store is used to map a variable to its flow-set via matrix multiplication. We use a CPS version of our original snippet of Scheme code for clarity.

```
((lambda (add5^x0)
   (add5^x0 10^d̂3 (lambda (result^x3) (halt)^l3)^d̂2)^l1)^d̂0
 (lambda (a^x1 add5k^x2)
   (prim + a^x1 5^d̂3 add5k^x2)^l2)^d̂1)^l0
```

Annotations show an assignment of labels to expressions, variables to vectors in $\vec{A}$, and abstract values to vectors in $\vec{V}$. The abstract value $\hat{d}_3$ represents `INT`.

**Example 20.** $\langle\langle \mathtt{a} \rangle\rangle \times \boldsymbol{\sigma} = \langle\langle \{\mathtt{INT}\} \rangle\rangle$

$$\begin{array}{c} \begin{array}{cccccccc} x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \end{array} \\ \left[\begin{array}{cccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array} \times \begin{array}{c} \begin{array}{cccc} \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \end{array} \\ \left[\begin{array}{cccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right] \end{array} \begin{array}{l} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \\ \hat{d}_0 \\ \hat{d}_1 \\ \hat{d}_2 \\ \hat{d}_3 \end{array} = \begin{array}{c} \begin{array}{cccc} \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \end{array} \\ \left[\begin{array}{cccc} 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

The notation $\langle\langle \cdot \rangle\rangle$ is used informally to denote the matrix representation of a given entity. Including an identity matrix in the store composes two mappings as one matrix so that all cases in $\hat{\mathcal{A}}$ may be handled together as a single multiplication.

A program's syntax tree can also be encoded as a series of matrices. For example, a matrix **Body** maps lambdas in $\vec{V}$ to their body expression in $\vec{S}$. The same can be done for the true and false branches of a conditional form.

$$\begin{aligned} \mathbf{Body}\colon \vec{V} \to \vec{S} \qquad & \mathbf{Fun}\colon \vec{S} \to \vec{A} \\ \mathbf{CondTrue}\colon \vec{S} \to \vec{S} \qquad & \mathbf{Arg_i}\colon \vec{S} \to \vec{A} \\ \mathbf{CondFalse}\colon \vec{S} \to \vec{S} \qquad & \mathbf{Var_i}\colon \vec{V} \to \vec{A} \end{aligned}$$

**Fun** maps call sites to the atomic-expression in call-position. If this is a variable, a value in the top portion of $\vec{A}$ will result; if it's a lambda or constant value, an entry in the lower portion of $\vec{A}$ results. $\mathbf{Arg_i}$ represents a similar encoding for argument $i$ of a callsite. $\mathbf{Var_i}$ encodes formal parameter $i$ of a lambda. For example, we can expect $\langle\!\langle$`(λ (a add5k) ...)`$\rangle\!\rangle \times$ $\mathbf{Var_2}$ to yield a value $\langle\!\langle$`add5k`$\rangle\!\rangle$.

For a callsite $\vec{s}$, the value of its second argument can be computed as $\vec{v}_2 = \vec{s} \times \mathbf{Arg_2} \times \boldsymbol{\sigma}$ and the value of the applied lambda as $\vec{v}_f = \vec{s} \times \mathbf{Fun} \times \boldsymbol{\sigma}$. The second formal parameter for $\vec{v}_f$ may then be computed as $\vec{a}_2 = \vec{v}_f \times \mathbf{Var_2}$ and with these two values, the store can be updated with a binding to $\vec{v}_2$ for $\vec{a}_2$. This is accomplished by using the outer product $\vec{a}_2 \otimes \vec{v}_2$ as this will give a store-update matrix with an entry at index $(m, n)$ whenever $\vec{a}_2$ has an entry at position $m$ and $\vec{v}_2$ has one at $n$. An update is applied to the current store using element-wise boolean-or. The example below shows the store update produced for `add5k`.

**Example 21.** $\langle\!\langle$`add5k`$\rangle\!\rangle \otimes \langle\!\langle$`(lambda (result) (halt))`$\rangle\!\rangle$

$$= \langle\!\langle [\texttt{add5k} \mapsto \texttt{(lambda (result) (halt))}] \rangle\!\rangle$$

$$\begin{array}{cccccccc} x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \\ \left[\begin{array}{cccccccc} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right] & \otimes & \multicolumn{4}{c}{\left[\begin{array}{cccc} \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \\ 0 & 0 & 1 & 0 \end{array}\right]} & = & \end{array} \begin{array}{c} \\ x_0 \\ x_1 \\ x_2 \\ x_3 \\ \hline \hat{d}_0 \\ \hat{d}_1 \\ \hat{d}_2 \\ \hat{d}_3 \end{array} \begin{array}{c} \begin{array}{cccc} \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \end{array} \\ \left[\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

An operation $\vec{v}_f \times \mathbf{Body}$ finds the body for $\vec{v}_f$, and boolean-or is used to extend the vector of reachable expressions $\vec{r}$. A full encoding for $f_e$ where $e$ is a call site of length $j$ can now be defined in full using these operations:

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

To handle `set!` forms, we may reuse the matrix **Fun** for encoding the continuation, $\mathbf{Arg_1}$ for encoding the variable being set, and $\mathbf{Arg_2}$ for encoding the atomic-expression it's being assigned to. The continuation receives a value $\langle\langle$`VOID`$\rangle\rangle$ we will denote as $\overrightarrow{void}$:

$$
\begin{aligned}
f_{\vec{s}_{set!}}(\vec{r},\ \boldsymbol{\sigma}) &= (\vec{r}',\ \boldsymbol{\sigma}') \\
\quad where \quad & \vec{v}_f = \vec{s}_{set!} \times \mathbf{Fun} \times \boldsymbol{\sigma} \\
& \vec{a}_{var} = \vec{v}_f \times \mathbf{Var_1} \\
& \vec{a}_{set} = \vec{s}_{set!} \times \mathbf{Arg_1} \\
& \vec{v}_{set} = \vec{s}_{set!} \times \mathbf{Arg_2} \times \boldsymbol{\sigma} \\
& \boldsymbol{\sigma}' = \boldsymbol{\sigma} + (\vec{a}_{var} \otimes \overrightarrow{void}) + (\vec{a}_{set} \otimes \vec{v}_{set}) \\
& \vec{r}' = \vec{r} + (\vec{v}_f \times \mathbf{Body})
\end{aligned}
$$

Conditionals make no changes to the store, but extend reachability to the subexpression for the true or false branches as appropriate. $\overrightarrow{false}$ is used to denote $\langle\langle$`FALSE`$\rangle\rangle$ and $\overrightarrow{notfalse}$ to denote its inverse – which is notably not the same as $\langle\langle$`TRUE`$\rangle\rangle$. A dot product is used to obtain a boolean value which is false exactly when the intersection of two sets is empty:

$$
\begin{aligned}
f_{\vec{s}_{if}}(\vec{r},\ \boldsymbol{\sigma}) &= (\vec{r}',\ \boldsymbol{\sigma}) \\
\quad where \quad & \vec{v}_{cond} = \vec{s}_{if} \times \mathbf{Arg_1} \times \boldsymbol{\sigma} \\
& tb = \vec{v}_{cond} \cdot \overrightarrow{notfalse} \\
& fb = \vec{v}_{cond} \cdot \overrightarrow{false} \\
& \vec{r}' = \vec{r} + tb(\vec{s}_{if} \times \mathbf{CondTrue}) + fb(\vec{s}_{if} \times \mathbf{CondFalse})
\end{aligned}
$$

### 6.4.1 A Fixed-Point Algorithm

To find a solution, we can iterate to a fix-point $(\vec{r},\ \boldsymbol{\sigma})$ over all $f_{\vec{s}}$ where $\vec{s}$ is drawn from the entries of $\vec{r}$. In practice, we may exploit both the fine-grain parallelism of matrix operations and the coarse-grain parallelism of running each $f_{\vec{s}}$ concurrently. As each individual $f_{\vec{s}}$ is

monotonic and continuous, our reasoning on termination and precision from section 6.2.3 remains applicable.

**while** $\boldsymbol{\sigma}$ or $\vec{r}$ changes **do**
  **foreach** $\vec{s}$ **in** $\vec{r}$ **do**
    $(\vec{r}, \boldsymbol{\sigma}) = f_{\vec{s}}(\vec{r}, \boldsymbol{\sigma})$
  **end**
**end**

## 6.5 Efficient Dynamic Matrix Updates

Sparse matrix-vector multiply (SpMV), the workhorse operation of many numerical simulations, has seen use in a wide variety of areas such as data mining (Im et al., 2000) and graph analytics (Gilbert et al., 2007). In scientific computing and numerical algorithms, a majority of the total processing is frequently spent on SpMV operations. Iterative computations such as the power method and conjugate gradient are commonly used in numerical simulations and require successive SpMV operations (Saad, 2003). GPUs are increasingly used for computing these operations as they are, in principle, highly parallelizable. GPUs have both a high computational throughput and a high memory bandwidth. Operations on sparse matrices are generally memory bound, which makes the GPU a good target platform due to its higher memory bandwidth compared to that of the CPU. However, it is still difficult to attain high performance with sparse matrices because of thread divergence and noncoalesced memory accesses.

Some applications require dynamic updates to the matrix; broadly construed, updates may include inserting or deleting entries. Fully compressed formats such as compressed sparse row (CSR) cannot handle these operations without rebuilding the entire matrix. Rebuilding the matrix is orders of magnitude more costly than performing an SpMV operation. The ellpack (ELL) format allocates a fixed amount of space for each row, allowing fast insertion of new entries and fast SpMV but limits each row to a predetermined number of entries and can be highly memory inefficient. The coordinate (COO) format stores a list of entries and permits both efficient memory use and fast dynamic updates but is unordered and slow to perform SpMV and SpMM operations. The hybrid-ellpack (HYB) format attempts a compromise between these by combining an ELL matrix with a COO matrix for overflow. Operations over rows may require examination of this overflow matrix,

however, and efficiency suffers.

Matrix representations of sparse graphs sometimes exhibit a power-law distribution (when the number of nodes with a given number of edges scales as a power of the number of edges). This distribution results in a long tail in which a few rows have a relatively high number of entries, but the rest have a relatively low number. Real-world phenomena often exhibit the power-law distribution—their corresponding matrices can represent adjacency graphs, web communication, and, in the case of our static analysis application, finite-state simulations. Such a matrix is also the pathological case for memory efficiency in the ELL format and requires significant use of the COO portion of a HYB matrix, making neither particularly well suited for dynamic sparse-graph applications.

Outside static analysis, sparse-matrix factorization is the essential step in direct methods for solving linear systems. This process is highly time and memory consuming, and could benefit from efficient dynamic updates to the factors being built or reduced. Existing methods for $LU$-decomposition (Gupta et al., 2009) and Cholesky decomposition (Avron and Gupta, 2012) make frequent use of sparse-matrix addition to union components of the overall workload during a recursive merging step. This union of matrices is done by allocating a fresh matrix all at once or by proprietary ad-hoc methods, which have gone undisclosed in the literature. Our work provides a general matrix format that allows such merging steps to incrementally extend an existing matrix.

Sparse matrix-matrix multiplication (SpMM) is another application for efficient dynamic updates. Existing approaches use an intermediate COO format matrix to compile a list of partial results before building the final product. A more efficient approach is to dynamically extend the final product with these intermediate results as they are asynchronously accumulated. Algebraic multigrid can be formulated in terms of SpMV, SpMM, and primitive vector operations, and this is often the preferred method on the GPU (Bell et al., 2012).

In this section, we review existing matrix formats and present our alternative approach to dynamic matrix allocation which allows an existing matrix to be modified arbitrarily in-place for certain operations. We also demonstrate that operations such as SpMM, which cannot be done in-place, still benefit greatly from our dynamic format due to improved memory efficiency. Our format, dynamic compressed sparse row (DCSR), has been designed for easy conversion with standard CSR, fast dynamic updates, and fast SpMV (King et al., 2016).

### 6.5.1 Sparse Matrices

Sparse matrices are those using one of a variety of formats which optimizes for values with only a small percentage of non-zero entries. As SpMV is arguably the most important sparse-matrix operation, we want to maintain efficient times for the problem $Ax = y$. A major design principle for sparse-matrix formats is to reduce irregularity in memory accesses. To begin, we review the most commonly used sparse-matrix formats.

The *coordinate* (COO) format is the simplest sparse-matrix format. It represents a matrix with three vectors holding the row indices, column indices, and values for all non-zero entries in the matrix. The entries within a COO format must be sorted by row in order to efficiently perform an SpMV operation. SpMV operations are conducted in parallel through segmented reductions over the length of the arrays. Tracking which thread has processed the final entry in a row requires explicit interthread communication.

The *compressed sparse row/column* (CSR/CSC) formats have arrays that fully store two of the three sets, either the column indices or the row indices in addition to the values. Either the rows or columns (in CSR or CSC, respectively) are compressed to store only the offsets into the other two arrays. For CSR, entry $i$ and $i+1$ in the row offsets array will store the starting and ending offsets for row $i$. CSR has been shown to be one of the best formats in terms of memory usage and SpMV efficiency due to its fully compressed nature and has become widely used (Greathouse and Daga, 2014). CSR has a greater memory efficiency than COO, which is a significant factor in speeding up SpMV operations due to decreased memory bandwidth usage.

The *ellpack* (ELL) format uses two arrays, each of size $m \times k$ (where $m$ is the number of rows and $k$ is a fixed width), to store the column indices and the values of the matrix (Garland, 2008; Garland and Kirk, 2010). These arrays are stored in column-major order to allow for efficient parallel access across rows. This format is best suited for matrices that have a fixed number of entries per row. Allocating enough memory in each row to store the entire matrix is prohibitively expensive for ELL when a matrix contains even one long row.

The *hybrid-ellpack* (HYB) format offers a compromise by using a combination of ELL and COO. It stores as many entries as possible in an ELL portion, and the overflow from rows with a number of entries greater than the fixed ELL width is stored in a COO portion. ELL and HYB have become popular on SIMD architectures due to the ability of thread warps to look through consecutive rows in an efficient parallel manner (Bell and Garland, 2008).

A number of other specialized sparse-matrix formats have been developed, including

jagged diagonal storage (JDS), block diagonal (BDIA), skyline storage (SKS), tiled COO (TCOO), block ELL (BELL), and sliced-ELL (SELL) (Monakov et al., 2010), all of which offer improved performance for specific matrix types. Blocked variants of these and other formats work by storing localized entries in blocks for better data locality and a reduction in index storage. "Cocktail" frameworks that mix and match matrix formats to fit specific subsets of the matrix have been developed, but they require significant preprocessing and are not easily modified dynamically (Su and Keutzer, 2012). Garland et al. have provided detailed reviews of the most common sparse matrix formats (Garland, 2008; Garland and Kirk, 2010; Vuduc, 2003), as well as an analysis of their performance on throughput-oriented many-core processors (Bell and Garland, 2009).

Block formats such as BRC (Ashari et al., 2014b) and BCCOO (Yan et al., 2014) have limited ability to add in additional entries. BRC can add new entries only if those entries correspond to zeros within blocks that have been stored. BCCOO can handle the addition of new entries, but it suffers from many of the same problems as COO. Also, new insertions will not always follow a blocked structure, so additional blocks may be especially sparse, which lowers memory efficiency.

Many sparse matrix formats are fully compressed and do not allow additional entries to be added to the matrix dynamically. Adding additional entries to a CSR matrix requires rebuilding the entire matrix, since there is no free space between entries. Of existing formats, COO is the most amenable to dynamic updates because new entries can be placed at the end of the data structure. However, updating a COO matrix in parallel requires atomic operations to keep track of currently available memory locations. The ELL/HYB formats allow for some additional entries to be added in a limited fashion. ELL cannot add in more entries per row than the given width of the matrix, and while the HYB format has a COO matrix to handle overflow from the ELL portion, it cannot be efficiently updated in parallel since atomic operations are required and the COO portion must maintain the sorted property.

### 6.5.2 Sparse Matrix Algorithms on the GPU

A great deal of research has been devoted to improving the efficiency of SpMV on both multi-core and many-core architectures. Williams et al. demonstrated the efficacy of using architecture-specific data structures to optimize performance (Liu et al., 2013; Williams et al., 2007). Since SpMV is a bandwidth-limited operation, research has also produced other methods, such as automatic tuning, blocking, and tiling, to increase cache hit rates

and decrease bandwidth usage (Choi et al., 2010; Reguly and Giles, 2012; Yang et al., 2011).

The two most common CSR SpMV algorithms are CSR-scalar and CSR-vector. CSR-scalar assigns one thread per row and CSR-vector assigns a vector of threads to each row. On SIMD architectures, the vector size generally never exceeds a full warp (to avoid explicit synchronization between threads). A vectorized approach allows for more efficient coalesced memory accesses. A hybrid approach has been shown to be effective. This method selectively picks between CSR-scalar and CSR-vector based on the row length (Greathouse and Daga, 2014). Adaptive algorithms that group rows together by length and assign separate kernels to each group have also been explored (Ashari et al., 2014a).

Graph applications often use sparse binary adjacency matrices to represent graphs and translate graph operations to linear algebraic operations (Kepner and Gilbert, 2011). Finding the transitive closure of a graph can be done through repeated multiplication of its adjacency matrix. The transitive closure of an adjacency matrix $R$ calculates $R^+ = \underset{i \in \{1,2,\ldots\}}{\cup} R^i$, where $R^i$ is the $i^{th}$ power of the matrix. The result is $R^i$ having a non-zero between any pair of nodes connected by a path of length $i$. Thus, the union (addition/binary-or) of all $R, \ldots R^n$ will have a non-zero entry for every pair of nodes that are connected by a path of length $\leq n$. This process of unioning successive powers of $R$ can be continued until a fixed point is reached. All nodes that are connected by a path of any length will be marked in the matrix.

Bandwidth limited sparse matrix-matrix operations such as sparse matrix-matrix addition $A + B = C$ and sparse matrix-matrix multiplication $AB = C$ remain difficult to compute efficiently. These operations require creating a new sparse matrix $C$ whose entries and sparsity will depend on the sparsity patterns of $A$ and $B$, and will often have a differing number of elements than either. Current implementations generally look globally at both matrices and find the intersection patterns using temporary workspace memory, after which the new matrix $C$ can be generated (Bell and Garland, 2012; Khronos Group, 2011). This often involves format conversions that consume additional time and memory.

### 6.5.3 Dynamic Allocation and SpMV

We now present our dynamic sparse-matrix allocation method that allows for efficient dynamic updates while still maintaining fast SpMV times (King et al., 2016). Our dynamic allocation uses a *row offset array*, representing a dense array of ordered rows, and for each a fixed number of *segment* offsets. The column indices and values are stored in arrays that are logically divided into these data segments in the same way that CSR row offsets

partition the column indices and values. Each such segment is a contiguous portion of memory that stores entries within a row. Segments may contain more space than entries to allow for future insertions. The contiguous layout of entries within the set of segments for a given row is equivalent to the corresponding row in CSR format. We next illustrate how dynamic allocation is performed, after which we provide details of how DCSR operations are implemented. We then present our implementation of an improved SpMM algorithm that utilizes DCSR for asynchronous dynamic writes to the resulting $C$ matrix.

Initializing the matrix can be done in one of two ways. Either a matrix can be loaded from another format (*e.g.*, COO or CSR) or the matrix can be initialized as blank. In the latter case, each row is assigned an initial number of entries (an initial segment size) in the column indices and values arrays. The row offset array is initialized with space for $k$ segment offset pairs, with either no allocated segments or a single allocated segment of size $\mu$ per row. The latter case will consume the same amount of memory as an ELL matrix with a row width of $\mu$, except in row-major order instead of column-major order. To allow for dynamic allocation we maintain a larger memory buffer than needed and use simple bump-pointer allocation to add new segments. This allocation pointer is set to the end of the currently used space ($rows \times \mu$ in the case of a new matrix). A maximum size of memory buffer for the columns and values arrays is specified by the user. Figure 6.1 provides a illustrative comparison of CSR, HYB, and DCSR formats.

The format consists of four arrays for column indices, values, row offsets, and row sizes, in addition to a memory allocation pointer. The row offsets array functions in a similar manner to that of its CSR counterpart, except that both a beginning and ending offset are stored and space exists for up to $k$ such pairs per row. This table is encoded as a strided array where the starting and ending offsets of segment $k$ in row $i$ are indexed by $(i * 2 + k * pitch)$ and $(i * 2 + k * pitch + 1)$, respectively. The $pitch$ may be defined as a value convenient for cache performance such that $pitch \geq 2 * rows$. Each set of offsets for a given segment lies within a different cache line, which serves to increase memory aligned accesses. The number of memory segment offset pairs (the max $k$) is an adjustable parameter specified at matrix construction. The column indices and values correspond 1:1, just as in CSR. Unlike CSR, however, there may be more than one memory segment assigned to a given row, and the segments need not be contiguous. As the last segment for a row may not be full, the actual row sizes are maintained so the used portion of each segment is known.

Explicitly storing row sizes allows for optimization techniques such as the adaptive binning strategy used in *adaptive CSR* (ACSR) (Ashari et al., 2014a). This optimization

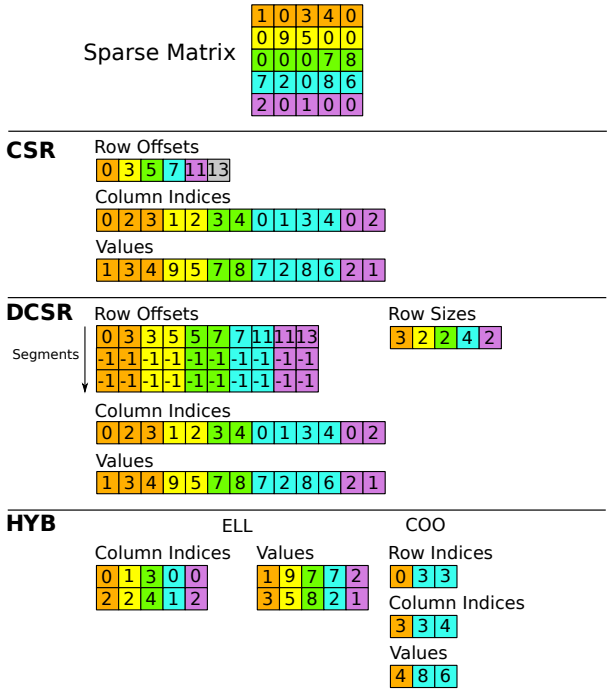



**Figure 6.1**. Comparison of CSR, DCSR, and HYB formats.

implements customized kernels to process bins of specified row-lengths. We make use of this optimization by binning rows together based on row size before SpMV or SpMM operations. Each row is given a bin label based on its size (1, 2-3, 4-8, 9-16, 17-32, ...). A permuted set of row indices is created by sorting according to these bin labels. Bin-specific kernels are launched with these permuted indices on separate streams which allows each kernel to easily access the rows that it needs to process without scanning over the matrix.

When inserting new elements within a row, the last allocated segment for that row is located and if space is available, the new elements are inserted in a contiguous fashion just after current entries. If that segment does not have enough room, a new segment will be allocated with the appropriate size plus an additional amount $\alpha$. The $\alpha$ value represents additional "slack space" and allows for a greater number of entries to be inserted without the creation of a new segment. If dynamic updates follow a power-law distribution, there will be a higher probability of additional entries being inserted into longer rows. Although we experimented with setting $\alpha$ to be a factor of the previous segment size, for our tests, we settled on a value of $\mu$ (average row size of matrix). When a new segment is allocated, the memory allocation pointer is atomically increased by the size of the new segment. A hard limit on these additions, before defragmentation is required, is fixed by the number of

segments $k$. The defragmentation operation always reduces the number of segments in each row to one, which allows the format to scale to an arbitrary number of allocations.

Algorithm 1 provides pseudo-code illustrating new segment allocation. This allocation function can be parallelized across rows, as each vector of threads will execute this function on a different row. Within a row, a vector of threads operate together to add new elements into matrix A from an array of values B (B_offsets, B_cols, B_vals). The segments could be of variable length, so the total size is computed by looping over the segments and summing the differences of the starting and ending offsets ($A\_start$, $A\_end$). The current available memory is calculated by computing the difference of the final segment ending offset and index of the last element ($A\_end - A\_start$). If there is enough room, the elements are inserted into the remaining space, otherwise a new segment must be allocated. This is performed by atomically incrementing the memory offset pointer to allocate a new segment of memory of size equal to new elements minus the remaining free space plus an $\alpha$ value. The returned address *addr* is the beginning offset of the new segment of size *size*. Afterward, the new elements are inserted via Algorithm 2.

When inserting new elements into the matrix, it is possible that duplicate non-zero entries (*i.e.*, two or more entries with the same row and column index) will be added. Duplicate entries are handled in one of two ways. The first method is to simply let them accumulate, which does not pose a problem for many operations. SpMV operations are tolerant of duplicate entries due to the distributive property of the inner product and will yield the same result to within floating point tolerance. For binary matrices, the row-vector inner products will produce the same result irrespective of duplicate non-zeros. A second solution is to perform a segmented reduction on the entries after sorting by row and column, which combines all entries with matching row and column indices into a single entry. This full reduction is generally not needed when performing only SpMV and addition operations. Sparse matrix-matrix multiplication (SpMM) operations may cause significant fill-in which would require such a reduction to be performed. In our SpMV tests, we let the values accumulate for all formats as they do not hinder the SpMV operations that are performed.

Algorithm 2 provides pseudo-code for the insertion operation. A vector of threads will operate together to add the elements into the segments. After a segment is full, the next segment indices are retrieved from the offsets table whose starting and ending offsets are $A\_start$ and $A\_end$, respectively. Column indices and values are copied from $B\_cols$ and $B\_vals$ to their respective locations in the $A$ matrix. After this is complete, a single thread will update the row sizes array to reflect the new size.

**Algorithm 1:** Allocate Segments

**Input:** sizes, offsets, Aj, Ax, B_offsets, B_cols, B_vals
**Output:** sizes, offsets, Aj, Ax

```
1  row ← vid ; // vector ID
2  while row < n_rows do
3      sid ← 0 ; // segment index
4      rl ← sizes[row] ; // row length
5      idx ← 0 ; // thread row index
6      start ← offsets[row * 2] ; // starting segment offset
7      end ← offsets[row * 2 + 1] ; // ending segment offset
8      free_mem ← 0;
9      B_start ←B_offsets[row * 2];
10     B_end ←B_offsets[row * 2 + 1];
11     rlB ← B_row_end − B_row_start;
12     if rlA ≥ 0 then
13         while A_idx < rlA do
14             idx ← idx + (A_end − A_start);
15             if idx < rlA then
16                 sid ← sid + 1;
17                 A_start ← offsets[sid*pitch+row * 2];
18                 A_end ← offsets[sid*pitch+row * 2 + 1];
19         idx ← A_end + rlA − idx;
20     else
21         idx ← A_start;
22     free_mem ← A_end − A_start;
23     if lane = 0 AND free_mem < rlB AND rlB > 0 then
           // allocate new space
24         size ← rlB − free_mem + α;
25         addr ← atomicAdd(sizes[n_rows], size);
           // allocate new row segment
26         offsets[(sid + 1)*pitch + row * 2] ← addr;
27         offsets[(sid + 1)*pitch + row * 2 + 1] ← addr + size;
       // Allocate new entries (Algorithm 2)
28     Insert_Elements();
29     row ← row + num_vectors;
```

An SpMV operation works as follows. The first pair of segment offsets is fetched. The entries within the corresponding segment are multiplied by the appropriate values in $x$ according to the algorithm being used (CSR-scalar, CSR-vector, etc.). If the row size is greater than the capacity of the current memory segment, the next pair of offsets is fetched. If the size of the current segment plus the running sum of the previous segment sizes is greater than or equal to the row size, this is the final segment of the row. In case the final

**Algorithm 2:** Insert Elements

**Input:** sizes, offsets, Aj, Ax, B_cols, B_vals
**Output:** sizes, Aj, Ax

```
1  B_idx ← B_start+lane ; // add thread lane
2  while B_idx < B_end do
3      if idx ≥ A_end then
4          pos ← idx − A_end;
5          sid ← sid + 1;
6          A_start ← offsets[sid∗pitch+row ∗ 2];
7          A_end ← offsets[sid∗pitch+row ∗ 2 + 1];
8          idx ← A_start + pos;
9      Aj[idx] ← B_cols[B_idx];
10     Ax[idx] ← B_vals[B_idx];
11     B_idx ← B_idx+ VECTOR_SIZE;
12     idx ← idx+ VECTOR_SIZE;
13 if lane = 0 then
14     sizes[row] ← sizes[row] + rlB;
```

segment is not full, the location of the last entry can be determined by the difference of the row size and the running sum. This process continues until the entire row has been read.

As the matrix accumulates more segments, SpMV performance decreases slightly. A fixed number of segments also means this process cannot continue forever. Our solution to both problems is to implement a defragmentation operation that compacts all the entries within the column indices and values arrays, eliminating empty space. This defragmentation step combines all the segments in a row into a single segment that compactly stores the entire row. This operation may be invoked periodically, or more conservatively when a row has reached its maximum capacity of segments. In practice, we do the latter and set a flag when any row reaches its maximum segment count. At this point, we consider defragmentation to be required. Algorithm 3 illustrates the SpMV operation. This is performed in a similar fashion to CSR-vector, except that there is an outer loop over the segments.

Defragmentation performs the equivalent to a sort by row operation on the entries of the matrix; we formulated a method that does not require an actual sort and is significantly faster than doing so. Since we explicitly store row sizes, we perform a prefix-sum operation on them to calculate the new row offsets in a compacted CSR form. The entries are then shuffled from their current indices to their new indices in newly allocated column indices and values buffers, after which we set a pointer in our data structure to these new arrays and free the old buffers (shallow copy). By using the knowledge of the row sizes to compute resulting offsets and indices, we eliminate the need to do any comparisons in this operation, which

**Algorithm 3:** DCSR SpMV

```
   Input: sizes, offsets, Aj, Ax, x, y
   Output: y
 1 tid ← thread index ; // thread ID
 2 lane ← tid % Vec_Size ; // lane ID
 3 vid ← tid / Vec_Size ; // vector ID
 4 for row ← vid to num_rows, row += num_vecs do
 5     idx ← 0 ; // thread row index
 6     rl ← sizes[row] ; // row length
 7     sid ← 0 ; // segment index
 8     while idx < rl do
 9         start ← offsets[sid*pitch + row * 2];
10         end ← offsets[sid*pitch + row * 2 + 1];
           /* accumulate local sums                                    */
11         for j ← start to end, j += Vec_Size do
12             sum += Ax[j] * x[Aj[j]];
13         idx += (end - start);
14     y[row] = sum;
```

greatly improves performance. The defragmentation process is described by Algorithm 4.

Figure 6.2 illustrates an example of inserting new elements into a DCSR matrix. Initially, the matrix has four populated rows with the memory allocation pointer being 16. Row 0 can insert 1 additional entry in its current segment before a new segment would need to be allocated. Rows 1 and 2 have enough room for two additional entries, but row 3 is full. Figure 6.2 shows a set of new entries that are inserted into rows 0, 2, and 3. In this case a new segment of size 4 is allocated for row 0 and row 3. The additional segments need not be consecutive nor in order of row since the exact offsets are stored for each segment. Finally, the defragmentation operation computes new segment offsets from the row sizes. The entries are shuffled to their new indices which results in a single compacted segment for

**Algorithm 4:** Defragment DCSR

```
   Input: sizes, offsets, Aj, Ax
   Output: offsets, Aj, Ax
   /* prefix sum on row sizes                                        */
 1 exclusive_scan(sizes, temp_offsets);
 2 new T_cols(size(Aj)), new T_vals(size(Ax));
 3 CompactIndices(T_cols, T_vals, temp_offsets, Aj, Ax, offsets, sizes);
   /* shallow copy, old arrays deleted                               */
 4 Aj = &T_cols, Ax = &T_vals;
 5 SetRowOffsets(offsets, sizes, temp_offsets);
```


**Figure 6.2**. Illustration of insertion and defragmentation operations with DCSR.

each row.

As CSR is the most commonly used sparse matrix format, we designed DCSR to be compatible with CSR algorithms and to allow for easy conversion between the formats. Minimal overhead is required to convert from CSR to DCSR and vice versa. When converting from CSR to DCSR, the column indices and values arrays are copied directly. For the row offsets array, the $i^{th}$ element is copied to indices $i*2-1$ and $i*2$ for all elements except the first and last one. A simple subtraction must be performed to calculate the row sizes from the row offsets. Converting back is equally simple, assuming the matrix is first defragmented; the column indices and values arrays are copied back, and the starting segment offset from

each row is copied into the row offsets array.

### 6.5.4 Sparse Matrix-Matrix Multiplication (SpMM)

It is a difficult task to efficiently compute $C = AB$ for sparse matrices in parallel. The sequential sparse matrix-matrix multiplication algorithm is not suitable for fine-grained parallelization. Sequential algorithms are efficient, but they rely on a large amount of (per thread) temporary storage. Specifically, to compute the sparse product $C = AB$, the sequential methods use $O(N)$ additional storage, where N is the number of columns in C. The parallel approach to sparse matrix-matrix multiplication is formulated in terms of highly scalable parallel primitives with no such limitations. As a result, a straightforward parallelization of the sequential scheme requires $O(n)$ storage per thread, which is not possible when using tens of thousands of independent threads of execution. Although it is possible to construct variations of the sequential method with lower per-thread storage requirements, any method that operates on the granularity of matrix rows (*i.e.*, distributing matrix rows over threads), requires a nontrivial amount of per-thread state and suffers load imbalances for certain input (Bell et al., 2012).

The standard algorithm for parallel SpMM that exposes fine-grained parallelism is:

1. Expansion of $A * B$ into an intermediate coordinate format $T$.
2. Sorting of T by row and column indices to form $\hat{T}$.
3. Compression of $\hat{T}$ by summing duplicate values for each matrix entry.

**Example 22.** $T$ and $\hat{T}$ are given for $C = AB$, where

$$A = \begin{bmatrix} 1 & 0 & 3 \\ 2 & 2 & 0 \\ 0 & 7 & 9 \end{bmatrix}, B = \begin{bmatrix} 4 & 3 & 7 \\ 0 & 5 & 0 \\ 2 & 0 & 8 \end{bmatrix},$$

$$C = \begin{bmatrix} 10 & 3 & 31 \\ 8 & 16 & 14 \\ 18 & 35 & 72 \end{bmatrix}$$

$$T = \begin{bmatrix} 0, & 0, & 4.0 \\ 0, & 1, & 3.0 \\ 0, & 2, & 7.0 \\ 0, & 0, & 6.0 \\ 0, & 2, & 24.0 \\ 1, & 0, & 8.0 \\ 1, & 1, & 6.0 \\ 1, & 2, & 14.0 \\ 1, & 1, & 10.0 \\ 2, & 1, & 35.0 \\ 2, & 0, & 18.0 \\ 2, & 2, & 72.0 \end{bmatrix} \hat{T} = \begin{bmatrix} 0, & 0, & 4.0 \\ 0, & 0, & 6.0 \\ 0, & 1, & 3.0 \\ 0, & 2, & 7.0 \\ 0, & 2, & 24.0 \\ 1, & 0, & 8.0 \\ 1, & 1, & 6.0 \\ 1, & 1, & 10.0 \\ 1, & 2, & 14.0 \\ 2, & 0, & 18.0 \\ 2, & 1, & 35.0 \\ 2, & 2, & 72.0 \end{bmatrix}$$

All three stages of the algorithm expose fine-grained parallelism that the GPU can take advantage of. The algorithm can be formulated in terms of efficient data-parallel computations — gather, scatter, scan, sort, etc. Like the sequential algorithm, this formulation is work efficient. It computes the exact number of partial products required for each non-zero without performing any additional operations with zero entries. It has the same computational complexity as the sequential method $O(nnz(T))$. The complexity is proportional to the size of the intermediate format $T$, and the work required at each stage is linear with respect to $T$. This process results in a relatively even load balancing across the GPU regardless of the sparsity patterns of the input matrices.

A limitation of this method is that the memory required to store the intermediate format is potentially large. If $A$ and $B$ are both square, $n \times n$ matrices with exactly K entries per row, then $O(nK^2)$ bytes of memory are needed to store T. Since the input matrices are generally large themselves ($O(nK)$ bytes), it is not always possible to store a $K$-times larger intermediate result in memory. In the limit, if A and B are dense matrices (stored in sparse format), then $O(n^3)$ storage is required. In such a case, the matrix-matrix multiplication $C = AB$ can be decomposed into several smaller operations that are computed in a workspace

of bounded size. The resulting slices are then concatenated together to produce the final result. This technique introduces some overhead, but in practice, it is relatively small as the workspace can be sized appropriately to saturate the device.

Our implementation of SpMM follows the same principles as the general algorithm, but we assign specialized kernels to process rows grouped by size. This algorithm allows for a more efficient use of shared memory when performing the sort and reduction operations. DCSR allows for asynchronous dynamic memory allocations when storing the rows products into $C$. This property of DCSR allows computation of the rows to be handled asynchronously. In the standard algorithm, the result of each previous row is required to know the offset when writing the final result into $C$. We precompute the number of partial products per row $i$ following:

$$\sum_{k=1}^{ARS_i} BRS_j$$

where $ARS_i$ is the number of entries in row $i$ of matrix $A$, and $j$ is the column index of element $a_{i,j}$. We then assign specific kernels, based on this row size, to process rows of length 1-32, 33-64, 65-128, 129-256, 257-512, 513-1024, 1025-2048, and 2049+.

The kernels process a row by computing the partial products, sorting them by column index, and reducing them before storing them in the resulting $C$ matrix. Since this is done on a per row basis, the row is implicit and we need only store the column indices and values for the sorting and reduction phases. For all kernels except the 2049+ kernel, the operations are computed within shared memory on the GPU, which provides a significant performance improvement over global memory. For the 2049+ kernel, we use dynamic parallelism to assign a compute kernel to each row, which performs these operations using global memory.

### 6.5.5 Experimental Results

To benchmark SpMV, SpMM, update, and conversion performance, we used a node with an Intel Xeon E5-2640 processor running at 2.50GHz, 128GB of memory, and a NVIDIA Tesla K20c GPU. We compiled using $g++$ 4.7.2, CUDA 7.5, CUSP 0.5.1, and Thrust 1.8.1, comparing our method against modern implementations in CUSP (Bell and Garland, 2012) and cuSPARSE (NVI, 2010). Table 6.1 provides a list of the matrices that we used in our tests as well as their sizes, number of non-zeros, and row-entry distributions. All the matrices can be found in the University of Florida sparse-matrix database (Davis and Hu, 2011).

Memory consumption is a major concern for sparse-matrix formats, as one of the primary reasons for eliminating the storage of zeros is to reduce the memory footprint. The ELL

**Table 6.1**. Matrices used in tests. NNZ: total number of non-zeros, $\mu$: average row size, $\sigma$: standard deviation of row sizes, Max: maximum row size

| Matrix | Abbr. | NNZ | Rows \ Cols | $\mu$ \ $\sigma$ \ Max |
|---|---|---|---|---|
| amazon-2008 | AMA | 5M | 735K | 7 \ 4 \ 10 |
| cnr-2000 | CNR | 3M | 325K | 9 \ 21 \ 2716 |
| dblp-2010 | DBL | 807K | 326K | 2 \ 4 \ 154 |
| enron | ENR | 276K | 69K | 3 \ 28 \ 1392 |
| eu-2005 | EU2 | 19M | 862K | 22 \ 29 \ 6985 |
| flickr | FLI | 9M | 820K | 11 \ 87 \ 10K |
| hollywood-2009 | HOL | 57M | 1139K | 50 \ 160 \ 6689 |
| in-2004 | IN2 | 16M | 1382K | 12 \ 37 \ 7753 |
| indochina-2004 | IND | 194M | 7414K | 26 \ 216 \ 6985 |
| internet | INT | 207K | 124K | 1 \ 4 \ 138 |
| kron-18 | KRO | 10M | 262K | 40 \ 261 \ 29K |
| ljournal-2008 | LJO | 79M | 5363K | 14 \ 37 \ 2469 |
| rail4284 | RAL | 11M | 4K \ 1M | 2633 \ 4K \ 56K |
| soc-LiveJournal1 | SOC | 68M | 4847K | 14 \ 35 \ 20K |
| webbase-1M | WEB | 3M | 1000K | 3 \ 25 \ 4700 |
| wikipedia-2005 | WIK | 19M | 1634K | 12 \ 31 \ 4970 |

component of HYB is best suited to store rows with an equal number of entries. If there is a large variance in row size, much of the ELL portion may end up storing zeros, which is inefficient. We provide a comparison of memory consumption for HYB, DCSR (using 2, 3, and 4 segments), and CSR formats in Table 6.2. We compute the storage size of the HYB format using an ELL width equal to the average number of non-zeros per row ($\mu$) for the given matrix. CSR has the smallest memory footprint since its row indices have been compressed to the number of rows in the matrix. We see that DCSR has a significantly smaller memory footprint in almost all test cases. Test cases such as AMA and DBL have lower memory consumption for HYB than for DCSR (with 3 and 4 segments), because these matrices have a low row size variance. DCSR with 4 segments uses 20% less memory on average than HYB.

Conversion times between formats are often a key factor when determining the efficacy of a particular format. High conversion times can be a significant hindrance to efficient performance. Architecture-specific formats may provide better performance, but unless the rest of the code base uses that format, the conversion time must be accounted for. We provide the overhead required to convert to and from CSR and COO matrices in Table 6.3. The conversion times have been normalized against the time required to copy CSR $\rightarrow$ CSR. The conversion times to DCSR are only slightly higher compared to that of CSR. HYB requires significant overhead as the entries must first be distributed throughout the ELL

**Table 6.2**. Comparison of memory consumption between HYB, CSR, and DCSR formats. Size of HYB is listed in bytes (using ELL width of $\mu$), and sizes for DCSR and CSR are listed as a percent of the HYB size.

| Matrix | HYB size | DCSR 2 segs. | DCSR 3 segs. | DCSR 4 segs. | CSR |
|---|---|---|---|---|---|
| AMA | 54M | 0.924 | 1.026 | 1.128 | 0.77 |
| CNR | 47M | 0.626 | 0.679 | 0.732 | 0.547 |
| DBL | 12M | 0.86 | 1.052 | 1.245 | 0.572 |
| ENR | 4M | 0.653 | 0.762 | 0.871 | 0.489 |
| EU2 | 236M | 0.675 | 0.703 | 0.731 | 0.633 |
| FLI | 160M | 0.546 | 0.585 | 0.624 | 0.487 |
| HOL | 859M | 0.531 | 0.541 | 0.551 | 0.516 |
| IN2 | 229M | 0.654 | 0.7 | 0.746 | 0.585 |
| IND | 2791M | 0.571 | 0.591 | 0.612 | 0.541 |
| INT | 4M | 0.761 | 0.969 | 1.177 | 0.449 |
| KRO | 171M | 0.493 | 0.505 | 0.516 | 0.475 |
| LJO | 1152M | 0.594 | 0.63 | 0.665 | 0.541 |
| RAL | 149M | 0.577 | 0.577 | 0.577 | 0.576 |
| SOC | 1009M | 0.595 | 0.631 | 0.668 | 0.54 |
| WEB | 40M | 0.966 | 1.155 | 1.344 | 0.682 |
| WIK | 276M | 0.635 | 0.68 | 0.725 | 0.567 |

portion and the remaining overflow entries distributed into the COO portion.

#### 6.5.5.1 Matrix Updates

To measure the speed of dynamic updates, we ran two series of tests, which involved streaming updates and iterative updates. In the streaming updates test, we incrementally build up the matrix by continuously inserting new entries. The elements are first buffered into three arrays, representing the rows indices, column indices, and values. We initialize the matrix sizes according to the average number of non-zeros for the given input. The entries are then added in a streaming parallel fashion to the matrices.

Updating a HYB matrix first requires checking the ELL portion, and if the row in question is full, inserting the new entry into the COO portion. Any updates to the COO portion require atomic operations to ensure synchronous writes between multiple threads. These atomic updates are prohibitive for fast parallel updates as all threads are contending to insert entries onto the end of the COO matrix.

Updating a DCSR matrix requires finding the last occupied (current) segment within a row. If that segment is not full, the new entry is added into it and the row size is increased.

**Table 6.3**. Comparison of relative conversion times. Conversions are normalized against time to copy CSR→CSR.

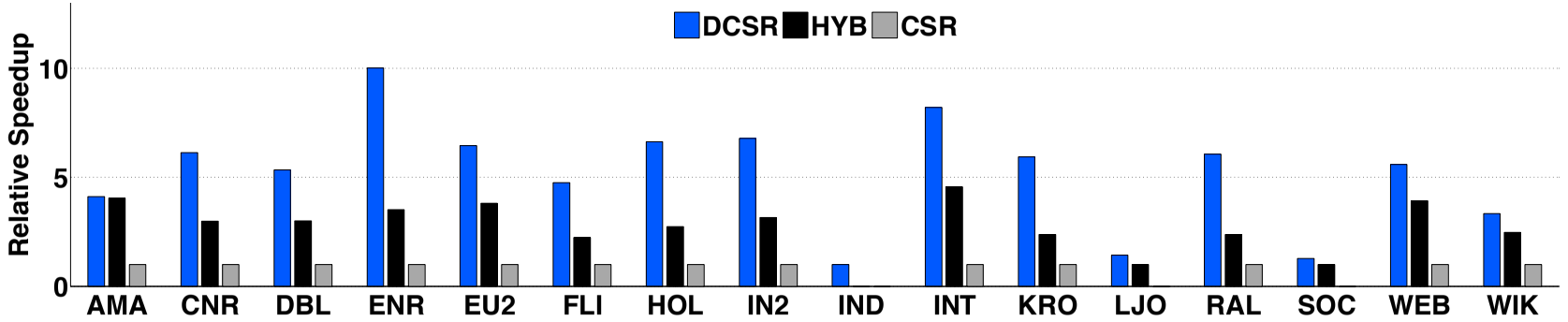
| From<br>To | COO<br>CSR | COO<br>DCSR | COO<br>HYB | CSR<br>DCSR | CSR<br>HYB | DCSR<br>CSR |
|---|---|---|---|---|---|---|
| AMA | 2.93 | 3.03 | 9.22 | 1.06 | 9.25 | 0.9 |
| CNR | 2.24 | 2.62 | 14.84 | 1.04 | 13.62 | 0.87 |
| DBL | 4.34 | 5.74 | 18.07 | 1.17 | 16.83 | 1.1 |
| ENR | 5.56 | 5.95 | 27.15 | 1.29 | 26.95 | 1.14 |
| EU2 | 2.1 | 2.29 | 16.08 | 1.06 | 15.67 | 0.99 |
| FLI | 2.13 | 2.5 | 23.29 | 1.06 | 19.74 | 0.96 |
| HOL | 1.82 | 1.9 | 20.37 | 1.01 | 20.3 | 0.99 |
| IN2 | 2.15 | 2.42 | 18.12 | 1.06 | 18.15 | 0.98 |
| IND | 1.93 | 1.98 | $\infty$ | 1.03 | $\infty$ | 1.01 |
| INT | 12.07 | 13.74 | 21.38 | 1.3 | 15.12 | 1.0 |
| KRO | 1.78 | 2.09 | 24.01 | 1.0 | 20.14 | 0.91 |
| LJO | 2.09 | 2.19 | 19.96 | 1.02 | 19.97 | 0.98 |
| RAL | 1.73 | 2.03 | 20.67 | 1.0 | 17.97 | 0.91 |
| SOC | 2.22 | 2.35 | 20.47 | 1.06 | 20.41 | 1.01 |
| WEB | 2.89 | 3.19 | 11.45 | 1.16 | 11.56 | 0.86 |
| WIK | 2.18 | 2.42 | 20.13 | 1.07 | 20.11 | 0.98 |

When the current segment for a row fills up, a new segment is allocated dynamically. Since atomic operations are required only for the allocation of new segments, and not for each individual element, synchronization overhead is kept low. By allowing for dynamically sized slack space within a row, we dramatically reduce the number of atomic operations that are required to allocate new entries. In this way, DCSR was designed to be updated in an efficient parallel manner.

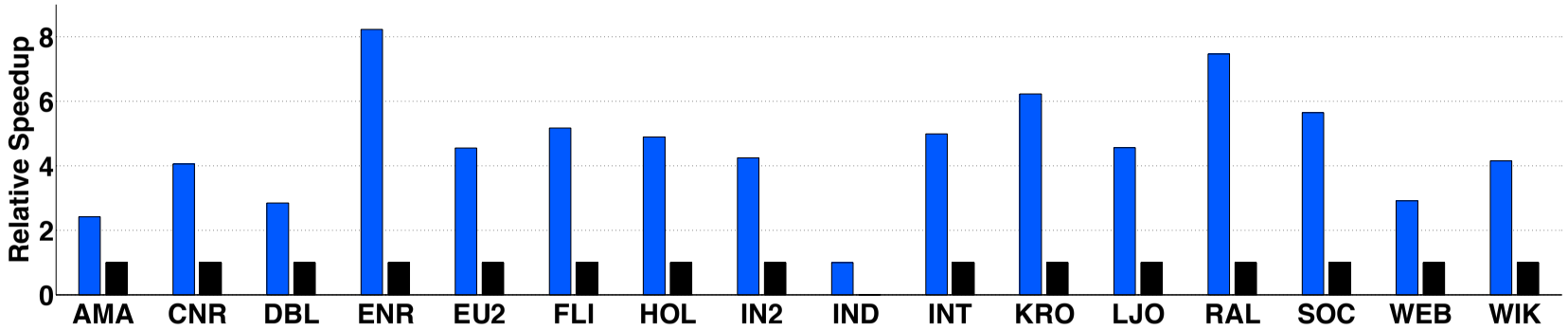
The number of segments, initial row width, and $\alpha$ value can be tuned for the problem to give a reasonable limit on updates. In our tests, we used four segments and an $\alpha$ value of $\mu$ (average row size of the matrix). When a row nears its limit, a defragmentation is required in order to reduce that row to a single segment.

Figure 6.3 provides the results of our iterative and streaming matrix update tests. We do not compare to CSR in the latter case, since it is not possible to dynamically add entries without rebuilding the matrix. This operation only loads the matrix and does not perform any insertion checks. DCSR saw an average speedup of 4.8× over HYB with streaming updates. In the case of IND, only DCSR was able to perform the operation within memory capacity.

We also executed an iterative update test to compare the ability of the formats to perform




(a) Relative speedup of DCSR compared to HYB for iterative updates with SpMV operations. The speedup is compared to a normalized CSR baseline.




(b) Relative speedup of DCSR compared to HYB for streaming matrix operations.

**Figure 6.3**. A comparison of update operations for DCSR and HYB.

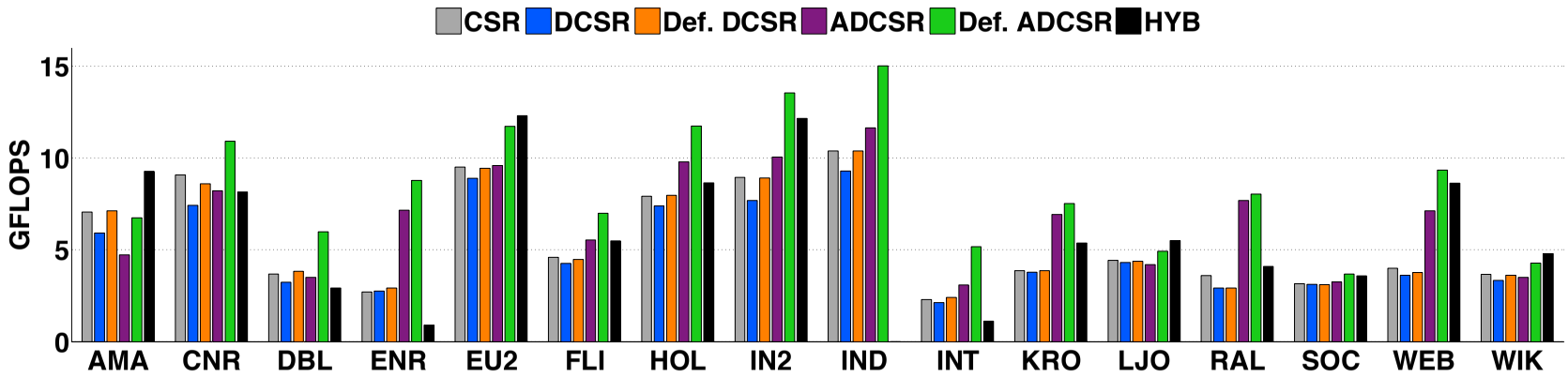
a combination of dynamic updates and SpMV operations. This test is analogous to what would be done in a graph application (such as CFA) where the graph is updated at periodic intervals. In the iterative updates test, we perform a series of iterations consisting of a matrix addition operation ($A = A + B$) followed by several SpMV operations $Ax = y$. Part (a) of Figure 6.3 provides the results for our iterative updates. Within each iteration, the matrix is updated with an additional 0.2% random non-zeros followed by 5 SpMV operations, which is repeated 50 times. This process yields a total increase of 10% to the number of non-zeros. We compare the DCSR and HYB results to a normalized CSR baseline. In the CSR case, a new matrix must be created to update the original matrix which causes a significant amount of overhead (in terms of computation and memory). In the cases of LJO and SOC, CSR was not able to complete within memory capacity, so we normalized against HYB.

DCSR shows significant improvement over HYB on streaming updates in all test cases (in some by as much as 8×). DCSR also outperforms HYB in all test cases on iterative updates, and in some cases by as much as 2.5×. The Amazon-2008 matrix has a low standard deviation, and the majority of its entries fit nicely into the ELL portion, which greatly speeds up SpMV operations. However, even in this case, DCSR slightly outperforms HYB on iterative updates due to having lower overhead for defragmentation. In all other cases DCSR exhibits noticeable performance improvements over HYB and CSR.

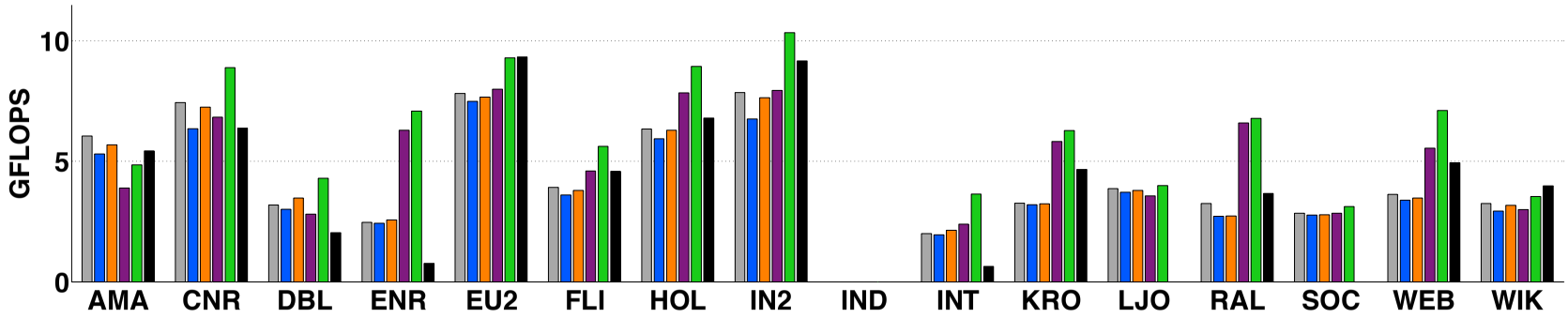
#### 6.5.5.2 SpMV Results

In the SpMV tests, we take the same set of matrices and perform SpMV operations with randomly generated dense vectors. We performed each SpMV operation 100× times and averaged the results. Figure 6.4 provides the results for these SpMV tests run using both single- and double-precision floating-point arithmetic. We implemented the adaptive binning optimization (ACSR) outlined in Ashari et al. (2014a), which we labeled ADCSR. This optimization requires relatively little overhead and provides noticeable speed improvements by using specialized kernels on bins of rows with similar row sizes. In these tests we compare across several variants of our format, including DCSR, defragmented DCSR, ADCSR, and defragmented ADCSR, in addition to standard implementations of HYB and CSR.

The fragmented DCSR times are 8% slower than the defragmented DCSR times on average. When the DCSR format is defragmented, it sees SpMV times competitive with those of CSR (1% slower on average). With the adaptive binning optimization applied, we see that ADCSR outperforms HYB in many cases. ADCSR performs 9% better on average than HYB across our benchmarks.




(a) Single precision.




(b) Double precision.

**Figure 6.4**. FLOP ratings of SpMV operations for CSR, DCSR, and HYB.

#### 6.5.5.3 Postprocessing Overhead

Postprocessing overhead is a concern when dealing with dynamic matrix updates. Dynamic segmentation allows for DCSR to be updated with new entries without requiring the entries to be defragmented. SpMV operations can be performed on the DCSR format regardless of the number and order of segments, in contrast to HYB matrices where a sort is required anytime an entry is added that overflows into the COO portion. The SpMV operation for HYB matrices assumes the COO entries are sorted by row (without this property, the COO SpMV would be dramatically slower). Table 6.4 provides postprocessing times for HYB and DCSR formats relative to a single SpMV operation. In the case of IND, HYB was unable to sort and update due to insufficient memory (represented as $\infty$).

The defragmentation operation gives us an opportunity to internally order rows by row-size at no additional cost. Our defragmentation algorithm is similar to the row sorting technique illustrated in Kreutzer et al. (2013), although we use a global sorting scope as opposed to a localized one. Because we explicitly manage segments within the columns and values arrays by both starting and ending index, the internal order of segments may be

**Table 6.4**. Overhead of DCSR defragmentation and HYB sorting is measured as the ratio of one operation against a single CSR SpMV. Update time is measured as the ratio of 1000 updates to a single CSR SpMV.

| Matrix | DCSR defrag | HYB sort | DCSR update | HYB update |
|---|---|---|---|---|
| AMA | 3.9 | 2.12 | 2.02 | 4.89 |
| CNR | 5.13 | 6.75 | 3.77 | 15.26 |
| DBL | 5.69 | 4.66 | 3.6 | 10.23 |
| ENR | 5.49 | 8.0 | 2.21 | 18.2 |
| EU2 | 2.32 | 4.28 | 2.65 | 12.05 |
| FLI | 1.58 | 4.22 | 1.94 | 10.01 |
| HOL | 1.54 | 5.57 | 2.55 | 12.45 |
| IN2 | 2.58 | 5.85 | 3.14 | 13.34 |
| IND | 2.15 | $\infty$ | 3.36 | $\infty$ |
| INT | 6.74 | 6.19 | 1.76 | 8.78 |
| KRO | 1.02 | 3.43 | 1.82 | 11.3 |
| LJO | 1.45 | 3.02 | 1.34 | 6.1 |
| RAL | 0.72 | 2.04 | 1.82 | 13.61 |
| SOC | 1.05 | 3.74 | 1.02 | 5.74 |
| WEB | 2.65 | 1.93 | 2.54 | 7.39 |
| WIK | 1.39 | 2.54 | 1.32 | 5.49 |

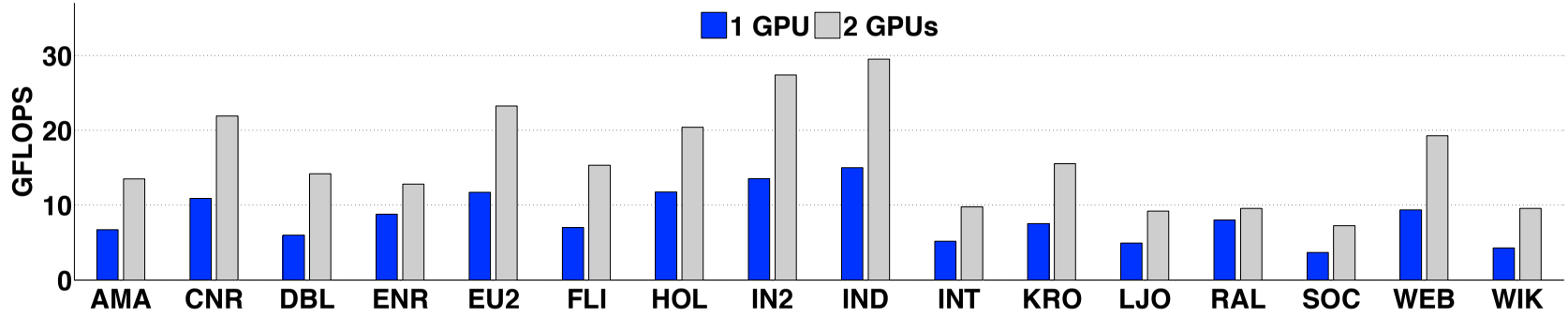
changed arbitrarily, and this permutation remains invisible from the outside. To accomplish this optimization, we permute row sizes according to the permuted row indices (which have already been binned and sorted by row size). The permuted row sizes can then be used to create new offsets for the monolithic segments produced by defragmentation. The column and value data can be internally reordered by row size at no additional cost. We observed this internal reordering to provide a noticeable SpMV performance improvement of 12%. This improvement is from an increased cache-hit rate via better correlation between bin-specific kernels and the memory they access.

The DCSR defragmentation incurs a lower overhead than HYB sort because entries can be shuffled to their new index without a sort operation. A DCSR defragmentation step is 2× faster on average than the HYB sorting step. More importantly, this is required infrequently, while HYB sorting must be performed at every insertion, which means that DCSR requires significantly lower total postprocessing overhead.

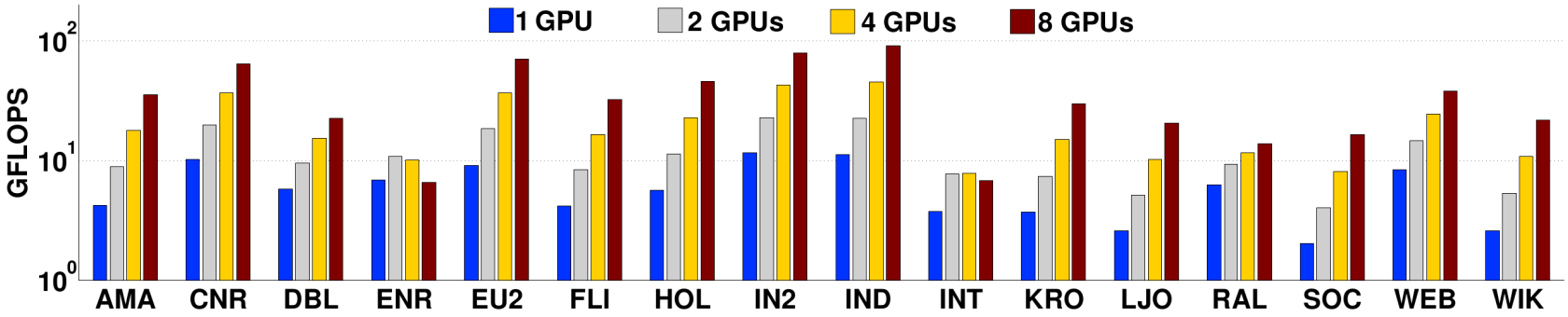
#### 6.5.5.4 Multi-GPU Implementation

DCSR can be effectively mapped to multiple GPUs. The matrix can be partitioned across $n$ devices by dividing rows between them (modulo n) after sorting by row size. This provides a relatively even distribution of non-zeros between the devices. Figure 6.5 provides scaling results for DCSR across two Tesla K20c GPUs and up to eight Tesla M2090 GPUs. We see an average speed up of 1.93× for the single precision and 1.97× across the set of test matrices. The RAL matrix sees a smaller performance gain due to our distribution strategy of dividing up the rows. The added parallelism is split across rows but, in this case, the matrix has few rows and many columns. We see nearly linear scaling for most test cases.

For the matrices INT and ENR, we see reduced scaling due to small matrix sizes. In these cases the kernel launch times account for a significant portion of the total time due to a relatively small workload. The total compute time can be approximately represented as $c + \frac{x}{n}$, where $c$ is the kernel launch overhead and the workload $x$ is divided amongst $n$ devices (assuming $x$ can be fully parallelized). As the number of devices increases, the work per device decreases while the kernel launch time remains constant. In our tests, we perform 100× iterations of each kernel, which leads to poor scaling performance on small matrices. We performed additional tests where we move the iterations into the kernel itself and call the kernel once, eliminating the additional kernel launch times. We see scaling for the INT matrix of 1.94×, 3.55×, and 6.03× and for the ENR matrix, we see scaling of 1.80×, 2.70×, and 3.76× for 2, 4, and 8 GPUs, respectively. This suggests the poor performance of those cases was due to having less work relative to the kernel launch overhead.




(a) One or two Tesla K20c GPUs.




(b) One, two, four, or eight Tesla M2090 GPUs.

**Figure 6.5**. Scaling results for SpMV operations.

**Table 6.5**. List of matrices used for AMG tests.

| Matrix | Abbr. | Unknowns | Non-zeros |
|---|---|---|---|
| 2D Poisson 5pt | 2-5-a | 262144 | 1310720 |
| 2D Poisson 9pt | 2-9-a | 262144 | 2359296 |
| 3D Poisson 7pt | 3-7-a | 262144 | 1810432 |
| 3D Poisson 27pt | 3-27-a | 262144 | 6859000 |
| 2D Poisson 5pt | 2-5-b | 1048576 | 5238784 |
| 2D Poisson 9pt | 2-9-b | 1048576 | 9424900 |
| 3D Poisson 7pt | 3-7-b | 2097152 | 14581760 |
| 3D Poisson 27pt | 3-27-b | 2097152 | 55742968 |

#### 6.5.5.5 SpMM

We test the efficiency of our SpMM method through its application to algebraic multigrid. We compare our method to a similar version that computes SpMM using CSR and COO matrices. AMG can be formulated in terms of SpMM, SpMV, and primitive parallel operations. Algorithm 5 illustrates the structure of the AMG preconditioner setup phase of AMG given a sparse matrix $A$ and a set of vectors $B$. In our tests, we used a constant vector which is a common default. The $(R_k A_k P_k)$ operation computes the Galerkin product of the three matrices using SpMM by first computing $A * P = AP$ followed by $R * AP = RAP$.

**Algorithm 5:** AMG Setup

**Input:** $A$, $B$
**Output:** $A_0,\ldots,A_M$, $P_0,\ldots,P_M$
$A_0 \leftarrow A$, $B_0 \leftarrow B$;
**for** $k = 0,\ldots,M$ **do**
  $C_k \leftarrow \text{strength}(A_k)$;
  $Agg_k \leftarrow \text{aggregate}(C_k)$;
  $T_k, B_{k+1} \leftarrow \text{tentative}(Agg_k, B_k)$;
  $P_k \leftarrow \text{prolongator}(A_k, T_k)$;
  $R_k \leftarrow P_k^T$;
  $A_{k+1} \leftarrow (R_k A_k P_k)$;

We compare the results for AMG on 2D and 3D Poisson problems with Dirichlet boundary conditions. It is known that AMG performs well as a preconditioner on such problems which allows us to focus on the merits of the SpMM method rather than on whether AMG is suited for the problem. Table 6.5 lists the set of matrices used in our tests as well as the number of unknowns and non-zeros. These tests were all computed with double precision.

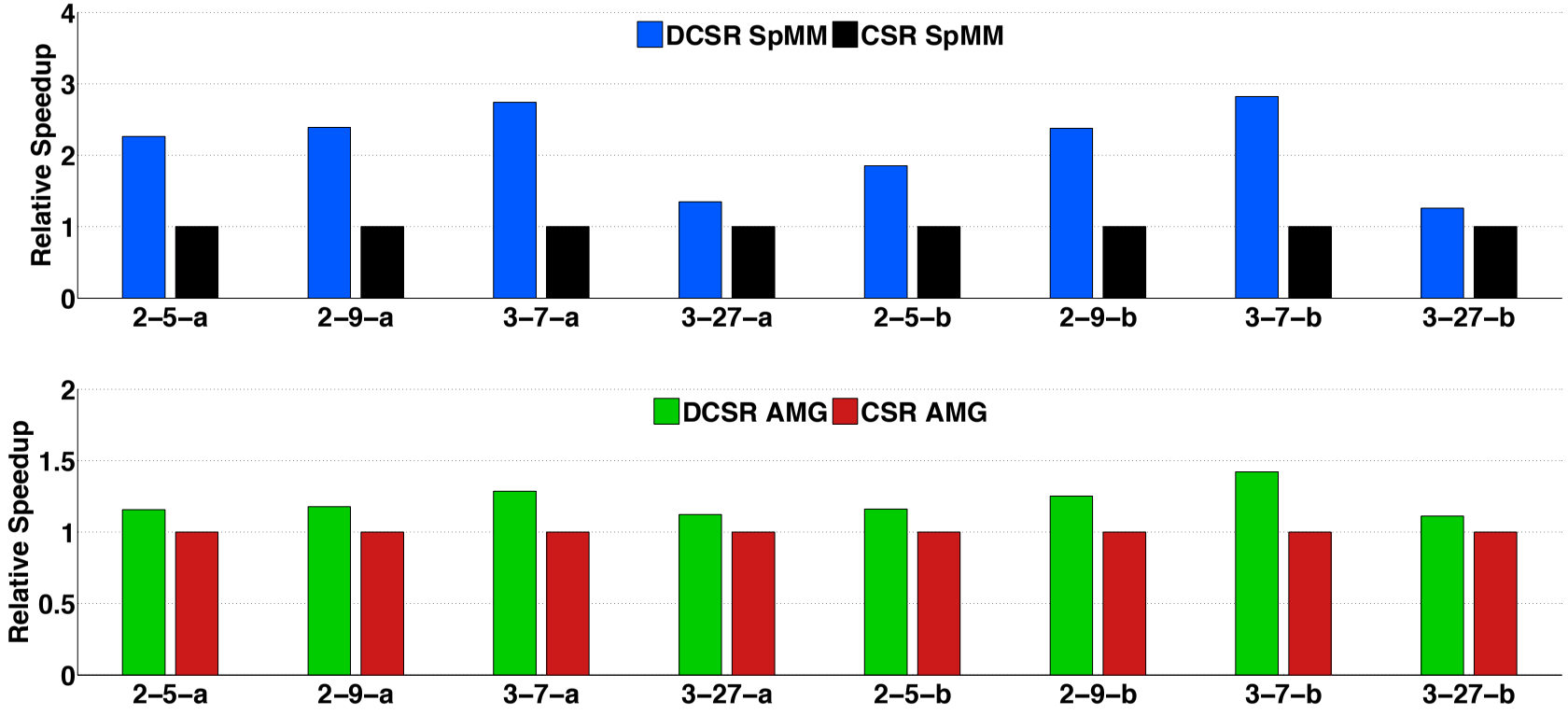



**Figure 6.6**. Relative speedup for SpMM (Above) and AMG (Below) using DCSR and CSR.

Figure 6.6 illustrates the results of our AMG tests with both the individual SpMM times and the overall AMG preconditioner times. Our method outperforms the baseline method by upwards of 3× in some cases. The Galerkin product represents 30% – 50% of total time required by the setup phase of the preconditioner. Results shown in Bell et al. (2012) indicate that the Galerkin product occupies 50% – 60% of the run time on similar matrices using a Nvidia Tesla C2050 GPU. This seems to indicate that the underlying architecture plays a role in the relative processing times across stages. In the case of matrix 3-7-a, the Galerkin product occupies roughly half of the setup time, and our SpMM method is nearly 3× faster in that case, resulting in a speedup of 40%. There is no guarantee what the resulting fill will be in the $C$ matrix, but in practice, the resulting fill is relatively sparse for multiplication with Poisson matrices.

By taking advantage of asynchronous updates enabled by DCSR, we are able to employ specialized kernels based on row lengths. These row length optimized kernels perform the sort and reduction operations within shared memory which is notably faster than performing these operations within global memory. The efficient use of shared memory leads to significant performance gains for the overall SpMM operation. The Galerkin product is by far the largest single component of the setup phase, so improvements in this area will lead to the greatest gains.

# BIBLIOGRAPHY


Michael D. Adams. *Flow-Sensitive Control-Flow Analysis in Linear-Log Time*. PhD thesis, Indiana University, 2011.

Ole Agesen. The cartesian product algorithm. In *Proceedings of the European Conference on Object-Oriented Programming*, page 226, 1995.

Torben Amtoft and Franklyn Turbak. Faithful translations between polyvariant flows and polymorphic types. In *Programming Languages and Systems*, pages 26–40. Springer, 2000.

Christopher Anderson and Paola Giannini. Type checking for javascript. In *In volume WOOD of Electronic Notes in Theoretical Computer Science*, 2004.

Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, February 2007. ISBN 052103311X.

Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 781–792, Piscataway, NJ, USA, 2014a. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.69.

Arash Ashari, Naser Sedaghati, John Eisenlohr, and P. Sadayappan. An Efficient Two-dimensional Blocking Strategy for Sparse Matrix-vector Multiplication on GPUs. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 273–282, New York, NY, USA, 2014b. ACM. ISBN 978-1-4503-2642-1. doi: 10.1145/2597652.2597678.

Haim Avron and Anshul Gupta. Managing Data-movement for Effective Shared-memory Parallelization of Out-of-core Sparse Solvers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 102:1–102:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5.

Anindya Banerjee. A modular, polyvariant and type-based closure analysis. In *ACM SIGPLAN Notices*, volume 32, pages 1–10. ACM, 1997.

Nathan Bell and Michael Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8. doi: 10.1145/1654059.1654078.

Nathan Bell and Michael Garland. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations, 2012. Version 0.3.0.

Nathan Bell, Steven Dalton, and Luke N. Olson. Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. *SIAM Journal on Scientific Computing*, 2012.

Frédéric Besson. CPA beats $\infty$-CFA. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, page 7. ACM, 2009.

Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *ACM SIGPLAN Notices*, volume 44, pages 243–262. ACM, 2009.

Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. *SIGPLAN Not.*, 45(5):115–126, January 2010. ISSN 0362-1340. doi: 10.1145/1837853.1693471.

Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 343–354, New York, NY, 1992. ACM.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of Computer Aided Verification*, pages 154–169. ACM, 2000.

Patrick Cousot. Types as abstract interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 316–331. ACM, 1997.

Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Paris, France, 1976.

Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. *ACM Sigplan Notices*, 12(8):1–12, 1977a.

Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *ACM SIGPLAN Notices*, volume 12, pages 77–94. ACM, 1977b.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977c. ACM Press, New York.

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, TX, 1979. ACM Press, New York.

Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming*, Leuven, Belgium, 13-17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

Olivier Danvy. A new one-pass transformation into monadic normal form. In *Compiler Construction*, pages 77–89. Springer, 2003.

Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011. ISSN 0098-3500.

Christopher Earl, Matthew Might, and David Van Horn. Pushdown control-flow analysis of higher-order programs: Precise, polyvariant and polynomial-time. In *Scheme Workshop*, August 2010.

Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. Introspective pushdown analysis of higher-order programs. In *International Conference on Functional Programming*, pages 177–188, September 2012.

ECMA. *ECMA-262 (ECMAScript Specification)*. ECMA, 5.1 edition, June 2011.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *ACM Sigplan Notices*, volume 28, pages 237–247. ACM, 1993.

Michael Garland. Sparse Matrix Computations on Manycore GPU's. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 2–6, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6. doi: 10.1145/1391469.1391473.

Michael Garland and David B. Kirk. Understanding Throughput-oriented Architectures. *Commun. ACM*, 53(11):58–66, November 2010. ISSN 0001-0782.

JohnR. Gilbert, Steve Reinhardt, and ViralB. Shah. High-Performance Graph Algorithms from Parallel Sparse Matrices. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 260–269. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-75754-2.

Thomas Gilray and Matthew Might. A survey of polyvariance in abstract interpretations. In *Proceedings of the Symposium on Trends in Functional Programming*, May 2013a.

Thomas Gilray and Matthew Might. A unified approach to polyvariance in abstract interpretations. In *Proceedings of the Workshop on Scheme and Functional Programming*, November 2013b.

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. *Proceedings of the Symposium on the Principals of Programming Languages (POPL)*, January 2016.

Joseph L. Greathouse and Mayank Daga. Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 769–780, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.68.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *Proceedings of the European Conference on Object-oriented Programming*, pages 126–150, Berlin, Heidelberg, 2010.

Anshul Gupta, Seid Koric, and Thomas George. Sparse Matrix Factorization on Massively Parallel Computers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 1:1–1:12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8. doi: 10.1145/1654059.1654061.

Dwight Guth. A formal semantics of python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, July 2013.

Williams Ludwell Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 1989.

Matthew Hennessy. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Wiley & Sons, 1990.

Stefan Holdermans and Jurriaan Hage. Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In *ACM Sigplan Notices*, volume 45, pages 63–74. ACM, 2010.

Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 351–363. ACM, 1986.

Eun-jin Im, Eun-jin Im, Katherine Yelick, and Katherine Yelick. Optimization of Sparse Matrix Kernels for Data Mining. In *First SIAM Conf. on Data Mining*, 2000.

Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 393–407, January 1995.

Suresh Jagannathan, Stephen Weeks, and Andrew Wright. Type-directed flow analysis for typed intermediate languages. In *International Static Analysis Symposium*, pages 232–249. Springer, 1997.

Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the symposium on Principles of programming languages*, pages 329–341. ACM, 1998.

Maria Jenkins, Leif Andersen, Thomas Gilray, and Matthew Might. Concrete and abstract interpretation: Better together. In *Workshop on Scheme and Functional Programming*, 2015.

J. Ian Johnson. AAC complexity analysis discussion. Unpublished correspondence., 2015.

J. Ian Johnson and David Van Horn. Abstracting abstract control. In *Proceedings of the ACM Symposium on Dynamic Languages*, October 2014.

J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. Optimizing abstract abstract machines. In *Proceedings of the International Conference on Functional Programming*, September 2013.

Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Symposium on principles of programming languages*, pages 66–74, 1982.

Gilles Kahn. *Natural Semantics*. Springer, 1987.

George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Notices*, volume 48, pages 423–434. ACM, 2013.

Andrew Kennedy. Compiling with continuations, continued. In *Proceedings of the International Conference on Functional Programming*, pages 177–190, New York, NY, 2007. ACM.

Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2011. ISBN 0898719909, 9780898719901.

Khronos Group. *The OpenCL Specification*, September 2011.

James King, Thomas Gilray, Robert M. Kirby, and Matthew Might. Dynamic Sparse-Matrix Allocation on GPUs. In *To Appear at the International SuperComputing Conference*, ISC 2016, June 2016.

Ruud Koot and Jurriaan Hage. Type-based exception analysis for non-strict higher-order functional languages with imprecise exception semantics. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, pages 127–138. ACM, 2015.

Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. A unified sparse matrix data format for modern processors with wide SIMD units. *CoRR*, abs/1307.6209, 2013.

Ondrej Lhoták. *Program analysis using binary decision diagrams*. PhD thesis, McGill University, 2006.

Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In *Compiler Construction*, pages 47–64. Springer, 2006.

Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(1):3, 2008.

Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Evaluating the impact of context-sensitivity on andersen's algorithm for java programs. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 6–12. ACM, 2005.

Shuying Liang and Matthew Might. Hash-flow taint analysis of higher-order pograms. In *Proceedings of the Conference on Programming Language Analysis for Security*, June 2012.

Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 273–282, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2130-3. doi: 10.1145/2464996.2465013.

Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 107–116, New York, NY, 2012. ACM.

Jan Midtgaard. Control-flow analysis of functional programs. *ACM Computing Surveys*, 44 (3):10:1–10:33, Jun 2012.

Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 2007.

Matthew Might. Abstract interpreters for free. In *Static Analysis Symposium*, pages 407–421, September 2010.

Matthew Might and Panagiotis Manolios. A posteriori soundness for non-deterministic abstract interpretations. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 260–274, January 2009.

Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *ACM SIGPLAN Notices*, volume 41, pages 13–25. ACM, 2006.

Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *Proceedings of the International Conference on Programming Language Design and Implementation*, pages 305–315, June 2010.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering Methodology*, 14(1):1–41, January 2005.

Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, july 1991. ISSN 0890-5401. doi: 10.1016/0890-5401(91)90052-4.

Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In YaleN. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *High Performance Embedded Architectures and Compilers*, volume 5952 of *Lecture Notes in Computer Science*, pages 111–125. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-11514-1.

Mayur Naik, Alex Aiken, and John Whaley. *Effective static race detection for Java*, volume 41. ACM, 2006.

Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2004.

*CUDA CUSPARSE Library*. NVIDIA, August 2010.

Oak Ridge National Lab. Titan supercomputer, 2016. URL `https://www.olcf.ornl.gov/titan/`.

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, volume 26, pages 80–113, 2007.

Nicholas Oxhøj, Jens Palsberg, and Michael I Schwartzbach. Making type inference practical. In *ECOOP'92 European Conference on Object-Oriented Programming*, pages 329–349. Springer, 1992.

Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of functional programming*, 11(03):263–317, 2001.

G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. In *Theoretical Computer Science 1*, pages 125–159, 1975.

Gordon D Plotkin. A structural approach to operational semantics. 1981.

Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, and Shriram Krishnamurthi. A tested semantics for getters, setters, and eval in javascript. In *Proceedings of the Dynamic Languages Symposium*, 2012.

Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. EigenCFA: Accelerating flow analysis with GPUs. In *Proceedings of the Symposium on the Principals of Programming Languages (POPL)*, pages 511–522, January 2010.

Racket Community. Racket programming language, 2015. URL `http://racket-lang.org/`.

James F. Ranson, Howard J. Hamilton, and Philip W.L. Fong. A semantics of Python in Isabelle/HOL. Technical Report CS-2008-04, Department of Computer Science, University of Regina, Regina, Saskatchewan, December 2008.

I. Reguly and M. Giles. Efficient sparse matrix-vector multiplication on cache-based GPUs. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12, May 2012. doi: 10.1109/InPar.2012.6339602.

Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003. ISBN 0898715342. URL `Saad:2003:IMS`.

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. Monadic abstract interpreters. In *ACM SIGPLAN Notices*, volume 48, pages 399–410. ACM, 2013.

Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. *Program flow analysis: Theory and applications*, pages 189–234, 1981.

Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1991.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhotak. Pick your contexts well: Understanding object-sensitivity. In *Symposium on Principles of Programming Languages*, pages 17–30, January 2011.

Gideon Joachim Smeding. An executable operational semantics for python. Master's thesis, Universiteit Utrecht, January 2009.

Bor-Yiing Su and Kurt Keutzer. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 353–364, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1316-2. doi: 10.1145/2304576.2304624.

Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

David Van Horn. *The Complexity of Flow Analysis in Higher-Order Languages*. PhD thesis, Mitchom School of Computer Science, Brandeis University, Boston, MA, August 2009.

David Van Horn and Harry G. Mairson. Deciding k-CFA is complete for EXPTIME. *ACM Sigplan Notices*, 43(9):275–282, 2008.

David Van Horn and Matthew Might. Abstracting abstract machines. In *International Conference on Functional Programming*, page 51, Sep 2010.

Dimitrios Vardoulakis and Olin Shivers. CFA2: a context-free approach to control-flow analysis. In *Proceedings of the European Symposium on Programming*, volume 6012, LNCS, pages 570–589, 2010.

Hidde Verstoep and Jurriaan Hage. Polyvariant cardinality analysis for non-strict higher-order functional languages: Brief announcement. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, pages 139–142. ACM, 2015.

Richard Wilson Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, 2003. AAI3121741.

Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3. doi: 10.1145/1362622.1362674.

Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. In *Proceedings of the ACM Transactions on Programming Languages and Systems*, pages 166–207, January 1998.

Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. yaSpMV: Yet Another SpMV Framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 107–118, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. doi: 10.1145/2555243.2555255.

Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. Fast Sparse Matrix-vector Multiplication on GPUs: Implications for Graph Mining. *Proc. VLDB Endow.*, 4(4): 231–242, January 2011. ISSN 2150-8097.