Synthesizing Control-Flow Graph Generators from Operational Semantics

JAMES KOPPEL, MIT
JACKSON KEARL, MIT
ARMANDO SOLAR-LEZAMA, MIT

Theoretically, given a complete semantics of a programming language, one should be able to automatically generate all desired tools. In this work, we take a first step towards that vision by automatically synthesizing many variants of control-flow graph generators from operational semantics, and prove a formal correspondence between the generated graphs and a language's semantics. The first half of our approach is a new algorithm for converting a large class of small-step operational semantics to abstract machines. The second half builds a CFG generator by using "abstract rewriting" to automatically abstract the semantics of a language. The final step generates a graph describing the control flow of all instantiations of each node type, from which the tool generates concise code that can generate a CFG for any program. We implement this approach in a tool called Mandate, and show that it can generate many types of control-flow graph generators for two realistic languages drawn from compilers courses.

1 INTRODUCTION

Control-flow graphs are ubiquitous in programming tools, from compilers to model-checkers. They provide a simple way to order the subterms of a program, and are usually taken for granted. Yet each tool shapes its control-flow graphs differently, and will give a differently-shaped CFG to the same program. In theory, this work should be redundant: they correspond to the same possible executions. Yet the theoretical semantics contain no mention of the nodes and program-points used by tools. There is a rift between the theory and practice.

Many Forms. Let us illustrate the decisions that shape a control-flow graph with an example. Fig. 1 is a fragment of a pretty-printer. What would a control-flow graph for it look like?

```
b := prec < 5;
if (b) then
  print("(")
else
  skip;
print(left);
print("+");
print(right);
if (b) then
  print(")")
else
  skip
```

Fig. 1

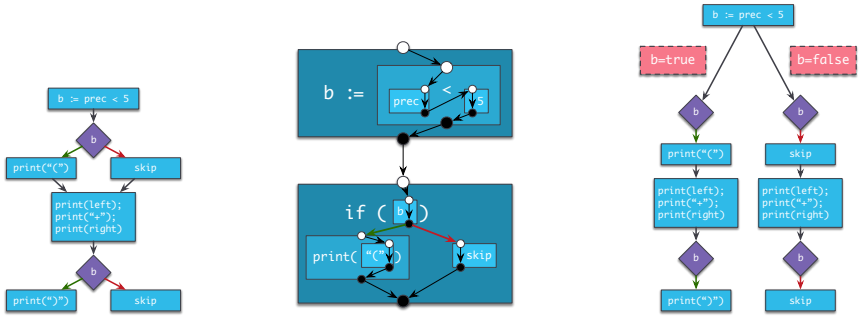
There are infinitely many possible answers. Here are three:

(1) Compilers need small graphs that use little memory for fast compilation. A compiler may only give one node per basic block, giving the graph in Fig. 2a.
(2) Many static analyzers give abstract values to the inputs and result of every expression. To that end, frameworks such as Polyglot [Nystrom et al. 2003] and IncA [Szabó et al. 2016] give two nodes for every expression: one each for entry and exit. Fig. 2b shows part of this graph.
(3) Consider trying to write an analyzer that proves this program's output has balanced parentheses. It must show that there are no paths in which one if-branch is taken but not the other. This can be easily written using a path-sensitive CFG that partitions on the value of `b` (Fig. 2c).

All three tools require separate CFG-generators.

Authors' addresses: James Koppel, MIT, Cambridge, MA, jkoppel@mit.edu; Jackson Kearl, MIT; Armando Solar-Lezama, MIT, Cambridge, MA, asolar@csail.mit.edu.

2019. XXXX-XXXX/2019/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) (b) (c)

Fig. 2. Variants of control-flow graphs

Whence CFGs? What if we had a formal semantics (and grammar) for a programming language? In principle, we should be able to automatically derive all tools for the language. In this dream, only one group needs to build a semantics for each, and then all tools will automatically become available — and semantics have already been built for several major languages [Bogdanas and Roşu 2015; Hathhorn et al. 2015; Park et al. 2015]. In this paper, we take a step towards that vision by deriving control-flow graph generators from a large class of operational semantics.

While operational semantics define each step of computation of a program, the correspondence with control-flow graphs is not obvious. The "small-step" variant of operational semantics defines individual steps of program execution. Intuitively, these steps should correspond to the edges of a control-flow graph. In fact, we shall see that many control-flow edges correspond to the second half of one rule, and the first half of another. We shall similarly find the nodes of a control-flow graph correspond to neither the subterms of a program nor its intermediate values. Existing CFG generators skip these questions, taking some notion of labels or "program points" as a given (e.g.: [Shivers 1991]).

Abstraction and Projection and Machines. Our first insight is to transform the operational semantics into another style of semantics, the abstract machine, via a new algorithm. Evaluating a program under these semantics generates an infinite-state transition system with recognizable control-flow. Typically at this point, an analysis-designer would manually specify some kind of abstract semantics which collapses this system into a finite-state one. Our approach does this automatically by interpreting the concrete semantics differently, using an obscure technique called *abstract rewriting*. From this reduced transition system, a familiar structure emerges: we have obtained our control-flow graphs!

Now all three variants of control-flow graph given in this section follow from different abstractions of the abstract machine, followed by an optional *projection*, or merging of nodes. With this approach, we can finally give a formal, proven correspondence between the operational semantics and all such variants of a control-flow graph.

Mandate: A CFG-Generator Generator. We have implemented our approach in Mandate, the first control-flow-graph generator generator. Mandate takes as input the operational semantics for a language, expressed in an embedded Haskell DSL that can express a large class of systems, along with a choice of abstraction and projection functions. It then outputs a control-flow-graph generator for that language. By varying the abstraction and projection functions, the user can generate any of a large number of CFG-generators.

Mandate has two modes. In the *interpreted mode*, Mandate abstractly executes a program with its language's semantics to produce a CFG. In cases where the control-flow of a node is independent

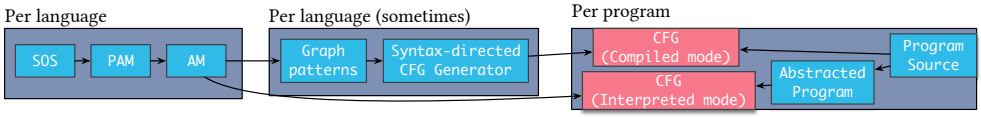


Fig. 3. Dataflow of our approach

from its context (e.g.: including variants (1) and (2) but not (3)). In *compiled mode*, MANDATE outputs a CFG-generator as a short program, similar to what a human would write.

We have evaluated MANDATE on several small languages, as well as two larger ones. The first is Tiger, an Algol-family language used in many university compilers courses, and made famous by a textbook of Appel [1998]. The second, BALISCRIPT[1], is a JavaScript-like language with objects and higher-order functions used in an undergraduate JIT-compilers course.

In summary, our work makes the following contributions:

(1) An elegant new algorithm for converting small-step structural operational semantics into abstract machines.
(2) A new approach using *abstract rewriting* to derive other artifacts from language semantics.
(3) A method of deriving many types of control-flow graph generator from an abstract machine, determined by choice of abstraction and projection functions, including standalone generators which execute without reference to the semantics ("compiled mode").
(4) A formal and proven correspondence between the operational semantics and many common variations of control-flow graphs
(5) The first CFG-generator generator, MANDATE, able to automatically derive many types of control-flow-graph generator for a language from an operational semantics for that language, and successfully used on two realistic languages.

2 CONTROL-FLOW GRAPHS FOR IMP

We shall walk through a simple example of generating a control-flow graph generator, using a simple imperative language called IMP. IMP features conditionals, loops, mutable state, and two binary operators. The syntax is in Fig. 4. In this syntax, we explicitly split terms into values and non-values to make the rules easier to write. We will do this more systematically in §3.2.

Variables	$x, y, \ldots$	$\in$	Var
Expr. Values	v	::=	$n \in \text{Int} \mid \textbf{true} \mid \textbf{false}$
Expressions	e	::=	$v \mid x \mid e + e \mid e < e$
Stmt. Values	w	::=	**skip**
Statements	s	::=	$w \mid s; s \mid \textbf{while } e \textbf{ do } s$
			$\mid x := e \mid \textbf{if } e \textbf{ then } s \textbf{ else } s$

Fig. 4. Syntax of IMP

The approach proceeds in three phases, corresponding roughly to the large boxes in Fig. 3. In the first phase (left box / §2.1), we transform the semantics of IMP into a form that reveals the control flow. In the second phase (right box / §2.2), we show how to interpret these semantics with abstract rewriting [Bert et al. 1993] to obtain CFGs. In the finale (middle box / §2.3), we show how to use abstract rewriting to discover facts about all IMP programs, resulting in human-readable code for a CFG-generator.

2.1 Getting Control of the Semantics

Semantics. The language has standard semantics, so we only show a few of the rules, given as structural operational semantics (SOS). Each rule relates an old term and environment (t, μ) to a

[1]This name has been changed for double-blinding reasons

new one (t', μ') following one step of execution. We focus on the rules for assignment, as they modify environments; we will later consider while-loops (which imply back-edges).

$$\frac{(e, \mu) \rightsquigarrow (e', \mu')}{(x := e, \mu) \rightsquigarrow (x := e', \mu')}\ \textit{AssnCong} \qquad \frac{}{(x := v, \mu) \rightsquigarrow (\textbf{skip}, \mu[x \to v])}\ \textit{AssnEval}$$

These rules give the basic mechanism to evaluate a program, but don't have the form of stepping from one subterm to another, as a control-flow graph would. So, from these rules, our algorithm will automatically generate an *abstract machine*. The abstract machine acts on states $\langle (t, \mu) \,|\, K \rangle$. K is the *context* or *continuation*, and represents what the rest of the program plans to do with the result of evaluating t. K is composed of a stack of *frames*. While the general notion of frames is slightly more complicated (§3.3), in most cases, a frame can be written as e.g.: $\left[(x := \square_t, \square_\mu)\right]$, which is a frame indicating that, once the preceding computation has produced a term and environment (t', μ'), the next step will be to evaluate $(x := t', \mu')$. Our algorithm generates the following rules. These match the textbook treatment of abstract machines [Felleisen et al. 2009].

$$\begin{aligned} \langle (x := e, \mu) \,|\, k \rangle &\to \langle (e, \mu) \,|\, k \circ \left[(x := \square_t, \square_\mu)\right] \rangle \\ \langle (v, \mu) \,|\, k \circ \left[(x := \square_t, \square_\mu)\right] \rangle &\to \langle (\textbf{skip}, \mu[x \to v]) \,|\, k \rangle \end{aligned}$$

The first abstract machine rule, on seeing an assignment $x := e$, will focus on e. Other rules, not shown, will then reduce e to a value. The second then takes this value and evaluates the assignment.

While these rules have been previously hand-created, this work is the first to derive them automatically from structural operational semantics. These two SOS rules become two abstract-machine rules. A naive interpretation is that the first SOS rule corresponds to the first abstract-machine rule, and likewise for the second. After all, the left-hand sides of the first rules match, as do the right-hand sides of the second. But notice how the right-hand sides of the firsts do not match, nor do the left-hand sides of the seconds. The actual story is more complicated. This diagram gives the real correspondence:

The first abstract machine rule is simple enough: it corresponds to the solid arrow, the first half of the first SOS rule. But the second abstract machine rule actually corresponds to the two dashed arrows, which are AssnEval and the second half of AssnCong. We shall find that jumping straight from the SOS to the abstract machine skips a step — there's a missing intermediate step which treats each of the three arrows separately.

This fusing of AssnEval and AssnCong happens because only two actions can follow the second half of AssnCong: AssnEval, and the first half of AssnCong — which is the inverse of the second half. Hence, only the pairing of AssnCong with AssnEval is relevant, and only two abstract machine rules are enough to describe all computations. And the second abstract-machine rule actually does two steps in one: it first plugs in (v, μ) to $\left[(x := \square_t, \square_\mu)\right]$ to obtain $(x := v, \mu)$, and then evaluates this result. So, by fusing these rules, the standard presentation of abstract machines obscures the multiple underlying steps, and hides the correspondence with the SOS.

This insight — that SOS rules can be split into several parts — powers our algorithm to construct abstract machines. The algorithm first creates a representation called the *phased abstract machine*, or PAM, which partitions the two SOS rules into three parts, corresponding to the diagram's three arrows, and gives each part its own rule. The algorithm then fuses some of these rules together, creating the final abstract machine, and completing the first stage of our CFG-creation pipeline.

§3.3 explains PAM in full, while §3.5 and §3.6 give the algorithm for creating abstract machines. §4 and Appendix A provide correctness results for the SOS-to-AM procedure.

2.2 Run Abstract Program, Get CFG

The abstract machine rules show how focus jumps into and out of an assignment during evaluation — the seeds of control-flow. But these transitions are not control-flow edges, as there are still a few important differences. The abstract machine allows for infinitely-many possible states, while control-flow graphs have finite numbers of states, potentially looping back on themselves. The abstract machine executes deterministically, with every state stepping into one other state. Even though we assume determistic languages, even for those, the control-flow graph may branch. We will see how abstraction solves both of these issues, turning the abstract machine into the interpreted-mode control-flow graph generator.

To give a complete example, we'll also need to evaluate an expression. Here is the rule for variable lookups, which looks up y in the present environment:

$$\langle (y, \mu) \,|\, k \rangle \;\rightarrow\; \langle (\mu(y), \mu) \,|\, k \rangle$$

Now consider the statement $x := y$. It can be executed with an infinite number of environments: the starting configuration $(x := y, [y \mapsto 1])$ yields the ending configuration $(\mathbf{skip}, [y \mapsto 1, x \mapsto 1])$; $(x := y, [y \mapsto 2])$ yields $(\mathbf{skip}, [y \mapsto 2, x \mapsto 2])$; etc.

To yield a control-flow graph, we must find a way to compress this infinitude of possible states into a finite number. The value-irrelevance abstraction replaces all values with a single *abstract value* representing any of them: $\star$. Under the value-irrelevance abstraction, all starting environments for this program will be abstracted into the single environment $[y \mapsto \star]$.

In a typical use of abstract interpretation, at this point a new abstract semantics must be manually defined in order to define executions on the abstract state. However, with this kind of syntactic abstraction, our system can interpret the exact same abstract machine rules on this abstract state, a process called *abstract rewriting*. Now, running in fixed context K, there is only one execution of this statement.

$$\begin{array}{rcl} \langle (x := y, [y \mapsto \star]) \,|\, K \rangle & \rightarrow & \langle (y, [y \mapsto \star]) \,|\, K \circ [(x := \square_t, \square_\mu)] \rangle \\ \rightarrow \quad \langle (\star, [y \mapsto \star]) \,|\, K \circ [(x := \square_t, \square_\mu)] \rangle & \rightarrow & \langle (\mathbf{skip}, [y \mapsto \star, x \mapsto \star]) \,|\, K \rangle \end{array}$$

This abstract execution is divorced from any runtime values, yet it still shows the flow of control entering and exiting the assignment and expression— exactly as in the expression-level control-flow graph from Fig. 2b. And thus we can take these abstract states to be our control-flow nodes. The control-flow graph is an abstraction of the transitions of the abstract machine.

Note that, because there are only finitely-many abstract states, this also explains loops in control-flow graphs: looping constructs lead to repeated states, which leads to back-edges in the transition graph. And abstractions also account for branching. Consider the rules for conditionals:

$$\begin{array}{l} \langle (\mathbf{true}, \mu) \,|\, k \circ [(\mathbf{if}\ \square_t\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \square_\mu)] \rangle \;\rightarrow\; \langle (s_1, \mu) \,|\, k \rangle \\ \langle (\mathbf{false}, \mu) \,|\, k \circ [(\mathbf{if}\ \square_t\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \square_\mu)] \rangle \;\rightarrow\; \langle (s_2, \mu) \,|\, k \rangle \end{array}$$

Under the value-irrelevance abstraction, the condition of an if-statement will evaluate to a configuration of the form $(\star, \mu)$. Because $\star$ can represent both **true** and **false**, **both** of the above rules will match. This gives control-flow edges from the if-condition into both branches, exactly as desired.

The value-irrelevance abstraction gives an expression-level CFG. But it is not the only choice of abstraction. §5 presents the CFG-derivation algorithm, and shows how other choices of abstraction

lead to other familiar control-flow graphs, and also introduces projections, which give the CFG-designer the ability to specify which transitions are "internal" and should not appear in the CFG.

2.3 A Syntax-Directed CFG-Generator

The previous section showed how to abstract away the inputs and concrete values of an execution, turning a program into a CFG. The per-program state-exploration of this algorithm is necessary for a path-sensitive CFG-generator, which may create an arbitrary number of abstract states from a single AST node. But for abstractions which discard all contextual information, only a few additional small ingredients are needed to generate a single artifact that describes the control-flow of all instances of a given node-type. This is done once per language, yielding the compiled-mode CFG-generator. We demonstrate how this works for while-loops, which shows how our algorithm can combine many rules to infer control-flow, abstracts away extra steps caused by the internal details of the semantics, and can discover loops in the control-flow even though they are not explicit in the rules.

The semantics of while-loops in IMP are given in terms of other language constructs, by the rule **while** e **do** $s \rightsquigarrow$ **if** e **then** $(s;$ **while** e **do** $s)$ **else skip**. Consider an AM state evaluating an arbitrary while-loop **while** e **do** s in an arbitrary context k, with an arbitrary abstract environment μ. Such a state can be written $\langle (\textbf{while } e \textbf{ do } s, \mu) \,|\, k \rangle$. Any such μ can be represented by the "top" environment $[\star \mapsto \star]$. This means that all possible transitions from any while-loop can be found by finding all rules that could match anything of the form $\langle (\textbf{while } e \textbf{ do } s, [\star \mapsto \star]) \,|\, k \rangle$. Repeatedly expanding these transitions results in a *graph pattern* describing the control-flow for every possible while-loop.

However, merely searching for matching rules will not result in a finite graph, because of states like $\langle (e, [\star \mapsto \star]) \,|\, K \rangle$. These states, which represents the intent to evaluate the unknown subterm e, can match rules for any expression. Instead, we note that, for any given e, other rules (corresponding to other graph patterns) would evaluate its control-flow, and their results can all be soundly overapproximated by assuming e is eventually reduced to a value. Hence, when the search procedure encounters such a state, it instead adds a "transitive edge" to a state $\langle (\star, [\star \mapsto \star]) \,|\, K \rangle$. With this modification, the search procedure finds only 8 unique states for while-loops. It hence terminates in the graph pattern of Fig. 5, which describes the control-flow of all possible while-loops. From this pattern, one could directly generate a CFG fragment for any given while-loop by instantiating e, s, and k. In combination with the graph patterns for all other nodes, this yields a control-flow graph with a proven correspondence to the original program.

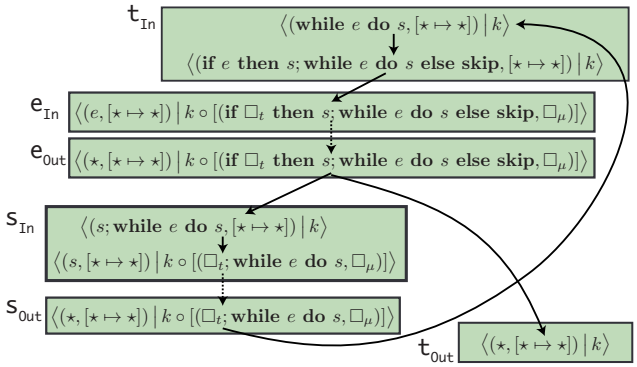


Fig. 5. Example graph-pattern. Dotted lines are transitive edges.

But, from these graph patterns, it is also straightforward to output code for a syntax-directed CFG-generator similar to what a human would write. Our heuristic code-generator traverses this graph pattern, identifying some states as the entrance and exit nodes of the entire while-loop and its subterms. All other states are considered internal steps which get merged with the enter and exit states (via a *projection*), resulting in a few "composite" states. Fig. 5 shows how the code-generator groups and labels the states of this graph pattern. From this labeling, it generates the code in Fig. 6.

After many steps transforming and analyzing the semantics of IMP, the algorithm has finally boiled down all aspects of the control flow into concise, human-readable code — for an expression-level CFG-generator. To generate a statement-level CFG-generator, the user must merely re-run the algorithm again with a different abstraction. For while-loops, the resulting pattern and code are similar to those of Fig. 5 and Fig. 6, except that they skip the evaluation of e.

The last few paragraphs already gave most of the details of graph-pattern construction. §6 gives the remaining details, while Appendix C proves the algorithm's correctness.

```
genCfg t@(While e s) = do
  (tIn, tOut) <- makeNodes t
  (eIn, eOut) <- genCfg e
  (sIn, sOut) <- genCfg s
  connect tIn eIn
  connect eOut sIn
  connect eOut tOut
  connect sOut tIn
  return (tIn, tOut)
```

Fig. 6

3 FROM OPERATIONAL SEMANTICS TO ABSTRACT MACHINES

This section presents our algorithm for converting structural operational semantics to abstract machines. Surprisingly, despite the large amount of work by Danvy and others on conversion between different kinds of semantics [Ager et al. 2003; Biernacka 2006; Danvy and Johannsen 2010; Danvy et al. 2012; Danvy and Nielsen 2004], we found no algorithms that take structural operational semantics as a starting point (§8 gives a thorough discussion of existing work). Our algorithm is unique for its use of a new style of semantics as an intermediate form, the phased abstract machine, which simulates a recursive program running the SOS rules. We believe this formulation is particularly elegant and leads to simple proofs, while being able to scale to the realistic languages Tiger and BaliScript.

There is a lot of notation required first. §3.1 gives a notation for all programming languages, while §3.2 gives an alternate notation for structural operational semantics, one more amenable to inductive transformation. §3.3 and §3.4 describe the phased and orthodox abstract machines, while §3.5 and §3.6 give the conversion algorithm. §4 provides correctness results.

3.1 Terms and Languages

Our presentation requires a uniform notation for terms in all languages. Fig. 7 gives this notation. Throughout this paper, we use the notation $\overline{\cdot}$ to represent lists, so that, e.g.: $\overline{\text{term}}$ represents the set of lists of terms.

const	::=	$n \in \text{Int} \mid \text{str} \in \text{String}$	
sym	::=	$+, <, \textbf{if}, \ldots \in \text{Symbol}$	
mt	::=	Val | NonVal | All	(Match Types)
		$a, b, c, \ldots \in \text{Var}$	(Raw Vars)
x	::=	a_{mt}	(Pattern Vars)
term	::=	$\text{nonval}(\text{sym}, \overline{\text{term}})$	
	|	$\text{val}(\text{sym}, \overline{\text{term}})$	
	|	const	
	|	x	
s	::=	State_I	(Reduction State)
c	::=	(term, s)	(Configurations)

Fig. 7. Universe of terms.

A typical presentation of operational semantics will have variable names like v_1 and e', where v_1 implicitly represents a *value* which cannot be reduced, while e' represents a *non-value* which can. We formalize this distinction by marking each node either value or nonvalue, and giving each variable a *match type mt* controlling whether a variable may match only values, only nonvalues, or either.

Each variable is specialized with one of three **match types**. Variables of the form a_{Val}, a_{NonVal}, and a_{All} are called **value**, **nonvalue**, and **general** variables respectively. Value variables may only match val constructors and constants; nonvalue variables match only nonval constructors; general variables match any.

We use a shorthand to mimic typical presentations of semantics. We will use e to mean a nonvalue variable, v or n to mean a value variable, and t or x for a general variable. But variables in a right-hand side will always be general variables unless said otherwise. For instance, in $e \rightsquigarrow e'$, e is a nonvalue variable, while e' is a general variable.

Each internal node of a term is tagged either *val* or *nonval*. For example, the IMP expression 1 + 1 is shorthand for nonval(+, val(1), val(1)). The IMP statement $x := 1$ may ambiguously refer to either the concrete term nonval(:=, "x", val(1)) or to the pattern nonval(:=, a_{All}, val(1)), but should be clear from context. Other presentations of semantics commonly have a similar ambiguity, using the same notation for patterns and terms.

Each language l is associated with a **reduction state** State_l containing all extra information used to evaluate the term. For example, $\text{State}_{\text{IMP}}$ is the set of *environments*, mapping variables to values. Formally:

$$\text{env} \quad ::= \quad \emptyset \mid \text{env}[\text{str} \to v]$$

Environments are matched using associative-commutative-idempotent patterns [Baader and Nipkow 1999], so that *e.g.:* the pattern $x["y" \to v]$ matches the environment $\emptyset["y" \to 1]["z" \to 2]$. The latter environment is abbreviated $[x \mapsto 2, y \mapsto 1]$. The environment-extension operator itself is not commutative. Specifically, it is right biased; e.g.: $\emptyset["z" \to 1]["z" \to 2] = \emptyset["z" \to 2]$.

Finally, the basic unit of reduction is a *configuration*, consisting of a term and reduction state, i.e.:

$$\text{Conf}_l = \text{term} \times \text{State}_l$$

We say that configuration $c = (t, s)$ is a value if t is a value.

3.2 Straightened Operational Semantics

Rules in structural operational semantics are ordinarily written like logic programs, allowing them to be used to run programs both forward and backwards, and allowing premises to be proven in any order. However, in most rules, there are dependences between the variables that effectively permit only one ordering. In this section, we give an **alternate syntax** for small-step operational semantics rules, which makes this order explicit. This is essentially the conversion of the usual notation into A-normal form. [Flanagan et al. 1993]

One can also gain this ordering property by imposing a restriction on rules without a change in notation: Ibraheem and Schimdt's "L-attributed semantics" is exactly this, but for big-step semantics [Ibraheem and Schmidt 1997]. However, there is an additional advantage of this new notation: it has an inductive structure, which will allow us to define a recursive algorithm over rules.

rule	::=	$c \rightsquigarrow$ rhs
rhs	::=	c
	|	**let** $[c \rightsquigarrow c]$ **in** rhs
	|	**let** $c = \text{semfunc}(\overline{c})$ **in** rhs

Fig. 8. Notation for operational semantics

Fig. 8 defines a grammar for operational semantics rules. These rules collectively define the step-to relation for a language l, $\rightsquigarrow_l$. These rules are relations rather than functions, as they may fail and may have multiple matches. We use the notation $R(A)$ for the image of a relation, i.e.: $R(x)$ refers to any y such that $x \; R \; y$.

A rule matches on a configuration, potentially binding several pattern variables. It then executes a right-hand side. Rule right-hand sides come in three alternatives. The two primary ones are that a rule's right-hand side may construct a new configuration from the bound variables, or may recursively invoke the step-to relation, matching the result to a new pattern. For example, the AssnCong rule from §2.1 would be rendered as:

$$(x := e, \mu) \rightsquigarrow \textbf{let } [(e, \mu) \rightsquigarrow (e', \mu')] \textbf{ in } (x := e', \mu')$$

As a third alternative, it may invoke an external semantic function. Semantic functions are meant to cover everything in an operational semantics that is not pure term rewriting. Common examples include basic arithmetic operations, as well as allocating fresh heap addresses. Each language has its own set of allowed semantic functions.

Definition 3.1. Associated with each language l, there is a set of **semantic functions** semfunc_l. Each is a relation R of type $\overline{\text{Conf}_l} \times \text{Conf}_l$, where, for each $\bar{c} \in \text{Conf}_l$, there are only finntely many d such that $\bar{c}\ R\ d$.

For instance, there are two semantic functions for the IMP language: $\text{semfunc}_{\text{IMP}} = \{+, <\}$. Both are only defined on arguments of the form $((n_1, \mu_1), (n_2, \mu_2))$. Since these functions only act on their term arguments and ignore their μ arguments, we invoke them using the abbreviated notation

$$\textbf{let } n_3 = +(n_1, n_2) \textbf{ in } \text{rhs}$$

which is short for $\textbf{let } (n_3, \mu_3) = +((n_1, \mu_1), (n_2, \mu_2)) \textbf{ in } \text{rhs}$, where μ_1, μ_2 are dummy values, and μ_3 is an otherwise unused variable.

Semantic functions give this notation a lot of flexibility. They can be used to encode side-conditions, e.g.: "$\textbf{let true} = \text{isvalid}(x) \textbf{ in } c$" would fail to match if x is not valid. And they may include external effects such as I/O. As an example using semantic functions, the rule

$$\frac{n = v_1 + v_2}{(v_1 + v_2, \mu) \rightsquigarrow (n, \mu)}\ \mathit{AddEval}$$

would be rendered as

$$(v_1 + v_2, \mu) \rightsquigarrow \textbf{let } n = +(v_1, v_2) \textbf{ in } (n, \mu)$$

where the first occurrence of "+" refers to a nonval node of the object language, and the second occurrence refers to a semantic function of the meta-language. Note that, in their general form, semantic functions may take and return configurations rather than just terms, but **all arguments and return terms must be values**.

There are two extra non-syntactic requirement on SOS rules. The first makes the value/term distinction meaningful, and is needed in the proofs:

Assumption 1 (Sanity of Values)**.** *The following must hold:*

- *In every rule,* $(t, s) \rightsquigarrow$ *rhs, the pattern* t *must not match a value.*
- *If* t *is a nonvalue, there are* s, t', s' *such that* $(t, s) \rightsquigarrow (t', s')$.

The second is necessary to create a tame notion of control-flow:

Assumption 2 (Determinism)**.** *For any* t, s*, there is at most one* t', s' *such that* $(t, s) \rightsquigarrow (t', s')$.

This assumption is not necessary for converting the semantics to PAM, which can faithfully emulate any structural operational semantics. It is necessary for conversion to an abstract machine, as abstract machines have difficulty expressing behaviors with top-level nondeterminism. Intuitively, this is desirable. Nondeterminism means that consecutive steps may occur in distant parts of a term: evaluating $((a * b) + (c - d)) + (e/f)$ may e.g.: evaluate first the multiplication on the left, then the division on the right, and then the subtraction on the left. It is difficult to imagine a notion of control-flow for such a space of executions.

This notation gives a simple inductive structure to SOS rules. We will see in §3.5 how it makes the SOS-to-PAM conversion straightforward. And it loses very little generality from the conventional SOS notation: it essentially only assumes that the premises can be ordered. (Converting this notation back into the conventional form is easy: turn all RHS fragments into premises.)

3.3 The Phased Abstract Machine

In a structural operational semantics, a single step may involve many rules, and each rule may perform multiple computations. The phased abstract machine (PAM) breaks each of these into distinct steps. In doing so, it simulates how a recursive functional program would interpret the operational semantics.

frame	::=	$[c \to \text{rhs}]$	
K	::=	**empty**	(Contexts)
	|	$K \circ \text{frame}$	
	|	k	(Context Vars)
$\updownarrow$	::=	$\uparrow \mid \downarrow$	(Phase)
pamState	::=	$\langle c \,\vert\, K \rangle \updownarrow$	
pamRhs	::=	pamState	
	|	**let** $c = \text{semfunc}(\overline{c})$ **in** pamRhs	
pamRule	::=	pamState $\hookrightarrow$ pamRhs	

Fig. 9. The Phased Abstract Machine

A PAM state takes the form $\langle c \,\vert\, K \rangle \updownarrow$. The configuration c is the same as for operational semantics. K is a *context* or *continuation*, and will be familiar to a reader with exposure to abstract machines. The phase $\updownarrow$ is the novel part. Every PAM state is either in the evaluating ("down") phase $\downarrow$, or the returning ("up") phase $\uparrow$. A down state $\langle c \,\vert\, K \rangle \downarrow$ can be interpreted as an intention for the PAM to evaluate c in a manner corresponding to one step of the operational semantics, yielding $\langle c' \,\vert\, K \rangle \uparrow$. In fact, in §4, we will provide theorems that a single step $c_1 \rightsquigarrow c_2$ in the operational semantics perfectly corresponds to a sequence of steps $\langle c_1 \,\vert\, K \rangle \downarrow \hookrightarrow^* \langle c_2 \,\vert\, K \rangle \uparrow$ in the PAM.

A PAM rule $\langle c_p \,\vert\, K_p \rangle \updownarrow_p \hookrightarrow_l \text{rhs}_p$ for language l is executed on a PAM state $\langle c_1 \,\vert\, K_1 \rangle \updownarrow_1$ as follows:

(1) Find a substitution σ such that $\sigma(c_p) = c_1$, $\sigma(K_p) = K_1$. Fail if no such σ exists, or if $\updownarrow_p \neq \updownarrow_1$.
(2) Recursively evaluate rhs_p as follows:
   - If $\text{rhs}_p = \textbf{let } c_{\text{ret}} = \text{func}(\overline{c_{\text{args}}}) \textbf{ in } \text{rhs}'_p$, with $\text{func} \in \text{semfunc}_l$, then pick $r \in \text{func}(\sigma(\overline{c_{\text{args}}}))$, and extend σ to σ' with $\sigma'(c_{\text{ret}}) = r$ and $\sigma'(x) = \sigma(x)$ for all $x \in \text{dom}(\sigma)$. Fail if no such σ' exists. Then recursively evaluate rhs'_p on σ'.
   - If $\text{rhs}_p = \left\langle c'_p \,\middle\vert\, K'_p \right\rangle \updownarrow'_p$, return the new PAM state $\left\langle \sigma(c'_p) \,\middle\vert\, \sigma(K'_p) \right\rangle \updownarrow'_p$.

Let us give some example rules. (An example PAM execution will be in §3.6). The AssnCong and AssnEval rules from §2.1 get transformed into the following three PAM rules:

$$\begin{aligned} \langle (x := e, \mu) \,\vert\, k \rangle \downarrow &\hookrightarrow \langle (e, \mu) \,\vert\, k \circ [(x := \square_t, \square_\mu)] \rangle \downarrow \\ \langle (t, \mu) \,\vert\, k \circ [(x := \square_t, \square_\mu)] \rangle \uparrow &\hookrightarrow \langle (x := t, \mu) \,\vert\, k \rangle \uparrow \\ \langle (x := v, \mu) \,\vert\, k \rangle \downarrow &\hookrightarrow \langle (\textbf{skip}, \mu[x \to v]) \,\vert\, k \rangle \uparrow \end{aligned}$$

The AssnCong rule becomes a pair of mutually-inverse PAM rules. One is a **down rule** which steps a down state to a down state, signaling a recursive call. The other is an **up rule**, corresponding to a return. This is a distinguishing feature of congruence rules, and is an important fact used when constructing the final abstract machine. Evaluation rules, on the other hand, typically become a **down-up** rule. Note also that frames may be able to store information; here, the frame $[(x := \square_t, \square_\mu)]$ stores the variable to be assigned, x.

The PAM and operational semantics share a language's underlying semantic functions. The AddEval rule of §3.2, for instance, becomes the following PAM rules:

$$\begin{array}{rcl} \left\langle (v_1 + v_2, \mu) \,\middle|\, k \right\rangle\!\downarrow & \hookrightarrow & \textbf{let } n = +(v_1, v_2) \textbf{ in } \left\langle (n, \mu) \,\middle|\, k \right\rangle\!\downarrow \\ \left\langle (v, \mu) \,\middle|\, k \right\rangle\!\downarrow & \hookrightarrow & \left\langle (v, \mu) \,\middle|\, k \right\rangle\!\uparrow \end{array}$$

The latter rule becomes redundant upon conversion to an abstract machine, which drops the phase.

Fig. 9 gives the full syntax for PAM rules. A PAM rule steps a left-hand state into a right-hand state, potentially after invoking a sequence of semantic functions. A PAM state contains a configuration, context, and phase. A context is a sequence of frames, terminating in **empty**.

Note that a pamRhs consists of a sequence of RHS fragments which terminate in a pamState $\left\langle c \,\middle|\, K \right\rangle\!\updownarrow$. Fig. 10 captures this into a notation for contexts, so that an arbitrary PAM rule can be written $\left\langle c_1 \,\middle|\, K_1 \right\rangle\!\updownarrow_1 \hookrightarrow C[\, \left\langle c_2 \,\middle|\, K_2 \right\rangle\!\updownarrow_2 \,]$.

Up until this point, we have not used the full notation for frames. In general, a frame represents a continuation of an SOS right-hand side. Because SOS right-hand sides may match on substructure of a result (*e.g.:* : the RHS **let** $[c \rightsquigarrow (a + b, \mu)]$ **in** $\square$ binds variables a and b to subterms of the result of stepping c), frames must similarly be able to destructure the values on which they await. So a frame takes the syntactic form of a function $[c \to \text{rhs}]$, and the frame $\left[(x := \square_t, \square_\mu)\right]$ is actually shorthand for $[(t', \mu') \to (x := t', \mu')]$.

3.4 Abstract Machines

Finally, the abstract machine is similar to the phased abstract machine, except that an abstract machine state does not contain a phase. Fig. 11 gives a grammar for abstract machine rules. We gave example rules for the abstract machine for IMP in §2.1.

C	::=	$\square$
	|	**let** c = semfunc($\overline{c}$) **in** C

Fig. 10. Abstract Machine: RHS contexts

Summary of Notation. This section has introduced three versions of semantics, each defining their own transition relation. In summary:

(1) $c_1 \rightsquigarrow c_2$ ("squiggly arrow") denotes one step of the operational semantics (§3.2).
(2) $c_1 \hookrightarrow c_2$ ("hook arrow") denotes one step of the PAM (§3.3).
(3) $c_1 \to c_2$ ("straight arrow") denotes one step of the abstract machine (§3.4).

amState	::=	$\left\langle c \,\middle|\, K \right\rangle$
amRhs	::=	amState
	|	**let** c = semfunc($\overline{c}$) **in** amRhs
amRule	::=	amState $\to$ amRhs

Fig. 11. Abstract Machines

Mnemonically, as the step-relation gets closer to the abstract machine, the arrow "flattens out."

Each language has its own set of rules, and hence its own reduction relations. So, $\rightsquigarrow_{\text{IMP}}$ refers to the single-step reduction in the structural operational semantics for IMP, while $\hookrightarrow_{\text{Bali}}$ is the transition relation of the PAM for BaliScript.

The conversion between PAM and abstract machine introduces a fourth system, the *unfused abstract machine*. The unfused abstract machine is identical to the abstract machine except that some rules of the abstract machine correspond to several rules of the unfused abstract machine. We thus do not distinguish it from the orthodox abstract machine, except in one of the proofs, where its transition relation is given the symbol $\longrightarrow$ ("long arrow").

In cases where we need to refer to individual rules, we use the notation $\left\langle c \,\middle|\, K \right\rangle \xrightarrow{F} \left\langle c' \,\middle|\, K' \right\rangle$ to denote that $\left\langle c \,\middle|\, K \right\rangle$ steps to $\left\langle c' \,\middle|\, K' \right\rangle$ by rule F.

3.5 Splitting the SOS

In this section, we present our algorithm for converting an operational semantics to PAM. Fig. 12 defines the sosToPam function, which computes this transformation.

The algorithm generates PAM rules for each SOS rule. For each SOS rule $c \rightsquigarrow$ rhs, it begins in the state $\langle c \,|\, k \rangle\downarrow$, the start state for evaluation of c. It then generates rules corresponding to each part of rhs. For a semantic function, it transitions to a down state, and begins the next rule in the same down state, so that they may match in sequence. For a recursive invocation, it transitions to a down state $\langle c \,|\, k' \rangle\downarrow$, but begins the next rule in the state $\langle c \,|\, k' \rangle\uparrow$, so that other PAM rules must evaluate c before proceeding with computations corresponding to this SOS rule. Finally, upon encountering the end of the step c, it transitions to a state $\langle c \,|\, k \rangle\uparrow$, returning c up the stack.

For each step, it also pushes a frame containing the remnant of the SOS rhs onto the context, both to to help ensure rules may only match in the desired order, and because the rhs may contain variables bound in the left-hand side, which must be preserved across rules.

After the algorithm finishes creating PAM rules for each of the SOS rules, it adds one special rule, called the **reset rule**:

$$\langle (t, s) \,|\, \textbf{empty} \rangle\uparrow \;\hookrightarrow\; \langle (t, s) \,|\, \textbf{empty} \rangle\downarrow$$

The reset rule takes a state which corresponds to completing one step of SOS evaluation, and changes the phase to $\downarrow$ so that evaluation may continue for another step. Note that it matches using a nonvalue-variable t so that it does not attempt to evaluate a term after termination.

Note also that the LHS and RHS of the reset rule differ only in the phase. The corresponding rule in the abstract machine, which omits the phases, would hence be a self-loop. For this reason, the translation from PAM to abstract machine removes this rule. It is also removed in the proof of Theorem 4.1.

3.6 Cutting PAM

The PAM evaluates a term in lockstep with the original SOS rules. Yet, in both, each step of computation always begins at the root of the term, rather than jumping from one subterm to the next. By optimizing these extra steps away, our algorithm will create the abstract machine from the PAM.

Consider how the PAM evaluates the term $(1 + (1 + 1)) + 1$, shown in Fig. 13. Notice how lines 6–7 mirror lines 8–9. After evaluating 1 + 1 deep within the term, the PAM walks up to the root, and then back down the same path. Knowing this, an optimized abstract machine could jump directly from line 6 to line 10.

Danvy used a similar insight to create his *refocusing* technique for converting a reduction semantics to an abstract machine [Danvy and Nielsen 2004]. But in the setting of PAM, the necessary property becomes particularly simple and mechanical:

Definition 3.2. An up-rule $\langle c_1 \,|\, K_1 \rangle\uparrow \;\hookrightarrow C[\, \langle c_2 \,|\, K_2 \rangle\uparrow \,]$ is **invertible** if $\langle c_2 \,|\, K_2 \rangle\downarrow \;\hookrightarrow^*\; \langle c_1 \,|\, K_1 \rangle\downarrow$ whenever c_1 is a non-value.

If an up-rule and its corresponding down-rules do not invoke any semantic functions, invertibility can be checked automatically via a reachability search. When all up-rules are invertible, we can show that $\langle c \,|\, K \rangle\uparrow \;\hookrightarrow^*\; \langle c \,|\, K \rangle\downarrow$ whenever c is a non-value, meaning that these transitions may be skipped (Lemma A.3). This means that all up-rules are redundant unless the LHS is a value, and hence they can be specialized to values. The phases now become redundant and can be removed, yielding the first abstract machine.

$$\boxed{\text{sosToPam}(\overline{\text{rules}})}$$

$$\text{sosToPam}(\overline{\text{rules}}) = \left(\bigcup_{r \in \overline{\text{rules}}} \text{sosRuleToPam}(r) \right) \cup \left\{ \left\langle (t,s) \,\middle|\, \textbf{empty} \right\rangle\!\uparrow \hookrightarrow \left\langle (t,s) \,\middle|\, \textbf{empty} \right\rangle\!\downarrow \right\}$$
$$\textbf{where } t,\ s \text{ are fresh variables}$$

$$\boxed{\text{sosRuleToPam}(\text{rule})}$$

$$\text{sosRuleToPam}(c \rightsquigarrow \text{rhs}) = \text{sosRhsToPam}(\left\langle c \,\middle|\, k \right\rangle\!\downarrow, k, \text{rhs}) \quad (1)$$
$$\textbf{where } k \text{ is a fresh variable}$$

$$\boxed{\text{sosRhsToPam}(\text{pamState}, K, \text{rhs})}$$

$$\text{sosRhsToPam}(s, k, c) = \left\{ s \hookrightarrow \left\langle c \,\middle|\, k \right\rangle\!\uparrow \right\} \quad (2)$$

$$\text{sosRhsToPam}(s, k, \textbf{let } [c_1 \rightsquigarrow c_2] \textbf{ in rhs}) = \left\{ s \hookrightarrow \left\langle c_1 \,\middle|\, k' \right\rangle\!\downarrow \right\} \cup \text{sosRhsToPam}(\left\langle c_2 \,\middle|\, k' \right\rangle\!\uparrow, k, \text{rhs}) \quad (3)$$
$$\textbf{where } k' = k \circ [c_2 \to \text{rhs}]$$

$$\text{sosRhsToPam}(s, k, \textbf{let } c_2 = \text{func}(\overline{c_1}) \textbf{ in rhs}) = \left\{ s \hookrightarrow \textbf{let } c_2 = \text{func}(\overline{c_1}) \textbf{ in } \left\langle c_2 \,\middle|\, k' \right\rangle\!\downarrow \right\} \cup \text{sosRhsToPam}(\left\langle c_2 \,\middle|\, k' \right\rangle\!\downarrow, k, \text{rhs}) \quad (4)$$
$$\textbf{where } k' = k \circ [c_2 \to \text{rhs}]$$

Fig. 12. The SOS-to-PAM algorithm. Labels are used in the proofs of Appendix A.

$$\begin{aligned}
&\left\langle ((1 + (1 + 1)) + 1,\ \emptyset) \,\middle|\, \textbf{empty} \right\rangle\!\downarrow && (1)\\
\hookrightarrow\ &\left\langle (1 + (1 + 1),\ \emptyset) \,\middle|\, \textbf{empty} \circ \left[(\square_t + 1, \square_\mu) \right] \right\rangle\!\downarrow && (2)\\
\hookrightarrow\ &\left\langle (1 + 1,\ \emptyset) \,\middle|\, \textbf{empty} \circ \left[(\square_t + 1, \square_\mu) \circ [(1 + \square_t,\ \mu)] \right] \right\rangle\!\downarrow && (3)\\
\hookrightarrow\ &\left\langle (2,\ \emptyset) \,\middle|\, \textbf{empty} \circ \left[(\square_t + 1, \square_\mu) \circ [(1 + \square_t,\ \mu)] \right] \right\rangle\!\downarrow && (4)\\
\hookrightarrow\ &\left\langle (2,\ \emptyset) \,\middle|\, \textbf{empty} \circ \left[(\square_t + 1, \square_\mu) \circ [(1 + \square_t,\ \mu)] \right] \right\rangle\!\uparrow && (5)\\
\hookrightarrow\ &\left\langle (1 + 2,\ \emptyset) \,\middle|\, \textbf{empty} \circ \left[(\square_t + 1, \square_\mu) \right] \right\rangle\!\uparrow && (6)\\
\hookrightarrow\ &\left\langle ((1 + 2) + 1,\ \emptyset) \,\middle|\, \textbf{empty} \right\rangle\!\uparrow && (7)\\
\hookrightarrow\ &\left\langle ((1 + 2) + 1,\ \emptyset) \,\middle|\, \textbf{empty} \right\rangle\!\downarrow && (8)\\
\hookrightarrow\ &\left\langle (1 + 2,\ \emptyset) \,\middle|\, \textbf{empty} \circ \left[(\square_t + 1, \square_\mu) \right] \right\rangle\!\downarrow && (9)\\
\hookrightarrow\ &\left\langle (3,\ \emptyset) \,\middle|\, \textbf{empty} \circ \left[(\square_t + 1, \square_\mu) \right] \right\rangle\!\downarrow && (10)\\
\hookrightarrow\ &\left\langle (3,\ \emptyset) \,\middle|\, \textbf{empty} \circ \left[(\square_t + 1, \square_\mu) \right] \right\rangle\!\uparrow && (11)\\
\hookrightarrow\ &\left\langle (3 + 1,\ \emptyset) \,\middle|\, \textbf{empty} \right\rangle\!\uparrow && (12)\\
\hookrightarrow\ &\left\langle (3 + 1,\ \emptyset) \,\middle|\, \textbf{empty} \right\rangle\!\downarrow && (13)\\
\hookrightarrow\ &\left\langle (4,\ \emptyset) \,\middle|\, \textbf{empty} \right\rangle\!\downarrow && (14)\\
\hookrightarrow\ &\left\langle (4,\ \emptyset) \,\middle|\, \textbf{empty} \right\rangle\!\uparrow && (15)
\end{aligned}$$

Fig. 13. Example PAM derivation.

There is one more technical requirement for the abstract machine to be valid: not having any up-down rules, rules of the form $\left\langle c \,\middle|\, k \right\rangle\!\uparrow \hookrightarrow C[\left\langle c' \,\middle|\, k' \right\rangle\!\downarrow]$. An up-down rule follows from any SOS rule which steps multiple subterms in a single step. One example is this lockstep-composition rule:

$$\frac{e_1 \rightsquigarrow e_1' \quad e_2 \rightsquigarrow e_2'}{e_1 \parallel e_2 \rightsquigarrow e_1' \parallel e_2'}\ \textit{LockstepComp}$$

The LockstepComp rule differs from normal parallel composition, in that both components must step simultaneously. Up-down rules like this break the locality of the transition system, meaning that whether one subterm can make consecutive steps may depend on different parts of the tree. Correspondingly, it also means that a single step of the program may step multiple parts of the tree, making it difficult to have a meaningful notion of program counter.

Most of the time, the presence of an up-down rule will also cause some up-rules to not be invertible, making a prohibition on up-down rules redundant. However, there are pathological cases where this is not so. For example, consider the expression $e_1 \parallel e_2$ with the LocksteppComp rule. The LocksteppComp rule splits into 3 PAM rules, of which the third is an up-rule, $\left\langle e_2' \,\middle|\, k \circ \left[e_1' \parallel \square\right]\right\rangle\uparrow \hookrightarrow \left\langle e_1' \parallel e_2' \,\middle|\, k\right\rangle\uparrow$. If it is possible for e_1' to be a value but not e_2', then this rule is not invertible. However, if $e_1 \rightsquigarrow e_1$ and $e_2 \rightsquigarrow e_2$ for all e_1, e_2, then this rule is invertible. Hence, the PAM-to-AM algorithm includes an additional check that there are no up-down rules save the reset rule.

Algorithm: PAM to Unfused Abstract Machine.

(1) Check that all up-rules for l are invertible. Fail if not.
(2) Check that there are no up-down rules other than the reset rule. Fail if not.
(3) Remove the reset rule.
(4) For each up-rule with LHS $\left\langle (t, s) \,\middle|\, K\right\rangle\uparrow$, unify t with a fresh value variable. The resulting t' will either have a value node at the root, or will consist of a single value variable. If t fails to unify, remove this rule.
(5) Remove all rules of the form $\left\langle c \,\middle|\, K\right\rangle\downarrow \hookrightarrow \left\langle c \,\middle|\, K\right\rangle\uparrow$, which would become self-loops.
(6) Drop the phase $\updownarrow$ from the pamState's in all rules

Fusing the Abstract Machine. This unfused abstract machine still takes more intermediate steps than a normal abstract machine. The final abstract machine is created by *fusing* successive rules together. A rule $\left\langle c_1 \,\middle|\, K_1\right\rangle \to C_1[\left\langle c_1' \,\middle|\, K_1'\right\rangle]$ is fused with a rule $\left\langle c_2 \,\middle|\, K_2\right\rangle \to C_2[\left\langle c_2' \,\middle|\, K_2'\right\rangle]$ by unifying (c_1', K_1') with (c_2, K_2), and replacing them with the new rule $\left\langle c_1 \,\middle|\, K_1\right\rangle \to C_1[C_2[\left\langle c_2' \,\middle|\, K_2'\right\rangle]]$.

Property 1 (Fusion). *Consider two AM rules F and G, and let their fusion be FG. Then* $\left\langle c \,\middle|\, K\right\rangle \xrightarrow{F} \left\langle c' \,\middle|\, K'\right\rangle \xrightarrow{G} \left\langle c'' \,\middle|\, K''\right\rangle$ *if and only if* $\left\langle c \,\middle|\, K\right\rangle \xrightarrow{FG} \left\langle c'' \,\middle|\, K''\right\rangle$.

There are two cases where rules should be fused. First, rules which invoke a semantic function always have only one possible successor, and should be fused. Without this, the abstract machine for AddEval would have an extra state for after it invokes the semantic computation $+(n_1, n_2)$, but before it plugs the result into a term. Second, up-rules should be fused with all possible successors. Without this, computing $e_1 + e_2$ would have an extra state where, after evaluating e_1, it revisits $e_1 + e_2$, rather than jumping straight into evaluating e_2. Both steps are strictly optional. However, doing so generates abstract machine rules which match the standard versions (as in e.g.: [Felleisen et al. 2009]), and also generate more intuitive control-flow graphs.

For example, here are the final rules for assignment:

$$\left\langle (x := e, \mu) \,\middle|\, k\right\rangle \to \left\langle (e, \mu) \,\middle|\, k \circ \left[(x := \square_t, \square_\mu)\right]\right\rangle$$
$$\left\langle (v, \mu) \,\middle|\, k \circ \left[(x := \square_t, \square_\mu)\right]\right\rangle \to \left\langle (\textbf{skip}, \mu[x \to v]) \,\middle|\, k\right\rangle$$

4 CORRECTNESS

This section provides the correspondence between the operational semantics and abstract machine. We present only the high-level theorems here, with the proofs available in **Appendix A**.

The core idea of the correspondence is simple: The PAM emulates the SOS because each PAM rule was explicitly constructed to correspond to an RHS fragment of the SOS:

Theorem 4.1. $c_1 \rightsquigarrow_l^* c_2$ *if and only if, for all contexts* K, $\left\langle c_1 \,\middle|\, K\right\rangle\downarrow \hookrightarrow_l^* \left\langle c_2 \,\middle|\, K\right\rangle\uparrow$

The PAM and AM are equivalent because the AM merely removes redundant steps from the PAM, and because fused rules in the AM each correspond to several rules in the PAM. However, a PAM

derivation may have "false starts" corresponding to a partially-applied SOS rule, and so we first give some technical definitions that determine which states are included in the correspondences.

Stuck States. The first kind of "false start" comes from steps that cannot be completed.

Definition 4.2. A configuration/context pair (c, K) is **non-stuck** if $\langle c \,|\, K\rangle\uparrow \hookrightarrow^* \langle c' \,|\, \mathbf{empty}\rangle\uparrow$ for some c'.

Because each PAM rule corresponds to part of an SOS rule, our definition of non-stuckness is different from the usual one: it is intended to exclude terms which correspond to a partial match on an SOS rule. A single step $c_1 \rightsquigarrow c_2$ in the SOS corresponds to a sequence $\langle c_1 \,|\, \mathbf{empty}\rangle\downarrow \hookrightarrow^* \langle c_2 \,|\, \mathbf{empty}\rangle\uparrow$ in the PAM, so a state is non-stuck if it can complete the current step. Stuck states result from SOS rules which only partially match a term. For example, the SOS rule

$$(a.b := v, \mu) \rightsquigarrow \mathbf{let}\ (r, \mu') = \text{Lookup}((a, \mu))\ \mathbf{in\ let\ false} = \text{ContainsField}(r, b)\ \mathbf{in}\ (\mathbf{error}, \mu)$$

decomposes into 3 PAM rules. If Lookup succeeds, the first brings $\langle (a.b := v, \mu) \,|\, K\rangle\downarrow$ into the state

$$\langle (r, \mu') \,|\, K \circ [\mathbf{let\ false} = \text{ContainsField}(\square_t, b)\ \mathbf{in}\ (\mathbf{error}, \mu)]\rangle\downarrow$$

If $\mathbf{false} \neq \text{ContainsField}(r, b)$, then this will be a stuck state.

Working Steps. As seen in the example in §3.6, many steps get removed when converting from PAM to AM. This causes the second form of "false start."

Definition 4.3. An **inversion sequence** beginning at $\langle c \,|\, K\rangle\uparrow$ is a sequence of transitions $\langle c \,|\, K\rangle\uparrow \hookrightarrow^* \langle c \,|\, K\rangle\downarrow$ which contains at most one application of the reset rule.

This idea of an inversion sequence partitions a derivation $\langle c_1 \,|\, K_1\rangle\downarrow \hookrightarrow^* \langle c_2 \,|\, K_2\rangle\downarrow$ into two parts: the inversion sequences, which do redundant work, and the remainder, which we call the **working steps**. A PAM state inside an inversion sequence might not correspond to any AM state.

Definition 4.4. A reduction $\langle c_1 \,|\, K_1\rangle\updownarrow_1 \hookrightarrow \langle c_2 \,|\, K_2\rangle\updownarrow_2$ within a derivation is a **working step** if the derivation cannot be extended so that $\langle c_1 \,|\, K_1\rangle\updownarrow_1$ is part of an inversion sequence.

PAM-AM Correspondence. The PAM and AM correspond as follows, via the Unfused AM.

Theorem 4.5 (PAM-Unfused AM: Forward). *Suppose* $\langle c \,|\, K\rangle\downarrow \hookrightarrow_l^* \langle c' \,|\, K'\rangle\downarrow$, $\langle c' \,|\, K'\rangle\downarrow$ *is non-stuck, and the derivation's last step is working. Then there is a derivation* $\langle c \,|\, K\rangle \longrightarrow_l^* \langle c' \,|\, K'\rangle$.

Theorem 4.6 (PAM-Unfused AM: Backward). *If* $\langle c \,|\, K\rangle \longrightarrow_l^* \langle c' \,|\, K'\rangle$, *then there are phases* $\updownarrow_c$ *and* $\updownarrow_{c'}$ *such that* $\langle c \,|\, K\rangle\updownarrow_c \hookrightarrow_l^* \langle c' \,|\, K'\rangle\updownarrow_{c'}$.

Reductions in the Unfused AM correspond to reductions in the normal AM unless the last rules used in the Unfused AM have been fused away.

Theorem 4.7 (Unfused AM-AM). $\langle c \,|\, K\rangle \rightarrow_l^* \langle c' \,|\, K'\rangle$ *if and only if* $\langle c \,|\, K\rangle \longrightarrow_l^* \langle c' \,|\, K'\rangle$ *by a sequence of rules whose last rule is not fused away.*

5 CONTROL-FLOW GRAPHS AS ABSTRACTIONS

The abstract machine is a transition system describing all possible executions of a program. Applying an *abstract interpretation* blows this down to a finite one. Some choices of abstraction yield a graph resembling a traditional control-flow graph (§5.3).

Yet to compute on abstract states, one must also abstract the transition rules, and this traditionally requires manual definitions. Fortunately, we will find that the desired families of CFG can be obtained by a specific class of *syntactic abstraction* which allow the transition rules to be abstracted automatically, via *abstract rewriting* (§5.1–§5.2).

After constructing the abstract transition graph, more control can be obtained by further combining states using a *projection function* (§5.4). A final choice of control-flow graph is then obtained by an abstraction/projection pair (α, π).

The development of abstract-rewriting in this section is broadly similar to that of Bert et al. [1993], but departs greatly in details to fit our formalism of terms and machines. In §6 and Appendix C, we explore connections to an older technique called *narrowing*.

5.1 Abstract Terms, Abstract Matching

Our goal is to find a notion of abstract terms flexible enough to allow us express desired abstraction functions, but restricted enough that we can find a way to automatically apply the existing abstract machine rules to them. We accomplish this by defining a set of generalized terms $\mathsf{term}\star$, satisfying $\mathsf{term} \subset \mathsf{term}\star$, where some nodes have been replaced with $\star$ nodes which represent any term.

$$\mathsf{term}\star \quad ::= \quad \mathrm{nonval}(\mathrm{sym}, \overline{\mathsf{term}\star}) \mid \mathrm{val}(\mathrm{sym}, \overline{\mathsf{term}\star}) \mid \mathrm{const} \mid x \mid \star_{\mathrm{mt}}$$

Here, mt is a match type as defined in Fig. 7, so that the allowed $\star$ nodes are $\star_{\mathsf{Val}}$, $\star_{\mathsf{NonVal}}$, and $\star_{\mathsf{All}}$.

Formally, we define an ordering $\prec$ on $\mathsf{term}\star$ as the reflexive, transitive, congruent closure of an ordering $\prec'$, defined by the following relations for all $t, s, \bar{t}$:

$$\mathrm{nonval}(s, \bar{t}) \prec' \star_{\mathsf{NonVal}} \qquad \mathrm{val}(s, \bar{t}) \prec' \star_{\mathsf{Val}} \qquad t \prec' \star_{\mathsf{All}}$$

A join operator $t_1 \sqcup t_2$ then follows as the least upper bound of t_1 and t_2. For instance, $(x := 1 + 1) \sqcup (y := 2 + 1) = (\star_{\mathsf{Val}} := \star_{\mathsf{Val}} + 1)$. We can the define the set of concrete terms represented by $\widehat{t} \in \mathsf{term}\star$ as:

$$\gamma\left(\widehat{t}\right) = \left\{t \in \mathsf{term} | t \prec \widehat{t}\right\}$$

The power of this definition of $\mathsf{term}\star$ is that it allows *abstract matching*, which allows the rewriting machinery behind abstract machines to be automatically lifted to abstract terms.

Definition 5.1 (Abstract Matching). A pattern $t_p \in \mathsf{term}$ matches an abstract term $\widehat{t} \in \mathsf{term}\star$ if there are is at least one $t \in \gamma(\widehat{t})$ and substitution σ_t such that $\sigma_t(t_p) = t$. The witness of the abstract match is a substitution $\widehat{\sigma}$ defined:

$$\widehat{\sigma}(x) = \bigsqcup \left\{\sigma_t(x) | t \in \gamma\left(\widehat{t}\right) \wedge \sigma_t(t_p) = t\right\}$$

For example, the abstract term $\star_{\mathsf{All}}$ will match the pattern $v_1 + v_2$, giving a witness $\widehat{\sigma}$ with $\widehat{\sigma}(v_1) = \widehat{\sigma}(v_2) = \star_{\mathsf{Val}}$. We are now ready to state the fundamental property of abstract matching.

Property 2 (Abstract Matching (for terms))**.** *Let t_p be a pattern, and $t \in$ term be a matching term, so that there is a substitution σ with $\sigma(t_p) = t$. Consider a $t' \in$ term$\star$ such that $t' \succ t$. Then t' matches t_p with witness σ', where σ' satisfies $\sigma(x) \prec \sigma'(x)$ for all $x \in dom(\sigma) = dom(\sigma')$, and $t \prec \sigma'(t_p)$.*

We assume there is some external definition of abstract reduction states $\widehat{\mathsf{State}_l}$ (discussed more in §5.2). After doing so, the definitions of abstract terms and states can be lifted to abstract configurations $\mathsf{Conf}\star_l$, lists of abstract configurations $\overline{\mathsf{Conf}\star}$, contexts $\mathsf{Context}\star$, abstract machine states $\mathsf{amState}\star$, and abstract semantic functions $\mathsf{semfunc}\star_l$ etc by transitively replacing all instances of

term and State_l in their respective definitions with term$\star$ and $\widehat{\text{State}_l}$. Abstract matching and the Abstract Matching Property are lifted likewise. To define abstract rewriting, we need a few more preliminaries.

5.2 Abstract Rewriting

Abstract rewriting works by taking the (P)AM execution algorithm of Section 3.3, and using abstract matching in place of regular matching. Doing so effectively simulates the possible executions of an abstract machine on a large set of terms. To define it, we must extend $\prec$ to other components of an abstract machine state.

First, we assume there is some externally-defined notion of abstract reduction states $\widehat{\text{State}_l} \supseteq \text{State}_l$ with ordering $\prec$. There must also be a notion of substitution satisfying $\sigma_1(s) \prec \sigma_2(s)$ for $s \in \widehat{\text{State}_l}$ if $\sigma_1(x) \prec \sigma_2(x)$ for all $x \in \text{dom}(\sigma_1)$. Finally, there must be an abstract matching procedure for states satisfying the Abstract Matching Property. In the common case where State_l is the set of environments mapping variables to terms, this all follows by extending the normal definitions above to associative-commutative-idempotent terms.

We can now abstract matching and the $\prec$ ordering over abstract contexts Context$\star$, abstract configurations $\text{Conf}\star_l$, and lists of abstract configurations $\overline{\text{Conf}\star_l}$ via congruence. We extend $\prec$ over sets of configurations $\mathbb{P}(\text{Conf}\star_l)$ via the multiset ordering[2], and extend $\prec$ over semantic functions pointwise, i.e.: for $f_1, f_2 \in \text{semfunc}_l$, $f_1 \prec f_2$ iff $f_1(x) \prec f_2(x)$ for all $x \in \overline{\text{Conf}\star_l}$.

Because normal AM execution may invoke an external semantic function, we need some way to abstract the result of semantic functions. We assume there is some externally-defined set $\widehat{\text{semfunc}_l}$. Abstract rewriting will be hence parameterized over a "base abstraction" $\beta : \text{semfunc}_l \to \widehat{\text{semfunc}_l}$, satisfying the following property:

$$\beta(f)(\widehat{\overline{c}}) \succ \bigsqcup\{f(\overline{c}) | \overline{c} \in \gamma(\widehat{\overline{c}})\}$$

We are now ready to present abstract rewriting. An AM rule $\left\langle c_p \,\middle|\, K_p \right\rangle \to_l \text{rhs}_p$ for language l is abstractly executed on an AM state $\left\langle c_1 \,\middle|\, K_1 \right\rangle$ using base abstraction β as follows:

(1) Compute the abstract match of $\left\langle c_p \,\middle|\, K_p \right\rangle$ and $\left\langle c_1 \,\middle|\, K_1 \right\rangle$, giving witness $\widehat{\sigma}$; fail if they do not abstractly match.
(2) Recursively evaluate rhs_p as follows:
   - If $\text{rhs}_p = \textbf{let}\, c_{\text{ret}} = \text{func}(\overline{c_{\text{args}}})\, \textbf{in}\, \text{rhs}'_p$, with $\text{func} \in \text{semfunc}_l$, then pick $r \in \beta(\text{func})(\widehat{\sigma}(\overline{c_{\text{args}}}))$ and compute $\widehat{\sigma_r}$ as the witness of abstractly matching c_{args} against r. Define $\widehat{\sigma'}$ by $\widehat{\sigma'}(x) = \widehat{\sigma}(x)$ for $x \in \text{dom}(\widehat{\sigma})$, and $\widehat{\sigma'}(y) = \widehat{\sigma_{\text{args}}}(y)$ for $y \in \text{dom}(\widehat{\sigma_{\text{args}}})$. Fail if no such $\widehat{\sigma'}$ exists. Then recursively evaluate rhs'_p using $\widehat{\sigma'}$ as the new witness.
   - If $\text{rhs}_p = \left\langle c'_p \,\middle|\, K'_p \right\rangle$, return the new abstract AM state $\left\langle \widehat{\sigma}(c'_p) \,\middle|\, \widehat{\sigma}(K'_p) \right\rangle$.

If $\left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle$ steps to $\left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ by abstractly executing an AM rule with base abstraction β, we say that $\left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle \underset{\beta}{\Rightarrow} \left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$. Here is the fundamental property relating abstract and concrete rewriting:

[2]The multiset ordering, also known as the Dershowitz-Manna ordering, extends a base ordering $\prec_D$ on a elements $d \in D$ to an ordering $\prec_{PD}$ on sets $A, B \in \mathbb{P}(D)$ as follows: $A \prec_{PD} B$ if A can be obtained from B by applying the following operations any number of times: (1) removing an element, and (2) replacing a single element $b \in B$ by a finite set of elements $a_1, \ldots, a_n$ such that each $a_i \prec_D b$.

Lemma 5.2 (Lifting Lemma). *If β is a base abstraction, $\langle c_1 \,|\, K_1 \rangle \prec \left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle$, and $\langle c_1 \,|\, K_1 \rangle \to \langle c_2 \,|\, K_2 \rangle$ by rule F, then there is a $\left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ such that $\langle c_2 \,|\, K_2 \rangle \prec \left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ and $\left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle \underset{\beta}{\overset{\frown}{\to}} \left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ by rule F.*

Note that, if β is the identity function, then $\underset{\beta}{\overset{\frown}{\to}}$ is the same as $\to$. Hence, this theorem follows from a more general statement.

Lemma 5.3 (Generalized Lifting Lemma). *Let β_1, β_2 be base abstractions where β_1 is pointwise less than β_2, i.e.: $\beta_1(f)(\overline{c}) \prec \beta_2(f)(\overline{c})$ for all f, c. Suppose $\langle c_1 \,|\, K_1 \rangle \prec \left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle$, and $\langle c_1 \,|\, K_1 \rangle \underset{\beta_1}{\overset{\frown}{\to}} \langle c_2 \,|\, K_2 \rangle$ by rule F. Then there is a $\left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ such that $\langle c_2 \,|\, K_2 \rangle \prec \left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ and $\left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle \underset{\beta_2}{\overset{\frown}{\to}} \left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ by rule F.*

Proof. See Appendix B. □

5.3 Machine Abstractions

In this section, we build the abstractions that define control-flow graphs, i.e.: that reduce the potentially-infinite graph of concrete state transitions into a finite number of CFG nodes. At the end of the day, these will all be trivial to implement, such as "replace all values and function calls with $\star_{\text{Val}}$" to give an expression-level CFG. However, actually defining the correspondence between the abstract and concrete states is quite subtle. Replacing a function call with $\star_{\text{Val}}$, for instance, skips over many intervening steps, potentially including infinite loops. If the language has mutable state, then an abstraction which skips a function call must also overapproximate all possible changes to the environment. Again, this is easy to implement (we'll use the "top" environment $[\star_{\text{Val}} \mapsto \star_{\text{Val}}]$, but it takes some work to pick a useful definition of "abstraction" which justifies this. Fortunately, for the definition we develop in this section, it will still be easy to show that any of the abstractions we use satisfies it.

We now work to define the abstraction preorder $\sqsubseteq$. Let us think about what may be done to turn a concrete state-transition graph into a control-flow graph. It must be able to create a single CFG node describing states with many possible inputs, so it should be able to replace concrete values with $\star_{\text{Val}}$, so it should include $(\prec)$. It should also be able to ignore some steps of computation, so as to e.g.: not create a node for desugaring a while-loop, so it should involve $(\underset{\beta}{\overset{\frown}{\to}})$ somehow. It should also be able to allow edges going out of loop or recursive function without first proving they terminate. So, how exactly may an abstraction skip over an infinite loop?

Definition 5.4 (Nontermination-cutting ordering). Let $\top_l \in \text{State}_l$ be a maximal element of $\widehat{\text{State}_l}$. Consider a state $a = \left\langle (\widehat{t}, \widehat{s}) \,\middle|\, \widehat{K} \right\rangle \in \text{amState}\star$, and let $\widehat{K}'$ be a subcontext of $\widehat{K}$. Suppose that, for all $\langle (t,s) \,|\, K \rangle \prec a$, and for all derivations of the form $\langle (t,s) \,|\, K \rangle \to^* \langle (t',s') \,|\, K' \rangle$, either K is a subterm of K', or there is a subderivation of the form $\langle (t,s) \,|\, K \rangle \to^* \langle (t'',s'') \,|\, K \rangle$ such that $t'' \prec \star_{\text{Val}}$. Then $a \triangleleft \left\langle (\star_{\text{Val}}, \top_l) \,\middle|\, \widehat{K}' \right\rangle$. If a does not satisfy this condition, then there is no a' such that $a \triangleleft a'$.

We now combine the $(\prec)$, $(\triangleleft)$, and $(\underset{\beta}{\overset{\frown}{\to}})$ orderings to fully describe what an abstraction may do. A naive approach would be to take the transitive closure of $(\prec) \cup (\triangleleft) \cup (\underset{\beta}{\overset{\frown}{\to}})$. But this is too loose, and permits abstracting $\langle \textbf{if } \star_{\text{Val}} \textbf{ then } A \textbf{ else } B \,|\, K \rangle$ to $\langle A \,|\, K \rangle$. We hence use a different manner of combining the three relations, which permits any use of $(\prec)$ and $(\triangleleft)$, but which only permits $(\underset{\beta}{\overset{\frown}{\to}})$ when the result is valid for all possible branches:

Definition 5.5 (Ordering $\sqsubseteq$ of amState$\star$). The relation $\sqsubseteq$ is defined inductively as follows: $a \sqsubseteq b$ if any of the following hold:

(1) $a = b$
(2) For some c, $a \prec c$ and $c \sqsubseteq b$
(3) For some c, $a \triangleleft c$ and $c \sqsubseteq b$
(4) For all c such that $a \underset{\beta}{\overset{\frown}{\rightarrow}} c$, $c \sqsubseteq b$

We can now define an abstract machine abstraction to be a pair (α, β), where β is a base abstraction and $\alpha : \text{amState}\star \rightarrow \text{amState}\star$ is a function which is an upper closure operator under the $\sqsubseteq$ ordering, meaning it is monotone and satisfies $x \sqsubseteq \alpha(x)$ and $\alpha(\alpha(x)) = \alpha(x)$. It is well-known that such an upper closure operator establishes a Galois connection between amState$\star$ and the image of α, $\alpha(\text{amState}\star)$ [Nielson et al. 2015]. We will assume that every α is associated with a unique β, and will abbreviate the machine abstraction (α, β) as just α. Appendix C adds a few additional technical restrictions on α. These restrictions have no bearing on interpreted-mode graph generation, but do simplify the proofs for compiled-mode graph generation and rule out some pathological cases.

We now define the abstract transition relation $\underset{\alpha}{\overset{\frown}{\rightarrow}}$ for α as: if $a \underset{\beta}{\overset{\frown}{\rightarrow}} b$, then $a \underset{\alpha}{\overset{\frown}{\rightarrow}} \alpha(b)$. In other words, the abstract transition relation is the same as abstract rewriting, except that the RHS of a transition must always lie within the image of the abstraction α. We now state the fundamental theorem of abstract transitions.

Theorem 5.6 (Abstract Transition). *For $a, b \in$ amState, if α is an abstraction with base abstraction β, and $a \rightarrow b$, then either $b \sqsubseteq \alpha(a)$, or there is a $g \in$ amState$\star$ such that $\alpha(a) \underset{\alpha}{\overset{\frown}{\rightarrow}} g$ and $b \sqsubseteq g$.*

Proof. See Appendix B. □

Following are some example abstractions.

Abstraction: Value-Irrelevance. The value-irrelevance abstraction maps each node val(sym, t) and each constant to $\star_{\text{Val}}$, and each semantic function to the constant $x \mapsto \star_{\text{Val}}$. Combining this with the abstract machine for IMP yields an expression-level control-flow graph, as in Fig. 2b. This also works for Tiger and BaliScript, with a modification discussed in §7.

Abstraction: Expression-Irrelevance. The expression-irrelevance abstraction is like value-irrelevance, but it also "skips" the evaluation expressions by mapping any expression under focus to $\star_{\text{Val}}$. In doing so, it also updates the state to encompass all possible modifications from running e. It can use an arbitrary amount of information about the current state and context to do this, but a safe choice is for is to add the mapping $[\star_{\text{Val}} \mapsto \star_{\text{Val}}]$.

Combining this with the abstract machines for IMP or BaliScript yields a statement-level control-flow graph. (In Tiger, everything is an expression, making this abstraction trivial.)

The Boolean-Tracking Abstraction. The boolean-tracking abstraction is similar to the value-irrelevance abstraction, except that it preserves some **true** and **false** values. A boolean-tracking abstraction is parameterized on a set of *tracking variables* V. For boolean-valued semantic functions such as the $<$ operator, it nondeterministically maps all inputs to the set {**true**, **false**}. In a configuration (t, μ), it preserves all **true** and **false** values in t, as well as the value of $\mu(v)$ for each $v \in V$.

Including the variable "b" in the tracked set, and combining this with the basic-block projection (§5.4) yields the path-sensitive control-flow graph seen in Fig. 2c.

5.4 Projections

A **projection**, also called a **quotient map**, is a function $\pi : \text{amState}\star \to \text{amState}\star$. They are used after constructing the initial CFG to merge together extra nodes, resulting in a blown-down graph.

The definition below comes from Manolios [2001], which presented it in the context of bisimulations. It differs slightly from the standard graph-theoretic definitions, in that it has an extra condition to prevent spurious self-loops.

Definition 5.7. Let (V, E) be a graph with $V \subseteq \text{amState}\star$ and $E \subseteq \left(\text{amState}\star^2\right)$, and let π be a projection. Then the **projected graph** (also: **quotient graph**) is the graph (V', E') satisfying:

(1) $V' = \pi(V)$
(2) For $a \neq b$, $(a, b) \in E'$ iff there is $(c, d) \in E$ such that $(a, b) = (\pi(c), \pi(d))$.
(3) $(a, a) \in E'$ iff, for all $b \in V$ such that $\pi(b) = a$, there is $c \in V$ such that $(b, c) \in E$.

Many of the uses of projections could be accomplished by instead using a coarser abstraction. However, projections have the advantage that they have no additional requirements to prove: they can be any arbitrary function.

Most projections are used to hide internal details of a language's semantics, such as in Fig. 5, which merges away the extra intermediate states when transitioning from the entrance to the loop to the entrance of its condition e, and from the exit of e to the entrance node of the body s. While in this case the projections are implicitly built automatically by the graph-pattern code generator (§6), they can also be defined manually by a projection which identifies nodes matching the first intermediate state, and projects them onto the corresponding second state. Older versions of Mandate did exactly this.

Still, there is one important general projection:

Projection: Basic Block. The basic-block projection inputs $\left\langle c \,\middle|\, K \right\rangle$ and removes all but the last top-level sequence-nodes from c and K, essentially identifying each statement of a basic-block with the last statement in the block. In combination with the expression-irrelevance abstraction, this yields the classic basic-block control-flow graph, as in Fig. 2a.

5.5 Termination

Will the algorithm of the preceding sections terminate in a finite control-flow graph? If the abstraction used is the identity, and the input program may have infinitely many concrete states, the answer is a clear no. If the abstraction reduces everything to $\star_{\text{Val}}$, the answer is a clear yes. In general, it depends both on the abstraction as well as the rules of the language.

If CFG-generation terminates for a program P, that means there are only finitely-many states reachable under the $\underset{\alpha}{\widehat{\rightarrow}}$ relation from P's start state. Term-rewriting researchers call this property "global finiteness," and have proven it is usually equivalent to another property, "quasi-termination" [Dershowitz 1987]. While the literature on these properties can help, there must still be a separate proof for the termination of CFG -generation for every language/abstraction pair. Such a proof may be a tedious one of showing that e.g.: every while-loop steps to **if** e **then** s; **while** e **do** s **else skip**, which eventually steps back to **while** e **do** s, and never grows the stack.

Fortunately, for abstractions which discard context, the graph-pattern generation algorithm of §6 does this analysis automatically. Because the transitions discovered by abstract rewriting for a specific program are a subset of the union of graph patterns for all AST nodes in that program, Theorem C.6 from Appendix C implies that, if graph-pattern generation for a given language and abstraction terminates, then so does interpreted-mode CFG generation for all programs.

6 SYNTAX-DIRECTED CFG GENERATORS

Although it may sound like a large leap to go from generating CFGs for specific programs to statically inferring the control-flow of every possible node, it is actually only little more than running the CFG-generation algorithm of §5 on a term with variables. Indeed, the core implementation in MANDATE is only 22 lines of code. Hence, the description in §2.3 was already mostly complete, and we have only a few details to add. For space reasons, we do not repeat facts given in §2.3. Correctness results and minor details are given in Appendix C.

There two primary points of simplification in §2.3. The chief one is that it did not explain how to run AM rules on a term with variables. A lesser one is that it ignored match types.

Executing Terms with Variables. Abstract rewriting can discover that $\star_{\mathsf{NonVal}} + \star_{\mathsf{NonVal}}$ steps to a state that executes a $\star_{\mathsf{NonVal}}$, but it cannot tell you which $\star_{\mathsf{NonVal}}$ it executes first. Yet there is a simple way to determine if an AM state with variables could be instantiated to match the LHS of an AM rule: if they unify[3]. The result is then the RHS of the rule with the substitution applied. This shows that, from the term $a_{\mathsf{NonVal}} + b_{\mathsf{NonVal}}$, a_{NonVal} is evaluated first.

That operation is called *narrowing*, a technique introduced in 1975 for reasoning with equational theories [Lankford 1975] and later used in functional-logic programming. Indeed, the abstract-rewriting of §5 can almost be viewed as an overapproximation of narrowing that follows each narrowing step with an abstraction that replaces each occurence of the same with distinct fresh variables (i.e.: $\star$ nodes), along with extensions to handle match types and semantic functions.

However, our abstract rewriting differs from conventional narrowing in an important way. Conventional narrowing is actually a ternary relation. For instance, the rule $f(f(x)) \to x$ enables the derivation $f(y) \rightsquigarrow_{[y \mapsto f(y')]} y'$, with the unifying substitution as the third component of the relation. We turn it into a binary relation by applying the substitution on the right, and ignoring it on the left. The resulting relation $\underset{\alpha}{\widehat{\rightsquigarrow}}$ is defined identically to the development of $\underset{\alpha}{\widehat{\rightarrow}}$ given in §5.2 and §5.3, except that the witness $\widehat{\sigma}$ is computed by unification instead of by matching.

This small tweak changes the interpretation of narrowing while preserving its ability to over-approximate rewriting. In conventional narrowing, $[v_1 \mapsto v_2]$ represents an environment with a single, to-be-determined key/value pair. v_1 may unify with both the names "x" and "y", but only in parallel universes. In abstract rewriting, $[v_1 \mapsto v_2]$ represents arbitrary environments, where any name maps to any value.

Graph-Pattern Generation (Now with Match-Types). The final requirement to generate graph patterns for a language l is that the ordering for State_l has a top value $\top_l$, such that, for all $\mu \in \mathsf{State}_l$, $\mu \prec \top_l$. Then, graph pattern generation works as follows: For a node type N, generate the abstract transition graph by narrowing from the start state $\left\langle (N(\overline{x^i_{\mathsf{NonVal}}}), \top_l) \,\middle|\, k \right\rangle$, where the x^i are arbitrary non-value variables, and k is a fresh context variable. Any time a state of the form $\left\langle (e_{\mathsf{NonVal}}, \mu) \,\middle|\, K \right\rangle$ is encountered, instead of narrowing, add a transitive edge to $\left\langle (\star_{\mathsf{Val}}, \top_l) \,\middle|\, K \right\rangle$. Halt at any state $\left\langle (v, \mu) \,\middle|\, k \right\rangle$, where k is the same context variable as the start state, and v, μ are an arbitrary value and reduction state.

From this, we see that the graph pattern in Fig. 5 was slightly simplified. The real graph pattern has each starting variable annotated with NonVal and replaces each $\star$ node with $\star_{\mathsf{Val}}$.

Code-Generation. To turn a graph-pattern into code for a syntax-directed CFG generator, we use a heuristic code-generator which, while unable to handle all possible graph patterns, does work

[3]This assumes a suitable unification procedure exists for State_l, such as AC-unification for environments.

for all graph patterns in IMP, Tiger, and BaliScript. Its high-level workings were in §2.3; some additional details are in Appendix C.

Example graph patterns and generated syntax-directed CFG generators are available in the supplementary material.

7 DERIVING CONTROL FROM A MANDATE

```
name "assn-cong" $
mkRule5 (\x e e' mu mu' ->
 let (gx, ne, ge') = (GVar x, NVar e, GVar e')
 in StepTo (conf (Assign gx ne) mu)
     (LetStepTo (conf ge' mu') (conf ne mu)
      (Build $ conf (Assign gx ge') mu')))
```

Fig. 14

We have implemented our approach in a tool called Mandate. Mandate takes as input an operational semantics for a language as an embedded Haskell DSL, and generates a control-flow graph generator for that language for every abstraction/projection supplied. It can then output a generated CFG to the graph-visualization language DOT. Mandate totals approximately 8300 lines of Haskell. 4200 of those lines are for our language definitions, with the rest in the core engine. Fig. 14 gives an example of this semantics DSL, depicting the AssnCong rule from §3.2.

The primary language implementations are Tiger, BaliScript, and IMP, along with smaller examples and some pathological languages. The Tiger implementation totals 1900 lines, with 650 of those lines giving the 80 SOS rules defining the language. The BaliScript implementation totals 1400 lines, with 500 lines for 60 SOS rules. The rest of the code for both define their syntax and reduction state, the semantic functions, and the interface with an external parser. Our Tiger implementation uses the parser from an open-source Tiger implementation written in Haskell; the BaliScript implementation interfaces with a parser written in C++.

Table 1 lists the CFG-generators we have generated using Mandate. The columns E, S, and P correspond to the expression-level, statement-level, and path-sensitive CFGs from §1, and which are generated by the value-irrelevance, expression-irrelevance, and boolean-tracking abstractions from §5.3. The expression- and statement-level CFG-generators come in interpreted-mode and compiled-mode flavors. In order for CFG-generation to terminate, value-irrelevance for BaliScript and Tiger was extended to also skip over function calls.

Tiger lacks a statement-level CFG-generator because it is a functional language, and everything is an expression (no statements). Unfortunately, Mandate cannot currently support the boolean-tracking abstraction for Tiger and BaliScript without significant upgrades to its unification algorithms, because heaps have a more complicated structure than in IMP.

Table 1

	Interpreted			Compiled	
	E	S	P	E	S
IMP	✓	✓	✓	✓	✓
Tiger	✓	N/A	×	✓	N/A
BaliScript	✓	✓	×	✓	✓

It is difficult to make anything other than a qualitative evaluation of the CFGs generated in interpreted mode other than to eyeball their shape; even so, there is still endless room for tuning them with custom projections. The compiled-mode CFG-generators, however, are short and readable. Particularly impressive is the code generated for Tiger for-loops, which cuts through many layers of syntactic-sugar. The supplemental material contains code and graph-patterns for all compiled-mode CFG-generators, many CFGs generated by interpreted-mode generators, and some additional discussion of specific outputs.

8 RELATED WORK

Work with Related Goals. Ours joins a small-but-growing body of work on mechanizing the generation of programming tools. Others include generating static analyzers via alternate executions of interpreters [Darais et al. 2017, 2015; Sergey et al. 2013], using an executable language semantics as a tool (by the K Framework [Roşu and Şerbănuţă 2010]), and work in the language-parametric construction of programming tools such as program transformations [Koppel et al. 2018; Lin et al. 2017]. Our work is similar to tools built with the K Framework in that both start with a semantics; ours differs by transforming the semantics into an applied tool, whereas K is limited to applications directly based on equational reasoning, namely interpreters and symbolic executors.

Work Using Similar Techniques. There are several works that transfrom semantic rules derive other artifacts. We can group the related work into three categories.

First, there are a few projects that transform semantics for one language into semantics for a related language. Examples include transforming static and dynamic semantics rules to support gradual types [Cimini and Siek 2016, 2017], and deriving new rules for types and scoping of syntactic sugar [Pombrio and Krishnamurthi 2018; Pombrio et al. 2017]. While our work is not in this category (as its output is not another semantics), we nonetheless took inspiration from these works to help communicate our own.

Second, there are projects that transform semantics presented in one formalism into identical semnatics in a different formalism. Danvy and his students are the main contributors here.

When we first began this project, we planned to simply look up an algorithm to convert SOS to abstract machines, but surprisingly found none existed. Almost all work in this space is by Danvy and associates, and, while their papers focus on individual formalisms, Danvy personally told us that he is not interested in small-step semantics because it would not require new techniques over his prior work, and that he is similarly uninterested in creating a general algorithm, when he's already sketched how to do the transformation for a single example [Danvy 2008].

We next planned to convert the SOS to reduction semantics and then apply Danvy's algorithm for converting reduction semantics to an abstract machine [Danvy and Nielsen 2004]. However, we found the assumptions of this algorithm too limiting, as it prohibits any use of external semantic functions or state. Rather than extend this algorithm, we found it much easier to use PAM as an intermediate step, because its nature as a modified term-rewriting system gives us access to unification-based techniques for analyzing and transforming it. For instance, while proving up-rules invertible is a reachability search taking 20 lines of code, proving the equivalent property for reduction semantics, unique decomposition, took a 20-page paper [Xiao et al. 2001].

Ultimately, this leaves our work as **the first published algorithm for converting SOS to abstract machines**, though we do use techniques pioneered by Danvy.

Another contrast between our work and previous works on the correspondence of formalisms is that prior work predominantly focuses on the call-by-name/value lambda calculi. To our knowledge, the Tiger and BaliScript languages in our work are the largest languages in any work to undergo automated conversion between two forms of semantics.

An older work on this topic is by Hannan and Miller [1992]. They give a few examples of manually deriving an abstract machine from big-step semantics, using a sequence of up to 6 hand-proven transformations. Ager [2004] continues this work, providing a simpler and automated approach, though with some additional limitations, such as needing nondeterminism to execute if-statements.

Third are the projects based on abstracting abstract machines. The first known work explicitly on abstract interpretation of abstract machines was by Midtgaard and Jensen [Midtgaard and Jensen 2008, 2009], though a modern reader may also consider Jones [1981] to be an earlier example. This was followed by the "abstracting abstract machines" line of work began by Van Horn and

Might [Johnson and Horn 2014; Johnson et al. 2013; Van Horn and Might 2010; Wei et al. 2018]. It takes a substantially different flavor from our own, due to its focus on higher-order analyses of pure functional languages. A key technical difference is the choice of abstraction operator: they use variants of store-bounding, whereas we use syntactic abstractions designed to make the abstract state space resemble a conventional CFG. The use of these syntactic abstractions allows our algorithm to automatically abstract the transition rules of an abstract machine via abstract rewriting. Their paper's approach only automatically abstracts reads and writes to an abstract store; the abstract transition steps are still manually defined. For example, the algorithm in our paper can take in a rule like $\left\langle \textbf{true} \,\middle|\, k \circ [(\textbf{if } \square_t \textbf{ then } s_1 \textbf{ else } s_2, \square_k)] \right\rangle \to \left\langle s_1 \,\middle|\, k \right\rangle$ and the corresponding rule for **false**, and deduce that if-statements may step into either branch because both rules match a single $\star$ state. The approach in their paper can at best nondeterministically evaluate an expression to separate {**true**, **false**} states and then match both if-rules, which produces an un-CFG-like graph where the branching happens prior to the if-statement.

Abstract Rewriting. Abstract rewriting was introduced by Bert and Echahed in the early 90's [Bert and Echahed 1995; Bert et al. 1993] and has received little attention since. We can thus only compare to their work. While the details differ substantially owing to their different focus (approximating the possible normal forms of a term), it has some major common elements with our development: they split nodes into "constructors" and "completely-defined operators," resembling our value/nonvalue split, and use a $\top$ node with similar semantics to our $\star$. A major point of departure in their development is that, in their system, each abstract step must overappproximate all concrete transitions from an abstract term.

Outside of Bert & Echahed, there are a few works which share minor details or terminology with our development of abstract rewriting. In the broadest sense, abstract unification means taking some algorithm that uses unification, and replacing the unification with some other operation. Prior work in this sense comes from work on the abstract interpretation of logic programs [Cousot and Cousot 1992; King and Longley 1995]. These algorithms commonly replace unification with a simple operation such as tracking sharing and linearity, for the purpose of, e.g.: eliminating backtracking or the occurs-check (see §10 of Cousot and Cousot [1992] for a literature review).

In contrast, the abstract matching procedure in our work (and in Bert & Echahed) is an extension of normal matching to a set of abstract terms, consisting of normal terms augmented with a set of "blown-down" terms. This technique appears absent from the literature on static analysis of logic programs, for it is not useful for traditional static analysis by abstract interpretation, as the domain of abstract terms is infinite (meaning: our algorithm cannot compute a single control-flow graph which usefully describes all possible programs).

In §6, we explained how abstract rewriting is similar to narrowing, but different in an important way. There is a long tradition in narrowing of proving lifting lemmas similar to our own; the first comes from Hullot [1980].

Apart from Bert & Echahed, we have found a couple papers that use syntactic abstraction, albeit in a different form. An early use is the "star abstraction" of Codish et al. [1993, 1991], which merges identical subterms of a tree, and is unrelated to the similarly-named abstraction in this paper. Schmidt [1996] uses this to merge identical processes in a CCS-like system, bounding executions to aid in model-checking. Although it is not discussed in the paper, Johnson et al. [2013] does some syntactic abstraction in the implementation, representing all numbers as identical number nodes.

The term "abstract matching" also has an unrelated meaning in the model-checking community, where it refers to finding equivalences between abstract states [Păsăreanu et al. 2005].

The ARM abstract rewriting machine [Kamperman and Walters 1993] provides a compact instruction set for executing term-rewriting systems efficiently. It handles ordinary rewriting, rather than abstract rewriting in our sense.

Control-Flow Analysis. Many papers have been written on control-flow analysis. Older research tries to manually construct a complicated analysis of programs with highly-dynamic control flow. Our work tries to automatically construct CFG-generators from first principles. Our goal is not to analyze complex programs, but to match the work of hand-written CFG-generators with a minimum of user input.

As such, we do not consider this work as part of the literature on control-flow analysis. Nonetheless, we give a brief overview of control-flow analysis here.

Research on control-flow analysis typically focuses on functional languages. Perhaps the most famous work is the k-CFA analysis of Shivers [1991]. This and many other works frame their analysis as an abstract interpretation of executions, though there are too many approaches to describe here; Midtgaard [2012] gives an extensive survey. Owing to their different emphasis, these works uniformly have three limitations that make them unsuitable for our problem of automatically deriving CFG-generators:

(1) While they explain how a human could define a new analysis for a different languages, their analyses are ultimately manually defined for each language. They typically further repeat this manual construction for every abstraction used.
(2) They check that their result safely approximates executions, but pay no attention to the shape of the graph.
(3) Most importantly, they manually partition program states into equivalence classes. That is, they manually annotate the program with labels or program-points, using these as CFG nodes. This is a hindrance to both automation and theory, as most type theories do not contain labels. A major motivation of this work is to support our ongoing work on combining analyses with different notions of program point (i.e.: partition program states differently), which makes not hardcoding them especially important.

Nonetheless, there are two works from this field which deserve special mention. The first is Jones [1981], which is the first to define control-flow as an abstract interpretation of executions. Through a modern lens, it is also the first to do so by abstracting abstract machines: it presents an abstraction of a custom abstract machine resembling a CC machine [Felleisen et al. 2009]. It shares the limitations mentioned above, although its representation of program points is subtle: it uses a set of tokens representing the different function applications of the source program.

The second is Jagannathan and Weeks [1995], due to its focus on generality, explicit construction of graphs, and attempt to relate their construction to an operational semantics. It is also the most extreme in its use of manual program-point annotations, going as far as to design an abstract machine with an explicit program counter.

9 CONCLUSION

This work presented an algorithm for constructing control-flow graphs from first principles, and provided the world's first CFG-generator generator. Yet our work also furthers three larger goals.

First, we have provided an answer to "what is a control-flow graph?" beyond the vague "a CFG is an abstraction of control-flow:" A CFG is a projection of the transition graph of abstracted abstract machine states. This fulfills our original impetus for this work, that of needing to create static analyzers with exotic notions of "program point."

Second, we have introduced abstract rewriting as a simple yet powerful technique for deriving tools from a language's semantics. We are excited by the prospect of applying this technique to

derive other artifacts from language semantics, such as a symbol-table generator from the typing abstract-machine [Sergey and Clarke 2011].

Third, we have used a language's semantics to derive a language tool entirely unlike a semantics. Though it's long been known that language semantics can be executed to obtain an interpreter or even a symbolic-executor [Roşu and Şerbănuţă 2010], we see our contribution as qualitatively different, and an important step towards the dream of being able to write down a language's syntax and semantics and automatically derive all desired tools.

REFERENCES

Mads Sig Ager. 2004. From Natural Semantics to Abstract Machines. In *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, 245–261.

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A Functional Correspondence between Evaluators and Abstract Machines. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming*. ACM, 8–19.

Andrew W. Appel. 1998. *Modern Compiler Implementation in ML*. Cambridge University Press.

Franz Baader and Tobias Nipkow. 1999. *Term Rewriting and All That*. Cambridge University Press.

Didier Bert and Rachid Echahed. 1995. Abstraction of Conditional Term Rewriting Systems. In *Logic Programming, Proceedings of the 1995 International Symposium, Portland, Oregon, USA, December 4-7, 1995*. 162–176. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6300583

Didier Bert, Rachid Echahed, and Bjarte M Østvold. 1993. Abstract Rewriting. In *International Workshop on Static Analysis*. Springer, 178–192.

Małgorzata Biernacka. 2006. *A Derivational Approach to the Operational Semantics of Functional Languages*. Ph.D. Dissertation. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark.

Denis Bogdanas and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 445–456.

Matteo Cimini and Jeremy G. Siek. 2016. The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 443–455. https://doi.org/10.1145/2837614.2837632

Matteo Cimini and Jeremy G. Siek. 2017. Automatically Generating the Dynamic Semantics of Gradually Typed Languages. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 789–803. http://dl.acm.org/citation.cfm?id=3009863

Michael Codish, Saumya K. Debray, and Roberto Giacobazzi. 1993. Compositional Analysis of Modular Logic Programs. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*. 451–464. https://doi.org/10.1145/158511.158703

Michael Codish, Moreno Falaschi, and Kim Marriott. 1991. Suspension Analysis for Concurrent Logic Programs. In *Logic Programming, Proceedings of the Eigth International Conference, Paris, France, June 24-28, 1991*. 331–345.

Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation and Application to Logic Programs. *The Journal of Logic Programming* 13, 2-3 (1992), 103–179.

Olivier Danvy. 2008. Defunctionalized Interpreters for Programming Languages. In *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. 131–142. https://doi.org/10.1145/1411204.1411206

Olivier Danvy and Jacob Johannsen. 2010. Inter-Deriving Semantic Artifacts for Object-Oriented Programming. *J. Comput. System Sci.* 76, 5 (2010), 302–323.

Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. 2012. On Inter-Deriving Small-Step and Big-Step Semantics: A Case Study for Storeless Call-by-Need Evaluation. *Theoretical Computer Science* 435 (2012), 21–42.

Olivier Danvy and Lasse R Nielsen. 2004. Refocusing in Reduction Semantics. *BRICS Report Series* 11, 26 (2004).

David Darais, Nicholas Labich, Phúc C Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 12.

David Darais, Matthew Might, and David Van Horn. 2015. Galois Transformers and Modular Abstract Interpreters: Reusable Metatheory for Program Analysis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 552–571. https://doi.org/10.1145/2814270.2814308

Nachum Dershowitz. 1987. Termination of Rewriting. *Journal of Symbolic Computation* 3, 1-2 (1987), 69–115.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. Mit Press.

Jeanne Ferrante and Joe D Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (1987), 319–349.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*. 237–247. https://doi.org/10.1145/155090.155113

John Hannan and Dale Miller. 1992. From Operational Semantics to Abstract Machines. *Mathematical Structures in Computer Science* 2, 4 (1992), 415–459.

Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 336–345. https://doi.org/10.1145/2737924.2737979

Jean-Marie Hullot. 1980. Canonical Forms and Unification. In *International Conference on Automated Deduction*. Springer, 318–334.

Husain Ibraheem and David A Schmidt. 1997. Adapting Big-Step Semantics to Small-Step Style: Coinductive Interpretations and "Higher-Order" Derivations. *Electronic Notes in Theoretical Computer Science* 10 (1997), 121.

Suresh Jagannathan and Stephen Weeks. 1995. A Unified Treatment of Flow Analysis in Higher-Order Languages. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. 393–407. https://doi.org/10.1145/199448.199536

James Ian Johnson and David Van Horn. 2014. Abstracting Abstract Control. In *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 11–22. https://doi.org/10.1145/2661088.2661098

James Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. 2013. Optimizing Abstract Abstract Machines. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 443–454. https://doi.org/10.1145/2500365.2500604

Neil D Jones. 1981. Flow Analysis of Lambda Expressions. In *International Colloquium on Automata, Languages, and Programming*. Springer, 114–128.

Jasper FT Kamperman and Humphrey Robert Walters. 1993. ARM Abstract Rewriting Machine. (1993).

Andy King and Mark Longley. 1995. Abstract Matching Can Improve on Abstract Unification. (1995).

James Koppel, Varot Premtoon, and Armando Solar-Lezama. 2018. One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 122.

Dallas S Lankford. 1975. *Canonical Inference*. University of Texas, Department of Mathematics and Computer Sciences.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 55–56.

Panagiotis Manolios. 2001. Mechanical Verification of Reactive Systems. (2001).

Jan Midtgaard. 2012. Control-flow Analysis of Functional Programs. *ACM Comput. Surv.* 44, 3, Article 10 (June 2012), 33 pages. https://doi.org/10.1145/2187671.2187672

Jan Midtgaard and Thomas P. Jensen. 2008. A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*. 347–362. https://doi.org/10.1007/978-3-540-69166-2_23

Jan Midtgaard and Thomas P. Jensen. 2009. Control-flow Analysis of Function Calls and Returns by Abstract Interpretation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 287–298. https://doi.org/10.1145/1596550.1596592

Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of Program Analysis*. Springer.

Nathaniel Nystrom, Michael R Clarkson, and Andrew C Myers. 2003. Polyglot: An Extensible Compiler Framework for Java. In *International Conference on Compiler Construction*. Springer, 138–152.

Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 346–356.

Corina S Păsăreanu, Radek Pelánek, and Willem Visser. 2005. Concrete Model Checking with Abstract Matching and Refinement. In *International Conference on Computer Aided Verification*. Springer, 52–66.

Justin Pombrio and Shriram Krishnamurthi. 2018. Inferring Type Rules for Syntactic Sugar. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 812–825.

Justin Pombrio, Shriram Krishnamurthi, and Mitchell Wand. 2017. Inferring Scope through Syntactic Sugar. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 44.

Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

David A Schmidt. 1996. Abstract Interpretation of Small-Step Semantics. In *LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*. Springer, 76–99.

Ilya Sergey and Dave Clarke. 2011. From Type Checking by Recursive Descent to Type Checking with an Abstract Machine. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*. ACM, 2.

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic Abstract Interpreters. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 399–410. https://doi.org/10.1145/2491956.2491979

Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. Dissertation. PhD thesis, Carnegie Mellon University.

Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the Definition of Incremental Program Analyses. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 320–331.

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *15th ACM SIGPLAN International Conference on Functional Programming, ICFP'10*.

Guannan Wei, James Decker, and Tiark Rompf. 2018. Refunctionalization of Abstract Abstract Machines: Bridging the Gap Between Abstract Abstract Machines and Abstract Definitional Interpreters (Functional Pearl). *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 105.

Yong Xiao, Amr Sabry, and Zena M. Ariola. 2001. From Syntactic Theories to Interpreters: Automating the Proof of Unique Decomposition. *Higher-Order and Symbolic Computation* 14, 4 (2001), 387–409. https://doi.org/10.1023/A:1014408032446

A CORRECTNESS OF SOS-AM TRANSLATION

This section proves the correspondence between the operational semantics and abstract machine. We begin by proving the correspondence between operational semantics and PAM. We then prove an important consequence of invertible up rules, and then prove the correspondence between PAM and the abstract machine.

The core idea of the correspondence is simple: The PAM emulates the SOS because each PAM rule was explicitly constructed to correspond to an RHS fragment of the SOS. The PAM and AM are equivalent because the AM merely removes some redundant steps from the PAM, and because the fused rules in the AM each correspond to several rules in the PAM. However, a PAM derivation may have some "false starts" corresponding to a partially-applied SOS rule, and so there are some additional technical details to explain which states are included in the correspondences.

A.1 SOS-PAM Correspondence

Theorem 4.1. *$c_1 \rightsquigarrow_l^* c_2$ if and only if, for all contexts K, $\langle c_1 \,|\, K\rangle\downarrow \hookrightarrow_l^* \langle c_2 \,|\, K\rangle\uparrow$*

The forward direction is easy, because the PAM rules were designed to follow in lockstep with each component of the SOS rules. The reverse-direction appears harder, but is rendered easy by two important facts:

Observation 1. *From the LHS of each PAM rule, it is possible to identify the arguments s, K, rhs of* sosRhsToPam *that generated it.*

This is because each case generates rules in a distinct form, and each generated rule contains all of the parameters of sosRhsToPam. (Note that LHSs originating on line (1) of Fig. 12 must be non-values, while those from line (4) must be a values.)

Property 3 (Sanity of Phase)**.** *The following three properties hold:*

(1) *If $\langle c \,|\, K\rangle\updownarrow_c \hookrightarrow \langle c' \,|\, K'\rangle\updownarrow_{c'}$ and K' contains strictly more stack frames than K, then $\updownarrow_c = \updownarrow_{c'} = \downarrow$, and $K' = K \circ f$ for some f.*
(2) *If $\langle c \,|\, K\rangle\updownarrow_c \hookrightarrow \langle c' \,|\, K'\rangle\updownarrow_{c'}$ and K' contains strictly fewer stack frames than K, then $\updownarrow_{c'} = \uparrow$, and $K = K' \circ f$ for some f.*
(3) *If the PAM rules for language l have no up-down rules, and $\langle c \,|\, K\rangle\uparrow \hookrightarrow_l \langle c' \,|\, K'\rangle\updownarrow_{c'}$ without using the reset rule, then $\updownarrow_{c'} = \uparrow$, and $K = K' \circ f$ for some f.*

These properties follow by inspection of the possible rules. We now prove Theorem 4.1 as a corollary of a stronger result:

Lemma A.1. *$c_1 \rightsquigarrow_l c_2$ if and only if, for all K, there is a derivation $\langle c_1 \,|\, K\rangle\!\downarrow \hookrightarrow_l^* \langle c_2 \,|\, K\rangle\!\uparrow$ which does not use the reset rule. This derivation must use the same sequence of rules regardless of K.*

Proof. We address each direction.

($\Rightarrow$): Let K be an arbitrary context, and consider a derivation of $c_1 \rightsquigarrow c_2$. By induction on the last SOS rule applied, we prove there exists a derivation $\langle c_1 \,|\, K\rangle\!\downarrow \hookrightarrow^* \langle c_2 \,|\, K\rangle\!\uparrow$.

Let the last SOS rule used take the form $c_1 \rightsquigarrow C_1[\ldots C_n[c_2]]$, where each C_i is a single RHS fragment. Let $R = C_{i+1}[\ldots C_n[c_2]]$. We induct again on i to show that $\langle c_1 \,|\, K\rangle\!\downarrow \hookrightarrow^* s_i$, where:

(1) If $i = 0$ (base case), then $s_i = \langle c_1 \,|\, K\rangle\!\downarrow$.
(2) If C_i is of the form **let** $[c \rightsquigarrow c']$ **in** $\square$, then $s_i = \langle c' \,|\, K \circ [c' \to R]\rangle\!\uparrow$.
(3) If C_i is of the form **let** $c' = f(\overline{c})$ **in** $\square$, then $s_i = \langle c' \,|\, K \circ [c' \to R]\rangle\!\downarrow$.

and further, the PAM system contains rules generated by sosRhsToPam(s_i, k, R) for some context variable k. We handle each case:

(1) Satisfied by the empty transition sequence and definition of sosRuleToPam.
(2) Then there is a rule that $s \hookrightarrow \langle c \,|\, K\,[c' \to R]\rangle\!\downarrow$. By inversion of the SOS derivation, we must have that $c \rightsquigarrow c'$. Then, by the outer induction hypothesis, $\langle c \,|\, K\,[c' \to R]\rangle\!\downarrow \hookrightarrow^* \langle c' \,|\, K\,[c' \to R]\rangle\!\uparrow$. The rest follows by line (3) of the definition of sosRhsToPam.
(3) Then there is a rule that $s \hookrightarrow \langle c' \,|\, K\,[c' \to R]\rangle\!\downarrow$. The rest follows by line (4) of the definition of sosRhsToPam.

The inner induction hypothesis for C_n tells us that a rule $s \hookrightarrow \langle c_2 \,|\, K\rangle\!\uparrow$ must exist, finishing the proof of the outer induction.

($\Leftarrow$): We proceed by a strong induction on all derivations of the form $\langle c_1 \,|\, K\rangle\!\downarrow \hookrightarrow_l^* \langle c_2 \,|\, K\rangle\!\uparrow$. Consider the first PAM rule of the derivation. Because it may not depend on K, it must have an LHS generated on line (1) of Fig. 12. By Observation 1, we can hence reconstruct the entire originating SOS rule, $c_1 \rightsquigarrow C_1[\ldots C_n[c_2]]$. We show that $c_1 \rightsquigarrow c_2$ by this rule.

The proof proceeds similarly to the forward direction, so we omit more details. We induct over i, and show that there must be a prefix of the derivation $\langle c_1 \,|\, K\rangle\!\downarrow \hookrightarrow^* s_i$, where $s_0 = \langle c_1 \,|\, K\rangle\!\downarrow$ and s_i is a state corresponding to the computation of C_i. Each s_i matches the LHS of the PAM rule used in the derivation; Observation 1 tells us this rule must be the one generated for C_{i+1}. The only interesting case is for recursive steps; there, $s_i = \langle c \,|\, K'\rangle\!\downarrow$, and the Sanity of Phase properties dictate there must be a later state in the derivation $\langle c' \,|\, K'\rangle\!\uparrow$; applying the outer inductive hypothesis finishes this case.

□

A.2 Invertibility

Our goal is to show that, if all up-rules are invertible (Definition 3.2), then $\langle c \,|\, K\rangle\!\uparrow \hookrightarrow^* \langle c \,|\, K\rangle\!\downarrow$ when c is a non-value, justifying the optimizations of §3.6, and characterizing which PAM states are and are not removed when translating a derivation to AM. However, there are a few technical restrictions on this.

Definition A.2. A configuration/context pair (c, K) is **non-stuck** if $\langle c \,|\, K\rangle\!\uparrow \hookrightarrow^* \langle c' \,|\, \mathbf{empty}\rangle\!\uparrow$ for some c'.

Because each PAM rule corresponds to part of an SOS rule, our definition of non-stuckness is different from the usual one: it is intended to exclude terms which correspond to a partial match

on an SOS rule. A single step $c_1 \rightsquigarrow c_2$ in the SOS corresponds to a sequence $\langle c_1 \,|\, \textbf{empty}\rangle\!\downarrow \hookrightarrow^* \langle c_2 \,|\, \textbf{empty}\rangle\!\uparrow$ in the PAM, so a state is non-stuck if it can complete the current step. Stuck states result from SOS rules which only partially match a term. For example, the SOS rule

$$(a.b := v, \mu) \rightsquigarrow \textbf{let } (r, \mu') = \text{Lookup}((a, \mu)) \textbf{ in let false} = \text{ContainsField}(r, b) \textbf{ in } (\textbf{error}, \mu)$$

decomposes into 3 PAM rules. Assuming Lookup succeeds, the first PAM rule brings $\langle (a.b := v, \mu) \,|\, K\rangle\!\downarrow$ into the state

$$\langle (r, \mu') \,|\, K \circ [\textbf{let false} = \text{ContainsField}(\square_t, b) \textbf{ in } (\textbf{error}, \mu)]\rangle\!\downarrow$$

If $\textbf{false} \neq \text{ContainsField}(r, b)$, then this will be a stuck state.

Excluding stuck states is enough to prove the general Invertibility Lemma:

Lemma A.3 (Invertibility). *If all up-rules for l are invertible, and there are no up-down rules for l other than the reset rule, then, for any non-stuck non-value (c, K), $\langle c \,|\, K\rangle\!\uparrow \hookrightarrow^* \langle c \,|\, K\rangle\!\downarrow$.*

Proof. Consider a derivation $\langle c \,|\, K\rangle\!\uparrow \hookrightarrow^* \langle c' \,|\, \textbf{empty}\rangle\!\uparrow$. Because there are no up-down rules, each step must follow from an up-rule. Hence, each step is invertible. Applying each inverted step gives a new derivation $\langle c' \,|\, \textbf{empty}\rangle\!\downarrow \hookrightarrow^* \langle c \,|\, K\rangle\!\downarrow$.

This requires that the term in the RHS of each step is a non-value, which also follows because each rule is invertible, and hence the RHS can be reduced. □

This motivates the definition of an **inversion sequence**. We add the condition about the reset rule to prevent the definition from including arbitrarily large subsequences of a nonterminating execution.

Definition A.4. An **inversion sequence** beginning at $\langle c \,|\, K\rangle\!\uparrow$ is a sequence of transitions $\langle c \,|\, K\rangle\!\uparrow \hookrightarrow^* \langle c \,|\, K\rangle\!\downarrow$ which contains at most one application of the reset rule.

This idea of an inversion sequence partitions a derivation $\langle c_1 \,|\, K_1\rangle\!\downarrow \hookrightarrow^* \langle c_2 \,|\, K_2\rangle\!\downarrow$ into two parts: the inversion sequences, which do redundant work, and the remainder, which we call the **working steps**. Sometimes a derivation must be extended to contain a complete inversion sequence, which is then eliminated upon the conversion to an abstract machine.

Definition A.5. A reduction $\langle c_1 \,|\, K_1\rangle\!\updownarrow_1 \hookrightarrow \langle c_2 \,|\, K_2\rangle\!\updownarrow_2$ within a derivation is a **working step** if the derivation cannot be extended so that $\langle c_1 \,|\, K_1\rangle\!\updownarrow_1$ is part of an inversion sequence.

Observation 2. *If (c, K) is a non-stuck non-value and all up-rules are invertible, then, by the Determinism Assumption, all sequences $\langle c \,|\, K\rangle\!\uparrow \hookrightarrow^* \langle c' \,|\, K'\rangle\!\updownarrow$ may be extended to contain an inversion sequence starting at $\langle c \,|\, K\rangle\!\uparrow$.*

Corollary A.6. *If a reduction $\langle c \,|\, K\rangle\!\uparrow \hookrightarrow \langle c' \,|\, K'\rangle\!\uparrow$ cannot be extended to contain an inversion sequence starting at $\langle c \,|\, K\rangle\!\uparrow$, then either (c, K) is stuck or c is a value.*

With these extra properties, we are now ready to exactly state the PAM-AM correspondence.

A.3 PAM-AM Correspondence

Intuitively, a derivation in the AM $\langle c_1 \,|\, K_1\rangle \rightarrow^* \langle c_2 \,|\, K_2\rangle$ is the same as a derivation in the PAM, but with the inversion sequences cut out and with some consecutive steps merged into one. We prove this in steps. First, we show that if $\langle c_1 \,|\, K_1\rangle\!\downarrow \hookrightarrow^* \langle c_2 \,|\, K_2\rangle\!\downarrow$, and the last transition is a working step, then $\langle c_1 \,|\, K_1\rangle \longrightarrow^* \langle c_2 \,|\, K_2\rangle$. Next we prove that every derivation in the Unfused AM corresponds to a derivation in the fused AM, but with some consecutive steps merged. We

also prove the reverse theorem, which is easier to state, as every state in an AM derivation has a corresponding state in the PAM.

The forward direction comes first. This version of the theorem starts with transitions between down-states, to simplify consideration of which states may be eliminated on conversion to AM.

Theorem 4.5 (PAM-Unfused AM: Forward). *Suppose* $\langle c \,|\, K\rangle\downarrow \hookrightarrow_l^* \langle c' \,|\, K'\rangle\downarrow$, $\langle c' \,|\, K'\rangle\downarrow$ *is non-stuck, and the derivation's last step is working. Then there is a derivation* $\langle c \,|\, K\rangle \longrightarrow_l^* \langle c' \,|\, K'\rangle$.

Proof. First, recall that, if $\longrightarrow_l$ is defined, then all up-rules for l are invertible.

Consider a derivation $\langle c \,|\, K\rangle\downarrow \hookrightarrow_l^* \langle c' \,|\, K'\rangle\downarrow$. Remove all maximal inversion sequences. Then remove all phases from the PAM states, resulting in AM states. This means that an inversion sequence $\langle c_1 \,|\, K_1\rangle\uparrow \hookrightarrow^* \langle c_1 \,|\, K_1\rangle\downarrow$ is replaced with a single state $\langle c_1 \,|\, K_1\rangle$.

If we can show that all PAM rules for all remaining steps of the derivation have a corresponding rule in the unfused abstract machines, then we will be done. Note that the only PAM rules without a corresponding rule in the AM are those of the form $\langle c \,|\, K\rangle\downarrow \hookrightarrow \langle c \,|\, K\rangle\uparrow$, and those of the form $\langle c_1 \,|\, K_1\rangle\uparrow \hookrightarrow \langle c_2 \,|\, K_2\rangle\uparrow$ where c_1 is a non-value. The former correspond to stutter steps in the AM and may be ignored. For the latter, because of the Determinism Assumption and Corollary A.6, all such transitions must be part an inversion sequence, and were hence removed. □

Theorem 4.6 (PAM-Unfused AM: Backward). *If* $\langle c \,|\, K\rangle \longrightarrow_l^* \langle c' \,|\, K'\rangle$, *then there are phases* $\updownarrow_c$ *and* $\updownarrow_{c'}$ *such that* $\langle c \,|\, K\rangle\updownarrow_c \hookrightarrow_l^* \langle c' \,|\, K'\rangle\updownarrow_{c'}$.

Proof. Consider a derivation $\langle c \,|\, K\rangle \longrightarrow_l^* \langle c' \,|\, K'\rangle$. For each rule of the unfused abstract machine used in this derivation, consider the corresponding PAM rule that generated it.

Let $\updownarrow_c$ be the phase of the LHS of the first such rule. We will show that there is $\updownarrow_{c'}$ such that $\langle c \,|\, K\rangle\updownarrow_c \hookrightarrow_l^* \langle c' \,|\, K'\rangle\updownarrow_{c'}$, obtained by replacing each AM rule with its corresponding PAM rule and by inserting inversion sequences. We proceed by induction on the derivation.

Consider the last step of the derivation $\langle c'' \,|\, K''\rangle \longrightarrow_l \langle c' \,|\, K'\rangle$. Consider the AM rule of this last step, and let G be the PAM rule from which it originated, $\langle c_G^1 \,|\, K_G^1\rangle\updownarrow_G^1 \overset{G}{\hookrightarrow}_l C_G[\,\langle c_G^2 \,|\, K_G^2\rangle\updownarrow_G^2\,]$. If G matches, it would finish the proof. In the base case where there is only one step, taking $\updownarrow_c$=$\updownarrow_G^1$ and $\updownarrow_{c'}$=$\updownarrow_G^2$ suffices to make it match.

If there is more than one step in the derivation, then, by the induction hypothesis, there are phases $\updownarrow_c$ and $\updownarrow_{c''}$ such that $\langle c \,|\, K\rangle\updownarrow_c \hookrightarrow_l^* \langle c'' \,|\, K''\rangle\updownarrow_{c''}$. Consider the second-to-last step of the AM derivation $\langle c''' \,|\, K'''\rangle \longrightarrow_l \langle c'' \,|\, K''\rangle$ and its generating AM rule, and let the corresponding PAM rule be F. If the RHS of F matches the LHS of G, we would be done. We hence must consider all PAM rules F, G such that the RHS of F and LHS of G match except for the phase, meaning they would erroneously match upon conversion to AM rules. Call such an (F, G) a *confused pair*. We perform case analysis on Fig. 12 to find all possible confused pairs, and for each find a derivation $\langle c'' \,|\, K''\rangle\updownarrow_{c''} \hookrightarrow_l^* \langle c' \,|\, K'\rangle\updownarrow_{c'}$.

Fig. 12 gives 3 possible forms of PAM RHSs, generated on lines (2), (3), and (4), and 3 possible LHSs, generated on lines (1), (3), and (4). Note that some of these only match values/non-values, and that, because of the prohibition on up-down rules, all rules using the LHS from line (3) will be restricted to only match values. This leaves only 3 possible forms for a confused pair: using the RHS/LHS generated on lines (2)/(1), (2)/(4), and (4)/(3). We analyze each in turn. We find that the first case is desirable, as it results from removing inversion sequences, while the other two are benign, as another rule must exist that does match.

- (2)/(1): In this case, the RHS of a rule F, $s \overset{F}{\hookrightarrow}_l \langle c_F \,|\, K_F\rangle\uparrow$, matches the LHS of a rule G, $\langle c_G \,|\, K_G\rangle\downarrow \overset{G}{\hookrightarrow}_l t$, where $c_F = c_G$ upon matching with c''. $c_F = c_G$ must be a non-value

because c_G originates from a SOS rule $c_G \rightsquigarrow r$, and by the Sanity of Values assumption. The invertibility lemma finishes this case.

- (2)/(4): In this case, the RHS of a rule F, $s \overset{F}{\hookrightarrow_l} \langle c_F \,|\, K_F\rangle\uparrow$, matches the LHS of a rule G, $\langle c_G \,|\, K_G\rangle\downarrow \overset{G}{\hookrightarrow_l} t$. Further, after unifying with the current state $\langle c'' \,|\, K''\rangle$, $\langle c_F \,|\, k_F\rangle = \langle c_G \,|\, k_G\rangle$, and $k_F = k_G$ can be written $k_F = k_G = k \circ [c_F \to \text{rhs}]$. By the induction hypothesis, there is a derivation $\langle c \,|\, K\rangle\updownarrow_c \hookrightarrow_l^* \langle c_F \,|\, K_F\rangle\uparrow$. Extend the last transition off this derivation to a maximal sequence $\langle c_F'' \,|\, K_F''\rangle\downarrow \overset{H}{\hookrightarrow_l} \langle c_F' \,|\, K_F\rangle\downarrow \hookrightarrow_l * \langle c_F \,|\, K_F\rangle\uparrow$ which does not use the reset rule; by the Sanity of Phase properties, this must exist, and Rule H must have been created by line (3). By Observation 1, we know that rule G was created by an invocation matching sosRhsToPam($\langle c_F \,|\, K_F\rangle\downarrow, k, \text{rhs}$), while rule H was created by an invocation matching sosRhsToPam($\langle c_F'' \,|\, k\rangle\downarrow, k,$ **let** $[x \rightsquigarrow c_F]$ **in** rhs), meaning there is also a rule J created by an invocation sosRhsToPam($\langle c_F \,|\, K_F\rangle\uparrow, k, \text{rhs}$) whose RHS is t. Using rule J completes this case.
- (4)/(3): In this case, the RHS of a rule F, $s \overset{F}{\hookrightarrow_l} \langle c_F \,|\, k_F\rangle\downarrow$, matches the LHS of a rule G, $\langle c_G \,|\, k_G\rangle\uparrow \overset{G}{\hookrightarrow_l} C_G[t]$. Further, after unifying with the current state $\langle c'' \,|\, k''\rangle$, $\langle c_F \,|\, k_F\rangle = \langle c_G \,|\, k_G\rangle$, and $k_F = k_G$ can be written $k_F = k_G = k \circ [x \to \text{rhs}]$. By Observation 1, we know that rule F was created by an invocation sosRhsToPam($s, k,$ **let** $[c_F \rightsquigarrow x]$ **in** rhs), and there is hence a rule H created by an invocation sosRhsToPam($\langle c_F \,|\, k_F\rangle\downarrow, k, \text{rhs}$). As rule G was created by an invocation sosRhsToPam($\langle c_G \,|\, k_G\rangle\uparrow, k, \text{rhs}$), rule H hence takes the form $\langle c_F \,|\, k_F\rangle\downarrow \overset{H}{\hookrightarrow_l} C_G[t]$. Using rule H completes this case.

□

Finally, to show the correspondence between the Unfused AM and the normal AM, we must show that fusing rules does not substantially alter the transition relation. This is very simple, thanks to the Fusion Property.

Lemma A.7. *Let M be an abstract machine whose transition relation is $\to_M$, containing a rule F. Let M' be M with Rule F fused with all possible successors, and let its transition relation be $\to_{M'}$. Then, for any state $\langle c \,|\, K\rangle$, $\langle c \,|\, K\rangle \to_{M'} \langle c' \,|\, K'\rangle$ if and only if $\langle c \,|\, K\rangle \to_M \langle c' \,|\, K'\rangle$, or $\langle c \,|\, K\rangle \overset{F}{\to_M} \langle c'' \,|\, K''\rangle \overset{G}{\to_M} \langle c' \,|\, K'\rangle$ for some rule $F \neq G$.*

Proof. By Property 1. □

Theorem 4.7 (Unfused AM-AM). *$\langle c \,|\, K\rangle \to_l^* \langle c' \,|\, K'\rangle$ if and only if $\langle c \,|\, K\rangle \longrightarrow_l^* \langle c' \,|\, K'\rangle$ by a sequence of rules whose last rule is not fused away.*

Proof. Corollary of Lemma A.7. □

Finally, we state some useful properties which are analogues of Sanity of Phase.

Property 4 (Sanity of Frame)**.** *The following properties hold:*

(1) *If $\langle c \,|\, K\rangle \to \langle c' \,|\, K'\rangle$, then either $K = K' \circ f$ for some f, $K' = K \circ f$ for some f, or there are f, f', K'' such that $K = K'' \circ f$ and $K' = K'' \circ f'$.*
(2) *If c is a nonvalue, and $\langle c \,|\, K\rangle \to \langle c' \,|\, K'\rangle$, then K is contained in K'.*

Why Didn't We Use Bisimulations? A common alternative way of stating results like Theorems 4.5 and 4.6 is by giving a stuttering bisimulation between the PAM and the AM. However, between any two (terminating) transition systems, without giving additional labels on the states, there always exists a trivial stuttering bisimulation. The interesting information lies in giving a specific stuttering bisimulation. We decided that dealing with the additional machinery of stuttering bisimulations would add to the background needed to understand the proof, without much benefit.

B PROOFS OF ABSTRACT REWRITING THEOREMS

Lemma 5.3 (Generalized Lifting Lemma). *Let β_1, β_2 be base abstractions where β_1 is pointwise less than β_2, i.e.: $\beta_1(f)(\overline{c}) \prec \beta_2(f)(\overline{c})$ for all f, c. Suppose $\langle c_1 \,|\, K_1 \rangle \prec \left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle$, and $\langle c_1 \,|\, K_1 \rangle \underset{\beta_1}{\overset{\frown}{\rightarrow}} \langle c_2 \,|\, K_2 \rangle$ by rule F. Then there is a $\left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ such that $\langle c_2 \,|\, K_2 \rangle \prec \left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ and $\left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle \underset{\beta_2}{\overset{\frown}{\rightarrow}} \left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ by rule F.*

Proof. We show a correspondence between the applications abstract rewriting algorithms for both β_1 and β_2. We know that the LHS of rule F matches $\langle c_1 \,|\, K_1 \rangle$ with witness σ; hence, by the Abstract Matching Property, it also matches $\left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle$ with some witness $\sigma' \succ \sigma$. Let the RHS of f be rhs_p. Then:

- If $\text{rhs}_p = \textbf{let } c_{\text{ret}} = \text{func}(\overline{c_{\text{args}}}) \textbf{ in } \text{rhs}'_p$, then the concrete rewriting of $\left\langle c_1 \,\middle|\, K_1 \right\rangle$ must have picked some $r \in \beta_1(\text{func})(\sigma(\overline{c_{\text{args}}}))$, where r matches c_{ret} with witness σ_r. Since β_1 is a base abstraction and $\widehat{\sigma}(\overline{c_{\text{args}}}) \succ \sigma(\overline{c_{\text{args}}})$, there is an $\widehat{r} \in \beta_2(\text{func})(\widehat{\sigma}(\overline{c_{\text{args}}}))$ with $\widehat{r} \succ r$. Then, by the abstract matching property, $\widehat{r}$ matches c_{ret} with witness $\widehat{\sigma_r} \succ \sigma_r$. The abstract rewriting algorithm then proceeds to recursively evaluate rhs_p with some $\widehat{\sigma'}$ and σ' respectively; by their definition and the previous argument, we must have $\widehat{\sigma'} \succ \sigma'$.
- If $\text{rhs}_p = \left\langle c'_p \,\middle|\, K'_p \right\rangle$, then the result is $\left\langle \widehat{\sigma}(c'_p) \,\middle|\, \widehat{\sigma}(K'_p) \right\rangle$. Since $\widehat{\sigma} \succ \sigma$, $\left\langle \widehat{\sigma}(c'_p) \,\middle|\, \widehat{\sigma}(K'_p) \right\rangle \succ \left\langle \sigma(c'_p) \,\middle|\, \sigma(K'_p) \right\rangle = \langle c_2 \,|\, K_2 \rangle$ by the Abstract Matching Property, finishing the proof.

□

Theorem 5.6 (Abstract Transition). *For $a, b \in \text{amState}$, if α is an abstraction with base abstraction β, and $a \longrightarrow b$, then either $b \sqsubseteq \alpha(a)$, or there is a $g \in \text{amState}^\star$ such that $\alpha(a) \underset{\alpha}{\overset{\frown}{\rightarrow}} g$ and $b \sqsubseteq g$.*

The proof uses the following observation:

Observation 3. *The nontermination-cutting relation ($\triangleleft$) satisfies the following properties:*

(1) *If $a \prec b$ and $b \triangleleft c$, then $a \triangleleft c$.*
(2) *If $a \longrightarrow b$ and $a \triangleleft c$, then $b \triangleleft c$.*

We now prove the Abstract Transition Theorem:

Proof. Because ($\prec$) is reflexive and transitive, there must be a *canonical derivation* of $a \sqsubseteq \alpha(a)$ of one of the following three forms:

- Case (1): $a \prec \alpha(a)$. Then, by the lifting lemma, there is a g with $b \prec g$ and $\alpha(a) \underset{\beta}{\overset{\frown}{\rightarrow}} g$.
- Case (2): There are x_1, x_2 such that $a \prec x \triangleleft x_2 \sqsubseteq \alpha(a)$. Then, by Observation 3, $a \triangleleft x_2$, and hence $b \triangleleft x_2$, and hence $b \sqsubseteq x_2 \sqsubseteq \alpha(a)$.
- Case (3): There is an x_1 such that $a \prec x_1$, and, for all x_2 such that $x_1 \underset{\beta}{\overset{\frown}{\rightarrow}} x_2$, $x_2 \sqsubseteq \alpha(a)$. First, by the lifting lemma, there is an x'_1 with $b \prec x'_1$ and $x_1 \underset{\beta}{\overset{\frown}{\rightarrow}} x'_1$. But, by assumption, $x'_1 \sqsubseteq \alpha(a)$. Hence, $b \prec x'_1 \sqsubseteq \alpha(a)$, so $b \sqsubseteq \alpha(a)$.

□

C MORE DETAILS ON SYNTAX-DIRECTED CFG-GENERATORS

C.1 Correctness of Graph Patterns

We first note that, by starting abstract execution on a node whose immediate children are all non-value variables, this algorithm assumes that the initial state of the program must contain no value nodes. This is not a real restriction; one can transform any language to meet this criterion by replacing each value-node $V(\overline{x})$ in initial program states with a non-value node $MkV(\overline{x})$ and adding the rule $MkV(\overline{x}) \rightsquigarrow V(\overline{x})$. So, pedantically speaking, the graph-patterns produced by the algorithm are not actually graph patterns of the original language, but rather of this normalized form.

We now build the setting of uor proofs. In this develpment, let term_{Var}, $\text{amState}_{\text{Var}}$, etc be variants of term$\star$, amState$\star$, etc that may contain free variables as well as $\star$ nodes. We extend the $\prec$ ordering to include the subsumption ordering, and with the relation $x_{\text{mt}} \prec \star_{\text{mt}}$ for any variable x, so that $a \prec b$ if a may be obtained from b by specializing a match type, expanding a star node, *or* substituting a variable.

As mentioned in §5.3, we add a few technical conditions to the definition of an abstraction α. The first condition prevents an antagonistically-chosen α from doing something substantially different when encountering a more abstract term.

Assumption 3. *On terms without variables, α must be monotone in the $\prec$ ordering. As the analogue for terms with variables, if there is a substitution σ such that $b = \sigma(a)$, then there must be a substitution σ' extending σ such that $\alpha(b) = \sigma'(\alpha(a))$.*

The next condition is stronger than we need, but greatly simplifies the discovery of the correspondence between graph patterns and interpreted-mode graphs. In plain words, it states that abstractions may not drop stack frames: they may skip over the execution of a subterm entirely, but may not skip over only the latter part of a computation. All abstractions discussed in this paper satisfy it.

Definition C.1. Define the *stack length* of a state $s = \langle c \mid K \rangle$ as:

- $\text{stacklen}(\langle c \mid \textbf{empty} \rangle) = 0$
- $\text{stacklen}(\langle c \mid K \circ f \rangle) = 1 + \text{stacklen}(\langle c \mid K \rangle)$

Assumption 4. *For all $a \in amState_{Var}$, $stacklen(a) \leq stacklen(\alpha(a))$. Further, there must be a derivation of $a \sqsubseteq \alpha(a)$ where none of the intervening states c, as in Definition 5.5, satisfy $stacklen(c) < stacklen(a)$.*

We now begin the proofs. We need a new version of the Generalized Lifting Lemma for narrowing. Mimicking the proof, and using the relation between matching and unification, gives the following:

Lemma C.2 (Lifting Lemma (Narrowing)). *Let $a \in amState_\star$, $b \in amState_{Var}$, $a \prec b$, and let β be a base abstraction. Suppose $a \underset{\beta}{\overset{\frown}{\rightarrow}} a'$. Then there exists b' such that $b \underset{\beta}{\overset{\frown}{\rightsquigarrow}} b'$ and $a' \prec b'$.*

We now prove the correspondence between a graph pattern and the relevant subgraph of an abstract transition graph. To isolate the relevant subgraphs, we use the concept of *hammocks* from graph theory, which are commonly used in the analysis of control-flow graphs (e.g.: Ferrante and Warren [1987]). A hammock of a control-flow graph is a single-entry single-exit subgraph. We use the modified term *weak hammock* to refer to a single-entry multiple-exit subgraph.

Definition C.3. Let $N^+(n)$ be the out-neighborhood of n in a graph G. Then the *weak hammock* of G bounded by entry node n and exit node-set $\mathcal{T}$ is the subgraph of G induced by the node set give by the least-fixed-point of Q, where

$$Q(S) = \{n\} \cup \bigcup_{m \in (S \setminus \mathcal{T})} N^+(m)$$

Our goal now is to, given the abstract transition graph of a program, discover the fragment that corresponds to the control-flow of a single node. We will then prove the correspondence between these fragments ad the relevant graph pattern.

Definition C.4. Let N be a non-value node type and α an abstraction, and consider some configuration $S = \left\langle (N(\overline{e_i}), \mu) \,\middle|\, K \right\rangle$. Let T be the abstract transition graph T of $(\underset{\alpha}{\widehat{\rightarrow}})$ starting from S. Let $E = \{t \in T | t \neq S \wedge \text{stacklen}(t) \leq \text{stacklen}(S)\}$. Then the *CFG fragment for* S is defined inductively as follows:

- If none of the e_i are non-values, then the CFG fragment for S is the weak hammock of T bounded by S and E.
- Otherwise, the CFG fragment for S is the weak hammock of T bounded by S and E, minus the edges of the CFG fragments for each non-value e_i, minus also the nodes which then become unreachable from both S and E.

We say there is a *transitive edge* from the start state of each sub-CFG-fragment to its end states. By default when discussing the edges of a CFG fragment, we do not include the transitive edges.

Lemma C.5. *Let $\widehat{e} \prec \star_{NonVal}$. Then, for any $\widehat{s}, \widehat{K}$, $\left\langle (\widehat{e}, \widehat{s}) \,\middle|\, \widehat{K} \right\rangle \triangleleft \left\langle (\star_{Val}, \top_l) \,\middle|\, \widehat{K} \right\rangle$.*

Proof. Consider $\left\langle c \,\middle|\, K \right\rangle \in \gamma(\left\langle (\widehat{e}, \widehat{s}) \,\middle|\, \widehat{K} \right\rangle)$. By the Sanity of Frame properties, for any derivation $\left\langle c \,\middle|\, K \right\rangle \rightarrow^* \left\langle c' \,\middle|\, K' \right\rangle$, either K' contains K or there is a subderivation of the form $\left\langle c \,\middle|\, K \right\rangle \rightarrow^* \left\langle c' \,\middle|\, K \right\rangle$. □

Theorem C.6 (Correctness of Graph Patterns). *Let N be a non-value node type, and P be its graph pattern under $(\underset{\alpha}{\widehat{\rightarrow}})$. For an abstraction α, consider the abstract transition graph T of $(\underset{\alpha}{\widehat{\rightarrow}})$ from some start state $S = \left\langle (N(\overline{e_i}), \mu) \,\middle|\, K \right\rangle$. Let F be the CFG fragment for S in T. Let σ be the substitution resulting from unifying the start state of P with S. Then, for every edge $a \underset{\alpha}{\widehat{\rightarrow}} b$ in F, there are $a', b' \in P$ with $a \prec \sigma(a')$, $b \prec \sigma(b')$ such that $a' \underset{\alpha}{\widehat{\rightarrow}} b'$ is in P.*

Proof. For any $a \in V(F), a' \in V(P)$ with $a \prec \sigma(a')$, this is true for all edges reachable from a by the Lifting Lemma and by Assumption 3. We hence must show that every node in F is reachable from some node x satisfying $\exists x' \in P, x \prec \sigma(x')$. But note that every node in F is reachable from either S or by the exit nodes of the CFG fragment for one of the e_i. By construction, S is equal to the start state of P'. It remains to show this condition for the exit nodes of the CFG fragments for the e_i, i.e.: the nodes which are the target of a transitive edge.

Pick such a node $x = \left\langle c_x \,\middle|\, K_x \right\rangle$, and note that, by construction and by Sanity of Frame, c_x must be a value. Then, using the Sanity of Frame properties and Assumption 4, the source of the corresponding transitive edge(s) must have the form $\left\langle (e_i, \mu) \,\middle|\, K_x \right\rangle$. By induction, we must have e_i', μ', K_x' with $e_i \prec \sigma(e_i')$ and $K_x \prec \sigma(K_x')$ with $\left\langle (e_i', \mu') \,\middle|\, K_x' \right\rangle \in P$. This node has a transitive edge to $\left\langle (\star_{\text{Val}}, \top) \,\middle|\, K_x' \right\rangle$ in P, which satisfies $\left\langle c_x \,\middle|\, K_x \right\rangle \prec \sigma(\left\langle (\star_{\text{Val}}, \top) \,\middle|\, K_x' \right\rangle)$, finishing the proof. □

One advantage of the compiled-mode is that it's done once per language/abstraction pair, so any corner case that causes an infinite-loop would be exposed up front. Yet interpreted-mode CFG-generation is always more precise, albeit slower and less predictable. Can we at least guarantee that interpreted-mode generation will terminate, i.e.: result in a finite graph?

It's easy to construct examples where interpreted-mode CFG-generation does not terminate (e.g.: the identity abstraction, and any non-terminating program), so there's no universal termination theorem. But here's a cool result: if compiled-mode generation terminates, then so does interpreted-mode. Intuitively, this is so because graph-pattern generation does the kind of case analysis that's needed to prove termination of interpreted-mode CFG-generation for a single language/abstraction pair

Theorem C.7 (Finiteness). *Consider a node type N, its graph pattern P, any state of the form $S = \left\langle (N(\overline{e_i}), \mu) \,\middle|\, K \right\rangle$, and the CFG fragment F for S in the abstract transition graph of S. If P is finite, then so is F.*

Proof. Suppose F is infinite. By König's Lemma, either F contains a node of infinite outdegree, or an infinite path. However, because all semantic functions func have the property that $\text{func}(\overline{c})$ is finite for all $\overline{c}$, the only possibility is for F to contain an infinite path.

Theorem C.6 establishes a relation M between F and P. Since P is finite, this implies that there is a path $f_1 \to f_2 \to \cdots \to f_n$ in F where each f_i is distinct, where the corresponding path in P $p_1 \to \cdots \to p_{n-1} \to p_1$ is a cycle, where each $\to$ is either a transitive edge, $\underset{\alpha}{\widehat{\to}}$ (in F), or $\underset{\alpha}{\widehat{\rightsquigarrow}}$ (in P).

However, by Theorem C.6, there is σ such that $\sigma(p_1) = f_1$, and there is σ' extending σ such that $\sigma'(p_1) = f_n$. Since $\sigma(p_1)$ must be ground, that means $\sigma(p_1) = \sigma'(p_1)$, and hence $f_1 = f_n$, contradicting the assumption. Hence, F must be finite. □

Corollary C.8. *Let $a \in amState_l$ and α be a machine abstraction. If the graph patterns under abstraction α for all nodes in a are finite, then only finitely many states are reachable from a under the $\underset{\alpha}{\widehat{\to}}$ relation.*

C.2 Code-Generation

The heuristic code generator traverses the graph pattern, identifying subterm enter and exit states, and greedily merges unrecognized intermediate states with adjacent ones if one dominates the other, essentially building a custom projection. In the end, for an AST-node with k children, there will be up to $2k + 2$ graph nodes, an enter and exit CFG-node for the outer AST node and each child. Once all abstract states have been merged into these graph nodes, the algorithm identifies the edges between the nodes, and outputs code like in Fig. 6.

This code generator is not complete for all possible graph patterns, and notably would fail when a single step in the source program corresponds to an arbitrary number of steps in the internal semantics (as in Java method resolution, which must traverse the class table). However, it is sufficient to generate code for both Tiger and BaliScript.

One subtle fact is that not every AST node will have a distinct exit node. For example, in all three languages, the graph pattern for **if** e **then** s_1 **else** s_2 will actually terminate in the exit nodes of s_1 and s_2, which both must be directly connected to whatever statement follows the if. The generated CFG-generators hence actually treat lists of enter/exit nodes as the atomic unit, rather than single nodes. Note that, if the language designer did want to have a distinct "join" node terminating the CFG for an if-statement, they could accomplish this by changing the semantics of if-statements to introduce an extra step after the body is evaluated.