Automatically Deriving Control-Flow Graph Generators from Operational Semantics

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

We develop the first theory of control-flow graphs from first principles, and use it to create an algorithm for automatically synthesizing many variants of control-flow graph generators from a language's operational semantics. Our approach first introduces a new algorithm for converting a large class of small-step operational semantics to an abstract machine. It next uses a technique called "abstract rewriting" to automatically abstract the semantics of a language, which is used both to directly generate a CFG from a program ("interpreted mode") and to generate standalone code, similar to a human-written CFG generator, for any program in a language. We show how the choice of two *abstraction* and *projection* parameters allow our approach to synthesize several families of CFG-generators useful for different kinds of tools. We prove the correspondence between the generated graphs and the original semantics. We provide and prove an algorithm for automatically proving the termination of interpreted-mode generators. In addition to our theoretical results, we have implemented this algorithm in a tool called Mandate, and show that it produces human-readable code on two medium-size languages with 60 − 80 rules, featuring nearly all intraprocedural control constructs common in modern languages. We then show these CFG-generators were sufficient to build two static analyses atop them. Our work is a promising step towards the grand vision of being able to synthesize all desired tools from the semantics of a programming language.

CCS Concepts: • **Theory of computation → Abstract machines**; **Operational semantics**.

Additional Key Words and Phrases: abstract machines, control flow, term rewriting

ACM Reference Format:
James Koppel, Jackson Kearl, and Armando Solar-Lezama. 2022. Automatically Deriving Control-Flow Graph Generators from Operational Semantics. *Proc. ACM Program. Lang.* 6, ICFP, Article 117 (August 2022), 30 pages. https://doi.org/10.1145/3547648

1 INTRODUCTION

Many programming tools use control-flow graphs, from compilers to model-checkers. They provide a simple way to order the subterms of a program, and are usually taken for granted. According to folklore, their definition is simple: "control-flow graphs are an abstraction of control-flow."

In fact, as we shall argue, CFGs are not well understood, and their definition is not so simple. Consider: Even for a single language, no two tools generate the same CFG for the same program, and we have found no prior attempt to define what it means for a given CFG to correctly abstract a program. Before diving deeper into the need for a theory of CFGs, let us illustrate the nuances of CFG-generation: Fig. 1 is a fragment of a pretty-printer. How would a CFG for it look? Here are three possible answers for different kinds of tools:

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

2475-1421/2022/8-ART117
https://doi.org/10.1145/3547648

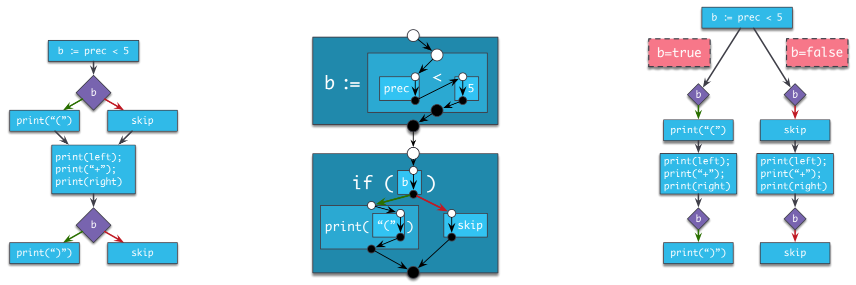


Fig. 2. Variants of control-flow graphs. Colors are for readability.

(1) Compilers want small graphs that use little memory. 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 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). Indeed, in §7, we build such an analyzer atop path-sensitive CFGs in under 50 lines of code.

All three tools require separate CFG-generators.

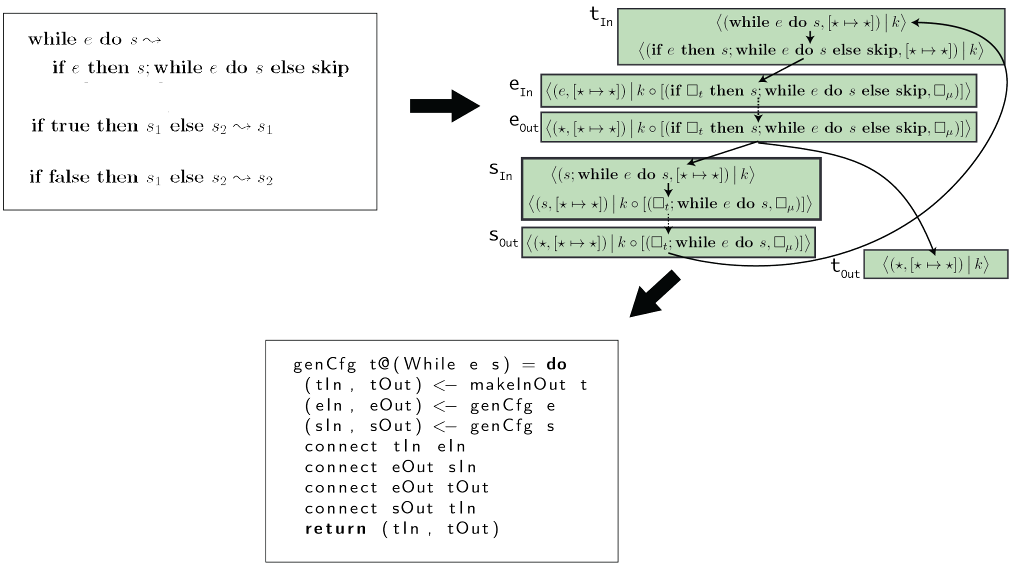
```
b := prec < 5;
if (b) then
  print("(")
else
  skip;
print(left);
print("+");
print(right);
if (b) then
  print(")")
else
  skip
```

Fig. 1. A program.

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 developing the formal correspondence between semantics and control-flow graphs, and use it to automatically derive CFG 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]). We instead develop CFGs from first principles, and, after much theory, discover that **"a CFG is a projection of the transition graph of abstracted abstract machine states."**

Abstraction and Projection and Machines. The 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



$$\text{while } e \text{ do } s \rightsquigarrow \text{if } e \text{ then } s; \text{while } e \text{ do } s \text{ else skip}$$

$$\text{if true then } s_1 \text{ else } s_2 \rightsquigarrow s_1$$

$$\text{if false then } s_1 \text{ else } s_2 \rightsquigarrow s_2$$

t_{In} $\langle(\text{while } e \text{ do } s, [\star \mapsto \star]) \mid k\rangle$
$\langle(\text{if } e \text{ then } s; \text{while } e \text{ do } s \text{ else skip}, [\star \mapsto \star]) \mid k\rangle$
e_{In} $\langle(e, [\star \mapsto \star]) \mid k \circ [(\text{if } \Box_t \text{ then } s; \text{while } e \text{ do } s \text{ else skip}, \Box_\mu)]\rangle$
e_{Out} $\langle(\star, [\star \mapsto \star]) \mid k \circ [(\text{if } \Box_t \text{ then } s; \text{while } e \text{ do } s \text{ else skip}, \Box_\mu)]\rangle$
s_{In} $\langle(s; \text{while } e \text{ do } s, [\star \mapsto \star]) \mid k\rangle$
$\langle(s, [\star \mapsto \star]) \mid k \circ [(\Box_t; \text{while } e \text{ do } s, \Box_\mu)]\rangle$
s_{Out} $\langle(\star, [\star \mapsto \star]) \mid k \circ [(\Box_t; \text{while } e \text{ do } s, \Box_\mu)]\rangle$
t_{Out} $\langle(\star, [\star \mapsto \star]) \mid k\rangle$

```
genCfg t@(While e s) = do
  (tIn, tOut) <- makeInOut 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. 3. (Top left) SOS rules for loops and conditionals (Top right) A graph-pattern generated from these rules, describing the control-flow of all while-loops (Bottom) Generated CFG-generation code

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. The primary goal of this paper is to develop the first theory of CFGs from first principles. Yet our theory immediately lends itself to automation. 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 control flow graph generators.

Mandate has two modes. In the *interpreted mode*, Mandate abstractly executes a program with its language's semantics to produce a CFG. For cases where the control-flow of a node is independent from its context (e.g.: including variants (1) and (2) but not (3)), Mandate's *compiled mode* can output a CFG-generator as a short program, similar to what a human would write (Fig. 3).

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 compilers courses, and made famous by a textbook of Appel [1998]. The second is MITScript [Carbin and Solar-Lezama 2018], a JavaScript-like language with objects and higher-order functions used in an undergraduate JIT-compilers course. While these are pedagogical languages without the edge-cases of C or SML, they nonetheless contain all common control-flow features except exceptions. And, since previous work on conversion of semantics features small lambda calculi, ours are the largest examples of automatically converting a semantics into a different form, by a large margin.

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

Fig. 5. Syntax of IMP

In summary, our work makes the following contributions:

(1) A formal and proven correspondence between the operational semantics and many common variations of CFGs, giving the first from-first-principles theoretical explanation of CFGs.
(2) An elegant new algorithm for converting small-step structural operational semantics into abstract machines.
(3) An algorithm which derives many types of control-flow graph generators 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) An "automated termination proof," showing that, if the compiled-mode CFG-generator terminates (run once per language/abstraction pair), then so does the corresponding interpreted-mode CFG generator (run once per program).
(5) The first CFG-generator generator, Mandate, able to automatically derive many types of CFG generators for a language from an operational semantics for that language, and successfully used to generate CFG-generators for two rich languages. The generated CFG-generators were then used to build two static-analyzers.

Further, our approach using *abstract rewriting* offers great promise in deriving other artifacts from language semantics.

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. Fig. 5 gives the syntax. 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 §3.3.

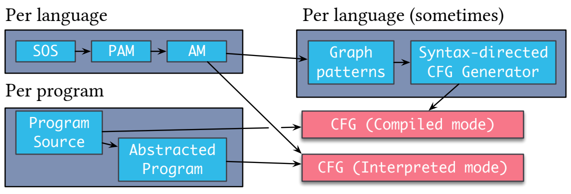


Fig. 4. Dataflow of our approach

The approach proceeds in three phases, corresponding roughly to the large boxes in Fig. 4. In the first phase (top-left box / §2.1), we transform the semantics of IMP into a form that reveals the control flow. In the second phase (bottom-left box / §2.2), we show how to interpret these semantics with abstract rewriting [Bert et al. 1993] to obtain CFGs. In the finale (top-right 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 new one (t', μ') following one step of execution. As our first **running example**, we use the rules for assignments. We will later introduce our two other running examples: addition and while-loops. Together, these cover all features of semantic rules: environments, external semantic functions, branching, and back-edges.

$$\frac{(e, \mu) \rightsquigarrow (e', \mu')}{(x := e, \mu) \rightsquigarrow (x := e', \mu')}\ \textit{AssnCong}$$

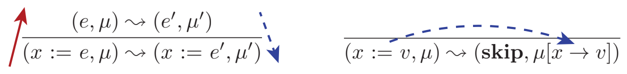
$$\frac{}{(x := v, \mu) \rightsquigarrow (\mathbf{skip}, \mu[x \to v])}\ \textit{AssnEval}$$

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* (AM). This machine acts on states $\langle (t, \mu) \,|\, K \rangle$. K is the *context* or *continuation*, and represents what the rest of the program will 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 (§3.4), in most cases, a frame can be written as e.g.: $[(x := \Box_t, \Box_\mu)]$, 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 AMs [Felleisen et al. 2009].

$$\begin{array}{ll}\langle (x := e, \mu) \,|\, k \rangle & \to \langle (e, \mu) \,|\, k \circ [(x := \Box_t, \Box_\mu)] \rangle \\ \langle (v, \mu) \,|\, k \circ [(x := \Box_t, \Box_\mu)] \rangle & \to \langle (\mathbf{skip}, \mu[x \to v]) \,|\, k \rangle\end{array}$$

The first AM rule, on seeing an assignment $x := e$, will focus on e. Other rules, not shown, 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 SOS. These two SOS rules become two AM rules. A naive interpretation is that the first SOS rule corresponds to the first AM 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 RHSs of the firsts do not match, nor do the LHSs of the seconds. The actual story is more complicated. This diagram gives the real correspondence:


$\frac{(e, \mu) \rightsquigarrow (e', \mu')}{(x := e, \mu) \rightsquigarrow (x := e', \mu')}$
$\frac{}{(x := v, \mu) \rightsquigarrow (\mathbf{skip}, \mu[x \to v])}$

The first AM rule is simple enough: it corresponds to the solid arrow, the first half of the first SOS rule. But the second AM rule actually corresponds to the two dashed arrows, AssnEval and the second half of AssnCong. We shall find that jumping straight from SOS to AM skips an 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 two AM rules are enough to describe all computations. And the second AM rule actually does two steps in one: it plugs (v, μ) into $[(x := \Box_t, \Box_\mu)]$ to obtain $(x := v, \mu)$, and then evaluates this result. So, by fusing these rules, the standard presentation of AM obscures the multiple underlying steps, hiding 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, creating the final AM, and completing the first stage of our CFG-creation pipeline.

§3.4 explains PAM in full, while §3.6 and §3.7 give the algorithm for creating abstract machines. §4 shows correctness.

```
valueIrrelevance t =
  mapTerm valToStar t
 where
  valToStar (Val _ _) = ValStar
  valToStar t         = t
```

Fig. 6. Value irrelevance abstraction

2.2 Run Abstract Program, Get CFG

The AM 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; there are still a few important differences. The AM allows for infinitely-many possible states, while control-flow graphs have finite numbers of states, potentially with loops. The AM 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 AM 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:

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

Now consider the statement $x := y$. It can be executed with an infinite number of environments: the starting configuration $(x := y, [y \mapsto 1])$ results in $(\mathbf{skip}, [y \mapsto 1, x \mapsto 1])$; $(x := y, [y \mapsto 2])$ 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, given by the code in Fig. 6, replaces all values with a single *abstract value* representing any of them: $\star$. Under this 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 AM rules on this abstract state, a process called *abstract rewriting*. Now, running in fixed context K, there is only one execution of this statement.

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

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 CFG 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:

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

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, showing 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 $\textbf{while}\ e\ \textbf{do}\ s \rightsquigarrow \textbf{if}\ e\ \textbf{then}\ (s; \textbf{while}\ e\ \textbf{do}\ s)\ \textbf{else skip}$. Consider an AM state evaluating an arbitrary while-loop $\textbf{while}\ e\ \textbf{do}\ s$ in an arbitrary context k, with an arbitrary abstract environment μ. Such a state can be written $\left\langle (\textbf{while}\ e\ \textbf{do}\ s, \mu) \middle| k \right\rangle$. 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 $\left\langle (\textbf{while}\ e\ \textbf{do}\ s, [\star \mapsto \star]) \middle| k \right\rangle$. 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 $\left\langle (e, [\star \mapsto \star]) \middle| k \right\rangle$. 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 $\left\langle (\star, [\star \mapsto \star]) \middle| k \right\rangle$. With this modification, the search procedure finds only 8 unique states for while-loops. It hence terminates in the graph pattern of Fig. 3 (with dotted lines for transitive edges), 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.

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 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. 3 shows how the code-generator groups and labels the states of this graph pattern as well as the generated code[1].

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

[1]This is verbatim MANDATE output except that, in actual output, (1) the `connect` statements are in no particular order, and (2) the actual return value is `([tIn], [tOut])`, as, in general, AST nodes such as conditionals may have multiple final CFG nodes.

the algorithm again with a different abstraction. For while-loops, the resulting pattern and code are similar to those of Fig. 3, 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 the extended version [Koppel et al. 2020] proves the algorithm's correctness.

3 FROM OPERATIONAL SEMANTICS TO ABSTRACT MACHINES

The first step in our algorithm is to convert the structural operational semantics for a language into an abstract machine which has a clear notion of control-flow. Surprisingly, no prior algorithm for this exists (see discussion in §9 and [Koppel et al. 2020]). This section hence presents **the first algorithm to convert SOS to AM**, which works on any language satisfying the conditions in §3.1. Our algorithm is unique in 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 MITScript.

There is a lot of notation required first. §3.2 gives a notation for all programming languages, while §3.3 gives an alternate notation for structural operational semantics, one more amenable to inductive transformation. §3.4 and §3.5 describe the phased and orthodox abstract machines, while §3.6 and §3.7 give the conversion algorithm. Correctness results are provided in §4 and [Koppel et al. 2020].

3.1 Setting and Assumptions

We take as our starting point a transition relation on term/state pairs $(t, \mu) \rightsquigarrow (t', \mu')$, defined using rules in the variant of structural operational semantics (SOS) to be given in §3.3. From this, our algorithms shall construct the Phased Abstract Machine, the Abstract Machine, and finally a CFG-generator. The middle step of PAM-to-AM conversion does the work of discovering which rules may follow which other rules, and imposes some additional requirements on the language semantics.

Assumption 1 (Sanity of Values). *All terms can be classified into either values or nonvalues, based only on the root node. For a rule, $(t, \mu) \rightsquigarrow rhs$, the pattern t must not match a value. If t is a nonvalue, there are μ, t', μ' such that $(t, \mu) \rightsquigarrow (t', \mu')$.*

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

Assumption 3 (No Up-Down Rules and All Up-Rules Invertible). *Discussed in §3.7.*

The Sanity of Values assumption is useful in correctness proofs, but imposes no real burden; a language definition can generally be tweaked to satisfy it.

The Determinism assumption may seem strong, but it is in fact necessary because **nondeterministic languages lack a clear notion of control-flow**. 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. The control flow of such programs cannot be given as a transition system (i.e.: a graph), which assumes a single program point. Instead, it is properly given as a Petri Net, called the "sea of nodes" representation [Click 1995]. We leave generating a sea-of-nodes-generator to future work.

The last assumption is most easily defined on the PAM, and discussed in the corresponding section. Its first part, "No Up-Down Rules," translates easily to SOS: it combines the superficial constraint that any rule which recursively evaluates a subterm must do any additional processing before the recursion, and the stronger constraint that there can be no single step of a term which simultaneously steps two independent subterms. The "All Up-Rules Invertible" assumption is less easy to translate, but it effectively means that successive steps in the SOS will focus on the same subterm, which is key to identifying the present "program point."

3.2 Terms and Languages

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

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

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 *nonvalue* which can. We formalize this distinction by marking each node either value or nonvalue, and giving each variable a *match type* controlling whether it 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 is tagged either *val* or *nonval*. For example, $1 + 1$ is shorthand for the term nonval(+, val(IntLit, 1), val(IntLit, 1)). The IMP statement $x := 1$ may ambiguously refer to either the concrete term nonval(:=, "x", val(IntLit, 1)) or to the pattern nonval(:=, x_{All}, val(IntLit, 1)), but should be clear from context. Others' presentations 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 $x[\text{"}y\text{"} \to v]$ matches the environment $\emptyset[\text{"}y\text{"} \to 1][\text{"}z\text{"} \to 2]$. The latter environment is abbreviated $[z \mapsto 2, y \mapsto 1]$. The environment-extension operator itself is not commutative. Specifically, it is right biased, e.g.: $\emptyset[\text{"}z\text{"} \to 1][\text{"}z\text{"} \to 2] = \emptyset[\text{"}z\text{"} \to 2]$.

Finally, the basic unit of reduction is a *configuration*, defined $\text{Conf}_l = \text{term} \times \text{State}_l$. We say that configuration $c = (t, \mu)$ is a value if t is a value.

3.3 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].

$$
\begin{array}{rcl}
\text{rule} & ::= & c \rightsquigarrow \text{rhs} \\
\text{rhs} & ::= & c \\
& | & \textbf{let } [c \rightsquigarrow c] \textbf{ in } \text{rhs} \\
& | & \textbf{let } c = \text{semfun}(\overline{c}) \textbf{ in } \text{rhs}
\end{array}
$$

Fig. 8. Notation for SOS

The most immediate benefit of the Straightened Operational Semantics notation is that it orders the premises of a rule. One can also gain the ordering property by imposing a restriction on rules without a change in notation: Ibraheem and Schmidt's "L-attributed semantics" is exactly this, but for big-step semantics [Ibraheem and Schmidt 1997]. But there is an additional advantage of this new notation: it has an inductive structure, which allows defining recursive algorithms over rules.

Fig. 8 defines a grammar for SOS 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. $R(A)$ denotes 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, e.g.: arithmetic operations. 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** semfun_l. Each is a relation[2] R of type $\overline{\text{Conf}_l} \times \text{Conf}_l$, subject to the restriction that (1) if $\overline{c}\ R\ d$, then $\overline{c}$ and d are values, and (2) for each $\overline{c} \in \text{Conf}_l$, there are only finitely many d such that $\overline{c}\ R\ d$.

For instance, there are two semantic functions for the IMP language: $\text{semfun}_{\text{IMP}} = \{+, <\}$. Both are partial functions, 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 straightened-operational-semantics 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. They may include external effects such as I/O. As an example using semantic functions, the rule

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

[2]We define semantic "functions" to actually be relations (i.e.: partial nondetermistic functions) so that this definition can be reused for abstract interpretation in §5.2.

$$\begin{array}{rcll}
\text{frame} & ::= & [c \rightarrow \text{rhs}] & \\
K & ::= & \mathbf{emp} & \text{(Contexts)} \\
 & \mid & K \circ \text{frame} & \\
 & \mid & k & \text{(Context Vars)} \\
\updownarrow & ::= & \uparrow \mid \downarrow & \text{(Phase)} \\
\text{pamState} & ::= & \langle c \mid K \rangle \updownarrow & \\
\text{pamRhs} & ::= & \text{pamState} & \\
 & \mid & \mathbf{let}\ c = \text{semfun}(\overline{c})\ \mathbf{in}\ \text{pamRhs} & \\
\text{pamRule} & ::= & \text{pamState} \hookrightarrow \text{pamRhs} &
\end{array}$$

(a)

$$\text{C} \quad ::= \quad \square \quad \mid \quad \mathbf{let}\ c = \text{semfun}(\overline{c})\ \mathbf{in}\ C$$

(b)

Fig. 9. (a) The Phased Abstract Machine (b) RHS contexts for Abstract Machines

would be rendered as

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

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.

Overall, this notation gives a simple inductive structure to SOS rules. We will see in §3.6 how it makes the SOS-to-PAM conversion straightforward. And it loses very little generality from the conventional SOS notation: it only assumes that the premises can be ordered. (And converting this notation back into the conventional form is easy: turn all RHS fragments into premises.)

3.4 The Phased Abstract Machine

In an SOS, 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.

A PAM state takes the form $\langle c \mid K \rangle \updownarrow$. The configuration c is the same as for operational semantics. K is a *context* or *continuation*, which represents the remainder of an SOS right-hand side. The phase is the novel part. Each PAM state is either in the evaluating ("down") phase $\downarrow$, or the returning ("up") phase $\uparrow$; an arbitrary phase is given the variable name "$\updownarrow$". A down state $\langle c \mid K \rangle \downarrow$ can be interpreted as an intention for the PAM to evaluate c in a manner corresponding to one step of the operational semantics, yielding $\langle c' \mid K \rangle \uparrow$. In fact, in the extended version, we prove that a single step $c_1 \rightsquigarrow c_2$ in the operational semantics perfectly corresponds to a sequence of steps $\langle c_1 \mid K \rangle \downarrow \hookrightarrow^* \langle c_2 \mid K \rangle \uparrow$ in the PAM.

The full syntax of PAM rules is in Fig. 9a. 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 **emp**.

Note that a pamRhs consists of a sequence of RHS fragments which terminate in a pamState $\langle c \mid K \rangle \updownarrow$. Fig. 9b captures this into a notation for RHS contexts, so that an arbitrary PAM rule may be written $\langle c_1 \mid K_1 \rangle \updownarrow_1 \hookrightarrow C[\langle c_2 \mid K_2 \rangle \updownarrow_2]$.

Finally, as alluded to in §2.1, a frame like $[(t', \mu') \rightarrow (x := t', \mu')]$ can be abbreviated to $[(x := \square_t, \square_\mu)]$, since t' and μ' are variables (i.e.: no destructuring).

A PAM rule $\langle c_p \mid K_p \rangle \updownarrow_p \hookrightarrow_l \text{rhs}_p$ for language l is executed on a PAM state $\langle c_1 \mid K_1 \rangle \updownarrow_1$ 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 $\text{rhs}_p = \mathbf{let}\ c_{\text{ret}} = \text{func}(\overline{c_{\text{args}}})\ \mathbf{in}\ \text{rhs}'_p$, with func $\in \text{semfun}_l$, pick $r \in \text{func}(\sigma(\overline{c_{\text{args}}}))$, and extend σ to σ' s.t. $\sigma'(c_{\text{ret}}) = r$ 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'.

$$\begin{array}{rcl} \text{amState} & ::= & \langle c \,|\, K \rangle \\ \text{amRhs} & ::= & \text{amState} \mid \textbf{let } c = \text{semfun}(\overline{c}) \textbf{ in } \text{amRhs} \\ \text{amRule} & ::= & \text{amState} \rightarrow \text{amRhs} \end{array}$$

Fig. 10. Abstract Machines

- If $\text{rhs}_p = \left\langle c'_p \,\middle|\, K'_p \right\rangle \updownarrow'_p$, return the new PAM state $\left\langle \sigma(c'_p) \,\middle|\, \sigma(K'_p) \right\rangle \updownarrow'_p$.

Let us give some example PAM rules. (An example execution will be in §3.7.) The AssnCong and AssnEval rules from §2.1 get transformed into the following three rules:

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

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 **down-up** rules. 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, for instance, §3.3 becomes the following PAM rules:

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

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

3.5 Abstract Machines

Finally, the AM is similar to the PAM, except that an AM state does not contain a phase. Fig. 10 gives a grammar for AM rules. We gave example rules for the AM for IMP in §2.1.

Summary of Notation. This section has introduced three versions of semantics, each defining their own transition relation. Mnemonically, as the step-relation gets closer to the abstract machine, the arrow "flattens out." They are:

(1) $c_1 \rightsquigarrow c_2$ ("squiggly arrow") denotes one SOS step (§3.3).
(2) $c_1 \hookrightarrow c_2$ ("hook arrow") denotes one PAM step (§3.4).
(3) $c_1 \rightarrow c_2$ ("straight arrow") denote one AM step (§3.5).

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

3.6 Splitting the SOS

In this section, we present our algorithm for converting an operational semantics to PAM. Fig. 11 defines the sosToPam function, which computes this transformation.

The algorithm generates PAM rules for each SOS rule. For each SOS rule $c \rightsquigarrow \text{rhs}$, 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

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

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

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

$$\begin{aligned}\text{sosRuleToPam}(c \rightsquigarrow \text{rhs}) \quad = \quad & \text{sosRhsToPam}(\langle c \,|\, k\rangle \downarrow,\, k,\, \text{rhs}) \qquad (1) \\ & \textbf{where } k \text{ is a fresh variable}\end{aligned}$$

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

$$\begin{aligned}\text{sosRhsToPam}(s,\, k,\, c) \quad &= \quad \{s \hookrightarrow \langle c \,|\, k\rangle\!\uparrow\} && (2) \\ \text{sosRhsToPam}(s,\, k,\, \textbf{let } [c_1 \rightsquigarrow c_2] \textbf{ in } \text{rhs}) \quad &= \quad \{s \hookrightarrow \langle c_1 \,|\, k'\rangle\!\downarrow\} && (3) \\ & \qquad \cup \text{ sosRhsToPam}(\langle c_2 \,|\, k'\rangle\!\uparrow,\, k,\, \text{rhs}) \\ & \quad \textbf{where } k' = k \circ [c_2 \to \text{rhs}] \\ \text{sosRhsToPam}(s,\, k,\, \textbf{let } c_2 = \text{func}(\overline{c_1}) \textbf{ in } \text{rhs}) \quad &= \quad \{s \hookrightarrow \textbf{let } c_2 = \text{func}(\overline{c_1}) \textbf{ in } \langle c_2 \,|\, k'\rangle\!\downarrow\} && (4) \\ & \qquad \cup \text{ sosRhsToPam}(\langle c_2 \,|\, k'\rangle\!\downarrow,\, k,\, \text{rhs}) \\ & \quad \textbf{where } k' = k \circ [c_2 \to \text{rhs}]\end{aligned}$$

Fig. 11. The SOS-to-PAM algorithm. Labels are used in the proofs of the extended version [Koppel et al. 2020]

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 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**: $\langle (t,\mu) \,|\, \textbf{emp}\rangle\!\uparrow \;\hookrightarrow\; \langle (t,\mu) \,|\, \textbf{emp}\rangle\!\downarrow$

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 translation from PAM to abstract machine hence removes this rule, as, upon dropping the phases, this rule would become a self-loop. It is also removed in the proof of Theorem 4.1.

3.7 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. 12. 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 machine could jump directly from line 6 to 10.

This insight is similar to the one behind Danvy's *refocusing*, which converts 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 $\langle c_1 \mid K_1 \rangle\uparrow \hookrightarrow C[\,\langle c_2 \mid K_2 \rangle\uparrow\,]$ is **invertible** if, for any c_1 nonvalue, $\langle c_2 \mid K_2 \rangle\downarrow \hookrightarrow^* \langle c_1 \mid K_1 \rangle\downarrow$.

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 $\langle c \mid K \rangle\uparrow \hookrightarrow^* \langle c \mid K \rangle\downarrow$ whenever c is a non-value, meaning that these transitions may be skipped (proof in [Koppel et al. 2020]). 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.

$$
\begin{aligned}
& \langle ((1 + (1 + 1)) + 1,\ \emptyset) \mid \mathbf{emp} \rangle\downarrow & (1)\\
\hookrightarrow\ & \langle (1 + (1 + 1),\ \emptyset) \mid \mathbf{emp} \circ [(\square_t + 1, \square_\mu)] \rangle\downarrow & (2)\\
\hookrightarrow\ & \langle (1 + 1,\ \emptyset) \mid \mathbf{emp} \circ [(\square_t + 1, \square_\mu) \circ [(1 + \square_t, \square_\mu)]] \rangle\downarrow & (3)\\
\hookrightarrow\ & \langle (2,\ \emptyset) \mid \mathbf{emp} \circ [(\square_t + 1, \square_\mu) \circ [(1 + \square_t, \square_\mu)]] \rangle\downarrow & (4)\\
\hookrightarrow\ & \langle (2,\ \emptyset) \mid \mathbf{emp} \circ [(\square_t + 1, \square_\mu) \circ [(1 + \square_t, \square_\mu)]] \rangle\uparrow & (5)\\
\hookrightarrow\ & \langle (1 + 2,\ \emptyset) \mid \mathbf{emp} \circ [(\square_t + 1, \square_\mu)] \rangle\uparrow & (6)\\
\hookrightarrow\ & \langle ((1 + 2) + 1,\ \emptyset) \mid \mathbf{emp} \rangle\uparrow & (7)\\
\hookrightarrow\ & \langle ((1 + 2) + 1,\ \emptyset) \mid \mathbf{emp} \rangle\downarrow & (8)\\
\hookrightarrow\ & \langle (1 + 2,\ \emptyset) \mid \mathbf{emp} \circ [(\square_t + 1, \square_\mu)] \rangle\downarrow & (9)\\
\hookrightarrow\ & \langle (3,\ \emptyset) \mid \mathbf{emp} \circ [(\square_t + 1, \square_\mu)] \rangle\downarrow & (10)\\
\hookrightarrow\ & \langle (3,\ \emptyset) \mid \mathbf{emp} \circ [(\square_t + 1, \square_\mu)] \rangle\uparrow & (11)\\
\hookrightarrow\ & \langle (3 + 1,\ \emptyset) \mid \mathbf{emp} \rangle\uparrow & (12)\\
\hookrightarrow\ & \langle (3 + 1,\ \emptyset) \mid \mathbf{emp} \rangle\downarrow & (13)\\
\hookrightarrow\ & \langle (4,\ \emptyset) \mid \mathbf{emp} \rangle\downarrow & (14)\\
\hookrightarrow\ & \langle (4,\ \emptyset) \mid \mathbf{emp} \rangle\uparrow & (15)
\end{aligned}
$$

Fig. 12. Example PAM derivation.

The last requirement for an abstract machine to be valid is not having any up-down rules, rules of the form $\langle c \mid k \rangle\uparrow \hookrightarrow C[\,\langle c' \mid k' \rangle\downarrow\,]$. An up-down rule follows from any SOS rule which simultaneously steps multiple subterms, and are not found in typical semantics. An example SOS rule which does result in an up-down rule is this lockstep-composition rule:

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

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 LockstepComp rule. The LockstepComp rule splits into 3 PAM rules, of which the third is an up-rule, $\langle e_2' \mid k \circ [e_1' \parallel \square] \rangle\uparrow \hookrightarrow \langle e_1' \parallel e_2' \mid k \rangle\uparrow$. 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 $\langle (t, \mu) \mid K \rangle\uparrow$, 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 $\langle c \,|\, K \rangle\!\downarrow \;\hookrightarrow\; \langle c \,|\, K \rangle\!\uparrow$, 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 $\langle c_1 \,|\, K_1 \rangle \to C_1[\langle c_1' \,|\, K_1' \rangle]$ is fused with a rule $\langle c_2 \,|\, K_2 \rangle \to C_2[\langle c_2' \,|\, K_2' \rangle]$ by unifying (c_1', K_1') with (c_2, K_2), and replacing them with the new rule $\langle c_1 \,|\, K_1 \rangle \to C_1[C_2[\langle c_2' \,|\, K_2' \rangle]]$.

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

There are two cases where rules should be fused. First, considering Fig. 11, rules which invoke a semantic function always have only one possible successor rule, 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:

$$\langle (x := e, \mu) \,|\, k \rangle \to \langle (e, \mu) \,|\, k \circ [(x := \Box_t, \Box_\mu)] \rangle$$
$$\langle (v, \mu) \,|\, k \circ [(x := \Box_t, \Box_\mu)] \rangle \to \langle (\mathbf{skip}, \mu[x \to v]) \,|\, k \rangle$$

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 **the extended version**.

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, $\langle c_1 \,|\, K \rangle\!\downarrow \;\hookrightarrow_l^*\; \langle c_2 \,|\, K \rangle\!\uparrow$

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 we first must explain 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 $\langle c \,|\, K \rangle\!\uparrow \;\hookrightarrow^*\; \langle c' \,|\, \mathbf{emp} \rangle\!\uparrow$ 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 $\langle c_1 \,|\, \mathbf{emp} \rangle\!\downarrow \;\hookrightarrow^*\; \langle c_2 \,|\, \mathbf{emp} \rangle\!\uparrow$ 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

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

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

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

If **false** $\neq$ ContainsField(r, b), then this will be a stuck state.

Working Steps. As seen in the example in §3.7, 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, for some c, K, $\langle c \,|\, K \rangle\!\downarrow \hookrightarrow_l^* \langle c' \,|\, K' \rangle\!\downarrow$, $\langle c' \,|\, K' \rangle\!\downarrow$ 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* shrinks this to a finite one. Some abstractions yield a graph resembling a traditional CFG (§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, 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 to 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 term$\star$, satisfying term $\subset$ term$\star$, where some nodes have been replaced with $\star$ nodes which represent any term.

$$\begin{aligned}\text{term}\star \quad ::= \quad & \text{nonval}(\text{sym}, \overline{\text{term}\star}) \mid \text{val}(\text{sym}, \overline{\text{term}\star}) \\ & \mid \text{const} \mid x \mid \star_{\text{mt}}\end{aligned}$$

Here, mt is a match type (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 term$\star$ as the reflexive, transitive, congruent closure of an ordering $\prec'$, defined by the following relations for all $t, s, \bar{t}$:

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

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 then define the set of concrete terms represented by $\widehat{t} \in$ term$\star$ as:

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

The power of this definition of 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 $t_p \in$ term matches an abstract term $\widehat{t} \in$ term$\star$ if there 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 $\star_{\mathsf{All}}$ matches the pattern $v_1 + v_2$ with a witness $\widehat{\sigma}$ with $\widehat{\sigma}(v_1) = \widehat{\sigma}(v_2) = \star_{\mathsf{Val}}$. We are now ready to state the main 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 $\widehat{\text{State}_l}$ (discussed in §5.2). After doing so, the definitions of abstract terms and states can be lifted to abstract configurations Conf$\star_l$, lists of abstract configurations $\overline{\text{Conf}\star}$, contexts Context$\star$, abstract machine states amState$\star$, and abstract semantic functions semfun$\star_l$, etc by transitively replacing all instances of term and State$_l$ in their 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 §3.4, 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 $\widehat{\text{State}_l} \supseteq$ State$_l$ with ordering $\prec$. There must also be a notion of substitution satisfying $\sigma_1(\mu) \prec \sigma_2(\mu)$ for $\mu \in \widehat{\text{State}_l}$ 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 Conf$\star_l$, and lists of abstract configurations $\overline{\text{Conf}\star_l}$ via congruence. We extend $\prec$

over sets of configurations $\mathbb{P}(\mathsf{Conf}\star_l)$ via the multiset ordering [Dershowitz 1987] [3] , and extend $\prec$ over semantic functions pointwise, i.e.: for $f_1, f_2 \in \mathsf{semfun}_l$, $f_1 \prec f_2$ iff $f_1(x) \prec f_2(x)$ for all $x \in \overline{\mathsf{Conf}\star_l}$.

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 $\widehat{\mathsf{semfun}_l}$. Abstract rewriting will be hence parameterized over a "base abstraction" $\beta : \mathsf{semfun}_l \to \widehat{\mathsf{semfun}_l}$, satisfying the following property:

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

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

(1) Compute the abstract match of $\langle c_p \,|\, K_p \rangle$ and $\langle c_1 \,|\, K_1 \rangle$, giving witness $\widehat{\sigma}$; fail if they do not abstractly match.
(2) Recursively evaluate rhs_p as follows:
   - If $\mathrm{rhs}_p = \textbf{let } c_{\mathrm{ret}} = \mathrm{func}(\overline{c_{\mathrm{args}}}) \textbf{ in } \mathrm{rhs}'_p$, with $\mathrm{func} \in \mathsf{semfun}_l$, then pick $r \in \beta(\mathrm{func})(\widehat{\sigma}(\overline{c_{\mathrm{args}}}))$ and compute $\widehat{\sigma_r}$ as the witness of abstractly matching c_{ret} against r. Define $\widehat{\sigma'}$ by $\widehat{\sigma'}(x) = \widehat{\sigma}(x)$ for $x \in \mathrm{dom}(\widehat{\sigma})$, and $\widehat{\sigma'}(y) = \widehat{\sigma_{\mathrm{r}}}(y)$ for $y \in \mathrm{dom}(\widehat{\sigma_{\mathrm{r}}})$. Fail if no such $\widehat{\sigma'}$ exists. Then recursively evaluate rhs'_p using $\widehat{\sigma'}$ as the new witness.
   - If $\mathrm{rhs}_p = \left\langle c'_p \,\middle|\, K'_p \right\rangle$, 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 $\left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle \underset{\beta}{\widehat{\to}} \left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$. Here is the fundamental property relating abstract and concrete rewriting:

Lemma 5.2 (Lifting Lemma). *If $\langle c_1 \,|\, K_1 \rangle \prec \left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle$, and $\langle c_1 \,|\, K_1 \rangle \to \langle c_2 \,|\, K_2 \rangle$ by rule F, then, for any β, 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 $\left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle \underset{\beta}{\widehat{\to}} \left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ by F.*

Note that, if β is the identity function, then $\underset{\beta}{\widehat{\to}}$ is the same as $\to$. Hence, the Lifting Lemma 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 also $\langle c_1 \,|\, K_1 \rangle \underset{\beta_1}{\widehat{\to}} \langle c_2 \,|\, K_2 \rangle$ 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 $\left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle \underset{\beta_2}{\widehat{\to}} \left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ by rule F.*

Proof. See extended version. □

5.3 Machine Abstractions

This section finally ties the knot on the adage "CFGs are an abstraction of control-flow" by defining the relation between a program's actual control flow (the concrete state transition graph) and its finite CFG. Intuitively, this relation involves skipping intermediate steps and combining states that correspond to the same program point. Yet getting the details right is tricky.

[3] 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 multisets $A, B \in \mathbb{N}^D$ 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 $a_i \prec_D b$.

The simple function call `f()` exemplifies the difficulties of defining this relationship. Running `f()` may evaluate arbitrary code. (In the language of abstract rewriting, without context on `f`, it evaluates to an arbitrary program $\star_{\mathsf{NonVal}}$.) An intraprocedural CFG generator must generate a small fixed number of nodes (typically, 1 evaluation node or part of a basic-block node) to represent the call to `f()`; any additional nodes depending on the definition of `f` makes it no longer intraprocedural. It must then choose to either (1) draw edges through these nodes continuing onwards, or (2) when `f()` provably does not terminate, it may optionally draw edges into but not out of these nodes. Under a naive "combine states and skip steps" definition of valid abstraction, an intraprocedural CFG-generator must first perform an interprocedural termination analysis to be sound. And if `f()` only conditionally diverges, then both options are incorrect.

We shall present a relation which does indeed justify abstracting `f()` to $\star_{\mathsf{Val}}$, using a definition with subtle treatment of nontermination and branching. The upshot of this work is the ability to write control-flow abstractions with formal guarantees whose actual implementation is trivial; abstracting function calls as described here is but a small modification to the 5 lines of Fig. 6. And, in spite of the definition's complexity, actually showing a candidate abstraction meets the definition is usually trivial.

Over the next paragraphs, we develop the abstraction preorder $a \sqsubseteq b$ between abstract states. From this, we obtain our first formal definition of a CFG: when G is the concrete transition graph for some program P, and H is a finite abstract transition graph, every state in G is $\sqsubseteq$ some state in H, and every state in H is $\sqsupseteq$ some state in G, then H is a valid CFG for P.

The $(\sqsubseteq)$ relation is built from three smaller relations. Clearly, $(\sqsubseteq)$ must contain the $(\prec)$ ordering, so that a single CFG node may describe concrete states that differ only in values. It should be able to ignore some steps of computation (e.g.: desugaring a while-loop), and so should involve $(\underset{\beta}{\widehat{\rightarrow}})$. It also must be able to skip over loops regardless of termination. We define the $(\lessdot)$ relation to skip over infinite loops. Intuitively, $a = \left\langle (\widehat{t}, \widehat{\mu}) \,\middle|\, \widehat{K} \right\rangle \lessdot \left\langle (\star_{\mathsf{Val}}, \top) \,\middle|\, \widehat{K'} \right\rangle$ if $\top$ is the "any" state (i.e.: overapproximates all possible effects) and all executions of a get "trapped" in some part of the program, with $\widehat{K'}$ never getting popped off the stack.

Definition 5.4 (Nontermination-cutting ordering). Let $\top_l$ be a maximal element of $\widehat{\mathsf{State}_l}$. Consider a state $a = \left\langle (\widehat{t}, \widehat{s}) \,\middle|\, \widehat{K} \right\rangle \in \mathsf{amState}\star$, and let $\widehat{K'}$ be a subcontext of $\widehat{K}$. Suppose that, for all $\left\langle (t, s) \,\middle|\, K \right\rangle \prec a$, and for all derivations of the form $\left\langle (t, s) \,\middle|\, K \right\rangle \rightarrow^* \left\langle (t', s') \,\middle|\, K' \right\rangle$, either K is a subterm of K', or there is a subderivation of the form $\left\langle (t, s) \,\middle|\, K \right\rangle \rightarrow^* \left\langle (t'', s'') \,\middle|\, K \right\rangle$ such that $t'' \prec \star_{\mathsf{Val}}$. Then $a \lessdot \left\langle (\star_{\mathsf{Val}}, \top_l) \,\middle|\, \widehat{K'} \right\rangle$. If a does not satisfy this condition, then $\nexists a'. a \lessdot a'$.

We now combine the $(\prec)$, $(\lessdot)$, and $(\underset{\beta}{\widehat{\rightarrow}})$ orderings to fully describe what an abstraction may do. The naive approach would be to take the transitive closure of $(\prec) \cup (\lessdot) \cup (\underset{\beta}{\widehat{\rightarrow}})$. But this would permit abstracting $\langle \textbf{if } \star_{\mathsf{Val}} \textbf{ then } A \textbf{ else } B \,|\, K \rangle$ to $\langle A \,|\, K \rangle$! Instead:

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, $a \lessdot c$ and $c \sqsubseteq b$
(4) *For all* c such that $a \underset{\beta}{\widehat{\rightarrow}} c$, $c \sqsubseteq b$

We can now define an abstract machine abstraction to be a pair (α, β), where β is a base abstraction and α : amState⋆ $\rightarrow$ amState⋆ 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⋆ and the image of α, α(amState⋆) [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. The extended versionadds a few additional technical restrictions on α. These restrictions have no bearing on interpreted-mode graph generation, but do rule out some pathological cases in the correctness proofs for compiled-mode graph generation.

We now define the abstract transition relation $\underset{\alpha}{\widehat{\rightarrow}}$ for α as: if $a \underset{\beta}{\widehat{\rightarrow}} b$, then $a \underset{\alpha}{\widehat{\rightarrow}} \alpha(b)$. 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, proved in the extended version.

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

Following are some example abstractions.

Abstraction: Value-Irrelevance. The value-irrelevance abstraction (code in Fig. 6) maps each node val(sym, t) and each constant to $\star_{\mathsf{Val}}$, and each semantic function to the constant $x \mapsto \star_{\mathsf{Val}}$. Combining this with the abstract machine for IMP yields an expression-level control-flow graph, as in Fig. 2b. Tiger and MITScript use a modified version described at the beginning of this section, which also abstracts all function calls to $\star_{\mathsf{Val}}$.

Abstraction: Expression-Irrelevance. This abstraction is like value-irrelevance, but it also "skips" the evaluation of expressions by mapping any expression under focus to $\star_{\mathsf{Val}}$. In doing so, it overapproximates modifications to the state from running e; the easiest implementation is to add the mapping $[\star_{\mathsf{Val}} \mapsto \star_{\mathsf{Val}}]$. Combining this with an abstract machine yields a statement-level control-flow graph.

The Boolean-Tracking Abstraction. This abstraction is similar to value-irrelevance, except that it preserves some **true** and **false** values. It differs in two ways: (1) boolean-valued semantic functions such as < nondeterministically return {**true**, **false**}, (2) for 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 a provided set of *tracking variables* 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 π : amState⋆ $\rightarrow$ amState⋆. They are used after constructing the initial CFG to merge together extra nodes.

The definition below comes from §5.4 of 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 $V \subseteq$ amState⋆, $E \subseteq \left(\text{amState}\star^2\right)$. Let π be a projection. Then the **projected graph** (or **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. The addition of projections gives our final definition of a CFG: when G is the concrete transition graph for some program P, and H is a finite abstract transition graph, and every state in G is $\sqsubseteq$ some state in H and vice versa, then for any projection π, the projected graph of H under π is a valid CFG for P. In short, a CFG is **a projection of the transition graph of abstracted abstract machine states**.

Projections can be defined either manually, or automatically by the graph-pattern code generator (§6), and are most often used to hide internal details of a language's semantics, such as in the graph pattern of Fig. 3, which merges away the internal steps of a while-loop which are mere artifacts of the SOS rules. But one important projection goes further:

Projection: Basic Block. The basic-block projection inputs $\langle c \mid K \rangle$ 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 interpreted-mode CFG-generation algorithm 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 $\star_{\mathrm{Val}}$, 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, we discuss in §6.2 and prove in the extended versionthat, 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. Correctness results are given in the extended version.

6.1 Algorithm

There are two primary points of simplification in §2.3. First, it did not explain how to run AM rules on terms with variables. Second, it ignored match types.

Executing Terms with Variables. A $\star$ term does not have identity. In a state $\left\langle (\star_{\mathsf{NonVal}}, \mu) \,\middle|\, K \right\rangle$, some work is needed to determine which (if any) term in the original program that $\star_{\mathsf{NonVal}}$ corresponds to. So 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, which is needed to tie a control-flow graph to the original AST.

Variables, in contrast, do have identity. And variables can serve in the place of $\star$, for 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. 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.

This operation is called *narrowing*, used since 1975 in decision procedures [Lankford 1975] and functional-logic programming. We present a variant of narrowing which makes it suitable for abstract interpretation of semantics: we say that $f \rightsquigarrow g$ if, for some rule $x \longrightarrow y$, f and x unify by substitution σ, and $g = \sigma(y)$. 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.

Note that the abstract-rewriting of §5 can be viewed as an overapproximation of our version of narrowing that follows each narrowing step with an abstraction that replaces each occurrence of the same with distinct fresh variables (i.e.: $\star$ nodes), along with extensions to handle match types and semantic functions.

Our abstract rewriting differs from conventional narrowing in an important way. Conventional narrowing is actually a ternary relation. For instance, the rule $f(f(x)) \longrightarrow x$ 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.

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 maximum value $\top_l$. Then, graph patterns are generated as follows: For each node type N, generate the abstract transition graph by narrowing from the start state $S = \left\langle (N(\overline{x^i_{\mathsf{NonVal}}}), \top_l) \,\middle|\, k \right\rangle$, where the x^i are arbitrary non-value variables, and k is fresh. 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 S, and v, μ are an arbitrary value and reduction state.

From this, we see that the graph pattern in Fig. 3 was slightly simplified. The real one has each starting variable annotated with NonVal and replaces each $\star$ node with $\star_{\mathsf{Val}}$.

Code-Generation. Given a graph pattern, the code generator will perform a greedy search to find a projection which groups the graph nodes into loop-free segments, each group associated with an AST node. From this graph, the final code generation step is trivial, with one `connect` statement per edge in the projected graph pattern (see Fig. 3). The discovered projection is guaranteed to be valid, and hence the generated CFG-generator is guaranteed to be correct. While they do not appear in the languages under study, there are also pathological cases where a projection of the desired form does not exist; in this case, the search terminates with failure.

The code generator's high-level workings were in §2.3; some additional details are in the extended version; examples are available in §8 and the supplementary material.

6.2 An Automated Termination-Prover

As the normal abstract state transition graph overapproximates all concrete executions of a program, a graph pattern for a language construct overapproximates the relevant fragment of all abstract state transition graphs.

Is it possible to have finite graph patterns but infinite abstract transition graphs, i.e.: for the compiled-mode CFG generator to terminate, but not the interpreted-mode one? With some light assumptions, we can show this is not the case. Hence, while the output of the compiled-mode CFG-generator will always be less precise than that of an interpreted-mode generator, it turns out running the compiled-mode generator once per language will prove that the interpreted-mode generator terminates on all programs in that language.

THEOREM 6.1. *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 $\underset{\alpha}{\widehat{\rightarrow}}$ relation.*

This proof requires a few more technical assumptions and a refinement to the definition of $\underset{\alpha}{\widehat{\rightsquigarrow}}$. The details are in the extended version.

7 DERIVING CONTROL FROM A MANDATE

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 9600 lines of Haskell: 4100 lines in the core engine, and 5500 lines for our language definitions and example analyzers. 1350 of those lines define the 80 SOS rules for TIGER and 60 rules for MITSCRIPT, using the DSL depicted in Fig. 13. 550 of those lines are automatically-generated CFG-generation code.

```
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. 13. Encoding of the ASSNCONG rule from §3.3

Table 1. Generatable CFG-generators.

	Interpreted			Compiled	
	E	S	P	E	S
IMP	✓	✓	✓	✓	✓
TIGER	✓	N/A	×	✓	N/A
MITSCRIPT	✓	✓	×	✓	✓

Table 2. Example analyzers

	LOC	IMP	TIGER	MIT
Constant-prop	115	✓	✓	✓
Paren-balancing	49	✓	×	×

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 three abstractions from §5.3. The expression- and statement-level CFG-generators come in interpreted-mode and compiled-mode flavors. TIGER lacks a statement-level CFG-generator, because everything in TIGER is an expression. §8 explains the structure of heaps in TIGER and MITSCRIPT, and the implementation that would be required for MANDATE to support the boolean-tracking abstraction for them.

The high readability of the generated CFG-generators (included in the supplementary material) allows for easy inspection and comparison to intuitive control-flow. But, to further test the usefulness of the generated CFGs, we built two example analyzers, summarized in Table 2. The first is a simple constant-propagation analysis on expression-level CFGs, supporting assignments and integer arithmetic. The second is the parenthesis-balancing analyzer described in §1, built atop the path-sensitive CFG. Even though Mandate was built as a demonstration of theory rather than as a practical tool, the simplicity of this exercise is further evidence that Mandate's output does indeed correspond to conventional hand-written CFG-generators, while their brevity reinforces our thesis that **having the appropriate kind of CFG-generator greatly simplifies tool construction**.

In the remainder of this section, we demonstrate the power of Mandate by showing how it generates concise, readable code even in the face of complicated language constructs.

8 CONTROL-FLOW GRAPHS FOR TIGER AND MITSCRIPT

Previous sections used IMP as the running language, which, in our implementation, has only 20 SOS rules, with low complexity. In this section, we explain how our techniques work when applied to two larger languages, Tiger and MITScript, which have 80 and 60 rules, respectively. It turns out that these do not introduce fundamental new challenges, although they do impose more stress on Mandate's term-rewriting engine.

The actual generated graph patterns and compiled-mode CFG-generators, as well as several example outputs of the interpreted-mode CFG-generators, are available in the supplemental material.

```
(ConsFrame (HeapAddr 0) NilFrame,
 JustSimpMap $ SimpEnvMap $ Map.fromList
  [ (HeapAddr 0,
     ReducedRec
     $ RedRecCons (RedRecPair (Name "print")
                   (RefVal $ HeapAddr 1))
     $ RedRecCons (RedRecPair (Name "read")
                   (RefVal $ HeapAddr 2))
     $ RedRecCons (RedRecPair (Name "intcast")
                   (RefVal $ HeapAddr 3))
     $ Parent $ HeapAddr $ -1)
  , (HeapAddr 1, builtinPrint)
  , (HeapAddr 2, builtinRead)
  , (HeapAddr 3, builtinIntCast)
  ])
```

Fig. 14. Starting state of MITScript programs

```
genCfg t@(Node "ForExp" [a, b, c, d]) =
  do (tIn, tOut) <- makeInOut t
     (bIn, bOut) <- genCfg b
     (cIn, cOut) <- genCfg c
     (dIn, dOut) <- genCfg d
     (aIn, aOut) <- genCfg a
     connect tIn bIn; connect dOut tOut
     connect dOut dIn; connect dOut tOut
     connect cOut dOut; connect bOut cIn
     return (inNodes [tIn], outNodes [tOut])
```

Fig. 15. Generated Tiger for-loop CFG generator

Reduction State. The main difference between IMP and the larger languages is in the structure of their heap. In IMP, the reduction state was a simple map of variable names to values. In Tiger and MITScript, the reduction state must allow for stack frames, pointers, and closures. This reduction state is merely **a particularly-shaped pair of environment and term**, and involves **no extension** to the mechanics already used in IMP. We show an example reduction state in Fig. 14, the starting state of all MTIScript programs, which contains hardcoded mappings of several strings to builtin functions. More details about these reduction states are available in the extended version.

A limitation of the current Mandate implementation is that it does not support associative-commutative matching for nested maps. Hence, each heap record is implemented as an assoc-list rather than a map, with record lookup implemented as a semantic function. This is the reason why the boolean-tracking abstraction is not implemented for Tiger or MITScript.

Examples. Our first example is MITScript if-statements, which illustrate how Mandate can generate multiple CFG-generators from the same semantics. When run in compiled-mode with the value-irrelevance abstraction, Mandate generates the code in Fig. 16a for if-statements:

Note how the generator returns multiple exit-nodes for the if-statement. This means that the graph-generator will draw edges from both nodes to whatever comes after the if-statement. When

```
genCfg t@(Node "If" [a, b, c]) =
  do (tIn, tOut) <- makeInOut t
     (aIn, aOut) <- genCfg a
     (bIn, bOut) <- genCfg b
     (cIn, cOut) <- genCfg c
     connect tIn aIn
     connect aOut cIn
     connect aOut bIn
     return (inNodes [tIn],
             outNodes [bOut,cOut])
```

(a)

```
genCfg t@(Node "If" [_, a, b]) =
  do (tIn, tOut) <- makeInOut t
     (bIn, bOut) <- genCfg b
     (aIn, aOut) <- genCfg a
     connect tIn bIn
     connect tIn aIn
     return (inNodes [tIn],
             outNodes [bOut,aOut])
```

(b)

Fig. 16. Generated expression-level (a) and statement-level (b) CFG-generators for MITScript if-statements.

run with the expression-irrelevance abstraction Mandate produces code generating smaller graphs which do not have nodes to represent the evaluation of the condition, code shown in Fig. 16b.

We next show our most impressive example, showcasing Mandate's ability to cut through syntactic sugar to determine the control-flow behavior of a node type: Tiger for-loops. Tiger for-loops have more parts than for-loops in most languages. A for-loop in Tiger looks like this.

```
for a = b to c do
  d
```

The semantics are given by a single rule which desugars the above to:

```
let a := b
    __hi := c
in
  while (a <= __hi) do
    (d; a := a + 1)
```

Mandate generates a graph pattern containing 46 nodes (shown in the extended version). Mandate's code-generator projects these into just 8 states, for the enter and exit nodes of *b*, *c*, *d*, and the entire loop, yielding the code in Fig. 15.
Mandate has blown that giant graph down into just 5 edges. While Mandate does not sort the `connect` statements, and does output one edge twice, it is still easy to see what the code is doing: it states that control first evaluates *b*, then *c*. The `connect cOut dOut` line is the most interesting one: it says that, after evaluating *c* (the upper boundary of the loop), control flows to the thing that happens after *d* is evaluated, namely the condition of the `while` loop, from which control flows either to the body of *d* or to the end of the entire loop.
It is impressive that Mandate can generate short code for this construct with no reference to these internal computations, particularly considering that `while` is defined by expansion into `if`. **This code is generated completely automatically** from the Tiger semantics and the (function-skipping variant of the) value-irrelevance abstraction. The user need not even provide a projection; one is generated automatically by the code generator, by greedily merging nodes as described in §6.

More examples are available in the supplementary material.

9 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 multiple executions of interpreters or semantics [Bodin et al. 2019; 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. Bodin et al [Bodin et al. 2019] briefly mention having a 0-CFA for the lambda calculus generated from their "skeletal semantics;" in personal communication, they described it as "working code, but not a principled approach" and "We don't do a CFG-generator."

Transformation of Semantics. There are a few projects that transform semantics for one language into semantics for a related language. Examples include transforming 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]. Supporting these, there are several languages for defining semantics [Klein et al. 2012; Lakin and Pitts 2007; Mulligan et al. 2014].

More closely related are projects that transform semantics presented in one formalism into identical semantics in a different formalism, mostly done by Danvy and his students [Ager et al. 2003; Biernacka 2006; Danvy 2008; Danvy and Johannsen 2010; Danvy et al. 2012; Danvy and Nielsen 2004; Xiao et al. 2001], with a few by others: Hannan and Miller [1992]; Huizing et al. [2010]; Poulsen and Mosses [2014]; Vesely and Fisher [2019].

At the start of this project, we attempted to build on this prior work. However, we were surprised to find that **there is no prior published algorithm for converting SOS to abstract machines**, and found the existing reduction-semantics-to-AM algorithm too weak to be used off-the-shelf as an intermediate step. After discovering PAM, we found it easier to work with: proving the "up-rules invertible" property of PAM is done automatically by 20 lines of code, whereas it took a 20-page paper to explain how to mechanically (not automatically) prove the corresponding property for reduction semantics [Xiao et al. 2001]. See the extended versionfor more discussion of attempts to use prior work. To our knowledge, the TIGER and MITSCRIPT languages in this paper are by far the largest languages to undergo automated conversion between two forms of semantics; prior work focuses on simple lambda calculi.

Abstracting Abstract Machines (AAM). There are several AAM projects [Glaze and Horn 2014; Glaze et al. 2013; Midtgaard and Jensen 2008, 2009; Van Horn and Might 2010; Wei et al. 2018], with the earliest example arguably being Jones [1981]. We explain in the extended versionhow their abstracting techniques differ fundamentally from abstract rewriting, and why this means that a control-flow analysis based on AAM will not resemble a CFG.

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 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 overapproximate all concrete transitions from an abstract term. A newer related technique from a different lineage is rewriting modulo SMT [Rocha et al. 2017], which operates on (numeric) symbolic terms constrained by an SMT formula (e.g.: linear arithmetic). We are interested in future work combining these techniques to automatically derive more-precise analyses.

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].

In the extended version, we discuss the use of several terms similar to "abstract matching" in the contexts of abstract interpretation of logic programs, model-checking, and term-rewriting engines, as well as several minor uses of syntactic abstraction.

Control-Flow Analysis. Many papers have been written on control-flow analysis [Jagannathan and Weeks 1995; Jones 1981; Midtgaard 2012; Shivers 1991]. Older research tries to manually construct a complicated analysis of programs with highly-dynamic control flow. Our work automatically constructs CFG-generators from first principles. Our goal is not to analyze complex programs, but to match the work of hand-written CFG-generators with minimal user input.

As such, we do not consider this work as part of the literature on control-flow analysis. Owing to their different emphasis, these works uniformly have three limitations that make them unsuitable for automatically deriving CFG-generators:

(1) While they explain how a human could define a new analysis for different languages, their analyses are ultimately manually defined for each language. They further repeat this manual construction for every abstraction used.
(2) They focus on safely approximating executions but not on 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.

The extended versiongives a brief overview of control-flow analysis, and discusses two works that deserve special mention, due to use of abstract machines and focus on generality.

10 CONCLUSION

This work presented both an algorithm for constructing CFGs from first principles and 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 idea of using it 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 tool entirely unlike a semantics. Though it's long been known that a semantics can be executed to obtain an interpreter or even a symbolic-executor [Roşu and Şerbănută 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.

Mandate is available from https://github.com/jkoppel/mandate.

ACKNOWLEDGMENTS

This material is based upon work supported by the US Air Force, AFRL/RIKE and DARPA under Contract No. FA8750-20-C-0208. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the US Air Force, AFRL/RIKE or DARPA.

REFERENCES

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.

Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. 2019. Skeletal semantics and their interpretations. *PACMPL* 3, POPL (2019), 44:1–44:31. https://doi.org/10.1145/3290357

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.

Michael Carbin and Armando Solar-Lezama. 2018. *MITScript Language Specification*. http://6.s081.scripts.mit.edu/sp18/handout-pdfs/specification.pdf

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

Clifford Noel Click. 1995. *Combining Analyses, Combining Optimizations*. Ph.D. Dissertation. Rice University.

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.

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

Dionna Amalie Glaze 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

Dionna Amalie Glaze, 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

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

Cornelis Huizing, Ron Koymans, and Ruurd Kuiper. 2010. A Small Step for Mankind. In *Concurrency, Compositionality, and Correctness*. Springer, 66–73.

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

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

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run Your Research: On the Effectiveness of Lightweight Mechanization. *ACM SIGPLAN Notices* 47, 1 (2012), 285–296.

James Koppel, Jackson Kearl, and Armando Solar-Lezama. 2020. Automatically Deriving Control-Flow Graph Generators from Operational Semantics. *arXiv preprint arXiv:2010.04918* (2020).

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.

Matthew R Lakin and Andrew M Pitts. 2007. A Metalanguage for Structural Operational Semantics. In *Symposium on Trends in Functional Programming*.

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* (Edinburgh, Scotland) *(ICFP '09)*. ACM, New York, NY, USA, 287–298. https://doi.org/10.1145/1596550.1596592

Dominic P Mulligan, Scott Owens, Kathryn E Gray, Tom Ridge, and Peter Sewell. 2014. Lem: Reusable Engineering of Real-World Semantics. *ACM SIGPLAN Notices* 49, 9 (2014), 175–188.

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.

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.

Casper Bach Poulsen and Peter D Mosses. 2014. Deriving Pretty-Big-Step Semantics from Small-Step Semantics. In *European Symposium on Programming Languages and Systems*. Springer, 270–289.

Camilo Rocha, José Meseguer, and César Muñoz. 2017. Rewriting Modulo SMT and Open System Analysis. *Journal of Logical and Algebraic Methods in Programming* 86, 1 (2017), 269–297.

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.

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.*

Ferdinand Vesely and Kathleen Fisher. 2019. One Step at a Time. In *European Symposium on Programming*. Springer, Cham, 205–231.

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