A Machine Equivalence Between Call-By-Push-Value and SSA Form

Dmitri Garbuzov William Manksy Steve Zdancewic
University of Pennsylvania
dmitri@seas.upenn.edu, wmansky@seas.upenn.edu, stevez@cis.upenn.edu

Abstract

This paper establishes a tight equivalence between (a variant of) Levy's call-by-push-value (CBPV) calculus and a virtual machine that operates on control flow graphs of instructions in static single assignment (SSA) form. The correspondence is made precise via a series of abstract machines that align the transitions of the structural operational semantics of the CBPV language with the computation steps of the SSA form.

The target machine, which is derived from the CBPV language, accurately captures the execution model of control flow graph formalisms, including direct jumps, mutually recursive code blocks, and multi-argument function calls. The closure-free subset is similar to the SSA intermediate representations found in modern compilers such as LLVM and GCC. The full language additionally supports explicit tail calls and closures, making it a suitable target for compiling higher-order languages.

The key technical result proves that the high-level and low-level operational semantics are different representations of the *same* transition system. This means that a wide variety of operational equivalences defined on CBPV terms can be transported to the abstract machines while preserving reasoning principles (for instance about execution costs). As such, this paper is a step towards translating results and techniques between lambda-calculus-based and control-flow-graph-based formalisms.

The definitions of all the language/abstract machine semantics and the theorems relating them are fully verified in Coq.

1. Introduction

Classical compiler theory, data-flow analysis, abstract interpretation, and optimization are all traditionally formulated in terms of control-flow-graph machines. (Muchnick 1997; Aho et al. 1986) Among the possible control-flow-graph (CFG) code representations, modern compilers like GCC and LLVM choose static single assignment (SSA) form for their implementations, in part because SSA offers an efficient representation for algorithms that work with low-level code, and in part because SSA form is conducive to many dataflow analyses and optimizations. SSA-based and other CFG-based machines serve as abstract, intuitive models of processor execution with (mostly) well-defined cost models that are suitable for code generation. They are clearly useful tools for compilation.

In the realm of verified compilation, the CompCert verified compiler (Leroy 2009) uses CFG-based machine models to define the semantics of various intermediate representations and to prove the correctness of all of its optimization passes. Similarly, formal treatments of SSA intermediate representations (Barthe et al. 2014; Zhao et al. 2012) invariably define their semantics in terms of CFG machines.

Despite their prevalence, working with such CFG machines is not all a bed of roses. CFGs, as an abstract model of processors, include many details that are useful for generating code but not for reasoning about the correctness of optimizations. Moreover, CFG program structure is typically not compositional, making some optimizations that radically alter the control flow graph (such as function inlining) difficult to implement correctly. Similarly, the imperative nature of their associated abstract machines makes formulating an equational theory for CFGs difficult, which can, in turn, make it hard to know what program transformations are valid.[1]

Structural operational semantics (Plotkin 2004) was developed specifically to address the shortcomings of working with such low-level machine models for reasoning about the operational behavior of programming languages. Defining operational semantics based on the structure of program terms and using meta-level operations like substitution have proven to be useful for defining and proving (contextual) program equivalences for a wide variety of realistic programming languages and features, especially those associated with the lambda calculus. (Pierce 2004; Pitts 2000)

This state of affairs is somewhat surprising given that Appel declared some time ago that "SSA *is* Functional Programming" (Appel 1998) and argued that SSA form is essentially equivalent to the CPS subset of lambda calculus.

While there is a large body of work on abstract machines for languages based on the lambda calculus and the precise relationships between different styles of operational semantics (see the discussion on related work in Section 9), these techniques have not yet been applied in the context of CFGs. We posit that this disconnect between lambda-calculus-based formalisms and CFG-based formalisms exists because prior approaches fail to capture some of the intensional properties implicit in CFG-based formalisms that are crucial for compilation to standard hardware.

In this paper, we seek to improve this situation by explaining Appel's slogan "SSA is Functional Programming" in a way that is both *precise* and *useful*—capturing enough relevant details of the CFG computation model to be used for verified compilation or for justifying compiler optimizations while still retaining the connection to lambda-calculus-style SOS.

[1] Indeed, early SSA compilers used incorrect program transformations that were subsequently corrected (Briggs et al. 1998). The problem had to do with correctly implementing the semantics of phi functions when translating out of SSA form after certain optimizations had been applied.

Contribution: Our main contribution is a proof of *machine equivalence* between (a variant of) Levy's call-by-push-value (CBPV) lambda calculus (Levy 1999) and a virtual machine that operates on control flow graphs of instructions in SSA form. This result shows that the CBPV and CFG semantics are different representations of the *same* transition system—this is a very strong machine equivalence, and we explain it in detail below.

The CFG machine accurately captures many features of the execution model intended for SSA formalisms, including direct jumps, mutually recursive code blocks, and multi-argument function calls. As such, the closure-free fragment is similar to SSA IRs in modern compilers such as LLVM and GCC. Additionally, because it is derived from CBPV, which is a higher-order language, our CFG formalism smoothly extends traditional SSA with higher-order features.

The rest of the paper proceeds as follows. Section 2 demonstrates some key features of an SSA machine and foreshadows the results of the paper by example. Section 3 introduces the call-by-push-value calculus, its SOS semantics, and a corresponding simple abstract machine. Section 4 sketches the high-level structure of the main result and explains our formulation of machine equivalence. The meat of the paper follows in sections 5 and 6. Inspired by the techniques in (Ager et al. 2003b), we derive the CFG machine via a series of abstract machines by systematically teasing apart the static content (*i.e.* the control flow graph of instructions) from the dynamic content (*i.e.* the environment and call stack) of a CBPV term. Section 7 shows how to refine the correspondence between CBPV and CFG semantics to account for different execution models by adjusting the compiler and the step size of the CBPV SOS. We conclude by comparing our result to existing work and suggesting some potential applications and future directions in Sections 8 and 9.

The definitions of all the language/abstract machine semantics and the theorems relating them are fully verified in Coq.[2]

2. Background: SSA Semantics

This section describes the key features of SSA-structured code by way of example and foreshadows our technical developments by showing the same program in both CFG and CBPV representations.

2.1 Static Single Assignment Form

Static single assignment form (SSA) is a popular family of compiler intermediate representations, used in both leading C implementations (Lattner and Adve 2004; Stallman and the GCC Developer Community 2009) and compilers for functional languages (Weeks 2006). It was introduced primarily to increase the efficiency of common dataflow-based optimizations (Cytron et al. 1991), and is both an efficient datastructure manipulated by the optimizer and a programming language in its own right. There are many different variants of SSA IRs, and so there is no canonical syntax or operational model, but these various IRs all share some features: SSA code is organized as a control flow graph of instructions with the constraint that each variable defined by an instruction is assigned exactly once, statically. The representation has no explicit nested scope, and blocks can refer to any label or identifier. However, to be well-formed, each use of an identifier must be dominated by its unique definition in the CFG.

The left side of Fig.1 shows a procedure power(n,m) that computes n^m by iterated addition and nested loops. This code is representative of SSA IRs. The function body consists of a sequence of labeled blocks of code, each terminated with a jump, conditional branch, or return. Each block optionally begins with a sequence of phi instructions, which take a list of arguments annotated with the labels of each control-flow predecessor.

Intuitively, the intraprocedural fragment of SSA is executed by a simple machine whose states consist of an instruction pointer and an assignment of identifiers (a.k.a. registers) to values. This SSA program's semantics are mostly straightforward—labels act as the targets of jumps but otherwise have no operational meaning. Most instructions compute a value and update the local environment. The semantics of a phi instruction is subtle, however. It binds its identifier to the value associated with the label of the dynamically last executed block and its right-hand side is interpreted using the register state available at the end of that block. This means that, semantically, the phi instructions at the beginning of a block should behave as though they are parallel-move operations that take place at each of the predecessor blocks.[3]

2.2 Execution models for CFG machines

As we noted in the introduction, SSA and other CFG-based intermediate representations model computation as the execution of an idealized processor. The machine state includes an instruction pointer, a register file, and a region of memory allocated according to a stack discipline. There is additionally a data structure representing the source program loaded into memory as instructions. The associated execution model is flexible enough that optimizations can be expressed generically for a range of concrete targets, but can take advantage of specific features of this model of computation. In other words, they make just enough intensional information about execution explicit to express optimizations that take advantage of common features of processors. This same information makes formal reasoning about program equivalence difficult for a number of reasons.

Instructions make regular use of an abstract register file: there are no specialized registers with distinct protocols. While each register has a single point of definition, SSA registers are updated in-place, unlike in environment-based abstract machines that implement lexical scoping for functional languages using environments. It is far from trivial to implement the SSA register file as lexical scoping, and there is existing work on when exactly the two forms of semantics coincide (Beringer et al. 2003; Schneider et al. 2015).

SSA typically has separate instructions for representing various kinds of control transfer that, semantically, can be modeled as just function calls. In addition to regular "machine function" calls that are executed by pushing a frame onto the stack, there are jumps with arguments within an SSA function that behave as tail calls. Both varieties are further divided into direct and indirect variants. Direct calls are represented as instructions with a literal label that identifies a location in the CFG, the data structure representing the program. Indirect jumps, on the other hand, fetch a label from the environment. For the purposes of generating machine code, and the low-level cost model used by compilers, these differences are important. While canonical abstract machines studied in the context of the lambda calculus distinguish between calls and tail calls, we are not aware of any that make the difference between direct and indirect jumps explicit.

Additionally, our CFG machine instruction set includes multi-argument function calls. This is only remarkable because our source calculus only includes functions of a single argument. Typically, reasoning about multi-argument functions requires either an extension of the source calculus, or else an execution model that handles curried function application (Leroy 1990).

[2] We have included the Coq development as non-anonymous supplementary material, and intend to submit a documented version for artifact evaluation.

[3] Even this view is naïve: the presence of conditional branches complicates the story and there have been several papers about correctly translating *out* of SSA form to more standard instruction sets (Cytron et al. 1991; Sreedhar et al. 1999).

```
function power(n, m):
e:
  jmp f
f:
  m1 ← phi(e:m, g:mk)
  a1 ← phi(e:1, g:a2)
  if0 m1 jmp f0 else f1
f0:
  ret a1
f1:
  m2 ← sub m1 1
  jmp g
g:
  m3 ← phi(f1:a1, g0:m4)
  a2 ← phi(f1:0, g0:a3)
  mk ← phi(f1:m2, g0:mk)
  if0 m3 jmp f else g0
g0:
  m4 ← sub m3 1
  a3 ← add a2 n
  jmp g
```

```
power:
0:  n = POP [1]
1:  m = POP [2]
2:  TAIL @3 m 1

3:  m1 = POP [4]
4:  a1 = POP [5]
5:  IF0 m1 [6; 7]

6:  RET a1

7:  m2 = SUB m1 1 [8]
8:  TAIL @9 a1 0 m2

9:  m3 = POP [10]
10: a2 = POP [11]
11: mk = POP [12]
12: IF0 m3 [13; 14]

13: TAIL @3 mk a2

14: m4 = SUB m3 1 [15]
15: a3 = ADD a2 n [16]
16: TAIL @9 m4 a3 mk
```

$$
\begin{aligned}
&\mathsf{power} = \\
&\lambda n.\lambda m. \\
&\quad 1 \cdot m \cdot \overline{\mathsf{prj}_0} \cdot \mathsf{force}(\mathsf{rec}\ x. \\
&\qquad \langle \lambda m_1.\lambda a_1. \\
&\qquad\quad \mathsf{if0}\ m_1 \\
&\qquad\qquad (\mathsf{prd}\ a_1) \\
&\qquad\qquad ((m_1 - 1)\ \mathsf{to}\ m_2\ \mathsf{in} \\
&\qquad\qquad\quad m_2 \cdot 0 \cdot a_1 \cdot \overline{\mathsf{prj}_1} \cdot (\mathsf{force}\ x))\ , \\
&\qquad \lambda m_3.\lambda a_2.\lambda m_k. \\
&\qquad\quad \mathsf{if0}\ m_3 \\
&\qquad\qquad (a_2 \cdot m_k \cdot \overline{\mathsf{prj}_0} \cdot (\mathsf{force}\ x)) \\
&\qquad\qquad ((m_3 - 1)\ \mathsf{to}\ m_4\ \mathsf{in} \\
&\qquad\qquad (a_2 + n)\ \mathsf{to}\ a_3\ \mathsf{in} \\
&\qquad\qquad\quad m_k \cdot a_3 \cdot m_4 \cdot \overline{\mathsf{prj}_1} \cdot (\mathsf{force}\ x)) \\
&\qquad \rangle)
\end{aligned}
$$

Figure 1. Three versions of a program that computes n^m via repeated addition. **Left**: The program written in informal SSA notation. **Middle**: The program written in our CFG notation. **Right**: The CBPV representation that compiles to the middle CFG program. The CFG and CBPV programs are machine equivalent—the transition steps they take are in one-to-one correspondence.

At a high level, we will be analyzing the correspondence between SSA and functional programming as the relationship between abstract machine and source language. This will allow us to map CFG machine states back to source terms with an operational semantics that makes fewer such low-level distinctions. We begin with an overview of the derived CFG machine and our chosen source language.

2.3 CFG form

The middle of Figure 1 shows the power program represented using the control-flow-graph virtual machine language that we derive in this paper. A CFG program is represented by a mapping from program points (numbers running down the left-hand side of the code) to instructions. As in the SSA representation, each CFG instruction binds a value to a local variable. So, for instance, the program fragment `0: n = POP [1]` pops the top element of the stack and assigns it to the local identifier `n`. Unlike the SSA representation, our CFG representation does not have implicit "fall through" for instruction sequences. The numbers, like `[1]`, written in square brackets at the end of an instruction indicate the program point to which control should pass after the instruction executes. A conditional branch like `5: IF0 m1 [6; 7]` selects one of two successor program points based on the value of its first operand.

In our CFG representation, phi nodes are replaced by `POP` instructions and the corresponding jmps are replaced with argument-taking `TAIL` call instructions. For example, the tail call instruction `13: TAIL @3 mk a2` transfers control to the program point `3`—the `@` marker indicates that the program point is being used as a code label that can be called. This particular tail call corresponds to the g entries of the two phi nodes at label f in the SSA version.

(As we will see later, in general CFG programs may treat labels as first-class values and can use indirect jumps.)

2.4 CBPV form

The right side of Figure 1 shows the same program yet again, but this time represented using a call-by-push-value (CBPV) term. We have chosen the variable names so that they line up precisely with the SSA representation. We will explain the semantics of CBPV programs and its correspondence with the CFG machine in more detail below. For now, it is enough to observe that each λ-abstraction corresponds to a `POP`. Application is written in reverse order compared to usual lambda calculus and CBPV also has a sequencing operation M to x in N.

Recursive code is explicitly marked with a rec keyword, and can be bundled together in tuples. In this example, there is a pair of computations such that the 0th component, accessed via the projection prj_0, corresponds to the f block in the SSA (code point 3 in the CFG) and the 1st component, accessed via the projection prj_1, corresponds to the g block (code point 5).

3. CBPV and its CEK Machine

3.1 CBPV Structural Operational Semantics

Figure 2 shows the syntax and operational semantics for our variant of Levy's call-by-push-value calculus (Levy 1999), which serves as the source language of the machine equivalence. As a functional language, CBPV is somewhat lower level and more structured than ordinary lambda calculus. It syntactically distinguishes values V, which include variables x, (numeric) contants n, and recursive thunks rec $x.M$, from general computation terms M. This value–computation distinction is a key feature of the CBPV design: its evaluation order is completely determined thanks to syntactic restrictions that ensure there is never a choice between a substitution step and a congruence rule.

For the most part, the operational semantics of CBPV is unsurprising. We write $\{V/x\}\,M$ for the usual notion of capture-avoiding subsitution of V for x in M. The term force (rec $x.M$) runs the suspended computation M, unrolling the recursive definition as shown in rule *force* of the operational semantics. CBPV computations are structured monadically, where the term prd V (pronounced "produce V") is the unit of the monad and M to x in N sequences the computation M before the computation N, binding

$$
\begin{array}{rrcl}
\text{Values} & \ni V & ::= & x \mid n \mid \mathsf{rec}\ x.M \\
\text{Terms} & \ni M, N & ::= & \mathsf{prd}\ V \mid M\ \mathsf{to}\ x\ \mathsf{in}\ N \mid V \cdot M \mid \lambda x.M \\
& & & \mid \langle M_1, .., M_n \rangle \mid \mathsf{prj}_n\ M \\
& & & \mid \mathsf{if0}\ V\ M_1\ M_2 \mid V_1 \oplus V_2 \\
& & & \mid \mathsf{force}\ V
\end{array}
$$

$$
\begin{array}{lrcll}
force & \mathsf{force}\,(\mathsf{rec}\ x.M) & \longrightarrow & \{\mathsf{rec}\ x.M/x\}\ M & \\
bind & \mathsf{prd}\ V\ \mathsf{to}\ x\ \mathsf{in}\ M & \longrightarrow & \{V/x\}\ M & \\
\beta/pop & V \cdot \lambda x.M & \longrightarrow & \{V/x\}\ M & \\
op & V_1 \oplus V_2\ \mathsf{to}\ x\ \mathsf{in}\ M & \longrightarrow & \{V_1 [\![\oplus]\!] V_2/x\}\ M & \\
cond_0 & \mathsf{if0}\ 0\ M_1\ M_2 & \longrightarrow & M_1 & \\
cond_n & \mathsf{if0}\ n\ M_1\ M_2 & \longrightarrow & M_2 & \text{if } n \neq 0 \\
prj & \mathsf{prj}_i\ \langle M_1, .., M_n \rangle & \longrightarrow & M_i & \\
push_N & M\ \mathsf{to}\ x\ \mathsf{in}\ N & \longrightarrow & M'\ \mathsf{to}\ x\ \mathsf{in}\ N & \text{if } M \longrightarrow M' \\
push_V & V \cdot M & \longrightarrow & V \cdot M' & \text{if } M \longrightarrow M' \\
push_i & \mathsf{prj}_i\ M & \longrightarrow & \mathsf{prj}_i\ M' & \text{if } M \longrightarrow M'
\end{array}
$$

Figure 2. Syntax and operational semantics for the CBPV language.

M's result to x in N (see rule $bind$). We will sometimes refer to $M\ \mathsf{to}\ x\ \mathsf{in}\ N$ as a "sequence" term.

Somewhat unusually, an application term $V \cdot M$ is written with the argument V, which is syntactically constrained to be a value, to the *left* of the function M, which should evaluate to an abstraction $\lambda x.N$ using the congruence rule $push_V$. Applications reduce via the usual β rule thereafter. An equivalent reading of these rules (which will be made much more formal when we lower CBPV to an abstract machine) is to think of rule $push_V$ as pushing V onto a stack while M continues to compute until the β step pops V off the stack for use in the function body. (This point of view is where call-by-push-value gets its name.)

The term $\langle M_1, .., M_n \rangle$ is a tuple of computations, one of which can be selected via a projection term $\mathsf{prj}_i\ M$. Note that such tuples are *lazy* and they are not, by themselves, values. In the machine correspondence with SSA, such a tuple represents a collection of basic blocks. That means that the index i in a projection corresponds to a code label of a basic block and that rule prj corresponds to a jump.

Binary operations take only values as arguments. We write $\oplus$ for a generic arithmetic operator and denote its interpretation by $[\![\oplus]\!]$; both are written infix and they evaluate as shown by rule op. Conditional computations (rules $cond_0$ and $cond_n$) are standard.

The structural rules (the last three rules of Figure 2) evaluate subterms only to the left of a sequencing bind, to the right of an application, and under a projection. As alluded to in the description above, each of these rules corresponds to a stack push operation at the abstract machine level.

Comparison to Levy's CBPV Our presentation of CBPV deviates from Levy's presentation in a few ways. First, we have combined his thunk M with the recursive binder rec because doing so leads to a cleaner formalism. Instead, we write thunk M as syntactic sugar for $\mathsf{rec}\ x.M$ when x does not occur free in M.

Second, this presentation of CBPV, unlike Levy's, is *untyped*. As a consequence, the operational semantics presented in Figure 2 can get stuck trying to evaluate ill-formed terms. However, using the type system to rule out stuck states is completely compatible with the correspondence with abstract machines—that correspondence means that a CBPV term is stuck exactly when the abstract machine is, so type safety at the CBPV level corresponds directly to type safety at the SSA level too. (This is an example of how results about CBPV can be translated to work on the SSA level.) Moreover, using an untyped language makes it easier to explore

$$
\begin{array}{rrcl}
\text{State} & \ni \sigma & = & \text{Term} \times \text{Env} \times \text{Kont} \\
\text{Term} & \ni c & ::= & M \\
\text{Env} & \ni e & = & \text{Var} \longrightarrow_{\text{fin}} \text{Val} \\
\text{Kont} & \ni k & ::= & \cdot \mid v \cdot \_ :: k \mid [\_\ \mathsf{to}\ x\ \mathsf{in}\ M, e] \mid \mathsf{prj}_n\ \_ :: k \\
\text{Val} & \ni v & ::= & x \mid n \mid [\mathsf{rec}\ x.M, e]
\end{array}
$$

$$
\begin{array}{rcl}
\gamma\ x\ e & = & e(x) \\
\gamma\ n\ e & = & n \\
\gamma\ \mathsf{rec}\ x.M\ e & = & [\mathsf{rec}\ x.M, e]
\end{array}
$$

Figure 3. CEK syntax and value-semantics function γ.

the semantics of features such as vararg functions that are useful for low-level code.

Finally, it is worth noting that the $\langle M_1, .., M_n \rangle$ product used here is from the *negative* ("computation") fragment of CBPV. We have not included Levy's *positive* ("value") tuples, which would correspond to records of values at the SSA level. Such tuples of values are simple to add, but since most low-level SSA languages don't include structured data, we omit them here for simplicity.

Why CBPV? CBPV is an attractive choice for modeling low-level control flow graphs and SSA-style IRs for several reasons. The main way that low-level code is structured is as mutually recursive collections of labeled instruction blocks connected by terminating jumps. CBPV's clear separation of values and computations is already inherent in the distinction low-level IRs make between first-class primitive values (e.g. integers and pointers that can be stored in temporaries) and the second-class code blocks that are used to describe computations.

Its simple operational model of curried functions means that CBPV already gives a good low-level account of multi-argument functions in a way that is compatible with the typical semantics of function-call operations found in SSA.

Moreover, CBPV supports a rich equational theory that can readily extended with effects, which means that it serves as a strong foundation for formal analysis of SSA programs.

From the lambda calculus point of view, we have not lost any expressiveness by starting from CBPV. There are entirely straightforward translations of ordinary lambda calculus into CBPV for both call-by-value and call-by-name evaluation strategies (Levy 1999). The main idea of these translations is to insert the thunks and sequencing needed to encode the desired behavior. For instance, the call-by-value translation is given below (where y, f, and v are fresh):

$$
\begin{array}{rcl}
[\![x]\!]_{cbv} & = & \mathsf{prd}\ x \\
[\![\lambda x.e]\!]_{cbv} & = & \mathsf{prd}(\mathsf{rec}\ y.\lambda x.[\![e]\!]_{cbv}) \\
[\![e_1\ e_2]\!]_{cbv} & = & [\![e_1]\!]_{cbv}\ \mathsf{to}\ f\ \mathsf{in}\ [\![e_2]\!]_{cbv}\ \mathsf{to}\ v\ \mathsf{in}\ (v \cdot \mathsf{force}\ f)
\end{array}
$$

While the primary aim of this paper is to give a model of control-flow-graph computations, exploring the possible applications of our approach for (verified) compilers for functional languages is a potentially fruitful avenue for future work.

3.2 CEK Machine

We define a CEK machine (Felleisen and Friedman 1986) for CBPV terms, the syntax of which is given in Figure 3. States of a CEK machine are triples $\langle c,\ e,\ k \rangle$ where c is a CBPV term (here we use the metavariable c rather than M to emphasize that this term lives at the CEK level), e is an environment mapping variables to values, and k is a stack of continuations, which we will also call frames. CEK values v include variables, number constants, and, unlike in CBPV, closures consisting of a thunk and captured environment , written $[\mathsf{rec}\ x.M, e]$. In CBPV, there are three search

$$
\begin{array}{rcll}
\langle \mathsf{force}\ V,\ e,\ k\rangle & \longrightarrow & \langle M,\ e[x \mapsto [\,\mathsf{rec}\,x.M, e']],\ k\rangle & \text{when } \gamma\ V\ e = [\,\mathsf{rec}\,x.M, e'] \\
\langle \mathsf{if0}\ V\ M_1\ M_2,\ e,\ k\rangle & \longrightarrow & \langle M_1,\ e,\ k\rangle & \text{when } \gamma\ V\ e = 0 \\
\langle \mathsf{if0}\ V\ M_1\ M_2,\ e,\ k\rangle & \longrightarrow & \langle M_2,\ e,\ k\rangle & \text{when } \gamma\ V\ e = n, n \neq 0 \\
\langle \mathsf{prd}\ V,\ e,\ [\_\ \mathsf{to}\ x\ \mathsf{in}\ M, e'] :: k\rangle & \longrightarrow & \langle M,\ e'[x \mapsto v],\ k\rangle & \text{when } \gamma\ V\ e = v \\
\langle V_1 \oplus V_2,\ e,\ [\_\ \mathsf{to}\ x\ \mathsf{in}\ M, e'] :: k\rangle & \longrightarrow & \langle M,\ e'[x \mapsto v_1 [\![\oplus]\!] v_2],\ k\rangle & \text{when } \gamma\ V_1\ e = v_1 \text{ and } \gamma\ V_2\ e = v_2 \\
\langle \lambda x.M,\ e,\ (v \cdot \_) :: k\rangle & \longrightarrow & \langle M,\ e[x \mapsto v],\ k\rangle & \\
\langle \langle M_1, .., M_n\rangle,\ e,\ (\mathsf{prj}_i\ \_) :: k\rangle & \longrightarrow & \langle M_i,\ e,\ k\rangle & \text{when } i \in \{1 \ldots n\} \\
\langle M\ \mathsf{to}\ x\ \mathsf{in}\ N,\ e,\ k\rangle & \longrightarrow & \langle M,\ e,\ [\_\ \mathsf{to}\ x\ \mathsf{in}\ N, e] :: k\rangle & \\
\langle V \cdot M,\ e,\ k\rangle & \longrightarrow & \langle M,\ e,\ (v \cdot \_) :: k\rangle & \text{when } \gamma\ V\ e = v \\
\langle \mathsf{prj}_i\ M,\ e,\ k\rangle & \longrightarrow & \langle M,\ e,\ (\mathsf{prj}_i\ \_) :: k\rangle &
\end{array}
$$

Figure 4. CEK Machine

rules in the SOS, and so we have three kinds of frames: application, sequence, and projection, each written with a "hole" _ in the place where the sub-expression's produced value will be placed.

Figure 4 gives the operational sematics of the CEK machine. It relies on a function γ, shown in Figure 3, for looking up values in the environment—it constructs closures as necessary. We write $e(x)$ for the value associated with variable x in environment e, and $e[x \mapsto v]$ for the environment e updated to map x to v. Each search construct pushes a corresponding frame onto the stack, while their corresponding reduction steps pop the frames off the stack and use them as context for reduction.

Unloading CEK Machine States. A CEK state is essentially a term with a delayed substitution: instead of performing substitution when reducing applications or sequences, we add bindings to the environment and only apply the environment to terms as needed. In a CEK state (c, e, k), c is the inner term that has not yet had e applied to it, and k is a list of outer constructors that have already had e applied to them (although they may contain other suspended environments in the form of sequence frames). Thus, a CEK state can be unloaded back into the corresponding CBPV term by first applying e to c (substituting values for all free variables), then unwrapping the frames in k around the result.[4]

Correctness with Respect to CBPV. Steps of the CEK machine do not correspond exactly to steps in the operational semantics of CBPV terms. Specifically, while the SOS includes recursive search rules, the CEK machine is not recursive. Where a CBPV term takes a step in context, the CEK machine repeatedly pushes frames to the stack until it reaches a non-control subterm. This means that the number of steps the CEK machine takes for each CBPV is computable: it is the *redex depth* of the c component, which is 1 plus the number of search constructs one must go under before reaching a non-search construct.

Lemma 1. *(CEK Correctness) For all CEK states σ,*

$\mathsf{CBPV.step}(\mathsf{CEK.unload}\,\sigma) = M'$ *iff* $\mathsf{CEK.step}^n\langle c,\ e,\ k\rangle = M'$

where n is the redex depth of c.

4. Proving Machine Equivalence

Figure 5 shows the structure of the machine correspondences proved in this paper: we translate call-by-push-value terms through a series of abstract machines, eventually reaching an SSA CFG language. At each level, the correctness lemma relates the behavior of states of the current machine to the behavior of corresponding states in the previous machine. For most of these levels, the correctness statement is lockstep: each step in the current machine

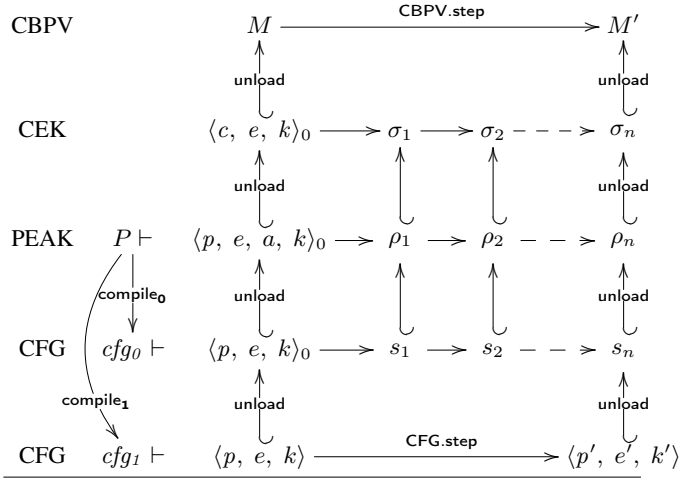

Figure 5. Structure of the machine correspondences.

matches exactly one step in the previous machine. At the top and bottom levels, multiple steps in the intermediate machines may correspond to a single step in a CBPV term or a CFG machine. The relation is something more precise than stuttering simulation, however; by examining the state involved, we can compute precisely the number of intermediate-machine steps that will be required to make up a single CBPV/CFG step.

More specifically, we define a function unload that translates CFG states into CBPV terms, defined as the composition of simpler unload functions between each of the layers. We then define a function depth such that each CFG machine step of a state s corresponds to exactly $\mathsf{depth}(s)$ steps of the last intermediate machine, and $\mathsf{depth}(s)$ steps of the first intermediate machine (the CEK machine of the previous section) correspond in turn to one step of $\mathsf{unload}(s)$. (In fact, the function depth is precisely the redex depth of the term being processed, which we have already seen gives us the number of steps of the CEK machine that make up one step of the SOS.) Since the steps of each intermediate machine are in lockstep correspondence, we can conclude that each step of the CFG machine corresponds exactly to a step of the corresponding CBPV term.

This conclusion still leaves open the possibility that not every CBPV term corresponds to a CFG state. Fortunately, it is easy to construct a state corresponding to any given CBPV term. We can define a function load at each level that translates a CBPV term into a machine state. For instance, we can load a CBPV term M into the CEK machine state $\langle M,\ \cdot,\ \cdot\rangle$. At every level, it is the case that $\mathsf{unload}(\mathsf{load}(M)) = M$, guaranteeing that every CBPV term has at least one corresponding CFG state.

[4] We elide the formal definition of unload for the CEK machine, and instead present the similar but more complicated function for the PEAK machine in Figure 8.

We also define a notion of well-formedness at each level. The machines may contain states that do not correspond to CBPV terms; the well-formedness property rules out these states, by translating the scoping constraints of terms into constraints on machine states. We show that initial states produced by loading are well-formed, well-formedness is preserved by steps of the machine, and well-formed states unload into terms. Together, these properties suffice to guarantee that starting from a CBPV term, we only encounter machine states that correspond to CBPV terms.

4.1 Machine equivalence

The net result is stated in Theorem 2.

Theorem 2. *(CBPV/CFG Machine Equivalence) For all CBPV terms P and CFG states s that are well-formed with respect to P,*

$$\mathsf{CBPV.step}\,(\mathsf{unload}\;P\;s) = \mathsf{unload}\;P\;(\mathsf{CFG.step}\,(\mathsf{compile}_1 P)\;s)$$

Proof. By combining the per-machine correctness lemmas, as shown in Figure 5. □

This theorem gives strong reasoning properties about the connection between CBPV terms and their corresponding machine semantics. As one illustration of its use, consider a definition of observational equivalence on closed CBPV terms M_1 and M_2 such that

$$M_1 \equiv M_2 \qquad \textit{iff} \qquad (M_1 \to^* \mathsf{prd}\,0 \wedge M_2 \to^* \mathsf{prd}\,0)$$

We can define what it means for two CFG programs to be equivalent by

$$cfg_1 \equiv' cfg_2 \quad \textit{iff} \begin{cases} cfg_1 \vdash s_0 \to^* s_1 \not\to \\ cfg_2 \vdash s_0 \to^* s_2 \not\to \\ \text{and } \mathsf{ret}(s_1) = 0 = \mathsf{ret}(s_2) \end{cases}$$

Here, the function $\mathsf{ret}(s)$ examines a CFG machine state to determine whether it has terminated at a `RET` instruction whose operand evaluates to 0 in the state s. s_0 denotes the initial CFG state. An immediate corollary of Theorem 2 is:

$$M_1 \equiv M_2 \implies (\mathsf{compile}_1 M_1 \equiv' \mathsf{compile}_1 M_2)$$

Note that if $M_1 \to M_1'$ then $M_1 \equiv M_1'$, so we immediately conclude that $\mathsf{compile}_1 M_1 \equiv' \mathsf{compile}_1 M_1'$. Many other optimizations also fall into this style of reasoning.

Importantly, the machine equivalence theorem holds even for *open* CBPV terms. We conjecture that this means that reasoning similar to the above can be generalized to contextual equivalences as well.

5. The PEAK Machine

The CEK machine gives a straightforward computation model for CBPV, but it falls short of the execution model of our desired CFG formalism in two important ways: 1. all control flow merge points require allocating a closure, then restoring the environment and 2. function arguments are pushed onto the stack individually, whereas our CFG language contains a multi-parameter jump-with-argument instruction (`TAIL`).

While the second point might seem to be a matter of the particular instruction set chosen, the transformation we present addresses both issues. To illustrate the first problem, consider a simple program that uses a conditional in the left position of a sequence to express a conditional assignment:

$$\mathsf{if0}\;U\;(\mathsf{prd}\;V_0)\;(\mathsf{prd}\;V_1)\;\mathsf{to}\;x\;\mathsf{in}\;M$$

Executed using the CEK machine, this program will first push the frame $[\_\;\mathsf{to}\;x\;\mathsf{in}\;M, \cdot]$ onto the stack, saving the environment, only to restore it after reducing the conditional. Using a CFG machine, we might write the same program as the following:

```
0:  IF0 U [1; 2]
1:  TAIL @3 1        // produce V0
2:  TAIL @3 2        // produce V1
3:  x = POP ...
```

which executes the branch, places an argument on the stack, pops it into `x` and continues. A CBPV program that executes roughly the same way on the CEK machine could be written as

$$\mathsf{if0}\;U\;(V_0 \cdot \lambda x.M)\;(V_1 \cdot \lambda x.M)$$

but this requires duplicating the term M. An attempt to avoid duplicating M by using a joint point function

$$\mathsf{prd}\,(\mathsf{thunk}\,(\lambda x.M))\;\mathsf{to}\;y\;\mathsf{in}\;\mathsf{if0}\;U\;(V_0 \cdot \mathsf{force}\;y)\;(V_1 \cdot \mathsf{force}\;y)$$

reintroduces the original problem: an environment is saved when the thunk is evaluated, only to be restored immediately after executing the conditional. An analogous situation occurs in programs that use rec simply as a label.

The principle underlying the PEAK machine's execution model is that the CEK machine does unnecessary work by saving an environment immediately when an expression is evaluated. By delaying closure allocation until the environment is actually changed, we avoid some paired save/restore operations. Furthermore, using an alternative representation of environments allows the machine to continue to add bindings to the environment as as long as the result of substitution does not change. As a result, the PEAK machine can express executions that contain control flow join points, like the CFG program above, without the need to save and restore an environment.

Figure 6 shows the syntax for the PEAK abstract machine, as well as some helper functions described below. Unlike the CEK machine, the PEAK machine works with program paths rather than directly with CBPV terms. It also uses an extra "argument stack" a to delay the creation of closures. A PEAK state ρ is thus a quadruple $\langle p,\ e,\ a,\ k\rangle$, where e is the environment, which maps paths to values, and k is the stack, whose frames mirror the CEK frames.

$$\begin{array}{rrcl}
\text{State} & \ni \rho & = & \text{Path} \times \text{Env} \times \text{Args} \times \text{Kont} \\
\text{Path} & \ni p, q & ::= & \cdot \mid n :: p \\
\text{Env} & \ni e & = & \text{Path} \longrightarrow_{\text{fin}} \text{Val} \\
\text{Args} & \ni a & ::= & \cdot \mid \texttt{ARG}\;p :: a \mid \texttt{SEQ}\;p :: a \mid \texttt{PRJ}\;n :: a \\
\text{Kont} & \ni k & ::= & \cdot \mid v \cdot _ :: k \mid [p, e, a] :: k \mid \mathsf{prj}_n\;_ :: k \\
\text{Val} & \ni v & ::= & x \mid n \mid [p, e]
\end{array}$$

$$\begin{array}{rcll}
\gamma\,p\,e & = & x & \text{when } P[p] = x,\ x \text{ free} \\
\gamma\,p\,e & = & e(p') & \text{when } P[p] = x, \\
 & & & \quad x \text{ bound by } P[p'] = \lambda x._ \text{ or} \\
 & & & \quad \text{by } P[p'] = _\;\mathsf{to}\;x\;\mathsf{in}\;_ \\
\gamma\,p\,e & = & [0 :: p', e] & \text{when } P[p] = x,\ x \text{ bound} \\
 & & & \quad \text{by } P[p'] = \mathsf{rec}\;x._ \\
\gamma\,p\,e & = & [0 :: p, e] & \text{when } P[p] = \mathsf{rec}\;x._ \\
\gamma\,p\,e & = & n & \text{when } P[p] = n
\end{array}$$

$$\begin{array}{rcl}
\delta\;e\;\cdot & = & \cdot \\
\delta\;e\;(\texttt{ARG}\;p :: a) & = & (\gamma\,(0 :: p)\,e \cdot _) :: \delta\;e\;a \\
\delta\;e\;(\texttt{SEQ}\;p :: a) & = & [p, e, a] :: \cdot \\
\delta\;e\;(\texttt{PRJ}\;i :: a) & = & \mathsf{prj}_i\;_ :: \delta\;e\;a
\end{array}$$

Figure 6. PEAK syntax and semantic functions.

start state	$\longrightarrow$	next state	when $P[p] =$	and when:
$\langle p,\ e,\ a,\ k\rangle$	$\longrightarrow$	$\langle 0::p',\ e',\ \cdot,\ \delta\ e\ a +\!\!+ k\rangle$	$\mathsf{force}\ {}_{-0}$	$\gamma\,(0::p)\,e = [p', e']$
$\langle p,\ e,\ a,\ k\rangle$	$\longrightarrow$	$\langle 1::p',\ e,\ a,\ k\rangle$	$\mathsf{if0}\ {}_{-0}\ {}_{-1}\ {}_{-2}$	$\gamma\,(0::p)\,e = 0$
$\langle p,\ e,\ a,\ k\rangle$	$\longrightarrow$	$\langle 2::p',\ e,\ a,\ k\rangle$	$\mathsf{if0}\ {}_{-0}\ {}_{-1}\ {}_{-2}$	$\gamma\,(0::p)\,e = n, n \neq 0$
$\langle p,\ e,\ \mathtt{SEQ}\ p'::a,\ k\rangle$	$\longrightarrow$	$\langle 1::p',\ e[p'\mapsto v],\ a,\ k\rangle$	$\mathsf{prd}\ {}_{-0}$	$\gamma\,(0::p)\,e = v$
$\langle p,\ e,\ \cdot,\ [p', e', a']::k\rangle$	$\longrightarrow$	$\langle 1::p',\ e'[p'\mapsto v],\ a',\ k\rangle$	$\mathsf{prd}\ {}_{-0}$	$\gamma\,(0::p)\,e = v$
$\langle p,\ e,\ \mathtt{SEQ}\ p'::a,\ k\rangle$	$\longrightarrow$	$\langle 1::p',\ e[p'\mapsto v_0[\![\oplus]\!]v_1],\ a,\ k\rangle$	${}_{-0}\oplus{}_{-1}$	$\gamma\,(0::p)\,e = v_0$ and $\gamma\,(1::p)\,e = v_1$
$\langle p,\ e,\ \cdot,\ [p', e', a']::k\rangle$	$\longrightarrow$	$\langle 1::p',\ e'[p'\mapsto v_0[\![\oplus]\!]v_1],\ a',\ k\rangle$	${}_{-0}\oplus{}_{-1}$	$\gamma\,(0::p)\,e = v_0$ and $\gamma\,(1::p)\,e = v_1$
$\langle p,\ e,\ \mathtt{ARG}\ q::a,\ k\rangle$	$\longrightarrow$	$\langle 0::p,\ e[p\mapsto v],\ a,\ k\rangle$	$\lambda x.{}_{-0}$	$\gamma\,(0::q)\,e = v$
$\langle p,\ e,\ \cdot,\ v\cdot\_::k\rangle$	$\longrightarrow$	$\langle 0::p,\ e[p\mapsto v],\ \cdot,\ k\rangle$	$\lambda x.{}_{-0}$	
$\langle p,\ e,\ \mathtt{PRJ}\ i::a,\ k\rangle$	$\longrightarrow$	$\langle i::p,\ e,\ a,\ k\rangle$	$\langle {}_{-0}, \cdots, {}_{-n}\rangle$	$i \in \{0, \ldots, n\}$
$\langle p,\ \cdot,\ \cdot,\ \mathsf{prj}_i\ \_::k\rangle$	$\longrightarrow$	$\langle i::p,\ e,\ \cdot,\ k\rangle$	$\langle {}_{-0}, \cdots, {}_{-n}\rangle$	$i \in \{0, \ldots, n\}$
$\langle p,\ e,\ a,\ k\rangle$	$\longrightarrow$	$\langle 0::p,\ e,\ \mathtt{SEQ}\ p::a,\ k\rangle$	${}_{-0}\ \mathsf{to}\ x\ \mathsf{in}\ {}_{-1}$	
$\langle p,\ e,\ a,\ k\rangle$	$\longrightarrow$	$\langle 1::p,\ e,\ \mathtt{ARG}\ p::a,\ k\rangle$	${}_{-0}\cdot{}_{-1}$	
$\langle p,\ e,\ a,\ k\rangle$	$\longrightarrow$	$\langle 0::p,\ e,\ \mathtt{PRJ}\ i::a,\ k\rangle$	$\mathsf{prj}_i\ {}_{-0}$	

Figure 7. PEAK semantics for a program P

Path Environments. Earlier, we remarked that CEK machine closures represent terms with delayed substitutions. Unloading carried out this substitution explicitly to recover exactly the residual of reduction using the SOS rules. It should therefore be possible to add bindings to an environment in a closure without changing the execution behavior or affecting the relationship with SOS states, as long as they do not bind additional free variables of the term. The main difficulty in sharing environments to attain an optimized evaluation strategy is that the desired property of substitutions is sensitive to the exact representation of environments and binders.

The two most popular representations of binders are names and DeBruijn indices. Neither of these is particularly well suited to sharing environments. In a DeBruijn representation, bindings are added to the front of the list as execution proceeds under binders, changing the interpretation of each index. Using a named representation of binders runs into problems with capturing variable names; one could require that all bound names be unique and distinct from the free variables of the source term, but our intended target language already contains an alternative solution.

We borrow an idea from SSA and use a representation of environments that associates dynamic values with *paths* into the source program. This turns out to have a number of advantages: 1. it imposes no conditions on the representation of binders in source terms, 2. we automatically get a unique name for each occurrence of a binder, which facilitates sharing environments, and 3. it results in a definition of machine states nearly identical to that of our desired CFG machine.

A PEAK machine *path* is simply a sequence of natural numbers interpreted relative to a CBPV term P.[5] The empty path $\cdot$ denotes the entire term P, whereas a path of the form $n :: p$ denotes the n^{th} subterm (ordered from left to right) of the term denoted by p, indexed from 0. For example, for $P = \mathsf{if0}\ V\ \langle M_1, M_2\rangle\ M_3$ the path $2::\cdot$ denotes the subterm M_3. We write $P[p]$ for the subterm of P at path p, and so for the term above we also have $P[0::\cdot] = V$, $P[0::1::\cdot] = M_1$, and $P[1::1::\cdot] = M_2$. (Clearly $P[-]$ is a partial function.)

PEAK operational semantics. Figure 7 shows the operational semantics for the PEAK abstract machine. The first component p of a PEAK state $\langle p,\ e,\ a,\ k\rangle$ is a path that tracks the location of the currently executing subterm, relative to a CBPV term P. Since P is constant for the duration of the execution, the step function of the PEAK machine is parameterized by it. The column labeled "when $P[p]$ = " shows the part of the subterm that needs to be consulted to determine which evaluation rule to apply. The indices on the _ "wildcard" pattern mark the numbers used to construct the paths. For instance, if $P[p] = \mathsf{if0}\ {}_{-0}\ {}_{-1}\ {}_{-2}$, then the subterm of the guard is $0::p$.

Sharing environments requires that the machine recognize when it is safe to delay creating a closure. PEAK machine configurations contain an additional "argument stack" component that logically contains a prefix of the continuation stack for which the current environment is a valid substitution. The syntax of argument frames is presented in Fig. 7. In the argument stack, `ARG`, `SEQ` and `PRJ`, correspond to application, sequencing, and projection continuation frames, respectively.

The Argument Stack The PEAK machine rules for sequencing, application, and projection push frames onto the argument stack, batching them up. The produce, lambda, and tuple rules each have two cases. When the argument stack is empty, the machine executes roughly as the CEK machine. The rule for force now evaluates the argument stack up until the nearest `SEQ` and pushes the resulting frames on the continuation stack—this operation, written as δ, is shown in Figure 6. The remaining argument frames are also saved in the sequence frame, and are restored when the sequence frame is popped by prd.

A similar strategy is used to delay allocating closures when execution passes under a rec. The PEAK version of the γ function is adapted from the CEK's γ function to work on paths rather than directly on values. Its definition (shown in Figure 6) inspects the binding location for a variable x to determine whether it should be found in the environment or treated as a pointer to a rec that should be put in a closure using the current environment. The construction of a closure is therefore delayed until the variable bound by a rec appears under a prd or is applied. If a variable bound by a rec is forced, the closure returned by γ just passes through the current environment.

Unloading PEAK Machine States to CEK. Recovering CEK states from PEAK states involves converting dynamic values, argument frames, and continuation frames. The PEAK.unload definition is given in Figure 8. The only interesting case for dynamic values is that of closures. Recovering a term from a path is a simple lookup in the source program, while unloading the PEAK environment proceeds by recursion on the path component of the closure. Every time a path corresponding to a binder is encountered, the associated dynamic value is unloaded and added to the CEK environment. The rec binder is a special case: since the PEAK machine does not allocate a closure when the binding is created, we cannot

[5] Here we use the metavariable P rather than M to range over CBPV terms to emphasize the fact that this term is being treated by a PEAK machine.

$$
\begin{array}{lcll}
\mathsf{unload}_e & : & \mathrm{Path} \to \mathrm{Env} \to \mathrm{Env}_{\mathrm{CEK}} & \\
\mathsf{unload}_e(\cdot,\ e) & = & \cdot & \\
\mathsf{unload}_e(n :: p,\ e) & = & (\mathsf{unload}_e(p,\ e))[x \mapsto \mathsf{unload}_v(v)] & \text{when } P[p] = \lambda x.\_,\ n = 0, \text{ and } e(p) = v \\
 & = & (\mathsf{unload}_e(p,\ e))[x \mapsto \mathsf{unload}_v(v)] & \text{when } P[p] = \_\ \mathsf{to}\ x\ \mathsf{in}\ \_,\ n = 1, \text{ and } e(p) = v \\
 & = & (\mathsf{unload}_e(p,\ e))[x \mapsto [\,\mathsf{rec}\,x.P[0 :: p], \mathsf{unload}_e(p,\ e)]] & \text{when } P[p] = \mathsf{rec}\ x.\_, \text{ and } n = 0 \\
 & = & \mathsf{unload}_e(p,\ e) & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{unload}_v & : & \mathrm{Val} \to \mathrm{Val}_{\mathrm{CEK}} \\
\mathsf{unload}_v(x) & = & x \\
\mathsf{unload}_v(n) & = & n \\
\mathsf{unload}_v([p, e]) & = & [\,\mathsf{rec}\,x.P[p], \mathsf{unload}_e(\mathsf{tail}(p),\ e)]
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{unload}_k & : & \mathrm{Env} \to \mathrm{Args} \to \mathrm{Kont} \to \mathrm{Kont}_{\mathrm{CEK}} \\
\mathsf{unload}_k(e,\ \cdot,\ \cdot) & = & \cdot \\
\mathsf{unload}_k(e,\ \texttt{ARG}\ p :: a,\ k) & = & \mathsf{unload}_v(\gamma\ p\ e) \cdot \_ :: \mathsf{unload}_k(e,\ a,\ k) \\
\mathsf{unload}_k(e,\ \texttt{SEQ}\ p :: a,\ k) & = & [\_\ \mathsf{to}\ x\ \mathsf{in}\ P[1 :: p], \mathsf{unload}_e(p,\ e)] :: \mathsf{unload}_k(e,\ a,\ k) \\
\mathsf{unload}_k(e,\ \texttt{PRJ}\ n :: a,\ k) & = & \mathsf{prj}_n\ \_ :: \mathsf{unload}_k(e,\ a,\ k) \\
\mathsf{unload}_k(e,\ \cdot,\ (v \cdot \_) :: k) & = & \mathsf{unload}_v(v) \cdot \_ :: \mathsf{unload}_k(e,\ \cdot,\ k) \\
\mathsf{unload}_k(e,\ \cdot,\ [p, e', a'] :: k) & = & [\_\ \mathsf{to}\ x\ \mathsf{in}\ P[1 :: p], \mathsf{unload}_e(p,\ e')] :: \mathsf{unload}_k(e',\ a',\ k) \\
\mathsf{unload}_k(e,\ \cdot,\ \mathsf{prj}_n\ \_ :: k) & = & \mathsf{prj}_n\ \_ :: \mathsf{unload}_k(e,\ \cdot,\ k)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{unload} & : & \mathrm{State}_{\mathrm{PEAK}} \to \mathrm{State}_{\mathrm{CEK}} \\
\mathsf{unload}(\langle p,\ e,\ a,\ k\rangle) & = & \langle P[p],\ \mathsf{unload}_e(p,\ e),\ \mathsf{unload}_k(e,\ a,\ k)\rangle
\end{array}
$$

Figure 8. PEAK unloading for a program P.

simply recursively unload the associated dynamic value. Instead, the CEK closure is recovered by combining the path of the binder with the result of unloading the rest of the environment at that path. This relies on the invariant that the CEK machine simply copies the current environment when execution passes under a rec.

To produce a CEK continuation stack from a PEAK machine state, the continuation stack is interleaved with argument frames. Starting with the current argument stack, each argument frame is converted to a CEK frame using the current environment. Then, the PEAK continuation stack is traversed, converting application and projection frames as necessary. When a sequence frame is encountered, the process is iterated using the saved environment and argument stack, and the remainder of the continuation stack.

Well-Formed States. The representation of environments used by the PEAK machine requires some extra well-formedness constraints on states in order to guarantee that free variables will not be captured.

Definition 3. *A PEAK state $\langle p,\ e,\ a,\ k\rangle$ is well formed iff*

1. *In each closure as well as the path and environment of the machine state, every suffix of the path that corresponds to a binder has an associated dynamic value in the environment*
2. *Every path occurring in an argument frame is a suffix of the path of the preceding* SEQ *frame, or the current path of the machine state or containing continuation frame*

The first condition is a scoping property that rules out states in which bound variables of the source term do not have associated dynamic values in the substitution. The second condition is necessary to maintain this invariant when argument frames are pushed onto and restored from the continuation stack.

As described in Section 4, we must prove that CBPV terms load to well-formed states, and that well-formedness is preserved by machine steps. Loading is simply the constant function that produces the initial state $\langle\cdot,\ \cdot,\ \cdot,\ \cdot\rangle$, so the first proof is simple; the second follows from the definition of the step function.

Correctness with Respect to the CEK Machine The PEAK step and unloading functions, as well as the state well-formedness predicate, are now defined with respect to an initial CBPV term. The statement of correctness is adjusted from that of the CEK machine accordingly.

Lemma 4. *(PEAK Correctness) For all terms P and PEAK states ρ that are well-formed with respect to P,*

$$\mathsf{CEK.step}\ (\mathsf{PEAK.unload}\ P\ \rho) = \mathsf{PEAK.unload}\ P\ (\mathsf{PEAK.step}\ P\ \rho)$$

6. CFG Machine

The PEAK machine is not intended to be used directly as a model of execution for CBPV terms. Instead, we use it as an intermediate step to derive a virtual machine, in Danvy's terminology (Danvy 2003). The abstract machine is factored into a compiler and interpreter for an instruction set that reflects the cases of the step function. The compiler pre-computes various aspects of execution that can be determined statically, so that the interpreter can be simpler and closer to real assembly languages.

Partially evaluating PEAK steps We observe that the argument stack of the PEAK machine is unique for a program P and path p, and can be computed statically as shown in Figure 12. This function traverses the source program from p up to the nearest enclosing rec or the root of the term, tracing the control flow of the current function call.

At the beginning of execution and whenever control passes under a rec, the argument stack is known to be empty. Every step except popping a sequence frame either has no effect or manipulates argument frames by examining P and the current path only, never using information from the environment or continuation. When prd pops a sequence frame and proceeds to path $1 :: p'$, the restored argument stack must be the one associated with p', captured by a force executed in the left-hand side of the associated sequence operation.

When examining values, all but one case of the γ function does not examine the environment. To do as much work as possible statically in the compiler, we split the PEAK machine's γ into two functions, $\overline{\gamma}$ and eval. The former takes a path to a syntactic value and computes an "operand", which may be a free variable $\texttt{VAR}\,x$,

$$
\begin{array}{rcl}
\text{State} \ni s & = & \text{Path} \times \text{Env} \times \text{Kont} \\
\text{CFG} \ni \mathit{cfg} & = & \text{Path} \to \text{Ins} \\
\text{Path} \ni p, q & ::= & \cdot \mid n :: p \\
\text{Env} \ni e & = & \text{Path} \longrightarrow_{\text{fin}} \text{Val} \\
\text{Kont} \ni k & ::= & \cdot \mid v \cdot \_ :: k \mid [p, e, a] :: k \mid \mathsf{prj}_n \ \_ :: k \\
\text{Val} \ni v & ::= & x \mid n \mid [p, e] \\
\text{Operand} \ni o & ::= & \texttt{VAR}\, x \mid \texttt{NAT}\, n \mid \texttt{LOC}\, p \mid \texttt{LBL}\, p \\
\text{Arg} \ni a & ::= & \texttt{ARG}\, o \mid \texttt{PRJ}\, n
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{eval}\, e\, \texttt{VAR}\, x & = & x \\
\mathsf{eval}\, e\, \texttt{NAT}\, n & = & n \\
\mathsf{eval}\, e\, \texttt{LOC}\, p & = & e(p) \\
\mathsf{eval}\, e\, \texttt{LBL}\, p & = & [p, e]
\end{array}
$$

$$
\begin{array}{rcll}
\overline{\gamma}\, p & = & \texttt{VAR}\, x & \text{when } P[p] = x,\ x \text{ free} \\
\overline{\gamma}\, p & = & \texttt{LOC}\, p' & \text{when } P[p] = x,\ x \text{ bound} \\
 & & & \quad \text{by } P[p'] = \lambda x.\_ \\
 & & & \quad \text{or } P[p'] = \_\ \mathsf{to}\ x\ \mathsf{in}\ \_ \\
\overline{\gamma}\, p & = & \texttt{LBL}\, 0 :: p' & \text{when } P[p] = x,\ x \text{ bound} \\
 & & & \quad \text{by } P[p'] = \mathsf{rec}\ x.\_ \\
\overline{\gamma}\, p & = & \texttt{LBL}\, 0 :: p & \text{when } P[p] = \mathsf{rec}\ x.\_ \\
\overline{\gamma}\, p & = & \texttt{NAT}\, n & \text{when } P[p] = n
\end{array}
$$

$$
\begin{array}{rcl}
\overline{\delta}\ e\ (\texttt{ARG}\, o) & = & \mathsf{eval}\, e\, o \cdot \_ \\
\overline{\delta}\ e\ (\texttt{PRJ}\, i) & = & \mathsf{prj}_i\ \_
\end{array}
$$

Figure 9. CFG machine states and semantic functions.

literal number $\texttt{NAT}\, n$, a bound local variable $\texttt{LOC}\, p$ represented by the path to its binder, or a label $\texttt{LBL}\, p$ representing a position in the source program. Note that we do not have to examine the environment to resolve variables bound by rec. The eval function only has to look up bound variables and attach an environment to labels in order produce dynamic values from operands. It is easy to verify that $\gamma\, p\, e = \mathsf{eval}\, e\, \overline{\gamma}\, p$.

The CFG virtual machine. The target machine operates on a simple instruction set, whose operational semantics are shown in Figure 10. At the CFG level, each program path $P[p]$ now corresponds to an instruction that communicates information about the source term and argument stack at p to the virtual machine's transition function. The instruction set contains 10 instructions:

- NOP, which does nothing
- CALL and TAIL, which perform regular and tail function calls (tail calls do not add return frames to the stack)
- PMOV, which does a parallel move of existing values to variables
- OP, which performs an arithmetic operation and stores the result in a variable
- RET and ORET, which pop return frames and return a direct value and a computed value respectively
- POP, which pops arguments pushed by CALL
- IFO, which performs a conditional branch
- SWI , which chooses the next label from a list

Its states are simply PEAK machine states minus the argument stack (shown in Figure 9). The step function also corresponds closely to that of the PEAK machine, where the case analysis on the argument stack is instead handled by specialized instructions. Some transitions that were distinguished in the PEAK machine are represented by a single instruction: for example, both lambdas and prd are represented by PMOV instructions in cases where the argument frame is known. The step for force, on the other hand, has been split into tail and non-tail calls. We will never emit PMOV instructions of more than one argument, but include the more general form which we consider when specializing certain instructions in Section 7.

Note that SEQ frames now contain two paths: the path bound to the returned value in the environment, and the path of the next instruction to execute. Though these will always be p and $1 :: p$, respectively, this change allows us to treat paths abstractly. The CFG machine never inspects the structure of paths or creates new paths other than the ones already present in instructions. Paths are only compared for equality by the environment and instruction lookup functions, and thus could be replaced with abstract identifiers.

Producing control flow graphs. The output of our compiler is a mapping from paths to instructions. Instructions can therefore reference positions in the CFG directly via "labels", a common feature of real world virtual machines. From

$$\mathsf{PEAK.step} : \text{Term} \to \mathsf{PEAK.state} \to \mathsf{PEAK.state}$$

we obtain

$$\mathsf{compile_0} : \text{Term} \to \text{Path} \to \text{Ins}$$
$$\mathsf{CFG.step} : (\text{Path} \to \text{Ins}) \to \mathsf{CFG.state} \to \mathsf{CFG.state}$$

For each subterm, the compiler records as much information about its execution as can be computed statically. For the control constructs, sequencing, application, and projection, this is just the location of the next instruction to execute, so they each compile to a NOP. The case of a tuple when the head of the argument stack contains a projection is identical, but when the argument stack is empty (and so we cannot statically compute which element will be chosen), we record the path of each element of the tuple. If the head of the argument stack is of any other form, we emit an explicit error instruction. For abstractions and produce when the corresponding argument frame is present, the path to bind in the environment, the next instruction, and operand computed from the value being bound are recorded. Otherwise, for a lambda we can only record the next instruction, and for a produce, the computed operand. Arithmetic operators follow a similar pattern. For the force instruction, we record the argument stack up to the nearest SEQ frame. If one is found, we record the the path to bind and the path to the next instruction.

Unloading, well-formedness of states, and correctness. Unloading CFG machine states to PEAK states is straightforward: the representation of dynamic values and environments is identical, and the machines traverse terms in lockstep. Application and projection frames in the continuation stack are also unchanged. For sequencing frames, we only have to throw away the extra path to the next instruction, and recover the stored argument stack. The argument stack in a sequence frame $[p, e, a]$ of the PEAK machine is always the one grabbed at path p, so we can simply reuse the frames function used in the compiler. Similarly, we can recover the argument stack of the top-level PEAK machine state. The well-formedness condition for CFG machine states is also simplified. We require only that paths point to values or computations as necesssary, and environments in the machine state, closures, and the continuations stack contain values for all binders in scope of the associated path. Loading, again, always produces the initial state $\langle \cdot,\ \cdot,\ \cdot \rangle$.

Lemma 5. *(CFG Machine Correctness) For all terms P and CFG states s that are well-formed with respect to P,*

$$\mathsf{PEAK.step}\ P\ (\mathsf{CFG.unload_0}\, P\, s) = \mathsf{CFG.unload_0}\, P\, (\mathsf{CFG.step}\ (\mathsf{compile_0}\, P)\ s)$$

start state $\longrightarrow$	next state	when $cfg[p] =$	and when:
$\langle p,\ e,\ k\rangle$	$\longrightarrow\ \langle p',\ e,\ k\rangle$	$\texttt{NOP}\ p'$	
$\langle p,\ e,\ k\rangle$	$\longrightarrow\ \langle p',\ e',\ \overline{\delta}\ e\ a_1 :: .. :: \overline{\delta}\ e\ a_n :: [q, p'', e] :: k\rangle$	$\texttt{CALL}\ o\ a_1\ ..\ a_n\ q\ p''$	$\mathsf{eval}\ e\ o = [p', e']$
$\langle p,\ e,\ k\rangle$	$\longrightarrow\ \langle p',\ e',\ \overline{\delta}\ e\ a_1 :: .. :: \overline{\delta}\ e\ a_n :: k\rangle$	$\texttt{TAIL}\ o\ a_1\ ..\ a_n$	$\mathsf{eval}\ e\ o = [p', e']$
$\langle p,\ e,\ k\rangle$	$\longrightarrow\ \langle p',\ e[\overline{p_i \mapsto v_i}^{\,i\in 1..n}],\ k\rangle$	$\texttt{PMOV}\ \overline{(p_i, o_i)}^{\,i\in 1..n}\ p'$	$\mathsf{eval}\ e\ o_i = v_i$
$\langle p,\ e,\ k\rangle$	$\longrightarrow\ \langle p',\ e[p_1 \mapsto v_1 [\![\oplus]\!] v_2],\ k\rangle$	$\texttt{OP}\ o_1 \oplus o_2\ p_1\ p'$	$\mathsf{eval}\ e\ o_1 = v_1$ and $\mathsf{eval}\ e\ o_2 = v_2$
$\langle p,\ e,\ [q, p', e'] :: k\rangle$	$\longrightarrow\ \langle p',\ e'[q \mapsto v_1 [\![\oplus]\!] v_2],\ k\rangle$	$\texttt{OPRET}\ o_1 \oplus o_2$	$\mathsf{eval}\ e\ o_1 = v_1$ and $\mathsf{eval}\ e\ o_2 = v_2$
$\langle p,\ e,\ [q, p', e'] :: k\rangle$	$\longrightarrow\ \langle p',\ e'[q \mapsto v],\ k\rangle$	$\texttt{RET}\ o$	$\mathsf{eval}\ e\ o = v$
$\langle p,\ e,\ v \cdot \_ :: k\rangle$	$\longrightarrow\ \langle p',\ e[p \mapsto v],\ k\rangle$	$\texttt{POP}\ p'$	
$\langle p,\ e,\ k\rangle$	$\longrightarrow\ \langle p_0,\ e,\ k\rangle$	$\texttt{IF0}\ o\ p_0\ p_1$	$\mathsf{eval}\ e\ o = 0$
$\langle p,\ e,\ k\rangle$	$\longrightarrow\ \langle p_1,\ e,\ k\rangle$	$\texttt{IF0}\ o\ p_0\ p_1$	$\mathsf{eval}\ e\ o = n,\ n \neq 0$
$\langle p,\ e,\ \mathsf{prj}_i\ \_ :: k\rangle$	$\longrightarrow\ \langle p_i,\ e,\ k\rangle$	$\texttt{SWI}\ \overline{p_j}^{\,j\in 1..n}$	

Figure 10. $\mathsf{CFG.step}\ cfg\ \langle p,\ e,\ k\rangle$) semantics.

$\mathsf{compile}_0 p =$	when $P[p] =$	and when:
$\texttt{TAIL}\ (\overline{\gamma}\ (0 :: p))\ args$	$\mathsf{force}\ \_0$	$\mathsf{args}\ p = (args, \mathsf{none})$
$\texttt{CALL}\ (\overline{\gamma}\ (0 :: p))\ args\ q\ (1 :: q)$	$\mathsf{force}\ \_0$	$\mathsf{args}\ p = (args, \mathsf{some}\ q)$
$\texttt{IF0}\ (\overline{\gamma}\ (0 :: p))\ (1 :: p)\ (2 :: p)$	$\mathsf{if0}\ \_0\ \_1\ \_2$	
$\texttt{PMOV}\ (q, \overline{\gamma}\ (0 :: p))\ (1 :: q)$	$\mathsf{prd}\ \_0$	$\mathsf{frames}\ p = \texttt{SEQ}\ q :: a$
$\texttt{RET}\ (\overline{\gamma}\ (0 :: p))$	$\mathsf{prd}\ \_0$	$\mathsf{frames}\ p = \cdot$
$\texttt{PMOV}\ (p, \overline{\gamma}\ (0 :: p'))\ (0 :: p)$	$\lambda x.\_0$	$\mathsf{frames}\ p = \texttt{ARG}\ p' :: a$
$\texttt{POP}\ (0 :: p)$	$\lambda x.\_0$	$\mathsf{frames}\ p = \cdot$
$\texttt{OP}\ (\overline{\gamma}\ (0 :: p)) \oplus (\overline{\gamma}\ 1 :: p)\ q\ (1 :: q)$	$\_0 \oplus \_1$	$\mathsf{frames}\ p = \texttt{SEQ}\ q :: a$
$\texttt{OPRET}\ (\overline{\gamma}\ (0 :: p)) \oplus (\overline{\gamma}\ 1 :: p)$	$\_0 \oplus \_1$	$\mathsf{frames}\ p = \cdot$
$\texttt{NOP}\ (i :: p)$	$\langle \_0, .., \_n \rangle$	$\mathsf{frames}\ p = \texttt{PRJ}\ i :: a$ and $i \in \{1, ..., n\}$
$\texttt{SWI}\ \overline{i :: p}^{\,i\in 1..n}$	$\langle \_0, .., \_n \rangle$	$\mathsf{frames}\ p = \cdot$
$\texttt{NOP}\ (0 :: p)$	$\_0\ \mathsf{to}\ x\ \mathsf{in}\ \_1$	
$\texttt{NOP}\ (1 :: p)$	$\_0 \cdot \_1$	
$\texttt{NOP}\ (0 :: p)$	$\mathsf{prj}_n\ \_0$	

```
frames (n :: p) =
  match n , P[n :: p] with
  | 1 , _0 · _1 ⇒ ARG p :: frames p
  | 0 , λx._0 ⇒ match frames p with
                | ARG q :: a ⇒ a
                | a ⇒ a
                end
  | 0 , _0 to x in _1 ⇒ SEQ p :: frames p
  | 1 , _0 to x in _1 | 1 , if0 _0 _1 _2
  | 2 , if0 _0 _1 _2 ⇒ frames p
  | 0 , prj_i _0 ⇒ PRJ i :: frames p
  | 0 , ⟨_0, .., _n⟩ ⇒ match frames p with
                       | PRJ i :: a ⇒ a
                       | a ⇒ a
                       end
  | _ ⇒ ·
  end
```

Figure 11. Lock-step compilation strategy for the term P.

6.1 Modified compilation strategy.

The steps of the CFG machine we have derived correspond exactly to those of the CEK machine. However, recall from Section 4 that steps of the CEK machine do not correspond exactly to steps of CBPV terms. In particular, the search rules in the SOS for CBPV are applied recursively, while each application requires a step in the CEK machine. These correspond to `NOP` instructions in the compiled program. Fortunately, we can eliminate these steps without needing a new virtual machine: we only need a new compilation strategy. This improved compilation strategy will never emit instructions that jump to structural paths, i.e., those that point to sequences, applications, and projections. Instead, the instructions emitted will always skip to the location of the next non-trivial instruction. We denote the improved compiler as $\mathsf{compile}_1$.

Unloading, well-formedness of states, and correctness for the improved compilation strategy. We have now seen how to translate from CBPV terms to CFG programs through a series of machines, each one of which comes closer to an assembly-language-like execution model. We define the top-level unload function as the composition of each of the unloading functions, translating a CFG state under the improved compilation strategy to a CBPV term. At each level, we have proved a lemma stating that the transformation preserves the semantics of machine states. As described in Section 4, we can combine the lemmas at each level to obtain correctness properties for the entire translation from CBPV to CFG.

7. Controlling Code Generation

While the execution model described in the previous section is provides reasonable cost model, and the virtual machine, by construction, are similar to machines used in the compiler literature, there are clearly CFG programs that are not in the image of the compilation function. The most glaring omission is that we haven't yet accurately captured mutually recursive blocks of code: since we've used CBPV's negative products and projections to express mutual recursion, the resulting CFG programs must push a projection argument and jump through a `SWI` instruction. While we could modify the CBPV language with a `letrec` construct directly, the operational semantics of general `letrec` is itself nontrivial and the modified compilation strategy of the previous section suggests an alternative.

Thus far, our goal was to have the CFG machine and the CBPV semantics implement the same transition system, modulo the syntax of states. In the CFG machine, rather than pushing a statically known projection and then executing a `SWI` instruction, we could easily write a program that jumps directly to the target; however, there is no single SOS reduction rule that will eliminate a thunk and tuple at once. In this section, we describe a general technique for using machine instructions that do not correspond to a single reduction in the SOS rules, while retaining the tight correspondence between execution models.

```
argsH a =
match a with
| · ⇒ (·, none)
| ARG p :: a′ ⇒
  map× (fun l ⇒ ARG (γ̄ (0 :: p)) :: l) id (argsH a′)
| PRJ i :: a′ ⇒
  map× (fun l ⇒ PRJ i :: l) id (argsH a′)
| SEQ p :: a′ ⇒ (·, some a′)
end

args p = argsH (frames p)
```

Figure 12. args compiler function.

7.1 Annotations for code generation.

The overall approach is to add ad-hoc annotations to the source language to control the associated execution model without changing the source semantics. Many real compiler intermediate representations include an interface for communicating additional non-semantic information to the code generation phase; our approach can be seen as an attempt to formalize this process. At the CBPV level, the annotations give rise to an SOS in which annotated terms take larger steps; at the CFG level, they allow more efficient compilation, removing some of the potentially undesirable features of straightforward translation from CBPV. We have used this approach to enable our cost model for CBPV terms to include mutually recursive blocks of instructions both with and without intervening dispatch using `SWI` instructions.

Extending the compiler. The only required change to our CBPV language is to allow terms to be tagged with some additional annotations. To support direct jumps under tuples, this is just an extra flag on projections $\overline{\mathsf{prj}}_n\ M$ indicating that reduction happens, for the purposes of the cost model, at compile time. At the CFG level, when compiling a force, we may take advantage of these annotations: if the target of the jump is a label, we examine the context for any immediately enclosing compile-time projections and execute them immediately, removing them from the generated args and extending the target label to skip over the associated tuples/`SWI` instructions. Aside from this modification, the compiler is unchanged from Section 6.1 and has been omitted for space.

Well-annotated terms and correctness. The optimized compilation strategy described above does not provide useful output for every possible annotated term. It ignores compile-time projections that do not occur immediately above a force, and if the source term gets stuck then it may produce jumps to undefined paths (which never occurred in the original compiler). It would be very difficult to reason about the results of this compilation strategy for arbitrary annotated terms, without accounting for the specifics of the compilation strategy. Instead, we define a subset of CBPV terms which we call "well-annotated", for which all annotations will be compiled correctly.

Definition 6. *A term M is well-annotated iff*

1. *Sequences of compile-time projections only occur directly above a* force*, and*
2. *The target of such a* force *is either a* rec *or a variable bound by a* rec*, and the* rec *immediately encloses a set of nested tuples for which the projections are valid.*

Note that terms not containing annotated projections are trivially well-annotated. We augment the SOS with the following rule:

$$\overline{prj}\quad \overline{\mathsf{prj}}_i\ M \longrightarrow M_i \qquad \text{if } M \longrightarrow \langle M_1, .., M_n\rangle$$

It is not necessary to specify whether the usual search and reduction rules for projections apply to the compile-time annotated variant, since those cases never arise in well-annotated terms. The statement of correctness is identical to Theorem 2 restricted to well-annotated terms, where the operational semantics are augmented with the rule above, and the optimized compilation strategy is used.

Other optimizations. We conjecture that this technique can support various ad-hoc extensions to our CBPV-CFG correspondence. We have formulated (but not yet verified) methods of adding the following features using code-generation annotations:

- Specializing direct tail calls into parallel move instructions. Many real compiler intermediate representations differentiate between tail calls and jumps within a machine procedure, and this optimization is a step towards a more literal implementation of phi nodes.
- Specializing the case where a variable bound by a rec only appears under a force within its scope. This corresponds to omitting the function preamble.
- Expressing function pointers, including those to mutually-recursively defined functions, as bare labels when it's syntactically apparent that the definitions do not close over any free variables.

These are typically optimizations done using analyses on low-level intermediate representations.

8. Discussion

The final form of the CFG virtual machine that we consider in this work strays from typical SSA intermediate representations in a number of ways. Most importantly, we claim that it accurately models key aspects of the execution of the intraprocedural fragment most commonly targeted by optimizations. We argue informally, but also consider what it might mean to prove this property.

8.1 Extensions

The most obvious extension to SSA form in our machine is that labels can appear as arguments to non-arithmetic instructions, and they allocate closures when evaluated. It's worth noting how small of a change this is: operational semantics given for full SSA languages (Zhao et al. 2012; Barthe et al. 2014) already allocate closures to model the stack frame.

Next, the use of the argument stack is more flexible than usual formulations of SSA. We conjecture that the restrictions present in SSA and functional IRs using multi-argument functions can be reintroduced by requiring that lambdas only appear directly under a rec and are never sequenced. However, this would rule out use of the argument stack for defining variable-argument functions. We believe these are useful for, among other things, efficiently representing curried functions when compiling ML-family languages to CBPV. The accompanying Coq development contains versions of the `eval` combinators (Marlow and Jones 2004) implemented in our CBPV fragment.

The `SWI` instruction is a form of jump-table instruction that is not commonly considered in CFG machines. Intuitively, it can be thought of as a control-flow analog of SSA's phi functions: each control flow predecessor of a block containing a `SWI` must have an associated projection on which to dispatch as well as arguments bound by phi nodes. Within a single function, it allows sharing a prefix of a block between CFG predecessors while requiring the associations to be statically known. It can always be rewritten by making a copy of the shared section of the CFG for each control flow predecessor. For our full language, where the control flow predecessor of a block is not always known, `SWI` supports a form of

dynamic dispatch, where "messages" are kept distinct from runtime values.

8.2 Future Work

As we alluded to in Section 7.1, CFG representations typically distinguish between "machine procedures" and intraprocedural control flow. While semantically both can be represented by functions, the distinction is important for code generation and optimization. Machine procedures require generating a function preamble implementing a standard parameter passing protocol, while SSA basic blocks do not. The key difference is that SSA block labels are assumed not to escape, and code generation can assume that every call site is known and can be specialized. Previous work on the SSA-functional programming correspondence only makes claims about a syntactically restricted first-order subset of functional programs (Kelsey 1995; Beringer et al. 2003; Schneider et al. 2015). We extend the correspondence to higher-order programs. Our CFG instruction set similarly contains both PMOV instructions and tail calls, and both can represent SSA phi functions by merging values at control flow join points, but our compiler only emits the former in simple cases. We have sketched an approach for using our code-generation annotations to distinguish additional intraprocedural control flow without modifying the source language.

While SSA intermediate representations do not typically include closures as primitive values , they do include function pointers. Our CFG machine currently only supports passing code values as closures. We believe that explicitly omitting allocating empty environments for closures that correspond to uses of function pointers in standard SSA is possible using our technique.

8.3 Completeness

A natural question to consider is whether our CBPV-CFG correspondence can be used to reason about any SSA program, or whether there are useful programs not in the image of our compilation function. This is a difficult property to state precisely, since it is dependent on the particular form of instructions chosen, and there is no canonical choice for SSA. We do claim, however, that there is a subset of CBPV terms that intuitively correspond to the intraprocedural fragment of SSA, for which our SOS and CFG machine accurately capture the intended semantics and execution model.

Consider the subset of well-annotated terms that contain only compile-time projections, where rec and variables bound by rec only occur directly under force. We also require that lambdas only appear directly under tuples, the root of the term, or other lambdas, and that only arithmetic operations appear in the left position of a sequence. The example in Fig. 1 is such a program along with its compiled CFG. It is clear by inspection that our compilation function will not emit calls with projection arguments, or any instructions that allocate closures. Furthermore, we can express arbitrary CFGs, modulo our particular choice of conditional instruction.

A related question is whether it is possible to express all reasonable CFG programs as CBPV terms. In Section 7 we showed that this is not the case, and provided several examples of CFG programs that are not in the image of the compilation function. For instance, the compiler only produces a subset of terms that use PMOV instructions at control flow join points, since they are only emitted for prd to the immediately enclosing sequence. One goal might be to characterize well-formed control flow graphs directly, rather than as the image of our compilation function. It would be interesting to formulate the higher-order variant of the SSA dominator condition, but we leave such considerations to future work.

9. Related Work

The similarity between SSA intermediate representations and functional languages using lexical scope was noticed early on by Appel (Appel 1992). Kelsey provided translations between the intraprocedural fragment of SSA and a subset of CPS with additional annotations (Kelsey 1995). Chakravarty et al. (Chakravarty et al. 2003) extended the correspondence to ANF forms, and presented a translation from SSA to ANF. They used this translation to port a variant of the sparse conditional constant propagation algorithm from SSA to ANF, and proved the associated analysis sound. Real compilers using ANF or CPS representations employ sophisticated analyses to avoid performing the extra work of saving and restoring the environment, and to turn the calls into direct jumps (Wand and Steckler 1994; Reppy 2001; Fluet and Weeks 2001). These intensional aspects of execution, however, are outside of the scope of the cost model provided by the ANF machine.

Beringer et al. (Beringer et al. 2003) define a first-order language that has both "functional" and "imperative" semantics. They present a big-step environment-based semantics where local functions are represented as closures in the environment, and a small-step semantics where variable bindings are interpreted as assignments and local function calls as jumps. They then identify a restricted subset of programs for which the two evaluation functions agree. The approach is similarly motivated by the more powerful reasoning principles available for the "functional" semantics.

Recent work by by Schneider et al. (Schneider et al. 2015) uses a similar approach, but their first-order language allows nested local function declarations as well as observable effects. The subset of terms for which the functional and imperative semantics of their language agree includes functions with free variables bound in an enclosing scope, using a condition they call *coherence*. They then formulate a register allocation transformation that puts programs in coherent form, and prove that it preserves a form of trace equivalence. They also provide machine-checked proofs.

The use of abstract machines is one of the oldest and most fruitful ideas in the programming languages literature, and dates back to Landin (Landin 1964). The CEK machine is due to Felleisen (Felleisen and Friedman 1986), as is the formulation of correctness using an "unloading" function (Felleisen and Flatt 1989), while the notion of machine equivalence modulo syntax is from Flanagan et al. (Flanagan et al. 1993)

The approach of linking existing calculi and virtual machines was inspired by the methods of Ager et al. (Ager et al. 2003b,a). The method of deriving compilers and virtual machine interpreters is attributed to Wand (Wand 1985). Many authors have investigated the relationships between various forms of operational semantics (Hannan and Miller 1990; Curien 1991), and it continues to be an active topic of research (Danvy and Nielsen 2004; Danvy 2008). Our contribution can be seen as an extension of this line of work to control flow graph formalisms found in the compiler literature.

Compared to the derivations presented in (Danvy 2003), our target language (and therefore our approach) is more ad-hoc: we do not make any connections to existing functional transformations such as defunctionalization or CPS. The resulting compiler also does more work by emitting different instructions based on context. Rather than beginning the derivation with an evaluation function, we use small-step operational semantics throughout.

Our overall approach to our abstract machines is inspired by Leroy's work on the Zinc abstract machine (Leroy 1990), though his goal is an efficient bytecode interpreter for the Caml language. He begins with the goal of avoiding allocating intermediate closures during curried function application, just as we aim to avoid allocating closures for the first-order subset of our language. From an existing abstract machine, he derives a modified machine that implements this optimization, in this case a variant of the Krivine machine. Finally, he modifies the operational semantics of the language to agree with the new execution model, as we do with our source annotations in Section 7.1.

Simplification phases for functional languages (Appel and Jim 1997; Benton et al. 2004; Kennedy 2007) often use graph representations of lambda terms to improve efficiency. While these techniques effectively compile source terms into a data structure with explicit use-def information in order to execute reductions quickly, unlike SSA they are not considered as execution models for the source language.

References

M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. *BRICS Report Series*, 10(14), 2003a.

M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 8–19, 2003b. doi: 10.1145/888251.888254.

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6.

A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN 0-521-41695-7.

A. W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4): 17–20, 1998. doi: 10.1145/278283.278285.

A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *J. Funct. Program.*, 7(5):515–540, 1997.

G. Barthe, D. Demange, and D. Pichardie. Formal verification of an ssa-based middle-end for compcert. *ACM Trans. Program. Lang. Syst.*, 36 (1):4, 2014. doi: 10.1145/2579080.

N. Benton, A. Kennedy, S. Lindley, and C. V. Russo. Shrinking reductions in SML.NET. In C. Grelck, F. Huch, G. Michaelson, and P. W. Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004, Lübeck, Germany, September 8-10, 2004, Revised Selected Papers*, volume 3474 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2004.

L. Beringer, K. MacKenzie, and I. Stark. Grail: a functional form for imperative mobile code. *Electr. Notes Theor. Comput. Sci.*, 85(1):3–23, 2003. doi: 10.1016/S1571-0661(05)80083-0.

P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw., Pract. Exper.*, 28(8):859–881, 1998. doi: 10.1002/(SICI)1097-024X(19980710)28:8<859::AID-SPE188>3.0.CO;2-8.

M. M. T. Chakravarty, G. Keller, and P. Zadarnowski. A functional perspective on SSA optimisation algorithms. *Electr. Notes Theor. Comput. Sci.*, 82(2):347–361, 2003. doi: 10.1016/S1571-0661(05)82596-4.

P. Curien. An abstract framework for environment machines. *Theor. Comput. Sci.*, 82(2):389–402, 1991. doi: 10.1016/0304-3975(91)90230-Y.

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. doi: 10.1145/115372.115320.

O. Danvy. A journey from interpreters to compilers and virtual machines. In *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, page 117, 2003. doi: 10.1007/978-3-540-39815-8_7.

O. Danvy. Defunctionalized interpreters for programming languages. In J. Hook and P. Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 131–142. ACM, 2008. doi: 10.1145/1411204.1411206.

O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. *BRICS Report Series*, 11(26), 2004.

M. Felleisen and M. Flatt. Programming languages and lambda calculi. *Unpublished lecture notes. http://www. ccs. neu. edu/home/matthias/3810-w02/readings. html*, 2003:1538, 1989.

M. Felleisen and D. P. Friedman. *Control Operators, the SECD-machine, and the λ-calculus*. Indiana University, Computer Science Department, 1986.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In R. Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247. ACM, 1993. doi: 10.1145/155090.155113.

M. Fluet and S. Weeks. Contification using dominators. In B. C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001.*, pages 2–13. ACM, 2001. doi: 10.1145/507635.507639.

J. Hannan and D. Miller. From operational semantics to abstract machines: Preliminary results. In *LISP and Functional Programming*, pages 323–332, 1990. doi: 10.1145/91556.91680.

R. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*, pages 13–23, 1995. doi: 10.1145/202529.202532.

A. Kennedy. Compiling with continuations, continued. In R. Hinze and N. Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 177–190. ACM, 2007. doi: 10.1145/1291151.1291179.

P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43 (4):363–446, 2009. doi: 10.1007/s10817-009-9155-4.

P. B. Levy. Call-by-push-value: A subsuming paradigm. In J. Girard, editor, *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, volume 1581 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 1999. doi: 10.1007/3-540-48959-2_17.

S. Marlow and S. L. P. Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In C. Okasaki and K. Fisher, editors, *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, pages 4–15. ACM, 2004. doi: 10.1145/1016850.1016856.

S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN 1-55860-320-4.

B. C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN 0262162288.

A. M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer, 2000. doi: 10.1007/3-540-45699-6_8.

G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

J. Reppy. Local CPS conversion in a direct-style compiler. In *Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW '01)*, pages 13–22. Citeseer, 2001.

S. Schneider, G. Smolka, and S. Hack. A first-order functional intermediate language for verified compilers. *CoRR*, abs/1503.08665, 2015.

V. C. Sreedhar, R. D. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In A. Cortesi and G. Filé, editors, *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*, volume 1694 of *Lecture Notes in*

Computer Science, pages 194–210. Springer, 1999. doi: 10.1007/3-540-48294-6_13.

R. M. Stallman and the GCC Developer Community. *Using The Gnu Compiler Collection*. CreateSpace, Paramount, CA, 2009. ISBN 144141276X, 9781441412768.

M. Wand. From interpreter to compiler: a representational derivation. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects, Proceedings of a Workshop, Copenhagen, Denmark, October 17-19, 1985*, volume 217 of *Lecture Notes in Computer Science*, pages 306–324. Springer, 1985.

M. Wand and P. Steckler. Selective and lightweight closure conversion. In H. Boehm, B. Lang, and D. M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 435–445. ACM Press, 1994. doi: 10.1145/174675.178044.

S. Weeks. Whole-program compilation in mlton. In A. Kennedy and F. Pottier, editors, *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, page 1. ACM, 2006. doi: 10.1145/1159876.1159877.

J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 427–440. ACM, 2012. doi: 10.1145/2103656.2103709.