Formal Small-step Verification of a Call-by-value Lambda Calculus Machine

Fabian Kunze, Gert Smolka, Yannick Forster

Saarland University, Saarbrücken, Germany
{kunze,smolka,forster}@ps.uni-saarland.de

Abstract. We formally verify an abstract machine for a call-by-value λ-calculus with de Bruijn terms, simple substitution, and small-step semantics. We follow a stepwise refinement approach starting with a naive stack machine with substitution. We then refine to a machine with closures, and finally to a machine with a heap providing structure sharing for closures. We prove the correctness of the three refinement steps with compositional small-step bottom-up simulations. There is an accompanying Coq development verifying all results.

1 Introduction

The call-by-value λ-calculus is a minimal functional programming language that can express recursive functions and inductive data types. Forster and Smolka [12] employ the call-by-value λ-calculus as the basis for a constructive theory of computation and formally verify elaborate programs such as step-indexed self-interpreters. Dal Lago and Martini [8] show that Turing machines and the call-by-value λ-calculus can simulate each other within a polynomial time overhead (under a certain cost model). Landin's SECD machine implements the call-by-value λ-calculus with closures eliminating the need for substitution [14,18].

In this paper we consider the call-by-value λ-calculus L from [12]. L comes with de Bruijn terms and simple substitution, and restricts β-reduction to terms of the form $(\lambda s)(\lambda t)$ that do not appear within abstractions. This is in contrast to Plotkin's call-by-value λ-calculus [18], which employs terms with named argument variables and substitution with renaming, and β-reduces terms of the forms $(\lambda x.s)(\lambda y.t)$ and $(\lambda x.s)y$. L and Plotkin's calculus agree for closed terms, which suffice for functional computation.

The subject of this paper is the formal verification of an abstract machine for L with closures and structure sharing. Our machine differs from the SECD machine in that it operates on programs rather than terms, has two flat stacks rather than one stack of frames, and provides structure sharing through a heap. Our goal was to come up with a transparent machine design providing for an elegant formal verification. We reach this goal with a stepwise refinement approach starting with a naive stack machine with programs and substitution. We then refine to a machine with closures, and finally to a machine with a heap. As

S. Ryu. (Ed): APLAS 2018, LNCS 11275, pp. 264-283, 2018.
The final authenticated publication is available online at: https://doi.org/10.1007/978-3-030-02768-1_15

it comes to difficulty of verification, the refinement to the naive stack machine is by far the most substantial.

We prove the correctness of the three refinement steps with compositional small-step bottom-up simulations (i.e., L is above the machines and simulates machine transitions). While L has only β-steps, our machines have β- and τ-steps. L simulates a machine by following β-steps and ignoring τ-steps, and a machine simulates a lower-level machine by following β-steps with β-steps and τ-steps with τ-steps. To obtain bisimulations, we require progress conditions: Reducibility must propagate downwards and machines must stop after finitely many τ-steps.

The first verification step establishes the naive stack machine as a correct implementation of L, the second verification step establishes the closure machine as a correct implementation of the naive stack machine, and the third verification step establishes the heap machine as a correct implementation of the closure machine. The second and third verification step are relatively straightforward since they establish strict simulations (no silent steps). Strict simulations suffice since the programs of the naive stack machine already provide the right granularity for the structure sharing heap machine.

The entire development is formalised with the Coq proof assistant [22]. Coq's type theory provides an ideal foundation for the various inductive constructions needed for the specification and verification of the machines. All reasoning is naturally constructive. In the paper we don't show Coq code but use mathematical notation and language throughout. While familiarity with constructive type theory is helpful for reading the paper, technical knowledge of Coq is not required. For the expert and the curious reader, the definitions and theorems in the paper are hyperlinked with their formalisations in an HTML rendering of the Coq development. The Coq formalisation is available at `https://www.ps.uni-saarland.de/extras/cbvlcm2/`.

Related Work

We review work concerning the verification of abstract machines for call-by-value λ-calculus.

Plotkin [18] presents the first formalisation and verification of Landin's SECD machine [14]. He considers terms and closures with named variables and proves that his machine computes normal forms of closed terms using a step-indexed evaluation semantics for terms and top-down arguments (from λ-calculus to machine). He shows that failure of term evaluation for a given bound entails failure of machine execution for this bound. Plotkin does not prove his substitution lemmas. Ramsdell [19] reports on a formalisation of a Plotkin-style verification of an SECD machine optimising tail calls using the Boyer-Moore theorem prover. Ramsdell employs de Bruijn terms and de Bruijn substitution.

Felleisen and Friedman [10] study Plotkin's call-by-value λ-calculus extended with control operators like J and call/cc. They prove correctness properties relating abstract machines, small-step reduction systems, and algebraic theories. Like Plotkin, they use terms and closures with named variables.

Rittri [20] seems to be the first who verifies an abstract machine for a call-by-value λ-calculus using a small-step bottom up simulation. Rittri's work is also similar to ours in that he starts from a λ-calculus with simple substitution reducing closed terms, and in that his machine uses a control and an argument stack. Rittri gives detailed informal proofs using terms with named variables. He does not consider a naive intermediate machine nor a heap realisation.

Hardin et al. [13] verify several abstract machines with respect to a fine-grained λ-calculus with de Bruijn terms and explicit substitution primitives. Like us, they simulate machine steps with reduction steps of the calculus and disallow infinitely many consecutive silent steps. They consider the Krivine machine [7] (call-by-name), the SECD machine [14,18] (call-by-value), Cardelli's FAM [5] (call-by-value), and the categorical abstract machine [6] (call-by-value) .

Accattoli et al. [1] verify several abstract machines for the linear substitution calculus with explicit substitution primitives. They simulate machine steps with reduction steps of the calculus and model internal steps of the calculus with a structural congruence. They employ a global environment acting as heap. Among other machines, they verify a simplified variant of the ZINC machine [15].

Leroy [17,16] verifies the Modern SECD machine for call-by-value λ-calculus specified with de Bruijn terms and an environment-based evaluation semantics in Coq. The modern SECD machine has programs and a single stack. Leroy's semantic setup is such that neither substitution nor small-step reduction of terms have to be considered. He uses top-down arguments and compiles terms into machine states. Using coinductive divergence predicates, Leroy shows that the machine diverges on states obtained from diverging terms. Leroy's proofs are pleasantly straightforward.

Danvy and Nielsen [9] introduce the refocusing technique, a general procedure transforming small-step reduction systems defined with evaluation contexts into abstract machines operating on the same syntax. Biernacka and Danvy [4] extend refocusing and obtain environment-based abstract machines. This yields a framework where the derived machines are provably correct with respect to small-step bisimulation. Biernacka et al. [3] formalise a generalisation of the framework in Coq.

Swierstra [21] formally verifies the correctness of a Krivine machine for simply typed λ-calculus in the dependently typed programming language Agda. Also following Biernacka and Danvy [4], Swierstra does this by showing the correctness of a Krivine-style evaluator for an iterative and environment-based head reduction evaluator. This way substitution does not appear. Swierstra's dependently typed constructions also provide normalisation proofs for simply typed λ-calculus. Swierstra's approach will not work for untyped λ-calculus.

Contribution of the Paper

We see the main contribution of the paper in the principled formal verification of a heap machine for a call-by-value λ-calculus using a small-step bottom-up simulation. A small-step bottom-up verification is semantically more informative than the usual evaluation-based top-down verification in that it maps every

reachable machine state to a term of L. The entire Coq development consists of 500 lines of proof plus 750 lines of specification. The decomposition of the verification in three refinement steps provides for transparency and reusability. The use of the naive stack machine as an intermediate machine appears to be new. We also think that our simple formalisation of structure sharing with code and heap is of interest.

We envision a formal proof showing that Turing machines can simulate L with polynomial overhead in time and constant overhead in space (under a suitable cost model) [11]. The verifications in this paper are one step into this direction.

Plan of the Paper

After some preliminaries fixing basic notions in Coq's type theory, we specify the call-by-value λ-calculus L and present our abstract framework for machines and refinements. We then introduce programs and program substitution and prove a substitution lemma. Next we specify and verify the naive stack machine for L. This is the most complex refinement step as it comes to proofs. Next we specify the closure machine and verify that it is an implementation of the naive stack machine and hence of L (by compositionality). Finally, we define abstractions for codes and heaps and verify that the heap machine is an implementation of the closure machine and hence of L.

2 Preliminaries

Everything in this paper is carried out in Coq's type theory and all reasoning is constructive. We use the following inductive types: N providing the *numbers* $n ::= 0 \mid \mathsf{S}n$, and $\mathcal{O}(X)$ providing the *options* $\emptyset$ and $^\circ x$, and $\mathcal{L}(X)$ providing the *lists* $A ::= [] \mid x :: A$.

For lists $A, B : \mathcal{L}(X)$ we use the functions *length* $|A| : \mathsf{N}$, *concatenation* $A +\!\!+ B : \mathcal{L}(X)$, *map* $f@A : \mathcal{L}(Y)$ where $f : X \to Y$, and *lookup* $A[n] : \mathcal{O}(X)$ where $(x :: A)[0] = {}^\circ x$, and $(x :: A)[\mathsf{S}n] = A[n]$, and $[][n] = \emptyset$. When we define functions that yield an option, we will omit equations that yield $\emptyset$ (e.g., the third equation $[][n] = \emptyset$ defining lookup $A[n] : \mathcal{O}(X)$ will be omitted).

We write $\mathbf{P}$ for the universe of propositions and $\bot$ for the proposition falsity. A *relation on X and Y* is a predicate $X \to Y \to \mathbf{P}$, and a *relation on X* is a predicate $X \to X \to \mathbf{P}$. A relation R is *functional* if $y = y'$ whenever Rxy and Rxy'. A relation R on X and Y is *computable* if there is a function $f : X \to \mathcal{O}(Y)$ such that $\forall x.\ (\exists y.\ fx = {}^\circ y \land Rxy) \lor (fx = \emptyset \land \neg\exists y.\ Rxy)$.

We use a recursive *membership* predicate $x \in A$ such that $(x \in []) = \bot$ and $(x \in y :: A) = (x{=}y \lor x \in A)$.

We define an inductive predicate $\mathsf{ter}_R\ x$ identifying the *terminating points* of a relation R on X:

$$\frac{\forall x'.\ Rxx' \to \mathsf{ter}_R\ x'}{\mathsf{ter}_R\ x}$$

If x is a terminating point of R, we say that R *terminates on* x or that x *terminates for* R. We call a relation *terminating* if it terminates on every point.

Let R be a relation on X. The *span of* R is the inductive relation $\rhd_R$ on X defined as follows:

$$\frac{\neg\exists y.\ Rxy}{x \rhd_R x} \qquad\qquad \frac{Rxx' \qquad x' \rhd_R y}{x \rhd_R y}$$

If $x \rhd_R y$, we say that y *is a normal form of* x *for* R.

Fact 1. *1. If R is functional, then $\rhd_R$ is functional.*
2. If R is functional and x has a normal form for R, then R terminates on x.
3. If R is computable, then every terminating point of R has a normal form for R.

A *reduction system* is a structure consisting of a type X and a relation R on X. Given a reduction system $A = (X, R)$, we shall write A for the type X and $\succ_A$ for the relation of A. We say that *a reduces to b in A* if $a \succ_A b$.

3 Call-by-value Lambda Calculus L

The call-by-value λ-calculus we consider in this paper employs de Bruijn terms with simple substitution and admits only abstractions as values.

We provide *terms* with an inductive type

$$s, t, u, v \ :\ \mathsf{Ter} \ ::=\ n \mid st \mid \lambda s \qquad (n : \mathsf{N})$$

and define a recursive function s_u^k providing *simple substitution*:

$$\begin{array}{lll} k_u^k := u & & (st)_u^k := (s_u^k)(t_u^k) \\ n_u^k := n & \text{if } n \neq k & (\lambda s)_u^k := \lambda(s_u^{\mathsf{S}k}) \end{array}$$

We define an inductive *reduction relation* $s \succ t$ on terms:

$$\frac{}{(\lambda s)(\lambda t) \succ s_{\lambda t}^0} \qquad \frac{s \succ s'}{st \succ s't} \qquad \frac{t \succ t'}{(\lambda s)t \succ (\lambda s)t'}$$

Fact 2. *$s \succ t$ is functional and computable.*

We define an *inductive bound predicate* $s < k$ for terms:

$$\frac{n < k}{n < k} \qquad \frac{s < k \qquad t < k}{st < k} \qquad \frac{s < \mathsf{S}k}{\lambda s < k}$$

Informally, $s < k$ holds if every free variable of s is smaller than k. A term is *closed* if $s < 0$. A term is *open* if it is not closed.

For closed terms, reduction in L agrees with reduction in the λ-calculus. For open terms, reduction in L is ill-behaved since L is defined with simple substitution. For instance, we have $(\lambda\lambda 1)(\lambda 1)(\lambda 0) \succ (\lambda\lambda 1)(\lambda 0) \succ \lambda\lambda 0$. Note that the second 1 in the initial term is not bound and refers to the De Bruijn index 0. Thus the first reduction step is capturing.

We define *stuck terms* inductively:

$$\frac{}{\mathsf{stuck}\ n} \qquad \frac{\mathsf{stuck}\ s}{\mathsf{stuck}\,(st)} \qquad \frac{\mathsf{stuck}\ t}{\mathsf{stuck}\,((\lambda s)t)}$$

Fact 3. (Trichotomy) *For every term s, exactly one of the following holds: (1) s is reducible. (2) s is an abstraction. (3) s is stuck.*

4 Machines and Refinements

We model machines as reduction systems. Recall that L is also a reduction system. We relate a machine M with L with a relation $a \gg s$ we call *refinement*. If $a \gg s$ holds, we say that a (a state of M) refines s (a term of L). Correctness means that L can simulate steps of M such that refinement between states and terms is preserved. Concretely, if a refines s and a reduces to a' in M, then either a' still *refines* s or s reduces to some s' in L such that a' refines s'. Steps where the refined term stays unchanged are called *silent*.

The general idea is now as follows. Given a term s, we compile s into a refining state a. We then run the machine on a. If the machine terminates with a normal form b of a, we decompile b into a term t such that b refines t and conclude that t is a normal form of s. We require that the machine terminates for every state refining a term that has a normal form.

Definition 4. *A* machine *is a structure consisting of a type A of* states *and two relations $\succ_\tau$ and $\succ_\beta$ on A. When convenient, we consider a machine A as a reduction system with the relation $\succ_A := \succ_\tau \cup \succ_\beta$.*

The letter X ranges over reduction systems and A and B range over machines.

Definition 5. *A* refinement *A to X is a relation $\gg$ on A and X such that:*

1. *If $a \gg x$ and x is reducible with $\succ_X$, then a is reducible with $\succ_A$.*
2. *If $a \gg x$ and $a \succ_\tau a'$, then $a' \gg x$.*
3. *If $a \gg x$ and $a \succ_\beta a'$, then there exists x' such that $a' \gg x'$ and $x \succ x'$.*
4. *If $a \gg x$, then a terminates for $\succ_\tau$.*

We say that a refines *x if $a \gg x$.*

Figure 1 illustrates refinements with a diagram. Transitions in X appear in the upper line and transitions in A appear in the lower line. The dotted lines represent the refinement relation. Note that conditions (2) and (3) of Definition 5 ensure that refinements are bottom up simulations (i.e., X can simulate A). Conditions (1) and (4) are progress conditions. They suffice to ensure that refinements also act as top-down simulations (i.e. A can simulate X), given mild assumptions that are fulfilled by L and all our machines.

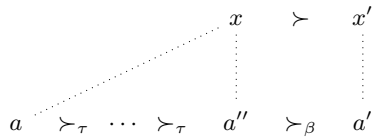

Fig. 1. Refinement diagram

Fact 6. (Correctness) *Let $\gg$ be a refinement A to X and $a \gg x$. Then:*

1. *If $a \rhd_A a'$, there exists x' such that $a' \gg x'$ and $x \rhd_X x'$.*
2. *If $a \rhd_A a'$, $a' \gg x'$, and $\gg$ is functional, then $x \rhd_X x'$.*
3. *If x terminates for $\succ_X$, then a terminates for $\succ_A$.*
4. *If x terminates for $\succ_X$ and $\succ_A$ is computable, then there exists a' such that $a \rhd_A a'$.*

Proof. (1) follows by induction on $a \rhd_A x$. (2) follows with (1) and Fact 1. (3) follows by induction on the termination of x for $\succ_X$ and the termination of a for $\succ_\tau$. (4) follows by induction on the termination of x. □

We remark that the concrete reduction systems we will consider in this paper are all functional and computable. Moreover, all concrete refinements will be functional and, except for the heap machine, also be computable.

A refinement may be seen as the combination of an invariant and a decompilation function. We speak of an invariant since the fact that a state is a refinement of a term is preserved by the reduction steps of the machine.

Under mild assumptions fulfilled in our setting, the inverse of a refinement is a stuttering bisimulation [2]. The following fact asserts the necessary top-down simulation.

Fact 7. *Let $\gg$ be a refinement A to X where $\succ_X$ is functional and $\succ_\tau$ is computable.*

1. *If $a \gg x \succ_X x'$, then there exist a' and a'' such that $a \rhd_\tau a'' \succ_\beta a' \gg x'$.*
2. *If $a \gg x \rhd_X x'$, then there exists a' such that $a \rhd_A a' \gg x'$.*

Proof. (1) follows with Fact 1. (2) follows by induction on $x \rhd_X x'$ using (1). □

We will also refine machines with machines and rely on a composition theorem that combines two refinements A to B and B to X to a refinement A to X. We define refinement of machines with strict simulation.

Definition 8. *A* refinement *A to B is a relation $\gg$ on A and B such that:*

1. *If $a \gg b$ and b is reducible with $\succ_B$, then a is reducible with $\succ_A$.*
2. *If $a \gg b$ and $a \succ_\tau a'$, then there exists b' such that $a' \gg b'$ and $b \succ_\tau b'$.*
3. *If $a \gg b$ and $a \succ_\beta a'$, then there exists b' such that $a' \gg b'$ and $b \succ_\beta b'$.*

Fact 9. (Composition) *Let $\gg_1$ be a refinement A to B and $\gg_2$ be a refinement B to X. Then the composition $\lambda ac.\, \exists b.\, a \gg_1 b \wedge b \gg_2 c$ is a refinement A to X.*

5 Programs

The machines we will consider execute programs. Programs may be seen as lists of commands to be executed one after the other. Every term can be compiled into a program, and programs that are images of terms can be decompiled. There are commands for variables, abstractions, and applications. We represent *programs* with a tree-recursive inductive type so that the command for abstractions can nest programs:

$$P, Q, R \ : \ \mathsf{Pro} \ ::= \ \mathsf{ret} \mid \mathsf{var}\, n; P \mid \mathsf{lam}\, Q; P \mid \mathsf{app}; P \qquad (n : \mathsf{N})$$

We define a tail recursive *compilation function* $\gamma : \mathsf{Ter} \to \mathsf{Pro} \to \mathsf{Pro}$ translating terms into programs:

$$\begin{aligned} \gamma n P &:= \mathsf{var}\, n; P & \gamma(\lambda s) P &:= \mathsf{lam}(\gamma s \mathsf{ret}); P \\ \gamma(st) P &:= \gamma s(\gamma t(\mathsf{app}; P)) \end{aligned}$$

The second argument of γ may be understood as a continuation.

We also define a *decompilation function* $\delta P A$ of type $\mathsf{Pro} \to \mathcal{L}(\mathsf{Ter}) \to \mathcal{O}(\mathcal{L}(\mathsf{Ter}))$ translating programs into terms. The function executes the program over a stack of terms. The optional result acknowledges the fact that not every program represents a term. We write A and B for lists of terms. Here are the equations defining the decompilation function:

$$\begin{aligned} \delta\, \mathsf{ret}\, A &:= {}^{\circ}A \\ \delta(\mathsf{var}\, n; P) A &:= \delta P(n :: A) \\ \delta(\mathsf{lam}\, Q; P) A &:= \delta P(\lambda s :: A) && \text{if } \delta\, Q\, [] = {}^{\circ}[s] \\ \delta(\mathsf{app}; P) A &:= \delta P(st :: A') && \text{if } A = t :: s :: A' \end{aligned}$$

Decompilation inverts compilation:

Fact 10. $\delta(\gamma s P) A = \delta P(s :: A)$.

Fact 11. *Let* $\delta P A = {}^{\circ}A'$. *Then* $\delta P(A +\!\!+ A'') = {}^{\circ}(A' +\!\!+ A'')$.

We define a predicate $P \gg s := \ \delta P [] = {}^{\circ}[s]$ read as P *represents* s.

The naive stack machine will use a *substitution operation* P^k_R for programs:

$$\begin{aligned} \mathsf{ret}\,{}^k_R &:= \mathsf{ret} & (\mathsf{lam}\, Q; P)^k_R &:= \mathsf{lam}(Q^{\mathsf{S}k}_R); P^k_R \\ (\mathsf{var}\, k; P)^k_R &:= \mathsf{lam}\, R; P^k_R & (\mathsf{app}; P)^k_R &:= \mathsf{app}; P^k_R \\ (\mathsf{var}\, n; P)^k_R &:= \mathsf{var}\, n; P^k_R \quad \text{if } n \neq k \end{aligned}$$

Note the second equation for the variable command that replaces a variable command with a lambda command. The important thing to remember here is the fact that the program R is inserted as the body of a lambda command.

For the verification of the naive stack machine we need a substitution lemma relating term substitution with program substitution. The lemma we need appears as Corollary 13 below. We prove the fact with a generalised version that can be shown by induction on programs. We use the notation $A^k_u := (\lambda s.\, s^k_u)@A$.

$$\begin{aligned} \mathsf{ret} :: T,\ V &\succ_\tau T,\ V \\ (\mathsf{lam}\, Q; P) :: T,\ V &\succ_\tau P :: T,\ Q :: V \\ (\mathsf{app}; P) :: T,\ R :: Q :: V &\succ_\beta Q^0_R :: P :: T,\ V \end{aligned}$$

Fig. 2. Reduction rules of the naive stack machine

Lemma 12. (Substitution) *Let $R \gg t$ and $\delta Q A = {}^\circ B$. Then $\delta\, Q^k_R\, A^k_{\lambda t} = {}^\circ B^k_{\lambda t}$.*

Corollary 13. (Substitution) *If $P \gg s$ and $Q \gg t$, then $P^k_Q \gg s^k_{\lambda t}$.*

We define a *bound predicate* $P < k$ for programs that is analogous to the bound predicate for terms and say that a program P is *closed* if $P < 0$:

$$\frac{}{\mathsf{ret} < k} \qquad \frac{n < k \qquad P < k}{\mathsf{var}\, n; P < k} \qquad \frac{Q < \mathsf{S}k \qquad P < k}{\mathsf{lam}\, Q; P < k} \qquad \frac{P < k}{\mathsf{app}; P < k}$$

Fact 14. *If $s < k$ and $P < k$, then $\gamma s P < k$.*

It follows that $\gamma s P$ is closed whenever s and P are closed.

6 Naive Stack Machine

The naive stack machine executes programs using two stacks of programs called *control stack* and *argument stack*. The control stack holds the programs to be executed, and the argument stack holds the programs computed so far. The machine executes the first command of the first program on the control stack until the control stack is empty or execution of a command fails.

The *states of the naive stack machine* are pairs

$$(T, V)\ :\ \mathcal{L}(\mathsf{Pro}) \times \mathcal{L}(\mathsf{Pro})$$

consisting of two lists T and V representing the control stack and the argument stack. We use the letters T and V since we think of the items on T as tasks and the items on V as values. The *reduction rules of the naive stack machine* appear in Figure 2. The parentheses for states are omitted for readability. We will refer to the rules as *return rule*, *lambda rule*, and *application rule*. The return rule removes the trivial program from the control stack. The lambda rule pushes a program representing an abstraction on the argument stack. Note that the programs on the control stack are executed as they are. This is contrast to the programs on the argument stack that represent bodies of abstractions. The application rule takes two programs from the argument stack and pushes an instantiated program obtained by β-reduction on the control stack. This way control is passed from the calling program to the called program. There is no reduction rule for the variable command since we will only consider states that represent closed terms.

Fact 15. *The relations $\succ_\tau$, $\succ_\beta$, and $\succ_\tau \cup \succ_\beta$ are functional and computable. Moreover, the relations $\succ_\tau$ and $\succ_\beta$ are terminating.*

We decompile machine states by executing the task stack on the stack of terms obtained by decompiling the programs on the value stack. To this purpose we define two decompilation functions. The *decompilation function δV for argument stacks* has type $\mathcal{L}(\mathsf{Pro}) \to \mathcal{O}(\mathcal{L}(\mathsf{Ter}))$ and satisfies the equations

$$\begin{aligned} \delta[] &:= {}^\circ[] \\ \delta(P :: V) &:= {}^\circ(\lambda s :: A) \qquad\qquad \text{if } P \gg s \text{ and } \delta V = {}^\circ A \end{aligned}$$

Note that the second equation turns the term s obtained from a program on the argument stack into the abstraction λs. This accounts for the fact that programs on the argument stack represent bodies of abstractions. The *decompilation function $\delta T A$ for control stacks* has type $\mathcal{L}(\mathsf{Pro}) \to \mathcal{L}(\mathsf{Ter}) \to \mathcal{O}(\mathcal{L}(\mathsf{Ter}))$ and satisfies the equations

$$\begin{aligned} \delta[]A &:= {}^\circ A \\ \delta(P :: T)A &:= \delta T A' \qquad\qquad \text{if } \delta P A = {}^\circ A' \end{aligned}$$

We now define the *refinement relation* between states of the naive stack machine and terms as follows:

$$(T, V) \gg s \;:=\; \exists A.\ \delta V = {}^\circ A \;\wedge\; \delta T A = {}^\circ[s]$$

We will show that $(T, V) \gg s$ is in fact a refinement.

Fact 16. *$(T, V) \gg s$ is functional and computable.*

Fact 17. (τ-Simulation) *If $(T, V) \gg s$ and $T, V \succ_\tau T', V'$, then $(T', V') \gg s$.*

Proof. We prove the claim for the second τ-rule, the proof for the first τ-rule is similar. Let $\mathsf{lam}\, Q; P :: T,\ V \ \succ_\tau \ P :: T,\ Q :: V$. We have

$$\begin{aligned} \delta(\mathsf{lam}\, Q; P :: T)(\delta V) &= \delta T(\delta(\mathsf{lam}\, Q; P)(\delta V)) \\ &= \delta T(\delta P(\lambda s :: \delta V)) \qquad\qquad Q \gg s \\ &= \delta(P :: T)(\delta(Q :: V)) \end{aligned}$$

□

Note that the equational part of the proof nests optional results to avoid cluttering with side conditions and auxiliary names.

Proving that L can simulate β-steps of the naive stack machine takes effort.

Fact 18. *If $\delta V = {}^\circ A$, then every term in A is an abstraction.*

Fact 19. $\delta(\mathsf{app}; P :: T)(t :: s :: A) = \delta(P :: T)(st :: A)$.

Fact 20. $\delta(P :: T)A = \delta T(s :: A)$ *if $P \gg s$.*

Proof. Follows with Fact 11. □

Lemma 21. (Substitution) $\delta(Q^0_R :: T)A = \delta T(s^0_{\lambda t} :: A)$ *if* $Q \gg s$ *and* $R \gg t$.

Proof. By Corollary 13 we have $Q^0_R \gg s^0_{\lambda t}$. The claim follows with Fact 20. □

We also need a special reduction relation $A \succ A'$ for term lists:

$$\frac{s \succ s' \qquad \forall t \in A.\ t \text{ is an abstraction}}{s :: A \succ s' :: A} \qquad\qquad \frac{A \succ A'}{s :: A \succ s :: A'}$$

Informally, $A \succ A'$ holds if A' can be obtained from A by reducing the term in A that is only followed by abstractions.

Lemma 22. *Let* $A \succ A'$ *and* $\delta P A = {}^\circ B$. *Then* $\exists B'.\ B \succ B' \wedge \delta P A' = {}^\circ B'$.

Proof. By induction on P. We consider the case $P = \mathsf{app}; P$.

Let $\delta(\mathsf{app}; P)(t :: s :: A) = {}^\circ B$ and $t :: s :: A \succ t' :: s' :: A'$. Then $\delta P(st :: A) = {}^\circ B$ and $st :: A \succ s't' :: A'$ (there are three cases: (1) $t = t'$, $s = s'$, and $A \succ A'$; (2) $t = t'$, $s > s'$, $A = A'$, and A contains only abstractions; (3) $t \succ t'$, $s :: A = s' :: A'$, and $s :: A$ contains only abstractions). By the inductive hypothesis we have $B \succ B'$ and $\delta P(s't' :: A') = {}^\circ B'$ for some B'. Thus $\delta(\mathsf{app}; P)(t' :: s' :: A') = {}^\circ B'$. □

Fact 23. (β-Simulation)
If $(T, V) \gg s$ *and* $T, V \succ_\beta T', V'$, *then* $\exists s'.\ (T', V') \gg s' \wedge s \succ s'$.

Proof. Let $\mathsf{app}; P :: T$, $R :: Q :: V \ \succ_\beta \ Q^0_R :: P :: T$, V. Moreover, let $R \gg t$, $Q \gg u$, and $\delta V = {}^\circ A$. We have:

$$\begin{array}{rcll}
{}^\circ[s] & = & \delta(\mathsf{app}; P :: T)(\delta(R :: Q :: V)) & \\
& = & \delta(\mathsf{app}; P :: T)(\lambda t :: \lambda u :: A) & \\
& = & \delta(P :: T)((\lambda u)(\lambda t) :: A) & \text{Fact 19} \\
& \succ & \delta(P :: T)(u^0_{\lambda t} :: A) & \text{Lemma 22 and Fact 18} \\
& = & \delta(Q^0_R :: P :: T)A & \text{Lemma 21} \\
& = & {}^\circ[s'] & \text{for some } s'
\end{array}$$

Note that s' exists since $\succ$ preserves the length of a list. We now have $s \succ s'$ by the definition of $\succ$ and $(Q^0_R :: P :: T,\ V) \gg s'$, which concludes the proof. □

It remains to show that states are reducible if they refine reducible terms. For this purpose, we define *stuck term lists*:

$$\frac{\mathsf{stuck}\ s \qquad \forall t \in A.\ t \text{ is an abstraction}}{\mathsf{stuck}\,(s :: A)} \qquad\qquad \frac{\mathsf{stuck}\ A}{\mathsf{stuck}\,(s :: A)}$$

Note that s is stuck iff $[s]$ is stuck.

Lemma 24. *Let* A *be stuck and* $\delta P A = {}^\circ B$. *Then* B *is stuck.*

Lemma 25. *Let A be stuck and $\delta T A = {}^\circ B$. Then B is stuck.*

Fact 26. (Trichotomy) *Let $T, V \gg s$. Then exactly one of the following holds:*

1. *(T, V) is reducible.*
2. *$(T, V) = ([], [P])$ and $P \gg s'$ with $s = \lambda s'$ for some P, s'.*
3. *$T = \mathsf{var}\, x; P :: T'$ for some x, P, T' and s is stuck.*

Proof. Let $\delta V = {}^\circ A$ and $\delta T A = {}^\circ[s]$, and s be reducible. By Fact 18 we know that A contains only abstractions. Case analysis on T.

$T = []$. Then $A = [s]$ and the second case holds by definition of δ.

$T = \mathsf{ret} :: T'$. Then (T, V) is reducible.

$T = \mathsf{var}\, n; P :: T'$. We have

$$^\circ[s] = \delta(\mathsf{var}\, n; P :: T')A = \delta T'(\delta(\mathsf{var}\, n; P)A) = \delta T'(\delta P(n :: A))$$

Since $n :: A$ is stuck, we know by Lemmas 24 and 25 that $[s]$ is stuck. Thus the third case holds.

$T = \mathsf{lam}\, Q; P :: T'$. Then (T, V) is reducible.

$T = \mathsf{app}; P :: T'$. Then $^\circ[s] = \delta(\mathsf{app}; P :: T')A = \delta T'(\delta(\mathsf{app}; P)A)$ and hence $A = t :: s :: A'$. Thus $V = R :: Q :: V'$. Thus (T, V) is reducible. □

Corollary 27. (Progress) *If $T, V \gg s$ and s is reducible, then (T, V) is reducible.*

Proof. Follows from Fact 26 using Fact 3. □

Theorem 28. (Naive Stack Machine to L) *The relation*

$$(T, V) \gg s \;:=\; \exists A.\ \delta V = {}^\circ A \;\wedge\; \delta T A = {}^\circ[s]$$

is a functional and computable refinement. Moreover, $([\gamma\, s\, \mathsf{ret}], []) \gg s$ holds for every term s.

Proof. The first claim follows with Facts 16, 27, 17, 23, and 15. The second claim follows with Fact 10. □

7 Closures

A closure is a pair consisting of a program and an environment. An environment is a list of closures representing a delayed substitution. With closures we can refine the naive stack machine so that no substitution operation is needed.

$$\begin{array}{rcll} e & : & \mathsf{Clo} ::= P/E & \textit{closure} \\ E, F, T, V & : & \mathcal{L}(\mathsf{Clo}) & \textit{environment} \end{array}$$

$$\begin{aligned}
(\mathsf{ret}/E) :: T,\ V &\ \succ_\tau\ T,\ V \\
(\mathsf{var}\, n; P/E) :: T,\ V &\ \succ_\tau\ (P/E) :: T,\ e :: V && \text{if } E[n] = {}^\circ e \\
(\mathsf{lam}\, Q; P/E) :: T,\ V &\ \succ_\tau\ (P/E) :: T,\ (Q/E) :: V \\
(\mathsf{app}; P/E) :: T,\ e :: (Q/F) :: V &\ \succ_\beta\ (Q/e :: F) :: (P/E) :: T,\ V
\end{aligned}$$

Fig. 3. Reduction rules of the closure machine

For the decompilation of closures into plain programs we define a *parallel substitution operation* P^k_W for programs (W ranges over lists of programs):

$$\begin{aligned}
\mathsf{ret}\,{}^k_W &:= \mathsf{ret} \\
(\mathsf{app}; P)^k_W &:= \mathsf{app}; P^k_W \\
(\mathsf{lam}\, Q; P)^k_W &:= \mathsf{lam}(Q^{\mathsf{S}k}_W); P^k_W \\
(\mathsf{var}\, n; P)^k_W &:= \text{if } \ n \geq k \wedge W[n-k] = {}^\circ Q \text{ then } \mathsf{lam}\, Q; P^k_W \text{ else } \mathsf{var}\, n; P^k_W
\end{aligned}$$

We will use the notation $W < 1 := \forall P \in W.\ P < 1$.

Fact 29. (Parallel Substitution)

1. $P^k_{[]} = P$.
2. *If* $P < k$ *and* $k \leq k'$, *then* $P < k'$.
3. *If* $P < k$, *then* $P^k_Q = P$.
4. *If* $W < 1$, *then* $P^k_{Q::W} = (P^{\mathsf{S}k}_W)^k_Q$.
5. *If* $W < 1$ *and* $P < |W| + k$, *then* $P^k_W < k$.

Note that Fact 29 (4) relates parallel substitution to single substitution .

We define a function $\delta_1 e$ of type $\mathsf{Clo} \to \mathsf{Pro}$ translating closures into programs:

$$\delta_1(P/E) \ := \ P^1_{\delta_1 @ E}$$

We also define an inductive *bound predicate* $e < 1$ for closures:

$$\frac{P < \mathsf{S}|E| \qquad E < 1}{P/E < 1} \qquad E < 1 \ := \ \forall e \in E.\ e < 1$$

Note the recursion through environments via the map function and via the membership predicate in the last two definitions.

Fact 30. *If* $e < 1$, *then* $\delta_1 e < 1$.

8 Closure Machine

We now refine the naive stack machine by replacing all programs on the control stack and the argument stack with closures, eliminating program substitution.

States of the closure machine are pairs

$$(T, V) \ : \ \mathcal{L}(\mathsf{Clo}) \times \mathcal{L}(\mathsf{Clo})$$

consisting of a *control stack* T and an *argument stack* V.

The *reduction rules of the closure machine* appear in Figure 3. The *variable rule* (second τ-rule) is new. It applies if the environment provides a closure for the variable. In this case the closure is pushed on the argument stack. We see this as delayed substitution of the variable. The variable rule will be simulated with the lambda rule of the naive stack machine.

The *application rule* (β-rule) takes two closures e and Q/F from the argument stack and pushes the closure $Q/e :: F$ on the control stack, which represents the result of β-reducing the abstraction represented by Q/F with the argument e.

We will show that the closure machine implements the naive stack machine correctly provided there are no free variables.

There is the complication that the closures on the control stack must be closed while the closures on the argument stack are allowed to have the free variable 0 representing the argument to be supplied by the application rule.

We define *closed states* of the closure machine as follows:

$$\begin{aligned} P/E < 0 \ &:= \ P < |E| \ \wedge \ E < 1 \\ T < 0 \ &:= \ \forall e \in T.\ e < 0 \\ \mathsf{closed}\,(T, V) \ &:= \ T < 0 \ \wedge \ V < 1 \end{aligned}$$

We define a function $\delta_0 e$ of type $\mathsf{Clo} \to \mathsf{Pro}$ for decompiling closures on the task stack:

$$\delta_0(P/E) \ := \ P^0_{\delta_1 @ E}$$

We can now define the *refinement relation* between states of the closure machine and states of the naive stack machine:

$$(T, V) \gg \sigma \ := \ \mathsf{closed}\,(T, V) \ \wedge \ (\delta_0 @ T, \delta_1 @ V) = \sigma$$

We show that $(T, V) \gg \sigma$ is a refinement.

Fact 31. *$(T, V) \gg \sigma$ is functional and computable.*

Fact 32. (Progress) *Let $(\delta_0 @ T, \delta_1 @ V)$ be reducible. Then (T, V) is reducible.*

Fact 33. *Let (T, V) be closed and $(T, V) \succ (T', V')$. Then (T', V') is closed.*

Fact 34. (τ-Simulation) *Let $(T, V) \succ_\tau (T', V')$.*
Then $(\delta_0 @ T, \delta_1 @ V) \succ_\tau (\delta_0 @ T', \delta_1 @ V')$.

Fact 35. (β-Simulation) *Let (T, V) be closed and $(T, V) \succ_\beta (T', V')$.*
Then $(\delta_0 @ T, \delta_1 @ V) \succ_\beta (\delta_0 @ T', \delta_1 @ V')$.

Proof. Follows with Facts 29 (4) and 30. □

Theorem 36. (Closure Machine to Naive Stack Machine) *The relation*

$$(T, V) \gg \sigma \;:=\; \mathsf{closed}\,(T, V) \;\wedge\; (\delta_0 @ T, \delta_1 @ V) = \sigma$$

is a functional and computable refinement. Moreover, $([P/[]], []) \gg ([P], [])$ *holds for every closed program* P*.*

Proof. The first claim follows with Facts 31 32, 33, 34, and 35. The second claim follows with Fact 29 (1). □

Note that Theorems 28 and 36 Facts 9 and 14 yield a refinement to L.

9 Codes

If a state is reachable from an initial state in the closure machine, all its programs are subprograms of programs in the initial state. We can thus represent programs as addresses of a fixed code, providing structure sharing for programs.

A code represents a program such that the commands and subprograms of the program can be accessed through addresses. We represent codes abstractly with a type Code, a type PA of program addresses, and two functions $\#$ and φ as follows:

$$\begin{array}{rcll} C & : & \mathsf{Code} & \textit{code} \\ p, q, r & : & \mathsf{PA} & \textit{program address} \\ \# & : & \mathsf{PA} \to \mathsf{PA} & \\ & & \mathsf{Com} \;:=\; \mathsf{ret} \mid \mathsf{var}\, n \mid \mathsf{lam}\, p \mid \mathsf{app} & \textit{command} \\ \varphi & : & \mathsf{Code} \to \mathsf{PA} \to \mathcal{O}(\mathsf{Com}) & \end{array}$$

Note that commands are obtained with a nonrecursive inductive type Com. The function $\#$ increments a program address, and the φ yields the command for a valid program address. We will use the notation $C[p] := \varphi C p$. We fix the semantics of codes with a relation $p \gg_C P$ relating program addresses with programs:

$$\frac{C[p] = {}^\circ\mathsf{ret}}{p \gg_C \mathsf{ret}} \qquad \frac{C[p] = {}^\circ\mathsf{var}\, n \qquad \#p \gg_C P}{p \gg_C \mathsf{var}\, n; P}$$

$$\frac{C[p] = {}^\circ\mathsf{lam}\, q \qquad q \gg_C Q \qquad \#p \gg_C P}{p \gg_C \mathsf{lam}\, Q; P} \qquad \frac{C[p] = {}^\circ\mathsf{app} \qquad \#p \gg_C P}{p \gg_C \mathsf{app}; P}$$

Fact 37. *The relation* $p \gg_C P$ *is functional.*

We obtain one possible implementation of codes as follows:

$$\begin{array}{rcl} \mathsf{PA} & := & \mathsf{N} \\ \mathsf{Code} & := & \mathcal{L}(\mathsf{Com}) \\ \#n & := & \mathsf{S}\, n \end{array} \qquad \begin{array}{rcll} \varphi C n & := & \mathsf{lam}\,(n + k) & \text{if } C[n] = \mathsf{lam}\, k \\ \varphi C n & := & C[n] & \text{otherwise} \end{array}$$

For this realisation of codes we define a function $\psi : \mathsf{Pro} \to \mathcal{L}(\mathsf{Com})$ compiling programs into codes as follows:

$$\begin{aligned} \psi\ \mathsf{ret} &:= [\mathsf{ret}] & \psi(\mathsf{lam}\, Q; P) &:= \mathsf{lam}\,(\mathsf{S}|\psi P|) :: \psi P +\!\!+ \psi Q \\ \psi(\mathsf{var}\, n; P) &:= \mathsf{var}\, n :: \psi P & \psi(\mathsf{app}; P) &:= \mathsf{app} :: \psi P \end{aligned}$$

The linear representation of a program $\mathsf{lam}\, Q; P$ provided by ψ is as follows: First comes a command $\mathsf{lam}\, k$, then the commands for P, and finally the commands for Q (i.e., the commands for the body Q come after the commands for the continuation P). The number k of the command $\mathsf{lam}\, k$ is chosen such that $n+\mathsf{S}k$ is the address of the first command for Q if n is the address of the command $\mathsf{lam}\, k$.

Fact 38. *$|C_1| \gg_{C_1 +\!\!+ \psi P +\!\!+ C_2} P$. In particular, $0 \gg_{\psi P} P$.*

10 Heaps

A heap contains environments accessible through addresses. This opens the possibility to share the representation of environments.

We model heaps abstractly based on an assumed code structure. We start with types for heaps and heap addresses and a function get accessing heap addresses:

$$\begin{aligned} H &: \mathsf{Heap} && \textit{heap} \\ a, b, c &: \mathsf{HA} && \textit{heap address} \\ g &: \mathsf{HC} := \mathsf{PA} \times \mathsf{HA} && \textit{heap closure} \\ & \mathsf{HE} := \mathcal{O}(\mathsf{HC} \times \mathsf{HA}) && \textit{heap environment} \\ \mathsf{get} &: \mathsf{Heap} \to \mathsf{HA} \to \mathcal{O}(\mathsf{HE}) \end{aligned}$$

We will use the notation $H[a] := \mathsf{get}\, H\, a$. We fix the semantics of heaps with an inductive relation $a \gg_H E$ relating heap addresses with environments:

$$\frac{H[a] = {}^{\circ}\emptyset}{a \gg_H []} \qquad \frac{H[a] = {}^{\circ\circ}((p, b), c) \qquad p \gg_C P \qquad b \gg_H F \qquad c \gg_H E}{a \gg_H (P/F) :: E}$$

Fact 39. *The relation $a \gg_H E$ is functional.*

We also need an operation $\mathsf{put} : \mathsf{Heap} \to \mathsf{HC} \to \mathsf{HA} \to \mathsf{Heap} \times \mathsf{HA}$ extending a heap with an environment. Note that put yields the extended heap and the address of the extending environment. We use the notation

$$H \subseteq H' := \forall a.\ H[a] \neq \emptyset \to H[a] = H'[a]$$

to say that H' is an *extension* of H. We fix the semantics of put with the following requirement:

HR If $\mathsf{put}\, H\, g\, a = (H', b)$, then $H'[b] = (g, a)$ and $H \subseteq H'$.

Fact 40. *If $H \subseteq H'$ and $a \gg_H E$, then $a \gg_{H'} E$.*

We define a relation $g \gg_H e$ relating heap closures with proper closures:

$$(p, a) \gg_H (P, E) \;:=\; p \gg_C P \wedge a \gg_H E$$

Fact 41. *If $H \subseteq H'$ and $g \gg_H e$, then $g \gg_{H'} e$.*

We define a *lookup function* $H[a, n] : \mathcal{O}(\mathsf{HC})$ yielding the heap closure appearing at position n of the heap environment designated by a in H:

$$\begin{aligned} H[a, 0] &:= {}^\circ(p, b) && \text{if } H[a] := {}^\circ((p, b), c) \\ H[a, \mathsf{S}\, n] &:= H[c, n] && \text{if } H[a] := {}^\circ((p, b), c) \end{aligned}$$

Fact 42. *Let $a \gg_H E$. Then:*

1. *If $E[n] = {}^\circ e$, then $H[a, n] = {}^\circ g$ and $g \gg_H e$ for some g.*
2. *If $H[a, n] = {}^\circ g$, then $E[n] = {}^\circ e$ and $g \gg_H e$ for some e.*

Here is one possible implementation of heaps:

$$\begin{aligned} \mathsf{HA} &:= \mathsf{N} \\ \mathsf{Heap} &:= \mathcal{L}(\mathsf{HC} \times \mathsf{HA}) \\ \mathsf{get}\; H\, 0 &:= {}^\circ\emptyset \\ \mathsf{get}\; H\, (\mathsf{S}\, n) &:= {}^{\circ\circ}(g, a) && \text{if } H[n] = {}^\circ(g, a) \\ \mathsf{put}\; H\, g\, a &:= (H +\!\!+ [(g, a)],\; \mathsf{S}\, |H|) \end{aligned}$$

Note that with this implementation the address 0 represents the empty environment in every heap.

Given that Coq admits only structurally recursive functions, writing a function computing $a \gg_H E$ is not straightforward. The problem goes away if we switch to a step-indexed function computing $a \gg_H E$.

11 Heap Machine

The heap machine refines the closure machine by representing programs as addresses into a fixed code and environments as addresses into heaps that reside as additional component in the states of the heap machine.

We assume a code structure providing types Code and PA, a code $C : \mathsf{Code}$, and a heap structure providing types Heap and HA. *States of the heap machine* are triples

$$(T, V, H) \;:\; \mathcal{L}(\mathsf{HC}) \times \mathcal{L}(\mathsf{HC}) \times \mathsf{Heap}$$

consisting of a control stack, an argument stack, and a heap. The *reduction rules of the heap machine* appear in Figure 4. They refine the reduction rules of the closure machine as one would expect.

$$
\begin{array}{rll}
(p,a) :: T,\ V,\ H & \succ_\tau\ T,\ V,\ H & \text{if } C[p] = {}^\circ\mathsf{ret} \\
(p,a) :: T,\ V,\ H & \succ_\tau\ (\#p,a) :: T,\ g :: V,\ H & \text{if } C[p] = {}^\circ\mathsf{var}\, n \\
 & & \text{and } H[a,n] = {}^\circ g \\
(p,a) :: T,\ V,\ H & \succ_\tau\ (\#p,a) :: T,\ (q,a) :: V,\ H & \text{if } C[p] = {}^\circ\mathsf{lam}\, q \\
(p,a) :: T,\ g :: (q,b) :: V,\ H & \succ_\beta\ (q,c) :: (\#p,a) :: T,\ V,\ H' & \text{if } C[p] = {}^\circ\mathsf{app} \\
 & & \text{and } \mathsf{put}\, H\, g\, b = {}^\circ(H', c)
\end{array}
$$

Fig. 4. Reduction rules of the heap machine

Note that the application rule is the only rule that allocates new environments on the heap. This is at first surprising since with practical machines (e.g., FAM and ZINC) heap allocation takes place when lambda commands are executed. The naive allocation policy of our heap machine is a consequence of the naive realisation of the lambda command in the closure machine, which is common in formalisations of the SECD machine. Given our refinement approach, smart closure allocation would be prepared at the level of the naive stack machine with programs that have explicit commands for accessing and constructing closure environments.

Proving correctness of the heap machine is straightforward:

Theorem 43. (Heap Machine to Closure Machine) *Let a code structure, a code C, and a heap structure be fixed. Let $T \gg_H \dot{T}$ and $V \gg_H \dot{V}$ denote the pointwise extension of $g \gg_H E$ to lists. Then the relation*

$$(T, V, H) \gg (\dot{T}, \dot{V}) \ :=\ T \gg_H \dot{T} \ \wedge\ V \gg_H \dot{V}$$

is a functional refinement. Moreover, $([(p,a)], [], H) \gg ([P/[]], [])$ for all p, a, H, and P such that $p \gg_C P$ and $a \gg_H []$.

Proof. Follows with Facts 37, 39, 40, 41, and 42. Straightforward. □

Using the refinement from the closure machine to L, Theorem 43 and Fact 9 we obtain a refinement from the Heap Machine to L. If we instantiate the heap machine with the realisation of codes from Section 9 and the realisation of heaps from Section 10 we obtain a function compiling closed terms into initial states. Moreover, given a function computing $a \gg_H E$, we can obtain a decompiler for the states of the heap machine.

12 Final Remarks

The tail call optimisation can be realised in our machines and accommodated in our verifications. For this subprograms app; ret are executed such that no trivial continuation (i.e., program ret) is pushed on the control stack.

The control stack may be merged with the argument stack. If this is done with explicit frames as in the SECD machine, adapting our verification should be straightforward. There is also the possibility to leave frames implicit as in the

modern SECD machine. This will require different decompilation functions and concomitant changes in the verification.

We could also switch to a λ-calculus with full substitution. This complicates the definition of substitution and the basic substitution lemmas but has the pleasant consequence that we can drop the closedness constraints coming with the correctness theorems for the closure and heap machines. The insight here is that a closure machine implements full substitution. With full substitution we may reduce β-redexes where the argument is a variable and show a substitutivity property for small-step reduction.

References

1. B. Accattoli, P. Barenbaum, and D. Mazza. Distilling abstract machines. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 363–376, 2014.
2. C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.
3. M. Biernacka, W. Charatonik, and K. Zielinska. Generalized refocusing: From hybrid strategies to abstract machines. In *LIPIcs*, volume 84. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2017.
4. M. Biernacka and O. Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic (TOCL)*, 9(1):6, 2007.
5. L. Cardelli. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 208–217. ACM, 1984.
6. G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.
7. P. Crégut. An abstract machine for lambda-terms normalization. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 333–340, 1990.
8. U. Dal Lago and S. Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008.
9. O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. *BRICS Report Series*, 11(26), 2004.
10. M. Felleisen and D. P. Friedman. *Control Operators, the SECD-machine, and the λ-calculus*. Indiana University, Computer Science Department, 1986.
11. Y. Forster, F. Kunze, and M. Roth. The strong invariance thesis for a λ-calculus. *Workshop on Syntax and Semantics of Low-Level Languages (LOLA)*, 2017.
12. Y. Forster and G. Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. In *Interactive Theorem Proving - 8th International Conference*, pages 189–206. Springer, LNCS 10499, 2017.
13. T. Hardin, L. Maranget, and B. Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–176, 1998.
14. P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
15. X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report, INRIA, 1990.
16. X. Leroy. Functional programming languages, Part II: Abstract machines, the Modern SECD. Lectures on Functional Programming and Type Systems, MPRI course 2-4, slides and Coq developments, `https://xavierleroy.org/mpri/2-4/`, 2016.

17. X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.
18. G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.
19. J. D. Ramsdell. The tail-recursive SECD machine. *Journal of Automated Reasoning*, 23(1):43–62, 1999.
20. M. Rittri. *Proving the correctness of a virtual machine by a bisimulation.* Chalmers University and University of Göteborg, 1988. Licentiate thesis.
21. W. Swierstra. From mathematics to abstract machine: A formal derivation of an executable Krivine machine. In *Proceedings Fourth Workshop on Mathematically Structured Functional Programming*, pages 163–177, 2012.
22. The Coq Proof Assistant. `http://coq.inria.fr`, 2018.