A Certified Extension of the Krivine Machine for a Call-by-Name Higher-Order Imperative Language

Leonardo Rodríguez, Daniel Fridlender, and Miguel Pagano

Universidad Nacional de Córdoba, FaMAF
Córdoba, Argentina
{lrodrig2,fridlend,pagano}@famaf.unc.edu.ar

Abstract

In this paper we present a compiler that translates programs from an imperative higher-order language into a sequence of instructions for an abstract machine. We consider an extension of the Krivine machine for the call-by-name lambda calculus, which includes strict operators and imperative features. We show that the compiler is correct with respect to the big-step semantics of our language, both for convergent and divergent programs.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases abstract machines, compiler correctness, big-step semantics

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.230

1 Introduction

Important advancements have been made during the last decade in the field of compiler certification; the project CompCert [19, 20] being the most significant achievement, since it deals with a realistic compiler for the C language. It is still an active research topic open for new techniques and experimentation. In this work we report our experiments in the use of known techniques to prove the correctness of a compiler for a call-by-name lambda calculus extended with strict operators and imperative features.

Most compilers are multi-pass, meaning that the process of translation usually involves successive symbolic manipulations from the source program to the target program. The compilation of a source program is often carried as a sequence of translations through several intermediate languages, each one closer to assembly code than the previous one. One common intermediate language consists of the instructions for some abstract machine; they are useful because they hide low-level details of concrete hardware, but also permit step-by-step execution of programs. At this level one can discover possible sources of optimization in the compilation.

Historically, several abstract machines have been developed and studied. Perhaps the best known ones are the SECD [15] machine and the CAM [8] machine, both for the call-by-value lambda calculus, and the Krivine machine [14] together with the G-machine [23], for call-by-name. We refer to Diehl *et al.* [11] for a, slightly dated, bibliographical review about different abstract machines. In this article we use the Krivine machine as the target of our compiler.

The Krivine machine has a very strong property: each transition rule of the machine corresponds directly to a reduction rule in the small-step semantics of the lambda calculus. This property is very useful to prove the correctness of the machine, since there is a relation of simulation between the machine and the calculus. This correspondence is, however, very difficult to maintain when one extends the source language, for example, by including

19th International Conference on Types for Proofs and Programs (TYPES 2013).
Editors: Ralph Matthes and Aleksy Schubert; pp. 230–250

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

imperative features. Indeed, the conventional small step semantics of the calculus and the usual transitions of the machine might not correspond, as there can be transition sequences which do not necessarily simulate reduction steps in the source language. However, when the goal is to get a proof of correctness of the compiler, simulation is a property stronger than the one we actually need.

A different way of proving the correctness of a compiler is by using big-step semantics [22]; in this setting one proves that if a term t evaluates to a value v, then, when the machine executes the code obtained in the compilation of t, it must stop in a state related to v, for some appropriate definition of the relation between the value and the final state. One benefit of this approach is that it can be adapted to handle the correctness of divergent programs by a coinductive definition of the big-step semantics and of the transition relation of the machine.

In this paper we present a compiler which translates programs from an imperative higher-order language into a sequence of instructions for an abstract machine. We consider an extension of the Krivine machine for the call-by-name lambda calculus, which includes strict operators and imperative features. We show that the compiler is correct with respect to the big-step semantics of our language, both for convergent and divergent programs. We formalized the compiler and the proof of its correctness in the Coq proof assistant; the source code is available at `http://cs.famaf.unc.edu.ar/~leorodriguez/compilercorrectness/`.

Except for the absence of the type system, the programming language we consider in this paper has all of the features that Reynolds [26] described as the essence of Algol-like languages: deallocation is automatically done at the end of a block (stack discipline), only the imperative fragment of the language can have side effects, and it includes a call-by-name lambda calculus as a procedure mechanism. We consider this work to be a step towards proving the correctness for a compiler designed by Reynolds [25] for Algol-like languages.

The paper is organized as follows: in Sec. 2 we analyze the calculus of closures and the Krivine machine; and we revisit the proof of correctness of the compiler with respects to the small-step semantics of the calculus. In the following sections, we gradually extend the source language and the machine to cope with the extensions: first we add, in Sec. 3, strict operators and then, in Sec. 4, imperative features. We also prove the compiler correctness with respects to the big-step semantics of those languages.

2 Call-by-name lambda calculus

In this section we revisit the call-by-name lambda calculus, as a calculus of closures, and the Krivine abstract machine. In this simple setting we briefly explain the methodology used to prove the correctness of the compilation function. The proof exploits the fact that the transitions of the machine simulate the small-step semantics of the calculus.

2.1 Calculus of closures

Our high-level language is the lambda calculus with de Bruijn indices, but the operational semantics is given for an extension, proposed by Biernacka and Danvy [4], of Curien's calculus of closures [9]. This calculus is an early version of the lambda calculus with explicit substitutions. The main difference of this calculus with respect to the usual presentation of the lambda calculus is the way in which substitution are treated. In the latter, substitution is a meta-level operation, which in zero steps solves a β-redex. In contrast, in the calculus of closures substitutions are represented as terms equipped with environments – these pairs are

called closures – and new rules are added to perform the reduction of a simple redex. A closed lambda term t can be seen as a closure just by pairing it with the empty environment: $t\,[[]]$.

▶ **Definition 1** (Terms and closures).

Terms	$\Lambda \ni t,\ t'$	$::=$	$\lambda\, t$	Abstraction
		$\mid$	$t\ t'$	Application
		$\mid$	$\overline{n}$	Variables
Closures	$C \ni c,\ c'$	$::=$	$t\,[e] \mid c\ c'$	
Environments	$E \ni e$	$::=$	$[] \mid c :: e$	

Terms are the usual ones of the lambda calculus with de Bruijn indices representing variables. We will use some notational convention for meta-variables, besides those used in the grammar we use $n \in \mathbb{N}$ and the overline is just to see them as variables. In Curien's work only the first production for closures is present; the second one is Biernacka and Danvy's extension and allows to define a small-step operational semantics. An environment is a sequence of closures, and represents a substitution for all of the free variables of a term. The reduction rules for the calculus of closures are given by the following rewriting system. Notice that the side condition in (Var) can be removed if one sticks to closed closures.

▶ **Definition 2** (Reduction rules).

$$
\begin{array}{lrcll}
(\beta) & (\lambda\, t)\,[e]\ c & \to & t\,[c :: e] & \\
(\textsc{App}) & (t\ t')\,[e] & \to & t\,[e]\ t'\,[e] & \\
(\textsc{Var}) & \overline{n}\,[e] & \to & e.n & \text{if } n < |e|
\end{array}
\qquad (\nu)\quad \frac{c_1 \to c_1'}{c_1\ c_2 \to c_1'\ c_2}
$$

These reduction rules evaluate a closed term up to a weak head normal form; i.e. values are closures of the form $\lambda\, t\,[e]$. The (β) rule associates the argument of the redex with the first free variable of the body of the abstraction. The rule (App) just propagates the environment inside the term. Finally, the rule (Var) performs a lookup inside the environment, and reduces to the closure associated with the variable position. The (ν) rule allows the reduction of the left part of an application until getting an abstraction.

It can be easily proved that the semantics is deterministic:

▶ **Lemma 3** (Determinism). *If $c \to c_1$ and $c \to c_2$, then $c_1 = c_2$, for all closures c, c_1 and c_2.*

2.2 The Krivine machine

Now we turn to the target of our compiler: the Krivine abstract machine [14]. This machine has three components: the code, the environment, and the stack. We also have machine-level closures $\gamma = (i, \eta)$, which are pairs of code i together with an environment η.

▶ **Definition 4** (Abstract Machine). A configuration w is a triple $i \mid \eta \mid s$, where

Code:	$I \ni i, i'$	$::=$	$\mathsf{Grab} \triangleright i$
		$\mid$	$\mathsf{Push}\ i \triangleright i'$
		$\mid$	$\mathsf{Access}\ n$
Environments:	$H \ni \eta, \eta'$	$::=$	$[] \mid (i, \eta') :: \eta$
Stacks:	$S \ni s$	$::=$	$[] \mid (i, \eta) :: s$

The environment and the stack are just a list of machine-level closures. We use the following notation for lists: [] denotes the empty list, append is $\_ :: \_$ as in ML tradition;

the length of the list xs is $|xs|$ and if $n < |xs|$ then projecting the nth-element from xs is written $xs.n$. There are only three instructions, whose action is defined by the following three transition rules.

▶ **Definition 5** (Transition of the machine).

$$\begin{array}{rlll} \mathsf{Grab} \triangleright i \mid \eta \mid (i',\eta') :: s & \longmapsto & i \mid (i',\eta') :: \eta \mid s & \\ \mathsf{Push}\ i' \triangleright i \mid \eta \mid s & \longmapsto & i \mid \eta \mid (i',\eta) :: s & \\ \mathsf{Access}\ n \mid \eta \mid s & \longmapsto & i' \mid \eta' \mid s & \text{if } n < |\eta| \text{ and } \eta.n = (i',\eta') \end{array}$$

The instruction $\mathsf{Grab} \triangleright i$ takes a closure from the top of the stack, puts it in the environment, and then continues with the execution of i. The instruction $\mathsf{Push}\ i' \triangleright i$ pushes a closure (i', η) (where η is the current environment) in the top of the stack and continues with the execution of i. Finally, the instruction $\mathsf{Access}\ n$ starts executing the closure associated with the position n inside the environment.

2.3 Compilation and correctness

The next step is the definition of the translation of terms into code. The compiler function is denoted with $[\![\_]\!]$ and it is easily defined by induction on terms. We also define a decompilation function denoted $\{\!| \_ |\!\}$ which is clearly the inverse of the compilation. This function is useful to describe some properties of the machine which in turn are helpful to prove the correctness of the compiler.

▶ **Definition 6** (Compilation and decompilation of terms).

$$\begin{array}{rlrl} [\![\_]\!] : \Lambda \to I & & \{\!| \_ |\!\} : I \to \Lambda & \\ [\![\lambda\, t]\!] = \mathsf{Grab} \triangleright [\![t]\!] & & \{\!| \mathsf{Grab} \triangleright i |\!\} = \lambda\, \{\!| i |\!\} & \\ [\![t\ t']\!] = \mathsf{Push}\ [\![t']\!] \triangleright [\![t]\!] & & \{\!| \mathsf{Push}\ i' \triangleright i |\!\} = \{\!| i |\!\}\ \{\!| i' |\!\} & \\ [\![\overline{n}]\!] = \mathsf{Access}\ n & & \{\!| \mathsf{Access}\ n |\!\} = \overline{n} & \end{array}$$

We homomorphically extend the definition of the decompilation function to machine-level closures and environments:

▶ **Definition 7** (Decompilation of closures and environments).

$$\begin{array}{c} \{\!| \_ |\!\}^c : I \times H \to C \\ \{\!| (i,\eta) |\!\}^c = \{\!| i |\!\}\,[\{\!| \eta |\!\}^e] \end{array} \qquad \begin{array}{c} \{\!| \_ |\!\}^e : H \to E \\ \{\!| [] |\!\}^e = [] \\ \{\!| (i',\eta') :: \eta |\!\}^e = \{\!| (i',\eta') |\!\}^c :: \{\!| \eta |\!\}^e \end{array}$$

We also need to decompilate configurations of the machine into source-level closures. To decompile a configuration $(i \mid \eta \mid s)$ we successively apply the decompilation of the current closure (i, η) to the decompilation of every closure in s.

▶ **Definition 8** (Decompilation of configurations). Let $s = \gamma_1, \ldots, \gamma_n$, then

$$\{\!| (i \mid \eta \mid s) |\!\} = (\ldots (\{\!| (i,\eta) |\!\}^c\, \{\!| \gamma_1 |\!\}) \ldots \{\!| \gamma_n |\!\})$$

We now continue by presenting some well-known lemmas about the behaviour of the machine with respect to the small-step semantics of the calculus. First, we state that every transition of the machine simulates a reduction step in the calculus:

▶ **Lemma 9** (Simulation). *If* $w \longmapsto w'$, *then* $\{\!| w |\!\} \to \{\!| w' |\!\}$.

There is another useful property of the machine: if a configuration w decompiles to the closure c, and c can reduce, then the machine does not stop but makes a transition from w.

▶ **Lemma 10** (Progress). *If $\{\!|\, w \,|\!\} \to c'$, then there exists a configuration w' such that $w \longmapsto w'$.*

We can use Lemma 9 to obtain a stronger version of the previous lemma that better characterizes the configuration to which the machine makes the transition:

▶ **Lemma 11** (Progress and simulate). *If $\{\!|\, w \,|\!\} \to c'$, then there exists a configuration w' such that $w \longmapsto w'$ and $\{\!|\, w' \,|\!\} = c'$.*

Proof. By the progress lemma we state the existence of w', then by the simulation lemma we know that $\{\!|\, w \,|\!\} \to \{\!|\, w' \,|\!\}$ and then we conclude $\{\!|\, w' \,|\!\} = c'$ using the fact that the semantics is deterministic. ◀

The Lemma 11 can be easily extended to the reflexive-transitive closure of the small-step reduction and the machine transitions:

▶ **Lemma 12.** *If $\{\!|\, w \,|\!\} \to^* c'$, then there exists a configuration w' such that $w \longmapsto^* w'$ and $\{\!|\, w' \,|\!\} = c'$.*

We are particularly interested in the case in which the reduction sequence of the previous lemma reaches an irreducible closure c'. In this case, we expect the machine to stop in an irreducible configuration w' which decompiles to c'. We say that a configuration is irreducible if the machine can not perform any transition from it.

▶ **Lemma 13.** *If $\{\!|\, w \,|\!\} \to^* c'$ and c' is irreducible, then there exists a configuration w' such that $w \longmapsto^* w'$, $\{\!|\, w' \,|\!\} = c'$ and w' is irreducible.*

Proof. It is a consequence of the Lemma 12 and the simulation lemma. It is important to note that the proof of this lemma can be done constructively since the property of being irreducible is decidable. ◀

Now we can use these results to prove the correctness of our compiler. The following lemma states the correctness of the compilation of a closed term whose reduction sequence reaches an irreducible closure:

▶ **Lemma 14** (Correctness for convergent closed terms). *If $t\,[[]] \to^* c'$ and c' is irreducible, then there exists a configuration w' such that $([\![\, t \,]\!] \mid [] \mid []) \longmapsto^* w'$, $\{\!|\, w' \,|\!\} = c'$ and w' is irreducible.*

Proof. This lemma is an instance of Lemma 13 since we have $\{\!|\, ([\![\, t \,]\!] \mid [] \mid []) \,|\!\} = t\,[[]]$. ◀

If the reduction sequence of a closed term does not terminate (it does not reach an irreducible closure), then the execution of the compiled code must diverge. We can capture the notion of divergence for both reduction and execution with the following coinductive rules, the double line indicates that the rules are to be interpreted coinductively:

▶ **Definition 15** (Divergence of reduction and execution).

$$\frac{c \to c' \qquad c' \overset{\infty}{\to}}{c \overset{\infty}{\to}} \qquad \frac{w \longmapsto w' \qquad w' \longmapsto^{\infty}}{w \longmapsto^{\infty}}$$

The following lemma states that the divergence of the reduction sequence forces the machine to diverge:

▶ **Lemma 16** (Progress forever). *If* $\{\!|\, w \,|\!\} \to^\infty$*, then* $w \longmapsto^\infty$.

Proof. The proof is obtained by coinduction and using Lemma 11. ◀

Finally, the correctness of the compilation of divergent closed terms can be stated as follows:

▶ **Lemma 17** (Correctness for divergent closed terms). *If* $t\,[[]] \to^\infty$*, then* $(\llbracket t \rrbracket \mid [] \mid []) \longmapsto^\infty$.

In general, obtaining a proof of compiler correctness with respect to the small-step semantics of the source language is a very complicated task. In this section, we avoided some of those complications due to the simplicity of the language, for example, we did not have to define a bisimilarity relation as in [12, 27], but instead we used a decompilation *function*.

For more sophisticated languages, the big-step semantics leads often to simpler proofs of compiler correctness [22]. In the following sections, we use big-step semantics to prove the correctness of the compilation of two languages: a call-by-name lambda calculus with strict operators and an imperative higher-order language. We follow an approach inspired in the work of Leroy [22] (a proof of compiler correctness for the call-by-value lambda calculus).

3 Call-by-name lambda calculus with strict operators

In this section we extend the source language with constants and a strict binary operator; the language is specified by a big-step semantics. Then we present the abstract machine and the corresponding compiler. The correctness of the compiler for convergent terms is a ternary relation involving terms, their values, and the execution of the abstract machine. Leroy defined this relation by compiling values and proving that the execution of the compilation of a term leads to the compilation of the value. Following the same path for our language would impose an artificial set of transition rules for the machine; we avoid this by defining a binary relation between values and configurations.

3.1 The calculus

We now extend the source language with integer constants and the addition operator. Everything in this section can be straightforwardly extended to a language with several strict binary operators, but for the sake of concreteness we restrict our exposition to addition.

▶ **Definition 18** (Terms and closures).

Terms	$\Lambda \ni t,\ t'$	$::=$	$\lambda\, t$	Abstraction
		$\mid$	$t\ t'$	Application
		$\mid$	$\overline{n}$	Variables
		$\mid$	$\underline{k}$	Constants
		$\mid$	$t + t'$	Addition
Closures	$C \ni c$	$::=$	$t\,[e]$	
Environments	$E \ni e$	$::=$	$[] \mid c :: e$	
Values	$V \ni v$	$::=$	$(\lambda\, t)\,[e] \mid k$	

The new terms are constants $\underline{k}$, for $k \in \mathbb{N}$, and addition. Notice that there is no application of closures, this is a consequence of passing from small-step reductions to a big-step semantics, where intermediate computations steps cannot be observed. Values are the canonical forms which are the result of the evaluation of a term: an abstraction with its environment, and a constant. We define now the big-step semantics of the language. The evaluation of a term t in the environment e to the value v is denoted by $e \vdash t \Rightarrow v$.

▶ **Definition 19** (Big-step semantics)**.**

$$(\textsc{Abs})\ \frac{}{e \vdash \lambda\, t \Rightarrow (\lambda\, t)\,[e]} \qquad (\textsc{Const})\ \frac{}{e \vdash \underline{k} \Rightarrow k}$$

$$(\textsc{App})\ \frac{e \vdash t_1 \Rightarrow (\lambda\, t)\,[e'] \qquad t_2\,[e] :: e' \vdash t \Rightarrow v}{e \vdash t_1\ t_2 \Rightarrow v} \qquad (\textsc{Var})\ \frac{e' \vdash t' \Rightarrow v}{e \vdash \overline{n} \Rightarrow v}\ e.n = t'\,[e']$$

$$(\textsc{Add})\ \frac{e \vdash t_1 \Rightarrow k \qquad e \vdash t_2 \Rightarrow k'}{e \vdash t_1\ +\ t_2 \Rightarrow k\ +\ k'}$$

The rules for abstractions and constants are trivial, since canonical forms evaluate to themselves. Notice that in the rule of the application the argument is not evaluated, but it is used to extend the environment during the evaluation of the body of the abstraction. In order to evaluate a variable one must do a lookup operation inside the environment, and start the evaluation of the corresponding closure. The rule for addition is quite conventional: one must first evaluate the two arguments and then obtain the final value by performing the addition of the two constants.

Now we show two simple examples of evaluation of terms:

▶ **Example 20.** A term that evaluates to an abstraction (partial application).

$$\frac{e \vdash (\lambda\,\lambda\, t) \Rightarrow (\lambda\,\lambda\, t)\,[e] \qquad t'\,[e] :: e \vdash \lambda\, t \Rightarrow (\lambda\, t)\,[t'\,[e] :: e]}{e \vdash (\lambda\,\lambda\, t)\ t' \Rightarrow (\lambda\, t)\,[t'\,[e] :: e]}$$

▶ **Example 21.** A term that evaluates to a constant.

$$\dfrac{\dfrac{}{e \vdash \lambda\,(\overline{0}\ +\ \underline{3}) \Rightarrow \lambda\,(\overline{0}\ +\ \underline{3})\,[e]} \qquad \dfrac{\dfrac{e \vdash \underline{2} \Rightarrow 2}{\underline{2}\,[e] :: e \vdash \overline{0} \Rightarrow 2} \qquad \dfrac{}{\underline{2}\,[e] :: e \vdash \underline{3} \Rightarrow 3}}{\underline{2}\,[e] :: e \vdash \overline{0}\ +\ \underline{3} \Rightarrow 5}}{e \vdash (\lambda\,(\overline{0}\ +\ \underline{3}))\ \underline{2} \Rightarrow 5}$$

3.2 A call-by-name machine with strict operations

The Krivine machine follows the call-by-name strategy, this implies that the argument of an application is evaluated only when it is needed. But if we want to incorporate some strict operation, like addition, we need a way to force the evaluation of the arguments before computing the operation. A known solution, cf. [28], to this issue is a data structure called *frame*, which is intended to store the code needed to compute the arguments along with the temporal values generated in the computation. The different components of the machine are defined as follows:

▶ **Definition 22** (Abstract machine).

$$
\begin{array}{lllll}
\text{Code:} & I \ni i, i' & ::= & \mathsf{Grab} \triangleright i \\
 & & \mid & \mathsf{Push}\ i \triangleright i' \\
 & & \mid & \mathsf{Access}\ n \\
 & & \mid & \mathsf{Const}\ k \\
 & & \mid & \mathsf{Add} \\
\text{Closures:} & \Gamma \ni \gamma & ::= & (i, \eta) \\
\text{Environments:} & H \ni \eta & ::= & [] \mid \gamma :: \eta \\
\text{Stack values:} & M \ni \mu & ::= & \gamma \mid [+ \bullet \gamma] \mid [+\ k\ \bullet] \\
\text{Stacks:} & S \ni s & ::= & [] \mid \mu :: s \\
\text{Configurations:} & W \ni w & ::= & (\gamma,\ s)
\end{array}
$$

As in the previous section, a closure is composed by a code together with its environment. The environment is a list of closures and a stack is a list of *stack values* which may be closures or frames. The frame $[+ \bullet \gamma]$ stores the code needed to compute the second argument of the addition, this closure remains stored in the stack while the first argument is being computed. On the other hand, the frame $[+\ k\ \bullet]$ stores the computed value of the first argument while the second argument is being computed. In the next section we generalize frames to support n-ary operations.

The following are the transitions of the machine; they are the same from the previous section and the new rules for operators and constants.

▶ **Definition 23** (Machine transitions).

$$
\begin{array}{rlll}
(\mathsf{Grab} \triangleright i, \eta) \mid \gamma :: s & \longmapsto (i, \gamma :: \eta) & \mid s & \\
(\mathsf{Push}\ i \triangleright i', \eta) \mid s & \longmapsto (i', \eta) & \mid (i, \eta) :: s & \\
(\mathsf{Access}\ n, \eta) \mid s & \longmapsto \eta.n & \mid s & \text{if } n < |\eta| \\
(\mathsf{Add}, \eta) \mid \gamma_1 :: \gamma_2 :: s & \longmapsto \gamma_1 & \mid [+ \bullet \gamma_2] :: s & \\
(\mathsf{Const}\ k, \eta) \mid [+ \bullet \gamma] :: s & \longmapsto \gamma & \mid [+\ k\ \bullet] :: s & \\
(\mathsf{Const}\ k, \eta) \mid [+\ k'\ \bullet] :: s & \longmapsto (\mathsf{Const}\ (k + k'), \eta) & \mid s &
\end{array}
$$

The instruction Add expects in the top of the stack one closure for each of the arguments of the addition. It pushes in the stack a new frame with the code of the second argument, and starts executing the code of the first one. For the case of the instruction Const k there are two transition rules, arising from two scenarios: k is the value of the first argument of an addition, and k is the value of the second argument. In the first case, it executes the code γ stored in the frame, and updates the frame with the constant k. In the second case, we can take the value of the first argument k' from the frame and execute Const $(k + k')$.

3.3 Compilation and its correctness

The compiler is defined by induction on the structure of the term, it maps source terms into a sequence of machine instructions:

▶ **Definition 24** (Compilation of terms)**.**

$$\begin{aligned}
[\![\,]\!] &: \Lambda \to I \\
[\![\lambda\, t]\!] &= \mathsf{Grab} \triangleright [\![t]\!] \\
[\![t\ t']\!] &= \mathsf{Push}\ [\![t']\!] \triangleright [\![t]\!] \\
[\![\overline{n}]\!] &= \mathsf{Access}\ n \\
[\![\underline{k}]\!] &= \mathsf{Const}\ k \\
[\![t_1 + t_2]\!] &= \mathsf{Push}\ [\![t_2]\!] \triangleright (\mathsf{Push}\ [\![t_1]\!] \triangleright \mathsf{Add})
\end{aligned}$$

The compilation of a term $\underline{k}$ is just the instruction $\mathsf{Const}\ k$. The code for an addition starts with a Push instruction for each argument, and continues with the Add instruction. By the time when the instruction Add is executed, the code for each argument is already stored in the stack, ready to be inserted inside a frame. We now extend the definition of the compiler for closures and environments:

▶ **Definition 25** (Compilation of closures and environments)**.**

$$\begin{aligned}
[\![_]\!]^{\mathrm{C}} &: C \to \Gamma & [\![_]\!]^{\mathrm{E}} &: E \to H \\
[\![t\,[e]]\!]^{\mathrm{C}} &= ([\![t]\!], [\![e]\!]^{\mathrm{E}}) & [\![[]]\!]^{\mathrm{E}} &= [] \\
& & [\![c :: e]\!]^{\mathrm{E}} &= [\![c]\!]^{\mathrm{C}} :: [\![e]\!]^{\mathrm{E}}
\end{aligned}$$

Here the functions $[\![\_]\!]^{\mathrm{C}}$ and $[\![\_]\!]^{\mathrm{E}}$ are mutually recursive. The compilation of a source-level closure is a machine-level closure which couples the code of the term and the code of its environment. On the other hand, the compilation of an environment is obtained by compiling each closure inside it.

In order to illustrate how the machine works, we take the same terms of the above examples and show the step-by-step execution of the corresponding code:

▶ **Example 26.** Execution of the code $[\![(\lambda\,\lambda\, t)\ t']\!]$.

$$[\![(\lambda\,\lambda\, t)\ t']\!] = \mathsf{Push}\ [\![t']\!] \triangleright \mathsf{Grab} \triangleright \mathsf{Grab} \triangleright [\![t]\!]$$

$$\begin{aligned}
&(\mathsf{Push}\ [\![t']\!] \triangleright \mathsf{Grab} \triangleright \mathsf{Grab} \triangleright [\![t]\!], \eta) \mid s \\
&\longmapsto (\mathsf{Grab} \triangleright \mathsf{Grab} \triangleright [\![t]\!], \eta) \mid ([\![t']\!], \eta) :: s \\
&\longmapsto (\mathsf{Grab} \triangleright [\![t]\!], ([\![t']\!], \eta) :: \eta) \mid s
\end{aligned}$$

▶ **Example 27.** Execution of the code $[\![(\lambda\,(\overline{0} + \underline{3}))\ \underline{2}]\!]$.

$$[\![(\lambda\,(\overline{0} + \underline{3}))\ \underline{2}]\!] = \mathsf{Push}\ (\mathsf{Const}\ 2) \triangleright \mathsf{Grab} \triangleright \mathsf{Push}\ (\mathsf{Const}\ 3) \triangleright \mathsf{Push}\ (\mathsf{Access}\ 0) \triangleright \mathsf{Add}$$

$$\begin{aligned}
&(\mathsf{Push}\ (\mathsf{Const}\ 2) \triangleright \mathsf{Grab} \triangleright \mathsf{Push}\ (\mathsf{Const}\ 3) \triangleright \mathsf{Push}\ (\mathsf{Access}\ 0) \triangleright \mathsf{Add}, \eta) \mid s \\
&\longmapsto (\mathsf{Grab} \triangleright \mathsf{Push}\ (\mathsf{Const}\ 3) \triangleright \mathsf{Push}\ (\mathsf{Access}\ 0) \triangleright \mathsf{Add}, \eta) \mid (\mathsf{Const}\ 2, \eta) :: s \\
&\longmapsto (\mathsf{Push}\ (\mathsf{Const}\ 3) \triangleright \mathsf{Push}\ (\mathsf{Access}\ 0) \triangleright \mathsf{Add}, (\mathsf{Const}\ 2, \eta) :: \eta) \mid s \\
&\longmapsto (\mathsf{Push}\ (\mathsf{Access}\ 0) \triangleright \mathsf{Add}, \eta') \mid (\mathsf{Const}\ 3, \eta') :: s \quad \text{where } \eta' = (\mathsf{Const}\ 2, \eta) :: \eta \\
&\longmapsto (\mathsf{Add}, \eta') \mid (\mathsf{Access}\ 0, \eta') :: (\mathsf{Const}\ 3, \eta') :: s \\
&\longmapsto (\mathsf{Access}\ 0, \eta') \mid [+ \bullet\ (\mathsf{Const}\ 3, \eta')] :: s \\
&\longmapsto (\mathsf{Const}\ 2, \eta) \mid [+ \bullet\ (\mathsf{Const}\ 3, \eta')] :: s \\
&\longmapsto (\mathsf{Const}\ 3, \eta') \mid [+\ 2\ \bullet] :: s \\
&\longmapsto (\mathsf{Const}\ 5, \eta') \mid s
\end{aligned}$$

Example 27 is, in fact, an instance of a more general behaviour of the machine: if a term t evaluates to a constant k in an environment e, then, the execution of the code $[\![t]\!]$ in the environment $[\![e]\!]^{\mathrm{E}}$ and an initial stack s leads to the configuration $(\mathsf{Const}\ k, \eta') \mid s$ for some environment η'. In a similar way, Example 26 can be generalized as follows: if a term t evaluates to a closure $(\lambda\, t')\,[e']$ in an environment e, then, the execution of $[\![t]\!]$ in the environment $[\![e]\!]^{\mathrm{E}}$ leads to the configuration $(\mathsf{Grab} \triangleright [\![t']\!], [\![e']\!]^{\mathrm{E}}) \mid s$. These facts can be taken as evidence of the correctness of the compiler:

▶ **Theorem 28** (Compiler Correctness). *For any $e \in E$, $t \in \Lambda$ and $v \in V$, if $e \vdash t \Rightarrow v$ then for all $s \in S$,*

- *if $v = k$ for some constant k, then $[\![t\,[e]]\!]^{\mathrm{C}} \mid s \longmapsto^* (\mathsf{Const}\ k, \eta') \mid s$ for some $\eta' \in H$,*
- *if $v = (\lambda\, t')\,[e']$ for some $t' \in \Lambda$ and $e' \in E$, then $[\![t\,[e]]\!]^{\mathrm{C}} \mid s \longmapsto^* (\mathsf{Grab} \triangleright [\![t']\!], [\![e]\!]) \mid s$.*

The statement of this theorem can significantly shortened by defining the relation $\rightarrowtail \subseteq W \times V$:

▶ **Definition 29.**

$$\begin{array}{lll} \gamma \mid s \rightarrowtail k & \text{iff} & \gamma \mid s \longmapsto^* (\mathsf{Const}\ k, \eta') \mid s \text{ for some } \eta' \in H \\ \gamma \mid s \rightarrowtail (\lambda\, t)\,[e] & \text{iff} & \gamma \mid s \longmapsto^* (\mathsf{Grab} \triangleright [\![t]\!], [\![e]\!]^{\mathrm{E}}) \mid s. \end{array}$$

The following property about this relation is expected and self-evident:

▶ **Lemma 30.** *For any $\gamma, \gamma' \in \Gamma$, $s \in S$ and $v \in V$, if $\gamma \mid s \longmapsto^* \gamma' \mid s$ and $\gamma' \mid s \rightarrowtail v$, then $\gamma \mid s \rightarrowtail v$.*

The relation $\rightarrowtail$ leads to simpler proofs (both in paper and in Coq) and can be generalized in the presence of more values, keeping the statement of correctness unchanged. Theorem 28 can be stated as follows:

▶ **Theorem 31** (Compiler Correctness). *For any $e \in E$, $t \in \Lambda$ and $v \in V$, if $e \vdash t \Rightarrow v$ then for all $s \in S$, $[\![t\,[e]]\!]^{\mathrm{C}} \mid s \rightarrowtail v$.*

Proof. This theorem can be proved by induction on the derivation of $e \vdash t \Rightarrow v$. We illustrate the proof with two cases: for rules (Const) and (App).

In the case of the rule $e \vdash \underline{k} \Rightarrow k$, we have

$$[\![\underline{k}\,[e]]\!]^{\mathrm{C}} \mid s = (\mathsf{Const}\ k, [\![e]\!]^{\mathrm{E}}) \mid s \longmapsto^* (\mathsf{Const}\ k, [\![e]\!]^{\mathrm{E}}) \mid s\ ,$$

for any $s \in S$, and therefore $[\![\underline{k}\,[e]]\!]^{\mathrm{C}} \mid s \rightarrowtail k$.

Now we turn to application; let us recall the rule (App):

$$(\textsc{App})\ \frac{e \vdash t_1 \Rightarrow (\lambda\, t)\,[e'] \qquad t_2\,[e] :: e' \vdash t \Rightarrow v}{e \vdash t_1\ t_2 \Rightarrow v}$$

We have one inductive hypothesis for each premise in the rule. In this case we have:

(i) for all $s' \in S$, $[\![t_1\,[e]]\!]^{\mathrm{C}} \mid s' \rightarrowtail (\lambda\, t)\,[e']$
(ii) for all $s' \in S$, $[\![t\,[t_2\,[e] :: e']]\!]^{\mathrm{C}} \mid s' \rightarrowtail v$.

Thus, by definition of $\rightarrowtail$ and (i), we get:

(iii) for all $s' \in S$, $[\![t_1\,[e]]\!]^{\mathrm{C}} \mid s' \longmapsto^* (\mathsf{Grab} \triangleright [\![t]\!], [\![e']\!]^{\mathrm{E}}) \mid s'$.

Using Lemma 30, we can now start with the configuration $[\![\, t_1\ t_2\,[e]\,]\!]^{\mathrm{C}} \mid s$ and try to reach the configuration $[\![\, t\,[t_2\,[e]::e']\,]\!]^{\mathrm{C}} \mid s$, which we know by (ii) is related with v by the $\rightarrowtail$ relation:

$$
\begin{array}{rcll}
[\![\, t_1\ t_2\,[e]\,]\!]^{\mathrm{C}} \mid s & = & ([\![\, t_1\ t_2\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid s & \text{by definition of } [\![\_]\!]^{\mathrm{C}} \\
& = & (\mathsf{Push}\ [\![\, t_2\,]\!] \triangleright [\![\, t_1\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid s & \text{by definition of } [\![\_]\!] \\
& \longmapsto & ([\![\, t_1\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid ([\![\, t_2\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: s & \text{by the } \mathsf{Push} \text{ rule} \\
& = & [\![\, t_1\,[e]\,]\!]^{\mathrm{C}} \mid ([\![\, t_2\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: s & \text{by definition of } [\![\_]\!]^{\mathrm{C}} \\
& \longmapsto^* & (\mathsf{Grab} \triangleright [\![\, t\,]\!], [\![\, e'\,]\!]^{\mathrm{E}}) \mid ([\![\, t_2\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: s & \text{by (iii)} \\
& \longmapsto & ([\![\, t\,]\!], ([\![\, t_2\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: [\![\, e'\,]\!]^{\mathrm{E}}) \mid s & \text{by the } \mathsf{Grab} \text{ rule} \\
& = & ([\![\, t\,]\!], [\![\, t_2\,[e]\,]\!]^{\mathrm{C}} :: [\![\, e'\,]\!]^{\mathrm{E}}) \mid s & \text{by definition of } [\![\_]\!]^{\mathrm{C}} \\
& = & ([\![\, t\,]\!], [\![\, t_2\,[e]::e'\,]\!]^{\mathrm{E}}) \mid s & \text{by definition of } [\![\_]\!]^{\mathrm{E}} \\
& = & [\![\, t\,[t_2\,[e]::e']\,]\!]^{\mathrm{C}} \mid s & \text{by definition of } [\![\_]\!]^{\mathrm{C}}.
\end{array}
$$

And that finishes the proof for (App). The remaining cases are similar. ◀

There is a third way to state the theorem of correctness that is a bit more intuitive and closer as how Leroy [22] stated it: one defines a compilation for values and then proves that the compilation of a term executes to the compilation of its value.

▶ **Definition 32** (Compilation of values)**.**

$$
\begin{aligned}
[\![\_]\!]^{\mathrm{V}} &: V \to \Gamma \\
[\![\,(\lambda\, t)\,[e]\,]\!]^{\mathrm{V}} &= (\mathsf{Grab} \triangleright [\![\, t\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \\
[\![\, k\,]\!]^{\mathrm{V}} &= (\mathsf{Const}\ k, [])
\end{aligned}
$$

The alternative version of the correctness theorem can be formally stated as follows:

▶ **Theorem 33** (Compiler Correctness, alternative)**.** *For any $e \in E$, $t \in \Lambda$ and $v \in V$, if $e \vdash t \Rightarrow v$, then, for all $s \in S$, $[\![\, t\,[e]\,]\!]^{\mathrm{C}} \mid s \longmapsto^* [\![\, v\,]\!]^{\mathrm{V}} \mid s$.*

The proof also proceeds by induction on derivations of the evaluation. Notice however that the compilation of a constant value pairs the constant with the empty environment, but the compilation of the constant (as a term) is paired with the compilation of the corresponding environment. So one needs to add a rule to discard the environment:

$$(\mathsf{Const}\ k, \gamma :: \eta) \mid s \longmapsto (\mathsf{Const}\ k, []) \mid s$$

But this change introduces some non-determinism in the machine, so one is forced to change the two rules for constants in Def. 23: those would require the environment to be empty. At first sight, it could seem possible to avoid this issue by generalizing the function $[\![\_]\!]^{\mathrm{V}}$ by taking an extra argument for the top-level environment; however, this also fails in the proof of the theorem for the case (Var).

3.3.1 Correctness for divergent terms

To complete the proof of correctness of the compiler, we need to ensure that, when the evaluation of a term t diverges, the execution of $[\![\, t\,]\!]$ will never terminate. We use the approach proposed by Leroy [22] of defining a coinductive big-step semantics to capture the notion of divergence of a term. We write $e \vdash t \Rightarrow \infty$ to denote the divergence of a term t in an environment e. The following are the rules of divergence for the high-level language of this section:

▶ **Definition 34** (Coinductive semantics).

$$(\textsc{App1})\ \frac{e \vdash t_1 \Rightarrow \infty}{e \vdash t_1\ t_2 \Rightarrow \infty} \qquad (\textsc{App2})\ \frac{e \vdash t_1 \Rightarrow (\lambda\, t)\,[e'] \qquad t_2\,[e] :: e' \vdash t \Rightarrow \infty}{e \vdash t_1\ t_2 \Rightarrow \infty}$$

$$(\textsc{Var})\ \frac{e' \vdash t' \Rightarrow \infty}{e \vdash \overline{n} \Rightarrow \infty}\ e.n = t'\,[e']$$

$$(\textsc{Add1})\ \frac{e \vdash t_1 \Rightarrow \infty}{e \vdash t_1\ +\ t_2 \Rightarrow \infty} \qquad (\textsc{Add2})\ \frac{e \vdash t_1 \Rightarrow k \qquad e \vdash t_2 \Rightarrow \infty}{e \vdash t_1\ +\ t_2 \Rightarrow \infty}$$

There are two possible reasons for an application $(t_1\ t_2)$ to diverge. The first possibility is that the function term t_1 diverges. The second one is that, when the term t_1 evaluates to an abstraction $(\lambda\, t)\,[e']$, then the evaluation of the body t diverges. Note that, since we are in a call-by-name setting, we do not make any claim about the evaluation of the argument t_2.

The next step is to capture divergence in the abstract machine; again, we use coinductive semantics. The following rule captures the notion of an infinite sequence of machine transitions:

▶ **Definition 35** (Divergence of execution).

$$\frac{w \longmapsto w' \qquad w' \longmapsto^{\infty}}{w \longmapsto^{\infty}}$$

Now we are able to state the following lemma that establishes that if a term t diverges, then the machine makes infinitely many transitions.

▶ **Theorem 36** (Correctness for divergent programs). *If* $e \vdash t \Rightarrow \infty$, *then* $[\![\, t\,[e] \,]\!]^{\mathrm{c}} \mid s \longmapsto^{\infty}$.

4 Higher-order imperative language

In this section we extend the language of the previous section with imperative features, namely we add the possibility to allocate, modify, and access memory locations. We adapt the abstract machine to reflect this extension of the source language and prove the correctness of the compiler with respect to a big-step semantics. For simplicity, the imperative variables of our language can only contain integer values, and we only consider memory locations storing integers.

4.1 The language

The imperative fragment of the language includes locations (natural numbers representing positions in the store), a dereferencing operator, variable declarations, composition and assignments. As in the previous section, we use de Bruijn indices to represent variables.

▶ **Definition 37** (Higher-order imperative language).

Terms	$\Lambda \ni t,\ t'$	$::=$	$\lambda\, t$	Abstraction
		$\mid$	$t\ t'$	Application
		$\mid$	$\overline{n}$	Variables
		$\mid$	$\underline{k}$	Constants
		$\mid$	$t \oplus t'$	Binary operators
		$\mid$	$\underline{\ell}$	Locations
		$\mid$	$!\, t$	Dereferencing
		$\mid$	**newvar** t	Variable declaration
		$\mid$	$t\,;\, t'$	Composition
		$\mid$	$t := t'$	Assignments
		$\mid$	**skip**	Skip command

Closures	$C \ni c$	$::=$	$t\,[e]$
Environments	$E \ni e$	$::=$	$[] \mid c :: e$
Stores	$\Sigma \ni \sigma$	$::=$	$[] \mid \sigma \triangleright k$
Values	$V \ni v$	$::=$	$(\lambda\, t)\,[e] \mid k \mid \ell \mid \sigma$

The initial configurations of the big-step semantics are triples (e, t, σ) and final configurations are values; we write $e \vdash^{\sigma} t \Rightarrow v$ to denote that (e, t, σ) evaluates to v. Of course, the values of commands are stores; moreover, since only commands can have side effects, the rules of the big-step semantics of the language of the previous sections remain unchanged, except that they propagate the state to each premise.

▶ **Definition 38** (Big-step semantics).

$$\text{(Abs)}\ \frac{}{e \vdash^{\sigma} \lambda\, t \Rightarrow (\lambda\, t)\,[e]} \qquad \text{(Var)}\ \frac{e' \vdash^{\sigma} t' \Rightarrow v}{e \vdash^{\sigma} \overline{n} \Rightarrow v}\ e.n = t'\,[e']$$

$$\text{(App)}\ \frac{e \vdash^{\sigma} t_1 \Rightarrow (\lambda\, t)\,[e'] \qquad t_2\,[e] :: e' \vdash^{\sigma} t \Rightarrow v}{e \vdash^{\sigma} t_1\ t_2 \Rightarrow v}$$

$$\text{(Const)}\ \frac{}{e \vdash^{\sigma} \underline{k} \Rightarrow k} \qquad \text{(Bop)}\ \frac{e \vdash^{\sigma} t_1 \Rightarrow k \qquad e \vdash^{\sigma} t_2 \Rightarrow k'}{e \vdash^{\sigma} t_1 \oplus t_2 \Rightarrow k \oplus k'}$$

$$\text{(Loc)}\ \frac{}{e \vdash^{\sigma} \underline{\ell} \Rightarrow \ell}\ \ell < |\sigma| \qquad \text{(Deref)}\ \frac{e \vdash^{\sigma} t \Rightarrow \ell}{e \vdash^{\sigma} !\, t \Rightarrow \sigma(\ell)}$$

$$\text{(Skip)}\ \frac{}{e \vdash^{\sigma} \textbf{skip} \Rightarrow \sigma} \qquad \text{(Newvar)}\ \frac{\underline{\ell}\,[e] :: e \vdash^{\sigma \triangleright 0} t \Rightarrow \sigma' \triangleright k}{e \vdash^{\sigma} \textbf{newvar}\ t \Rightarrow \sigma'}\ \ell = |\sigma|$$

$$\text{(Assign)}\ \frac{e \vdash^{\sigma} t \Rightarrow \ell \qquad e \vdash^{\sigma} t' \Rightarrow k}{e \vdash^{\sigma} t := t' \Rightarrow \sigma[\ell \mapsto k]} \qquad \text{(Comp)}\ \frac{e \vdash^{\sigma} t \Rightarrow \sigma' \qquad e \vdash^{\sigma'} t' \Rightarrow \sigma''}{e \vdash^{\sigma} t\,;\, t' \Rightarrow \sigma''}$$

Here we use some conventional notations to access and modify stores. We consider locations to be a position (an index) of the store. Thus, the store $\sigma[\ell \mapsto k]$ contains the same values as σ except perhaps in the position ℓ, in which the value is k. We denote by $\sigma(\ell)$ the

constant allocated in the position ℓ of the store. Extension of the store σ with value k in the new location is written $\sigma \triangleright k$.

In the rule of the term **newvar** we observe that, in order to evaluate the inner command t, we need to extend the store and the environment. The store is extended with the value 0, which is the default value we chose for newly created locations. The environment is extended with the location $\ell = |\sigma|$ which points to the new (and last) position of the extended store.

It is worth noting that, since the access to a location could be done exclusively through the use of a variable bound by **newvar**, we can "hide" to the user the existence of explicit locations as terms. In other words, the user does not need to know that he can write explicit locations, since all of them are created by **newvar** and bound to variables. This kind of explicit locations have been used before, for example in [13, page 3].

Another observation to make is that it is impossible for a location created by **newvar** to leave its lexical scope. In the assignment command $t_1 := t_2$ the term t_2 must evaluate to a constant, and not to a location. The store is also restricted to contain integer constants only.

A distinct feature of Algol-like languages is that the execution of a command should not leave inaccessible locations in the store; as the following lemmas show, our semantics respects that condition.

▶ **Lemma 39** (Store size preservation). *If* $e \vdash^{\sigma} t \Rightarrow \sigma'$ *then* $|\sigma| = |\sigma'|$.

▶ **Lemma 40** (Safe locations). *If* $e \vdash^{\sigma} t \Rightarrow \ell$ *then* $\ell < |\sigma|$.

4.2 A Krivine abstract machine with store

In order to cope with the extensions in the source language we need to make some changes to the abstract machine of Sec. 3. We generalize the treatment of the binary operator of the previous section so as to capture at once addition, dereferencing, and assignment; in order to do so, some instructions carry the arity of the operator. Besides that generalization, we need two instructions for allocating and deallocating memory cells; and yet another one to signal the end of the execution of the current command.

▶ **Definition 41** (Abstract machine).

$$
\begin{array}{lrcl}
\text{Code:} & I \ni i, i' & ::= & \mathsf{Grab} \triangleright i \\
& & \mid & \mathsf{Push}\ i \triangleright i' \\
& & \mid & \mathsf{Access}\ n \\
& & \mid & \mathsf{Const}\ V \\
& & \mid & \mathsf{Op}\ \ominus^{\mathsf{n}} \\
& & \mid & \mathsf{Frame}\ \ominus^{\mathsf{n}} \\
& & \mid & \mathsf{Alloc} \triangleright i \\
& & \mid & \mathsf{Dealloc} \\
& & \mid & \mathsf{Cont}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Closures:} & \Gamma \ni \gamma & ::= & (i, \eta) \\
\text{Environments:} & H \ni \eta & ::= & [] \mid \gamma :: \eta \\
\text{Operators:} & \mathit{Ops} \ni \ominus^{\mathsf{n}} & ::= & \oplus^2 \mid !^1 \mid :=^2 \\
\text{Operator Arguments:} & N \ni \nu & ::= & k \mid \ell \\
\text{Stack values:} & M \ni \mu & ::= & \gamma \mid [\ominus^{\mathsf{n}}\ \overline{\nu} \bullet \overline{\gamma}] \\
\text{Stacks:} & S \ni s & ::= & [] \mid \mu :: s \\
\text{Stores:} & \Sigma \ni \sigma & ::= & [] \mid \sigma \triangleright k \\
\text{Configurations:} & W \ni w & ::= & (\gamma,\ \sigma,\ s)
\end{array}
$$

Here, a frame $[\ominus^{\mathsf{n}}\ \overline{\nu}\ \bullet\ \overline{\gamma}]$ is a data structure that contains: (a) an operator $\ominus^{\mathsf{n}}$, which is always associated with an operation supported by the machine, (b) a list $\overline{\nu}$ with the arguments of the operation which have been already computed, and (c) a list $\overline{\gamma}$ with the code required to compute the rest of the arguments.

The transitions of the machine are given in the following definition. Notice that the execution of code corresponding to expressions will, eventually, finish with a numeric value in the closure part; while the execution of an imperative command will finish with Cont.

▶ **Definition 42.**

$$
\begin{array}{llllll}
(\mathsf{Grab} \triangleright i, \eta) & \mid \sigma \mid \gamma :: s & \longmapsto & (i, \gamma :: \eta) \mid \sigma \mid s & & \\
(\mathsf{Push}\ i \triangleright i', \eta) & \mid \sigma \mid s & \longmapsto & (i', \eta) \mid \sigma \mid (i, \eta) :: s & & \\
(\mathsf{Access}\ n, \eta) & \mid \sigma \mid s & \longmapsto & \eta.n \mid \sigma \mid s & & \text{if } n < |\eta| \\
(\mathsf{Frame}\ \ominus^{\mathsf{n}}, \eta) & \mid \sigma \mid \gamma_1 :: \overline{\gamma} :: s & \longmapsto & \gamma_1 \mid \sigma \mid [\ominus^{\mathsf{n}}\ \bullet\ \overline{\gamma}] :: s & & \text{if } |\overline{\gamma}| < n \\
(\mathsf{Op} \oplus, \eta) & \mid \sigma \mid [\oplus\ k, k' \bullet] :: s & \longmapsto & (\mathsf{Const}\ \hat{k}, \eta) \mid \sigma \mid s & & \text{where } \hat{k} = k \oplus k' \\
(\mathsf{Op}\ :=, \eta) & \mid \sigma \mid [:=\ \ell, k \bullet] :: s & \longmapsto & (\mathsf{Cont}, \eta) \mid \sigma' \mid s & & \text{where } \sigma' = \sigma[\ell \mapsto k] \\
(\mathsf{Op}\ !, \eta) & \mid \sigma \mid [!\ \ell \bullet] :: s & \longmapsto & (\mathsf{Const}\ k, \eta) \mid \sigma \mid s & & \text{where } k = \sigma(\ell) \\
(\mathsf{Alloc} \triangleright i, \eta) & \mid \sigma \mid s & \longmapsto & (i, \gamma :: \eta) \mid \sigma \triangleright 0 \mid s & & \text{where } \gamma = (\mathsf{Const}\ |\sigma|, \eta) \\
(\mathsf{Dealloc}, \eta) & \mid \sigma \triangleright k \mid s & \longmapsto & (\mathsf{Cont}, \eta) \mid \sigma \mid s & & \\
(\mathsf{Cont}, \eta) & \mid \sigma \mid \gamma :: s & \longmapsto & \gamma \mid \sigma \mid s & & \\
(\mathsf{Const}\ \nu, \eta) & \mid \sigma \mid [\ominus^{\mathsf{n}}\ \overline{\nu}\ \bullet\ \gamma_1, \overline{\gamma}] :: s & \longmapsto & \gamma_1 \mid \sigma \mid [\ominus^{\mathsf{n}}\ \overline{\nu}, \nu\ \bullet\ \overline{\gamma}] :: s & & \\
(\mathsf{Const}\ \nu, \eta) & \mid \sigma \mid [\ominus^{\mathsf{n}}\ \overline{\nu} \bullet] :: s & \longmapsto & (\mathsf{Op}\ \ominus^{\mathsf{n}}, \eta) \mid \sigma \mid [\ominus^{\mathsf{n}}\ \overline{\nu}, \nu \bullet] :: s & &
\end{array}
$$

The instruction $\mathsf{Frame}\ \ominus^{\mathsf{n}}$ expects n closures in the top of the stack. Then it executes the code of the first argument, and creates a frame containing the rest of them.

As in the previous section, the instruction $\mathsf{Const}\ k$ updates the frame with the constant k – which is the value of an argument – and executes the next closure stored in the frame, if there is any. When all the arguments have been computed, the instruction $\mathsf{Op}\ \ominus^{\mathsf{n}}$ is executed. This instruction expects a frame with all the arguments computed and applies the built-in operation associated with $\ominus^{\mathsf{n}}$. For example, if $\ominus^{\mathsf{n}}$ is the assignment operator $(:=)$, then $\mathsf{Op}\ (:=)$ expects a frame $[:=\ \ell, k \bullet]$ and then updates the store in the location ℓ with the constant value k.

4.3 Compilation and correctness

The compilation of the applicative part of the language remains unchanged with respect to the previous section. The translation of an n-ary operator $\ominus$ consists in compiling all its operands and putting the instruction $\mathsf{Frame}\ \ominus^{\mathsf{n}}$ after their code. Notice, however, that the code for the operands will be executed after constructing the appropriate frame in the stack. To compile the allocation of a new variable, we prepare the deallocation of the new location –to be executed after the body of the block–, then we generate an allocation instruction followed by the code of the body.

▶ **Definition 43** (Compilation of terms)**.**

$$
\begin{array}{rl}
[\![\_]\!] \colon \Lambda \to I & [\![t_1 \oplus t_2]\!] = \mathsf{Push}\ [\![t_2]\!] \triangleright \mathsf{Push}\ [\![t_1]\!] \triangleright \mathsf{Frame}\ (\oplus) \\
[\![\lambda\, t]\!] = \mathsf{Grab} \triangleright [\![t]\!] & [\![!\, t]\!] = \mathsf{Push}\ [\![t]\!] \triangleright \mathsf{Frame}\ (!) \\
[\![t\ t']\!] = \mathsf{Push}\ [\![t']\!] \triangleright [\![t]\!] & [\![\mathbf{newvar}\ t]\!] = \mathsf{Push}\ (\mathsf{Dealloc}) \triangleright \mathsf{Alloc} \triangleright [\![t]\!] \\
[\![\overline{n}]\!] = \mathsf{Access}\ n & [\![t_1\,;\, t_2]\!] = \mathsf{Push}\ [\![t_2]\!] \triangleright [\![t_1]\!] \\
[\![\underline{k}]\!] = \mathsf{Const}\ k & [\![t_1 := t_2]\!] = \mathsf{Push}\ [\![t_2]\!] \triangleright \mathsf{Push}\ [\![t_1]\!] \triangleright \mathsf{Frame}\ (:=) \\
[\![\underline{\ell}]\!] = \mathsf{Const}\ \ell & [\![\mathbf{skip}]\!] = \mathsf{Cont}
\end{array}
$$

The following is the definition of compilation functions for closures and environments. Note that these functions are mutually recursive:

▶ **Definition 44** (Compilation of closures and environments)**.**

$$\begin{array}{ll} [\![\_]\!]^{\mathrm{C}} \colon C \to \Gamma & [\![\_]\!]^{\mathrm{E}} \colon E \to H \\ [\![\, t\,[e]\,]\!]^{\mathrm{C}} = ([\![\, t\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) & [\![\,[]\,]\!]^{\mathrm{E}} = [] \\ & [\![\, c :: e\,]\!]^{\mathrm{E}} = [\![\, c\,]\!]^{\mathrm{C}} :: [\![\, e\,]\!]^{\mathrm{E}} \end{array}$$

As in the previous section, we use a relation $\rightarrowtail$ between configurations and values to state the correctness theorem. We extend Definition 29 as follows:

▶ **Definition 45** ($\rightarrowtail \subseteq W \times V$)**.**

$$\begin{array}{lll} \gamma \mid \sigma \mid s \rightarrowtail k & \text{iff} & \gamma \mid \sigma \mid s \longmapsto^* (\mathsf{Const}\ k, \eta') \mid \sigma \mid s \ \text{ for some } \eta' \in H \\ \gamma \mid \sigma \mid s \rightarrowtail (\lambda\, t)\,[e] & \text{iff} & \gamma \mid \sigma \mid s \longmapsto^* (\mathsf{Grab} \triangleright [\![\, t\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid s \\ \gamma \mid \sigma \mid s \rightarrowtail \ell & \text{iff} & \gamma \mid \sigma \mid s \longmapsto^* (\mathsf{Const}\ \ell, \eta') \mid \sigma \mid s \ \text{ for some } \eta' \in H \\ \gamma \mid \sigma \mid s \rightarrowtail \sigma' & \text{iff} & \gamma \mid \sigma \mid s \longmapsto^* (\mathsf{Cont}, \eta') \mid \sigma' \mid s \ \text{ for some } \eta' \in H. \end{array}$$

Now we can state the theorem of correctness for convergent terms:

▶ **Theorem 46** (Correctness for convergent terms)**.** *For any $e \in E$, $t \in \Lambda$, $\sigma \in \Sigma$, $v \in V$, if $e \vdash^{\sigma} t \Rightarrow v$ then, for all $s \in S$, $[\![\, t\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s \rightarrowtail v$.*

Proof. The proof is by induction in the derivation of $e \vdash^{\sigma} t \Rightarrow v$. We illustrate the proof for the case of the assignment command. Let us recall the rule for assignment:

$$(\textsc{Assign})\ \frac{e \vdash^{\sigma} t \Rightarrow \ell \qquad e \vdash^{\sigma} t' \Rightarrow k}{e \vdash^{\sigma} t := t' \Rightarrow \sigma[\ell \mapsto k]}$$

We have an inductive hypothesis for each premise of the rule:

(i) for all $s' \in S$, $[\![\, t\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s' \rightarrowtail \ell$

(ii) for all $s' \in S$, $[\![\, t'\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s' \rightarrowtail k$.

Therefore, by definition of $\rightarrowtail$, we get:

(iii) for all $s' \in S$, $[\![\, t\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s' \longmapsto^* (\mathsf{Const}\ \ell, \eta_1) \mid \sigma \mid s'$ for some $\eta_1 \in H$

(iv) for all $s' \in S$, $[\![\, t'\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s' \longmapsto^* (\mathsf{Const}\ k, \eta_2) \mid \sigma \mid s'$ for some $\eta_2 \in H$.

Now we can make the following sequence of transitions:

$$
\begin{array}{lr}
[\![\, t := t'\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s & \\
= ([\![\, t := t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid s & \text{by definition of } [\![\_]\!]^{\mathrm{C}} \\
\longmapsto (\mathsf{Push}\ [\![\, t'\,]\!] \triangleright \mathsf{Push}\ [\![\, t\,]\!] \triangleright \mathsf{Frame}\ (:=), [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid s & \text{by definition of } [\![\_]\!] \\
\longmapsto (\mathsf{Push}\ [\![\, t\,]\!] \triangleright \mathsf{Frame}\ (:=), [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid ([\![\, t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: s & \text{by the Push rule} \\
\longmapsto (\mathsf{Frame}\ (:=), [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid ([\![\, t\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: ([\![\, t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) :: s & \text{by the Push rule} \\
\longmapsto ([\![\, t\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid [:= \ \bullet\, ([\![\, t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}})] :: s & \text{by the Frame rule} \\
= \ [\![\, t\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid [:= \ \bullet\, ([\![\, t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}})] :: s & \text{by definition of } [\![\_]\!]^{\mathrm{C}} \\
\longmapsto^{*} (\mathsf{Const}\ \ell, \eta_1) \mid \sigma \mid [:= \ \bullet\, ([\![\, t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}})] :: s & \text{by (iii)} \\
\longmapsto ([\![\, t'\,]\!], [\![\, e\,]\!]^{\mathrm{E}}) \mid \sigma \mid [:= \ \ell \bullet\,] :: s & \text{by a Const rule} \\
= [\![\, t'\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid [:= \ \ell \bullet\,] :: s & \text{by definition of } [\![\_]\!]^{\mathrm{C}} \\
\longmapsto^{*} (\mathsf{Const}\ k, \eta_2) \mid \sigma \mid [:= \ \ell \bullet\,] :: s & \text{by (iv)} \\
\longmapsto (\mathsf{Op}\ :=, \eta_2) \mid \sigma \mid [:= \ \ell,\, k \bullet\,] :: s & \text{by a Const rule} \\
\longmapsto (\mathsf{Cont}, \eta_2) \mid \sigma[\ell \mapsto k] \mid s & \text{by the Op } (:=) \text{ rule .}
\end{array}
$$

Thus, we have proved $[\![\, t := t'\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s \rightarrowtail \sigma[\ell \mapsto k]$. The remaining cases are similar. ◀

4.3.1 Correctness for divergent terms

We continue with the definition of the coinductive big-step semantics. In the following definition we present the rules for the imperative fragment of our language, since the rules for the other terms are similar to those in Definition 34, except for the propagation of the store through the premises:

▶ **Definition 47** (Coinductive semantics).

$$
(\textsc{Deref})\ \frac{e \vdash^{\sigma} t \Rightarrow \infty}{e \vdash^{\sigma} !\, t \Rightarrow \infty} \qquad (\textsc{Newvar})\ \frac{\underline{\ell}\,[e] :: e \vdash^{\sigma \triangleright 0} t \Rightarrow \infty}{e \vdash^{\sigma} \mathbf{newvar}\ \ t \Rightarrow \infty}
$$

$$
(\textsc{Comp1})\ \frac{e \vdash^{\sigma} t_1 \Rightarrow \infty}{e \vdash^{\sigma} t_1 \,;\, t_2 \Rightarrow \infty} \qquad (\textsc{Comp1})\ \frac{e \vdash^{\sigma} t_1 \Rightarrow \sigma' \qquad e \vdash^{\sigma'} t_2 \Rightarrow \infty}{e \vdash^{\sigma} t_1 \,;\, t_2 \Rightarrow \infty}
$$

$$
(\textsc{Assign1})\ \frac{e \vdash^{\sigma} t_1 \Rightarrow \infty}{e \vdash^{\sigma} t_1 := t_2 \Rightarrow \infty} \qquad (\textsc{Assign2})\ \frac{e \vdash^{\sigma} t_1 \Rightarrow \ell \qquad e \vdash^{\sigma} t_2 \Rightarrow \infty}{e \vdash^{\sigma} t_1 := t_2 \Rightarrow \infty}
$$

We can prove, by coinduction, that if the machine executes the code of a divergent term, then it never stops:

▶ **Theorem 48** (Correctness for divergent terms). *If $e \vdash^{\sigma} t \Rightarrow \infty$, then $[\![\, t\,[e]\,]\!]^{\mathrm{C}} \mid \sigma \mid s \longmapsto^{\infty}$ for all $s \in S$.*

4.4 About the formalization

In Coq, most of the definitions above are represented using inductive types. For example, the following is the definition of the evaluation rules for abstractions, applications and variables:

```
Inductive eval (e : env) (q : store) : term → value → Prop :=
| eval_abs : forall t, eval e q (term_abs t) (value_abs t e)
| eval_app : forall t1 t2 t e' v,
               eval e q t1 (value_abs t e') →
               eval (t2 [e] :: e') q t v →
               eval e q (term_app t1 t2) v
| eval_var : forall n t' e' v,
               lookup e n = Some (t' [e']) →
               eval e' q t' v →
               eval e  q (term_var n) v
[...]
```

Here, each constructor corresponds to one of the rules of evaluation. For example, the constructor `eval_abs` corresponds to the rule (Abs) of Definition 38.

We rely on an important feature of Coq that is its built-in support for coinductive definitions and proofs, which allowed us to handle proof involving infinite sequences of transitions or coinductive evaluation in a simple manner. For example, the following is the definition of the coinductive evaluation rules for the case of the application:

```
CoInductive diverges (e : env) (q : store) : term → Prop :=
| diverges_app_fst :
    forall t1 t2,
      diverges e q t1 →
      diverges e q (term_app t1 t2)
| diverges_app_snd :
    forall t1 t t2 e' ,
      eval e q t1 (value_abs t e') →
      diverges (t2[e] :: e') q t →
      diverges e q (term_app t1 t2)
[...]
```

The constructor `diverges_app_fst` covers the case when the application diverges due to its operator (the term `t1`), and `diverges_app_snd` covers the case when the operator evaluates to an abstraction but the divergence occurs after the contraction of the redex. The correctness lemma for divergent terms is proved using the `cofix` tactic that permits proofs by coinduction:

```
Lemma correctness_for_divergent :
  forall e q t,
    diverges e q t →
    forall s,
      let g := closure_code (compile t) (compile_env e) in
      infseq (plus trans) (g, q, s).
Proof.
  cofix.
  [...]
Qed.
```

We have used Coq's `Ltac` tactic language to define tactics useful to automate some of the proofs of the formalization. For example, the following tactic is used in the proof of compiler correctness for convergent terms to make as many machine transitions as possible:

```
Ltac progress_until_possible :=
  repeat
    match goal with
```

```
    | [|- star trans _ _ ] ⇒
      first [
          eassumption
        | apply star_refl
        | eapply star_step ; [econstructor | eauto]
        | eapply star_trans ; [eassumption | eauto]
        ]
    | [|- _ ] ⇒ simpl ; progress eauto
  end
```

Here, the inductive type `star trans` represents a sequence of machine transitions. This tactic tries to prove a goal where the conclusion has the form `star trans _ _`. First, it tries to use an assumption to prove the goal, but if it is not possible, it will try to make zero, one or more steps (in that order) to reach the desired configuration.

We have measured the size of the formalization using the tool `coqwc` that prints the number of lines of code designated to specifications or proofs. The next table shows the results for the formalization of each of the three languages we considered in the paper:

Language	Specifications	Proofs
Call-by-name lambda calculus	345	531
Call-by-name lambda calculus with strict operators	336	199
Imperative higher-order language	468	272

The formalization of the first section has larger proofs scripts than the others. This is due to the fact that the use of small-step semantics requires to prove more results to capture the notion of correctness of the compiler and to a less extensive use of the `Ltac` mechanism.

5 Conclusion

In this paper we used well-known techniques [21, 22, 2] to mechanize in Coq the correctness of a compiler for a higher-order imperative language to a variant of the Krivine abstract machine. As far as we know, this is the first proof of correctness of a compiler combining call-by-name lambda calculus extended with a store and strict operators.

This formalization is also one of our first steps towards proving the correctness of a compiler for an Algol-like language [25]. Our next steps towards that goal involve (i) to add booleans with non-strict binary operations, (ii) to impose a type system on the source language, and (iii) to add a recursion operator.

Most of those changes planned for the language also entail modifications in the design of the compiler or the machine. For example, if we impose a type system in the language, the compiler might be designed to compile typing derivations instead of raw terms, as we do in this paper. The type system should also enable us to eliminate the need for a dereferencing operator, since we can detect during type-checking the different roles of the occurrences of a variable.

We plan to make some improvements in the abstract machine and also consider the use of the refocusing technique [10] to derive an abstract machine for the imperative language. One downside of our machine is the overhead incurred by the use of *frames* to implement strict operators; one possible remedy for this is the use of *stack markers* as in the ZAM machine [18]. Since we are using call-by-name evaluation, we could get some improvements in the execution by also considering sharing as in [17, 16].

Related work. Leroy [22] defined an abstract machine for a call-by-value lambda calculus, and used coinductive big-step semantics to describe the behavior of divergent programs. He also used Coq to prove the correctness of the compiler and some additional semantic properties like evaluation determinism and progress. A similar approach has been used by Leroy [21] and Bertot [3] for the simple imperative language.

Danvy and Nielsen [10] introduced the *refocusing* technique, that allows to systematically derive abstract machines from reduction semantics, by applying successive program transformations. Sieczkowski *et al.* [29] formalized in Coq and proved correct the technique for some applicative languages. We have taken Sieckzowski's formalization and adapted it for the language in Sec. 3; the resulting formalization is longer than our original mechanization. This happens because that method requires to prove several technical lemmas for each language; it could be interesting to investigate the possibility of stating refocusing more abstractly in order to prove some of those lemmas in a more general setting. It is not immediate if this technique can be applied to imperative languages, like the one in Section 4.

Chlipala [5, 6, 7] and Benton [1] used denotational semantics and logical relations to structure the proof of correctness of compilers for several programming languages, including typed lambda calculus and impure functional languages. Peter Selinger [28] derived extensions of the Krivine machine from the CPS translations of the $\lambda\mu$-calculus. Piróg *et al.* [24] derived a lazy abstract machine for an applicative language and formalized that derivation in Coq.

Acknowledgements. We would like to thank three anonymous reviewers for their comments and suggestions on an earlier version of this article.

References

1 Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. *SIGPLAN Not.*, 44(9):97–108, August 2009.

2 Yves Bertot. A certified compiler for an imperative language. Technical Report RR-3488, INRIA, September 1998.

3 Yves Bertot. Theorem proving support in programming language semantics. *CoRR*, abs/0707.0926, 2007.

4 Malgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Log.*, 9(1), 2007.

5 Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. *SIGPLAN Not.*, 42(6):54–65, June 2007.

6 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. *SIGPLAN Not.*, 43(9):143–156, September 2008.

7 Adam Chlipala. A verified compiler for an impure functional language. In *POPL*, pages 93–106, 2010.

8 G. Cousineau and P.-L. Curien. The categorical abstract machine. *Sci. Comput. Program.*, 8(2):173–202, April 1987.

9 Pierre-Louis Curien. An abstract framework for environment machines. *Theor. Comput. Sci.*, 82(2):389–402, 1991.

10 Olivier Danvy and Lasse Nielsen. Refocusing in reduction semantics. Research report BRICS RS-04-26, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark, November 2004.

11 Stephan Diehl and Peter Sestoft. Abstract machines for programming language implementation. *Future Gener. Comput. Syst.*, 16(7):739–751, May 2000.

12 Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *J. Funct. Program.*, 8(2):131–176, March 1998.

13 Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 141–152. ACM, 2006.

14 Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207, September 2007.

15 P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, January 1964.

16 John Launchbury. Lazy imperative programming. In *ACM Sigplan Workshop on State in Programming Languages*, pages 46–56, 1993. (available as YALEU/DCS/RR968, Yale University).

17 John Launchbury. A natural semantics for lazy evaluation. In *POPL*, pages 144–154, 1993.

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

19 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

20 Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.

21 Xavier Leroy. Mechanized semantics – with applications to program proof and compiler verification. In *Logics and Languages for Reliability and Security*, pages 195–224. IOS Press BV, 2010.

22 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, February 2009.

23 Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

24 Maciej Pirog and Dariusz Biernacki. A systematic derivation of the STG machine verified in Coq. *SIGPLAN Not.*, 45(11):25–36, September 2010.

25 John C. Reynolds. Using functor categories to generate intermediate code. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'95, pages 25–36, New York, NY, USA, 1995. ACM.

26 John C. Reynolds. The essence of Algol. In Peter W. O'Hearn and Robert D. Tennent, editors, *ALGOL-like Languages, Volume 1*, pages 67–88. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

27 M. Rittri and Institutionen för informationsbehandling (Göteborg). *Proving the Correctness of a Virtual Machine by a Bisimulation*. Department of computer sciences, 1988.

28 Peter Selinger. From continuation passing style to Krivine's abstract machine. Manuscript, 2003. Available in Peter Selinger's web site.

29 Filip Sieczkowski, Małlgorzata Biernacka, and Dariusz Biernacki. Automating derivations of abstract machines from reduction semantics: A generic formalization of refocusing in Coq. In *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages*, IFL'10, pages 72–88, Berlin, Heidelberg, 2011. Springer-Verlag.