Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 323 (2016) 197–214

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Proving Correctness of a Compiler Using Step-indexed Logical Relations

Leonardo Rodríguez[1] Miguel Pagano[2] Daniel Fridlender[3]

FaMAF
Universidad Nacional de Córdoba
Argentina

Abstract

In this paper we prove the correctness of a compiler for a call-by-name language using step-indexed logical relations and biorthogonality. The source language is an extension of the simply typed lambda-calculus with recursion, and the target language is an extension of the Krivine abstract machine. We formalized the proof in the Coq proof assistant.

Keywords: Compiler verification, proof assistants, biorthogonality, step-indexed logical relations.

1 Introduction

There are many tools and frameworks available to analyze programs and to prove desirable properties about them, for instance, that they meet their specification. Several methods of static analysis such as program verification, and abstract interpretation can be used to lower the chance of letting errors go into deployed programs. However, a machine seldom executes source programs directly. Instead, they are translated into low-level programs with the help of a compiler. Therefore, we must consider the potential errors that the compilation process might introduce: a naive translation of a source program may easily invalidate its properties, making the effort initially invested useless. Dynamic program analysis, such as testing, may help finding errors in the executable code, but it is not enough when it comes to critical systems, which demand greater guarantees of security and reliability. It becomes necessary to prove that the compiler preserves semantics, that is, that the program

[1] Email:lrodrig2@famaf.unc.edu.ar

[2] Email:mpagano@famaf.unc.edu.ar

[3] Email:dfridlender@famaf.unc.edu.ar

http://dx.doi.org/10.1016/j.entcs.2016.06.013
1571-0661/© 2016 The Author(s). Published by Elsevier B.V.

generated by the compiler behaves exactly as the semantics of the source program indicates.

Since the first proofs of compiler correctness appeared many years ago [20,21], there has been a considerable amount of progress in the topic. Of particular importance is the work of the CompCert project [17], a certified compiler for a large subset of the C programming language. In the case of functional languages we can mention [8] which is a certified compiler for the the simply-typed lambda calculus, and [4] where the source language is a call-by-value functional language, and the target is a variant of the SECD machine [16].

In order to prove a compiler correct it is necessary to find a connection between the semantics of the source language and the semantics of the target language. In general, the latter is described operationally: we define which are the instructions available in the machine (a real microprocessor or an abstract machine) and how those instructions modify the configuration as they are executed. On the other hand, there are many ways to describe the semantics of the source language, and the structure of the proof of correctness is highly dependent on which method is used. There are proofs of compiler correctness based on the big-step semantics [9,19], small-step semantics [2], or denotational semantics [4,8] of the source language, among others.

In this work we prove the correctness of a compiler for a typed call-by-name functional language, and the proof is based on the domain-theoretic denotational semantics of the language. The compiler translates a well-typed term of the source language into a list of instructions for the Krivine machine (KAM) [15]. We use step-indexed logical relations [1,3] and biorthogonality [23] to capture the notion of correctness in a compositional and modular way. These two techniques have been used before in combination to obtain proofs of compiler correctness [4,5,12] and also applied in other topics such as program equivalence [11]. As far as we know, no previous work has applied these techniques to prove the correctness of a compiler targeting the KAM and for a call-by-name language.

The approach we follow in this paper has been used before by [4] but applied to a call-by-value language and the SECD machine. In this work we revise the method in such a way that it becomes applicable in a call-by-name language and the KAM machine, and we obtained a simpler definition of the logical relations and a cleaner proof of correctness.

We formalized all the results in the Coq proof assistant, and the code is available online [25]. We used and extended a domain theory library [6] as a basis for the formalization of the semantics and the logical relations.

The rest of the paper is organized as follows. In Section 2 we present the source language and its denotational semantics. We continue in Section 3 with the target language and its operational semantics. We present a general explanation of biorthogonality in Section 4 and then we apply this technique in Section 5 in which we present our first logical relation that we called "denotational approximation". In Section 6 we introduce step-indexing and some results about its combination with biorthogonality. We apply both biorthogonality and step-indexing in Section

7 to construct the second logical relation called "operational approximation". We comment on the formalization in Coq in Section 8 and in Section 9 we conclude.

2 The Source Language

The terms of the source language are the following:

Definition 2.1 (Language terms).

$$\mathcal{T} \ni t ::= \lambda t \mid t_1\, t_2 \mid \overline{n} \mid \mathit{rec}\, t \mid \underline{m} \mid \ominus^n (t_1, \ldots, t_n)$$
$$\mid (t_0,\, t_1) \mid \mathit{fst}\ t \mid \mathit{snd}\ t \mid \mathit{ifz}\, t\ .\ t'$$

Hereafter we use the notation $\mathcal{T} \ni t$ to specify both the set defined by the grammar and our naming convention for meta-variables ranging over it. The first three constructors correspond to the lambda calculus with de Bruijn indices. The language also includes a fixed-point operator, integer constants, strict arithmetic operators, pairs and projections. The last constructor is a conditional projection. We choose this form of conditional for convenience, but a more familiar constructor of the form $\mathit{ifz}\, t\ \mathit{then}\, t_1\ \mathit{else}\, t_2$ can be expressed as $\mathit{ifz}\, t\ .\ (t_1,\, t_2)$. We write $\ominus^n$ to represent any strict arithmetic operator with arity $n > 0$; operators are written in prefix position and cannot be partially applied.

The type system is rather simple. We have a single basic type **int**, and also arrow and product types. A context is defined to be a list of types, accordingly with the use of de Bruijn indices.

Definition 2.2 (Types and contexts).

$$\Theta \ni \theta ::= \mathbf{int} \mid \theta \to \theta' \mid \theta \times \theta'$$

$$\Theta^* \ni \pi ::= [] \mid \theta :: \pi$$

We present the typing rules for the language in Figure 1, which are quite familiar. The conclusion of a typing rule is a judgment of the form $\pi \vdash t : \theta$ which states that the term t has type θ under the context π.

2.1 Denotational Semantics

The denotational semantics of the source language is given in a domain-theoretic setting because of the presence of the fixed-point operator. In this section, and in the rest of the paper, we will follow a traditional treatment of domain theory – for example, we will not comment on how one calculates the supremum of a chain. In contrast, our formalization in Coq is based on a constructive domain theory library [6] where the supremum is given by a function (in Coq's language).

Before coming to the semantics of the language, we recall some concepts and notations of domain theory. The domain of continuous functions from a domain P

$$\text{(TyAbs)}\ \frac{\theta :: \pi \vdash t : \theta'}{\pi \vdash \lambda t : \theta \to \theta'} \qquad \text{(TyApp)}\ \frac{\pi \vdash t_1 : \theta \to \theta' \qquad \pi \vdash t_2 : \theta}{\pi \vdash t_1\, t_2 : \theta'}$$

$$\text{(TyVar)}\ \frac{n < |\pi| \qquad \pi\,.\,n = \theta}{\pi \vdash \overline{n} : \theta} \qquad \text{(TyRec)}\ \frac{\pi \vdash t : \theta \to \theta}{\pi \vdash \mathit{rec}\, t : \theta} \qquad \text{(TyConst)}\ \frac{}{\pi \vdash \underline{m} : \mathbf{int}}$$

$$\text{(TyOp)}\ \frac{\pi \vdash t_i : \mathbf{int} \qquad i \in \{1, \ldots, n\}}{\pi \vdash \ominus^n (t_1, \ldots, t_n) : \mathbf{int}}$$

$$\text{(TyFst)}\ \frac{\pi \vdash t : \theta \times \theta'}{\pi \vdash \mathit{fst}\, t : \theta} \qquad \text{(TySnd)}\ \frac{\pi \vdash t : \theta \times \theta'}{\pi \vdash \mathit{snd}\, t : \theta'}$$

$$\text{(TyPair)}\ \frac{\pi \vdash t_0 : \theta \qquad \pi \vdash t_1 : \theta'}{\pi \vdash (t_0, t_1) : \theta \times \theta'} \qquad \text{(TyCond)}\ \frac{\pi \vdash t : \mathbf{int} \qquad \pi \vdash t' : \theta \times \theta}{\pi \vdash \mathit{ifz}\, t\, .\, t' : \theta}$$

Fig. 1: Typing rules.

to a domain Q is written as $[P \to Q]$. Any set P can be turned into the *flat* domain $P_\perp$ by adjoining a least element $\perp$; this construction can be turned into a Kleisli triple whose unit is $\iota_\uparrow \colon P \to P_\perp$ and its extension operation for a $f \colon P \to Q_\perp$ is given by $f^* \perp = \perp$ and $f^* (\iota_\uparrow x) = f\, x$, for $x \in P$. The semantics of strict arithmetic operators are based on abstract considerations: for a total function $\oplus \colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ we can get a function $\oplus^* \colon (\mathbb{Z} \times \mathbb{Z})_\perp \to \mathbb{Z}_\perp$ by $\oplus^* = (\iota_\uparrow \cdot \oplus)^*$. Since the denotation of operands will be an unlifted tuple of lifted values, we compose $\oplus^*$ with the strength, $\tau_{A,B} \colon A_\perp \times B_\perp \to (A \times B)_\perp$, to get the function $\oplus_\perp = \oplus^* \cdot \tau_{\mathbb{Z},\mathbb{Z}}$; notice that one can apply the same construction for an n-ary operation. Given a function $f \colon P \to D$ from the predomain P to a domain D, we write $f_{\perp\!\!\perp} \colon P_\perp \to D$ for the function such that $f_{\perp\!\!\perp} \perp = \perp$ and $f_{\perp\!\!\perp} (\iota_\uparrow x) = f\, x$, for $x \in P$. If $f \colon D \to D$ is a continuous function over the domain D, then $\mathbf{Y}_D\, f$ denotes the least fixed-point of f.

As usual, once we choose a domain for the denotation of atomic types, in our case **int**, the semantics of arrow types and contexts are determined by the exponentials and finite products of the underlying category.

Definition 2.3 (Semantics of types and contexts).

$$\begin{aligned} [\![\, \mathbf{int} \,]\!] &= \mathbb{Z}_\perp \\ [\![\, \theta \to \theta' \,]\!] &= [[\![\, \theta \,]\!] \to [\![\, \theta' \,]\!]] \\ [\![\, \theta \times \theta' \,]\!] &= [\![\, \theta \,]\!] \times [\![\, \theta' \,]\!] \end{aligned} \qquad \begin{aligned} [\![\, [] \,]\!] &= \{*\} \\ [\![\, \theta :: \pi \,]\!] &= [\![\, \theta \,]\!] \times [\![\, \pi \,]\!] \end{aligned}$$

In Figure 2 we present the denotational semantics for typing derivations of the source language. We use the symbol $\hat{\lambda}$ as a meta-binder to avoid confusion with the symbol λ used in abstractions. Also, if γ is a finite product, we write $\gamma \downarrow n$ for its n-th projection. The *coherence* of this semantics –meaning that different judgments of the same expression have the same denotation– has already been proved for a language even larger than the one we use in this paper [24].

$$\llbracket \_ \rrbracket : |\pi \vdash t : \theta| \to \llbracket \pi \rrbracket \to \llbracket \theta \rrbracket$$

$$\llbracket \pi \vdash \lambda t : \theta \to \theta' \rrbracket \gamma = \hat{\lambda} a . \llbracket \theta :: \pi \vdash t : \theta' \rrbracket (a, \gamma)$$

$$\llbracket \pi \vdash t_1\, t_2 : \theta' \rrbracket \gamma = (\llbracket \pi \vdash t_1 : \theta \to \theta' \rrbracket \gamma)\, (\llbracket \pi \vdash t_2 : \theta \rrbracket \gamma)$$

$$\llbracket \pi \vdash \overline{n} : \pi . n \rrbracket \gamma = \gamma \swarrow n$$

$$\llbracket \pi \vdash rec\, t : \theta \rrbracket \gamma = \mathbf{Y}_{\llbracket \theta \rrbracket} (\llbracket \pi \vdash t : \theta \to \theta \rrbracket \gamma)$$

$$\llbracket \pi \vdash \underline{m} : \mathbf{int} \rrbracket \gamma = \iota_{\uparrow} m$$

$$\llbracket \pi \vdash \ominus^n (t_1, \dots, t_n) : \mathbf{int} \rrbracket \gamma = \ominus^n_{\perp} (d_1, \dots, d_n)$$

$$\text{where } d_j = \llbracket \pi \vdash t_j : \mathbf{int} \rrbracket \gamma$$

$$\llbracket \pi \vdash fst\, t : \theta \rrbracket \gamma = d_0$$

$$\text{where } (d_0, d_1) = \llbracket \pi \vdash t : \theta \times \theta' \rrbracket \gamma$$

$$\llbracket \pi \vdash snd\, t : \theta \rrbracket \gamma = d_1$$

$$\text{where } (d_0, d_1) = \llbracket \pi \vdash t : \theta \times \theta' \rrbracket \gamma$$

$$\llbracket \pi \vdash (t_0, t_1) : \theta \times \theta' \rrbracket \gamma = (\llbracket \pi \vdash t_0 : \theta \rrbracket \gamma, \llbracket \pi \vdash t_1 : \theta' \rrbracket \gamma)$$

$$\llbracket \pi \vdash ifz\, t\, .\, t' : \theta \rrbracket \gamma = (\hat{\lambda} z . \text{if } z = 0 \text{ then } d_0 \text{ else } d_1)_{\perp\!\!\perp}\, d$$

$$\text{where } d = \llbracket \pi \vdash t : \mathbf{int} \rrbracket \gamma \text{ and } (d_0, d_1) = \llbracket \pi \vdash t' : \theta \times \theta \rrbracket \gamma$$

Fig. 2: Denotational semantics.

Semantic Chain
The semantics of a typing judgment can be thought of as the limit of increasingly better defined denotational values. In Figure 3 we define a family of functions $\llbracket \_ \rrbracket_i$, indexed by natural numbers. It is easy to see that $\llbracket \pi \vdash t : \theta \rrbracket_i \sqsubseteq \llbracket \pi \vdash t : \theta \rrbracket_{i+1}$, thus the sequence $\llbracket \pi \vdash t : \theta \rrbracket_i$ forms a chain in the domain $[\llbracket \pi \rrbracket \to \llbracket \theta \rrbracket]$ and its supremum is $\llbracket \pi \vdash t : \theta \rrbracket$.

Later we prove that each element of the semantic chain $\llbracket \pi \vdash t : \theta \rrbracket_i$ approximates the compiled code for t, a key point in the correctness proof.

3 The Target Language

3.1 The Abstract Machine

We use an abstract machine as our target language. Abstract machines are often used as an idealized model of execution; they are in general simpler than a real machine since they lack certain hardware details that would otherwise complicate the reasoning and the analysis of its behaviour. They are therefore suitable as an intermediate language for compilation [10]. We proceed with the definition of the

$$
\begin{aligned}
&[\![\_]\!]_i : |\pi \vdash t : \theta| \to [\![\pi]\!] \to [\![\theta]\!] \\
&[\![\pi \vdash t : \theta]\!]_0\, \gamma = \bot \\
&[\![\pi \vdash \lambda t : \theta \to \theta']\!]_{i+1}\, \gamma = \hat{\lambda}\, a\, .\, [\![\theta :: \pi \vdash t : \theta']\!]_i\, (a, \gamma) \\
&[\![\pi \vdash t_1\, t_2 : \theta']\!]_{i+1}\, \gamma = ([\![\pi \vdash t_1 : \theta \to \theta']\!]_i\, \gamma)\, ([\![\pi \vdash t_2 : \theta]\!]_i\, \gamma) \\
&[\![\pi \vdash \overline{n} : \pi\, .\, n]\!]_{i+1}\, \gamma = \gamma \swarrow n \\
&[\![\pi \vdash rec\, t : \theta]\!]_{i+1}\, \gamma = [\![\pi \vdash t : \theta \to \theta]\!]_i\, \gamma\, ([\![\pi \vdash rec\, t : \theta]\!]_i\, \gamma) \\
&[\![\pi \vdash \underline{m} : \mathbf{int}]\!]_{i+1}\, \gamma = \iota_{\uparrow}\, m \\
&[\![\pi \vdash \ominus^n\, t_1, \ldots, t_n : \mathbf{int}]\!]_{i+1}\, \gamma = \ominus^n_{\bot}\, (d_1, \ldots, d_n) \\
&\quad \text{where } d_j = [\![\pi \vdash t_j : \mathbf{int}]\!]_i\, \gamma \\
&[\![\pi \vdash \mathit{fst}\, t : \theta]\!]_{i+1}\, \gamma = d_0 \\
&\quad \text{where } (d_0, d_1) = [\![\pi \vdash t : \theta \times \theta']\!]_i\, \gamma \\
&[\![\pi \vdash snd\, t : \theta]\!]_{i+1}\, \gamma = d_1 \\
&\quad \text{where } (d_0, d_1) = [\![\pi \vdash t : \theta \times \theta']\!]_i\, \gamma \\
&[\![\pi \vdash \mathit{ifz}\, t\, .\, t' : \theta]\!]_{i+1}\, \gamma = (\hat{\lambda}\, z\, .\, \text{if } z = 0 \text{ then } d_0 \text{ else } d_1)_{\perp\!\!\!\perp}\, d \\
&\quad \text{where } d = [\![\pi \vdash t : \mathbf{int}]\!]_i\, \gamma \text{ and } (d_0, d_1) = [\![\pi \vdash t' : \theta \times \theta]\!]_i\, \gamma
\end{aligned}
$$

Fig. 3: Semantic chain.

components of our machine. The instructions are the following:

Definition 3.1 (Machine instructions).

$$
\begin{aligned}
I \ni c, c' ::= \;& \mathsf{Grab} \triangleright c \mid \mathsf{Push}\; c \triangleright c' \mid \mathsf{Access}\; n \\
\mid \;& \mathsf{Fix} \triangleright c \mid \mathsf{Pair}\; (c, c') \mid \mathsf{Fst} \mid \mathsf{Snd} \\
\mid \;& \mathsf{Frame}\; \ominus^n \mid \mathsf{Op} \mid \mathsf{Const}\; m
\end{aligned}
$$

The first three instructions correspond to the classic KAM that are sufficient to evaluate the pure lambda calculus. We have added new instructions to handle recursion, strict arithmetic operators, pairs, and conditionals. We now define the components of the machine and some meta-variables as follows:

Definition 3.2 (Machine components).

$$
\begin{array}{ll}
\text{Closures:} & \Gamma \ni \alpha ::= (c, \eta) \\
\text{Environments:} & H \ni \eta ::= [] \mid \alpha :: \eta \\
\text{Operators:} & Ops \ni \ominus^n \\
\text{Stack values:} & M \ni \mu ::= \alpha \mid [\ominus^n\ \overline{m}\ \bullet\ \overline{\alpha}] \mid \langle \alpha, \alpha' \rangle \\
\text{Stacks:} & S \ni s ::= [] \mid \mu :: s \\
\text{Configurations:} & W = \Gamma \times S \ni w ::= (\alpha, s)
\end{array}
$$

A machine configuration is a pair (α, s) where α is a closure and s is a stack. A closure is also a pair (c, η) where c is an instruction and η is a machine environment; which is itself a list of closures. A stack value can be either a closure, a frame, or a pair of closures. We use *frames* [27] to store the arguments of operators throughout execution: a frame $[\ominus^n\ \overline{m}\ \bullet\ \overline{\alpha}]$ has three components: the list $\overline{m}$ of arguments already computed, a hole to indicate the argument that is being computed at the time, and a list $\overline{\alpha}$ of closures for computing the remaining arguments. The transition rules of the KAM are shown in Figure 4; they define a deterministic relation $\longmapsto\ \subseteq\ W \times W$. We use the symbol "|" in the rules to help the reader to distinguish the components of the configuration.

3.2 Compilation

Now that we have defined both the source and the target language, we present in Figure 5 the compilation function: each well-typed term is mapped into KAM's code. For a closed term t of type **int**, we expect that the execution of $((\llparenthesis\, t\, \rrparenthesis_{,\mathbf{int}}, []), [])$ leads to a configuration $((\mathsf{Const}\ m, \eta), [])$ if $[\![\ \vdash\ t\ :\ \mathbf{int}\,]\!]() = \iota_{\uparrow} m$. In the next sections we will prove that statement.

As a simple example, consider the compilation of the term $(\lambda x.\, x\, *\, \underline{2})\, \underline{3}$ that with our syntax is written $(\lambda\, (*)\, (\overline{0},\, \underline{2}))\, \underline{3}$,

$$
\begin{gathered}
\llparenthesis\, (\lambda\, (*)\, (\overline{0},\, \underline{2}))\, \underline{3}\, \rrparenthesis_{\pi,\,\mathbf{int}} = \\
\mathsf{Push}\ (\mathsf{Const}\ 3) \triangleright \mathsf{Grab} \triangleright \mathsf{Push}\ (\mathsf{Const}\ 2) \triangleright \mathsf{Push}\ (\mathsf{Access}\ 0) \triangleright \mathsf{Frame}\ (*)\ .
\end{gathered}
$$

A step-by-step execution of this code might be useful to understand the transition rules. In particular, this example illustrates how frames are used to compute

$$
\begin{array}{lllll}
(\mathsf{Grab} \rhd c, \eta) & \mid \alpha :: s & \longmapsto & (c, \alpha :: \eta) & \mid s \\
(\mathsf{Push}\ c \rhd c', \eta) & \mid s & \longmapsto & (c', \eta) & \mid (c, \eta) :: s \\
(\mathsf{Access}\ n, \eta) & \mid s & \longmapsto & \eta\,.\,n & \mid s \\
 & \quad \text{if } n < |\eta| & & & \\
(\mathsf{Fix} \rhd c, \eta) & \mid s & \longmapsto & (c, \eta) & \mid \alpha :: s \\
 & \quad \text{where } \alpha = (\mathsf{Fix} \rhd c, \eta) & & & \\
(\mathsf{Pair}\ (c_0, c_1), \eta) & \mid \alpha :: s & \longmapsto & \alpha & \mid \langle \alpha_0,\, \alpha_1 \rangle :: s \\
 & \quad \text{where } \alpha_i = (c_i, \eta) & & & \\
(\mathsf{Fst}, \eta) & \mid \langle \alpha_0,\, \alpha_1 \rangle :: s & \longmapsto & \alpha_0 & \mid s \\
(\mathsf{Snd}, \eta) & \mid \langle \alpha_0,\, \alpha_1 \rangle :: s & \longmapsto & \alpha_1 & \mid s \\
(\mathsf{Frame}\ \ominus^n, \eta) & \mid \alpha_1 :: \overline{\alpha} :: s & \longmapsto & \alpha_1 & \mid [\ominus^n \ \bullet\ \overline{\alpha}] :: s \\
 & \quad \text{if } |\overline{\alpha}| = n - 1 & & & \\
(\mathsf{Const}\ m_0, \eta) & \mid [\ominus^n\ \overline{m}\ \bullet\ \alpha', \overline{\alpha}] :: s & \longmapsto & \alpha' & \mid [\ominus^n\ \overline{m}, m_0\ \bullet\ \overline{\alpha}] :: s \\
(\mathsf{Const}\ m_0, \eta) & \mid [\ominus^n\ \overline{m} \bullet] :: s & \longmapsto & (\mathsf{Const}\ m, \eta) & \mid \text{s} \\
 & \quad \text{where } m = [\![\, \ominus^n \,]\!]\, \overline{m}, m_0 \text{ and } |\overline{m}| = n - 1 & & & \\
(\mathsf{Const}\ m_0, \eta) & \mid \langle \alpha_0,\, \alpha_1 \rangle :: s & \longmapsto & \alpha_i & \mid \text{s} \\
 & \quad \text{where } i = 0 \text{ if } m_0 = 0 \text{ and } i = 1 \text{ otherwise.} & & &
\end{array}
$$

Fig. 4: Machine transitions.

the arguments of a strict operator:

$$
\begin{array}{lr}
\mathsf{Push}\ (\mathsf{Const}\ 3) \rhd \mathsf{Grab} \rhd \mathsf{Push}\ (\mathsf{Const}\ 2) \rhd \mathsf{Push}\ (\mathsf{Access}\ 0) \rhd \mathsf{Frame}\ (*),\ \eta,\ s & \longmapsto \\
\mathsf{Grab} \rhd \mathsf{Push}\ (\mathsf{Const}\ 2) \rhd \mathsf{Push}\ (\mathsf{Access}\ 0) \rhd \mathsf{Frame}\ (*),\ \eta,\ (\mathsf{Const}\ 3, \eta) :: s & \longmapsto \\
\mathsf{Push}\ (\mathsf{Const}\ 2) \rhd \mathsf{Push}\ (\mathsf{Access}\ 0) \rhd \mathsf{Frame}\ (*),\ (\mathsf{Const}\ 3, \eta) :: \eta,\ s & \longmapsto \\
\mathsf{Push}\ (\mathsf{Access}\ 0) \rhd \mathsf{Frame}\ (*),\ \eta',\ (\mathsf{Const}\ 2, \eta') :: s \ \text{ where } \eta' = (\mathsf{Const}\ 3, \eta) :: \eta & \longmapsto \\
\mathsf{Frame}\ (*),\ \eta',\ (\mathsf{Access}\ 0, \eta') :: (\mathsf{Const}\ 2, \eta') :: s & \longmapsto \\
\mathsf{Access}\ 0,\ \eta',\ [(*)\ \bullet\ (\mathsf{Const}\ 2, \eta')] :: s & \longmapsto \\
\mathsf{Const}\ 3,\ \eta,\ [(*)\ \bullet\ (\mathsf{Const}\ 2, \eta')] :: s & \longmapsto \\
\mathsf{Const}\ 2,\ \eta',\ [(*)\ 3 \bullet] :: s & \longmapsto \\
\mathsf{Const}\ 6,\ \eta',\ s\ . &
\end{array}
$$

$$
\begin{aligned}
\llparenthesis\, \lambda\, t \,\rrparenthesis_{\pi,\,\theta \to \theta'} &= \mathsf{Grab} \rhd \llparenthesis\, t \,\rrparenthesis_{\theta \,::\, \pi,\,\theta'} \\
\llparenthesis\, t_1\ t_2 \,\rrparenthesis_{\pi,\,\theta'} &= \mathsf{Push}\ (\llparenthesis\, t_2 \,\rrparenthesis_{\pi,\,\theta}) \rhd \llparenthesis\, t_1 \,\rrparenthesis_{\pi,\,\theta \to \theta'} \\
\llparenthesis\, \overline{n} \,\rrparenthesis_{\pi,\,\pi\,.\,n} &= \mathsf{Access}\ n \\
\llparenthesis\, rec\ t \,\rrparenthesis_{\pi,\,\theta} &= \mathsf{Fix} \rhd (\llparenthesis\, t \,\rrparenthesis_{\pi,\,\theta \to \theta}) \\
\llparenthesis\, \underline{m} \,\rrparenthesis_{\pi,\,\mathbf{int}} &= \mathsf{Const}\ m \\
\llparenthesis\, \ominus^n\ (t_1, \ldots, t_n) \,\rrparenthesis_{\pi,\,\mathbf{int}} &= \mathsf{Push}\ (\llparenthesis\, t_n \,\rrparenthesis_{\pi,\,\mathbf{int}}) \rhd \ldots \rhd \mathsf{Push}\ (\llparenthesis\, t_1 \,\rrparenthesis_{\pi,\,\mathbf{int}}) \rhd \mathsf{Frame}\ \ominus^n \\
\llparenthesis\, (t_0,\, t_1) \,\rrparenthesis_{\pi,\,\theta \times \theta'} &= \mathsf{Pair}\ (\llparenthesis\, t_0 \,\rrparenthesis_{\pi,\,\theta},\ \llparenthesis\, t_1 \,\rrparenthesis_{\pi,\,\theta'}) \\
\llparenthesis\, fst\ t \,\rrparenthesis_{\pi,\,\theta} &= \mathsf{Push\ Fst} \rhd \llparenthesis\, t \,\rrparenthesis_{\pi,\,\theta \times \theta'} \\
\llparenthesis\, snd\ t \,\rrparenthesis_{\pi,\,\theta} &= \mathsf{Push\ Snd} \rhd \llparenthesis\, t \,\rrparenthesis_{\pi,\,\theta \times \theta'} \\
\llparenthesis\, ifz\, t\ .\ t' \,\rrparenthesis_{\pi,\,\theta} &= \mathsf{Push}\ \llparenthesis\, t \,\rrparenthesis_{\pi,\,\mathbf{int}} \rhd \llparenthesis\, t' \,\rrparenthesis_{\pi,\,\theta \times \theta}
\end{aligned}
$$

Fig. 5: Compilation function.

Note that, unlike the SECD machine, the Krivine abstract machine does not store the final value in the top of stack. Instead, one has to look at the entire final closure to infer the value computed by the machine. In this example, assuming $s = []$, the final closure is $(\mathsf{Const}\ 6, \eta')$.

4 Biorthogonality

Biorthogonality is a well-known technique that has been used in program equivalence [23], realizability [14], and compiler correctness [4,12]. The general idea can be explained as follows.

Let $\mathcal{E}$ and $\mathcal{T}$ be two sets, and $\models\ \subseteq\ \mathcal{E} \times \mathcal{T}$ a relation between those two sets. If we think $\mathcal{T}$ as a set of *tests*, and $\models$ as a satisfability relation, then $e \models t$ states whether an element $e \in \mathcal{E}$ satisfies the test $t \in \mathcal{T}$. Suppose $\mathcal{T}_0$ is a subset of $\mathcal{T}$, we write $\mathcal{T}_0^{\top}$ for the set of elements that satisfy *all* the tests in $\mathcal{T}_0$:

$$\mathcal{T}_0^{\top} = \{e \in \mathcal{E} \mid \text{for all}\, t \in \mathcal{T}_0,\ e \models t\} \ .$$

As a concrete example, if $\mathcal{T}$ are formulas and $\mathcal{E}$ are models of a particular logic, then $\mathcal{T}_0^{\top}$ is the set of models that satisfy all the formulas in $\mathcal{T}_0$. We can also define a dual operation to obtain the set of tests that are satisfied by all the elements in a subset $\mathcal{E}_0 \subseteq \mathcal{E}$:

$$\mathcal{E}_0^{\perp} = \{t \in \mathcal{T} \mid \text{for all}\, e \in \mathcal{E}_0,\ e \models t\} \ .$$

The operators $\perp$ and $\top$ are often called *orthogonal*, and it is a well-known fact that they form an antitone Galois connection [7,22]. As a consequence, the function $(\_)^{\perp\top} : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ is a closure operator for the poset $\langle \mathcal{P}(\mathcal{E}), \subseteq \rangle$.

The key point of biorthogonality is that for a given set $\mathcal{E}_0$ we can obtain the set $\mathcal{E}_0^{\perp\top}$ which is an extension of $\mathcal{E}_0$ that satisfies all the tests in $\mathcal{E}_0^{\perp}$. That is, we are able to extend the set $\mathcal{E}_0$ without "losing" any test and hence maintaining the satisfability relation. In the next section we present the concrete use of biorthogonality that is useful for our purposes.

5 Denotational Approximation

In this section we prove the correctness of the compiler for terms whose denotation is different from bottom. The strategy is to define a logical relation which states when a denotational value d approximates a closure α at type θ; then we prove the fundamental lemma of logical relations, finally concluding that the compilation of a term is approximated by every element of its semantic chain.

Our logical relation is parameterized over a set of *observations* of the KAM. Given a set of *observations* $R \subseteq W = \Gamma \times S$, we use biorthogonality to define this logical relation – following the terminology of the previous section, we will say that stacks are *tests* for closures. All the reasoning of this section assumes that R is closed by anti-execution, i.e. $(\alpha,\, s) \in R$ and $(\alpha',\, s') \longmapsto^* (\alpha,\, s)$ implies $(\alpha',\, s') \in R$; moreover, to keep our reasoning constructive we will also ask for the existence of an "excluded" closure: that is an $\hat{\alpha}$ such that $(\hat{\alpha},\, s) \notin R$ for any $s \in S$ (a closure that does not satisfy any test). Termination is an example of an observation which is closed by anti-execution and has an excluded closure (think of the compilation of $rec\ \lambda x.x$).

The relations $\rhd^{\theta}$, $\blacktriangleright^{\theta} \subseteq \Gamma \times [\![\, \theta \,]\!]$ are defined by mutual recursion over types as follows:

Definition 5.1 (Denotational approximation).

$$\begin{array}{l}
\alpha \ \rhd^{\theta} \ \perp \ \text{for any closure } \alpha \ , \\
(\mathsf{Const}\ m, \eta) \ \rhd^{\mathbf{int}} \ \iota_{\uparrow} m \ , \\
(\mathsf{Grab} \rhd c, \eta) \ \rhd^{\theta \to \theta'} \ f \quad \text{iff} \quad \text{for all } \alpha \text{ and } d, \text{ if } \alpha \ \blacktriangleright^{\theta} \ d \text{ then } (c, \alpha :: \eta) \ \blacktriangleright^{\theta'} \ f\, d \ , \\
(\mathsf{Pair}\ (c_0, c_1), \eta) \ \rhd^{\theta \times \theta'} \ (d_0,\, d_1) \quad \text{iff} \quad (c_0, \eta) \ \blacktriangleright^{\theta} \ d_0 \text{ and } (c_1, \eta) \ \blacktriangleright^{\theta'} \ d_1 \ , \\
\alpha \ \blacktriangleright^{\theta} \ d \quad \text{iff} \quad \alpha \in \Gamma^{\theta}(d)^{\perp_R \top_R}, \quad \text{where} \quad \Gamma^{\theta}(d) = \{\, \alpha \mid \alpha \ \rhd^{\theta} \ d \,\} \ .
\end{array}$$

Let $\alpha \in \Gamma$ and $d \in [\![\, \theta \,]\!]$, then $\alpha \ \blacktriangleright^{\theta} \ d$ is read "d is an approximation of type θ to the closure α" but we often omit the type and just write "d approximates the closure α". In a sense, the set $\Gamma^{\theta}(d)$ contains the closures that are "strongly approximated" by d, and $\Gamma^{\theta}(d)^{\perp_R \top_R}$ is the extension of this set obtained through the orthogonal operators. Note that, in this definition, the transitions of the machine are not relevant except in the restrictions we imposed to the set of observations.

We let $\perp$ to be an approximation of any closure; this is consistent with the idea of approximation since $\perp$ is a value with a minimum amount of information. Since $\Gamma^{\theta}(\perp) = \Gamma$, by the "excluded closure" assumption, we know $\Gamma^{\theta}(\perp)^{\perp_R} = \emptyset$; consequently $s \in \Gamma^{\theta}(d)^{\perp_R}$ always implies $d \neq \perp$. The fact that R is closed by anti-execution leads to the following (trivial) lemma:

Lemma 5.2 *Let $\alpha, \alpha' \in \Gamma$ and $s, s' \in S$. If $(\alpha', s') \longmapsto^* (\alpha, s)$, $\alpha \blacktriangleright^\theta d$ and $s \in \Gamma^\theta(d)^{\perp_R}$ then $(\alpha', s') \in R$.*

We define a relation $\blacktriangleright^\pi$ between machine-level environments and denotational environments, as a point-wise extension of the $\blacktriangleright^\theta$ relation.

Definition 5.3 (Denotational approximation for environments).

$$[] \blacktriangleright^{[]} ()$$
$$\alpha :: \eta \blacktriangleright^{\theta :: \pi} (d, \gamma) \quad \text{iff} \quad \alpha \blacktriangleright^\theta d \quad \text{and} \quad \eta \blacktriangleright^\pi \gamma.$$

If we follow the general schema of biorthogonality presented before, the elements of $\Gamma^\theta(d)^{\perp_R}$ are the tests that a closure must satisfy to be approximated by the value d. Since this set depends on the type θ, we can also talk about "tests of type θ" (which is a frequent terminology in the literature about *realizability*). In Krivine's realizability tests of arrow types $\theta \to \theta'$ are stacks $\alpha :: s$ where α is a realizer of θ and s is a test of type θ'; as the following lemma shows, that is a good characterization in our setting.

Lemma 5.4 *Let $\alpha \in \Gamma$, $s \in S$, $f \in [\![\theta \to \theta']\!]$, $d \in [\![\theta]\!]$. If $\alpha \blacktriangleright^\theta d$ and $s \in \Gamma^{\theta'}(f\,d)^{\perp_R}$ then $\alpha :: s \in \Gamma^{\theta \to \theta'}(f)^{\perp_R}$.*

Proof. In order to prove $\alpha :: s \in \Gamma^{\theta \to \theta'}(f)^{\perp_R}$ we take $\alpha' \in \Gamma^{\theta \to \theta'}(f)$ and prove $(\alpha', \alpha :: s) \in R$. Since $\Gamma^{\theta'}(f\,d)^{\perp_R}$ is not empty we know $f\,d \neq \perp$ and hence $f \neq \perp$. Therefore, we have by inversion that α' is a closure of the form $(\mathsf{Grab} \rhd c, \eta)$ where $\eta \in H$ and $c \in I$. Moreover, since $\alpha \blacktriangleright^\theta d$ we have $(c, \alpha :: \eta) \blacktriangleright^{\theta'} f\,d$. Consequently, since $(\alpha', \alpha :: s) \longmapsto ((c, \alpha :: \eta), s)$ and $s \in \Gamma^{\theta'}(f\,d)^{\perp_R}$ we conclude $(\alpha', \alpha :: s) \in R$. □

Analogously, it is easy to see that tests for a product type $\theta \times \theta'$ can be defined as those stacks having a "projection" at the top followed by a test for the projected type. For the sake of brevity we do not show characterization of tests for **int**, which can be found in the formalization.

Lemma 5.5 *Let $s \in S$, $\eta \in H$, $d_0 \in [\![\theta]\!]$, $d_1 \in [\![\theta']\!]$. If $s \in \Gamma^\theta(d_0)^{\perp_R}$ then $(\mathsf{Fst}, \eta) :: s \in \Gamma^{\theta \times \theta'}((d_0, d_1))^{\perp_R}$. In a similar manner, if $s \in \Gamma^{\theta'}(d_1)^{\perp_R}$ then $(\mathsf{Snd}, \eta) :: s \in \Gamma^{\theta \times \theta'}((d_0, d_1))^{\perp_R}$.*

The next lemma provides various ways to combine closures using machine instructions, in order to obtain new approximations of different types. This is an important property since it essentially says that we can merge "correct" code fragments (potentially generated by different compilers, or hand-written) to obtain a larger code fragment that is also correct.

Lemma 5.6 (i) *If $(c, \eta) \blacktriangleright^{\theta \to \theta'} f$ and $(c', \eta) \blacktriangleright^\theta d$, then $(\mathsf{Push}\ c' \rhd c, \eta) \blacktriangleright^{\theta'} f\,d$.*

(ii) *If $\eta \blacktriangleright^\pi \gamma$ and $n < |\pi|$, then $(\mathsf{Access}\ n, \eta) \blacktriangleright^{\pi.n} \gamma \downarrow n$.*

(iii) *If $(c, \eta) \blacktriangleright^{\theta \to \theta} f$ and $(\mathsf{Fix} \rhd c, \eta) \blacktriangleright^\theta d$, then $(\mathsf{Fix} \rhd c, \eta) \blacktriangleright^\theta f\,d$.*

(iv) *If* $(c_i, \eta) \blacktriangleright^{\mathbf{int}} d_i$ *for all* $i \in \{1, \dots, n\}$*, then* $(\mathsf{Push}\ c_n \rhd \dots \rhd \mathsf{Push}\ c_1 \rhd \mathsf{Frame}\ \ominus^n, \eta) \blacktriangleright^{\mathbf{int}} \ominus^n_{\perp}(d_1, \dots, d_n)$.

(v) *If* $(c, \eta) \blacktriangleright^{\theta \times \theta'} (d_0, d_1)$*, then* $(\mathsf{Push\ Fst} \rhd c, \eta) \blacktriangleright^{\theta} d_0$ *and* $(\mathsf{Push\ Snd} \rhd c, \eta) \blacktriangleright^{\theta'} d_1$.

(vi) *If* $(c, \eta) \blacktriangleright^{\theta \times \theta} (d_0, d_1)$ *and* $(c', \eta) \blacktriangleright^{\mathbf{int}} d$*, then* $(\mathsf{Push}\ c' \rhd c, \eta) \blacktriangleright^{\theta} (\hat{\lambda} z \,.\, \mathit{if}\ z = 0\ \mathit{then}\ d_0\ \mathit{else}\ d_1)_{\perp\!\!\perp}\, d$.

By using Lemma 5.6, we can easily prove that the compilation of a typing derivation is related with every element of its semantic chain.

Lemma 5.7 (Denotational approximation of compiled code) *If* $\eta \blacktriangleright^{\pi} \gamma$ *then for all* i, $((\!|\, t \,|\!)_{\pi,\theta}, \eta) \blacktriangleright^{\theta} [\![\, \pi \vdash t : \theta \,]\!]_i\, \gamma$.

Proof. The proof is by induction in the typing derivation. However, in the case of the fixed-point operator, we need a nested induction over the index i. We now show the proof for that case.

Let $c = (\!|\, t \,|\!)_{\pi,\theta \to \theta}$, let $f_i = [\![\, \pi \vdash t : \theta \to \theta \,]\!]_i\, \gamma$, and $d_i = [\![\, \pi \vdash \mathit{rec}\, t : \theta \,]\!]_i\, \gamma$. We want to prove $(\mathsf{Fix} \rhd c, \eta) \blacktriangleright^{\theta} d_i$ by induction over i. The case $i = 0$ is trivial since $d_0 = \perp$ (and $\perp$ is always an approximation). In the inductive case, we assume $(\mathsf{Fix} \rhd c, \eta) \blacktriangleright^{\theta} d_i$ and prove $(\mathsf{Fix} \rhd c, \eta) \blacktriangleright^{\theta} d_{i+1}$. We have $(c, \eta) \blacktriangleright^{\theta \to \theta} f_i$ by inductive hypothesis, and hence by Lem. 5.6 we get $(\mathsf{Fix} \rhd c, \eta) \blacktriangleright^{\theta \to \theta} f_i\, d_i = d_{i+1}$.□

It is possible to relate the compilation of a term directly to its semantics by defining an admissible extension of $\blacktriangleright^{\theta}$; the interested reader is invited to consult this extension in the formalization.

Note that Lemma 5.7 holds for any choice of R that satisfies the two conditions we stated before: it must be closed by anti-execution and there must be an excluded closure. In particular, to prove a "standard" version of the compiler correctness theorem for closed terms of type **int** one fixes the set of observation to be $R^m = \{ w \in W \mid w \longmapsto^* ((\mathsf{Const}\ m, \eta), [\,]), \text{for any environment } \eta \}$.

Lemma 5.8 *If* t *is a closed term, and* $[\![\, [\,] \vdash t : \mathbf{int} \,]\!]\, () = \iota_{\uparrow} m$*, then there is some environment* η *such that* $(((\!|\, t \,|\!)_{[\,],\mathbf{int}}, [\,]), [\,]) \longmapsto^* ((\mathsf{Const}\ m, \eta), [\,])$.

Proof. In order to prove this result, we use Lemma 5.7 choosing R^m as the set of observations. We have then $((\!|\, t \,|\!)_{[\,],\mathbf{int}}, [\,]) \blacktriangleright^{\mathbf{int}} [\![\, [\,] \vdash t : \mathbf{int} \,]\!]_i\, ()$ for all $i \in \mathbb{N}$. But since $[\![\, \mathbf{int} \,]\!]$ is a flat domain, there is a $j \in \mathbb{N}$ such that $[\![\, [\,] \vdash t : \mathbf{int} \,]\!]_j\, () = \iota_{\uparrow} m$ and hence we have $((\!|\, t \,|\!)_{[\,],\mathbf{int}}, [\,]) \blacktriangleright^{\mathbf{int}} \iota_{\uparrow} m$. Since $[\,] \in \Gamma^{\mathbf{int}}(\iota_{\uparrow} m)^{\perp_{R^m}}$, by Lemma 5.2 we have $(((\!|\, t \,|\!)_{[\,],\mathbf{int}}, [\,]), [\,]) \in R^m$, which is what we wanted by the definition of R^m.□

In order to prove a similar lemma for *divergent* terms we use another logical relation with similar properties.

6 Step-indexed Logical Relations

To prove the correctness of the compiler for terminating terms it was necessary to relate code fragments with each element of the semantic chain; as the proof of

Lemma 5.7 shows, this allowed us to make a nested induction when considering the case for $rec\, t$. If the source language were strong normalizing (i.e., by setting aside the fixed point operator), there would be no need to introduce the semantic chain and correctness would directly relate the compiled code with the semantics of the term.

A more general approach to deal with the subtleties introduced by the recursion operator is using *step-indexed* logical relations. This method has been used alone and in combination with biorthogonality to obtain proofs of compiler correctness [5,12] and program equivalence [1,11], among other topics. The basic idea is that the logical relation is defined incrementally through a *family* of relations indexed by natural numbers. Thus, one can prove different properties about this relation using induction in the index. Step-indexing is helpful to capture a notion of approximation at the operational side, analogous to that provided by the semantic chain at the denotational side. In this section we introduce step-indexed families and show some results regarding the combination of these families with the orthogonal operators.

Definition 6.1 (Step-indexed family). A family $R_i \subseteq A$ is step-indexed if $R_0 = A$ and for all $i \in \mathbb{N}$, $R_{i+1} \subseteq R_i$.

An example of a step-indexed family over the set of KAM configurations is given by letting R_i be the set of configurations that can make at least i transition steps. While in the previous section we parameterized all the development over a set of observations, in the next section we will work with any step-indexed family of observations closed by anti-execution.

Given a family of observations $R_i \subseteq \mathcal{E} \times \mathcal{T}$, we can define a binary relation $\boldsymbol{R} \subseteq \hat{\mathcal{E}} \times \hat{\mathcal{T}}$ over indexed elements $\hat{\mathcal{E}} = \mathbb{N} \times \mathcal{E}$ and indexed tests $\hat{\mathcal{T}} = \mathbb{N} \times \mathcal{T}$.

Definition 6.2 Let $R_i \subseteq \mathcal{E} \times \mathcal{T}$ be an indexed family, then $\boldsymbol{R} \subseteq \hat{\mathcal{E}} \times \hat{\mathcal{T}}$ is given by $(i, e)\, \boldsymbol{R}\, (j, t)$ iff $(e, t) \in R_{min(i,j)}$.

Let us make explicit the definition of the orthogonal operator $(\_)^{\perp_{\boldsymbol{R}}}$ for the relation $\boldsymbol{R}$:

$$X^{\perp_{\boldsymbol{R}}} = \{\, (j, t) \in \hat{\mathcal{T}} \mid \text{for all}\, (i, e) \in X, (i, e)\ \boldsymbol{R}\ (j, t) \,\}\ ,$$

which means that to prove $(j, t) \in X^{\perp_{\boldsymbol{R}}}$ one has to check that *every* element (i, e) in X is related with (j, t) via $\boldsymbol{R}$. Now we prove that one can simplify the reasoning when R_i is step-indexed.

Definition 6.3 (Down-closed set). For any set $\mathcal{E}$, we say that $X \subseteq \hat{\mathcal{E}}$ is down-closed if whenever $(i, e) \in X$ and $j \leqslant i$, then $(j, e) \in X$.

Lemma 6.4 *Let $R_i \subseteq \mathcal{E} \times \mathcal{T}$ be step-indexed and $X \subseteq \hat{\mathcal{E}}$, then $X^{\perp_{\boldsymbol{R}}}$ is down-closed.*

Proof. Let $(j, t) \in X^{\perp_{\boldsymbol{R}}}$ and $i \leqslant j$. Suppose $(k, e) \in X$, then $(e, t) \in R_{min(k,j)}$. Since $min(k, i) \leqslant min(k, j)$, we have $R_{min(k,j)} \subseteq R_{min(k,i)}$ and hence $(e, t) \in R_{min(k,i)}$. Therefore, $(i, e) \in X^{\perp_{\boldsymbol{R}}}$. □

As the following lemma shows, when restricted to down-closed sets, one can give a simpler definition of $(\_)^{\perp_{\boldsymbol{R}}}$ in which it is only necessary to check those elements that satisfy $i \leqslant j$.

Lemma 6.5 *Let $R_i \subseteq \mathcal{E} \times \mathcal{T}$. If X is down-closed, then*

$$X^{\perp_R} = \{\, (j,t) \in \hat{\mathcal{T}} \mid \textit{for all}\,(i,e) \in X,\ i \leqslant j \ \textit{ implies } \ (e,t) \in R_i \,\} \ .$$

Since Lemmas 6.4 and 6.5 can also be proved for the operator $(\_)^{\top_R}$, we can construct an alternative definition of the closure operator.

Lemma 6.6 *Let $R_i \subseteq \mathcal{E} \times \mathcal{T}$ be step-indexed and $X \subseteq \hat{\mathcal{E}}$, then*

$$X^{\perp_R \top_R} = \{\, (j,e) \in \hat{\mathcal{E}} \mid \textit{for all}\,(i,t) \in X^{\perp_R},\ i \leqslant j \ \textit{ implies } \ (e,t) \in R_i \,\} \ .$$

7 Operational Approximation

In this section we define a relation of approximation from machine closures to denotational values. In the same vein as in section 5, these relations are defined by means of the closure operator associated to a set of observations. The logical relation is parameterized by a step-indexed family of relations $R_i \subseteq \Gamma \times S$ satisfying the following condition: $(\alpha, s) \in R_i$ and $(\alpha', s') \longmapsto (\alpha, s)$ imply $(\alpha', s') \in R_{i+1}$. Note that this condition implies that each relation of the family is closed by anti-execution.

This time, instead of working with a single relation of approximation, we define simultaneously two *families* of relations, $\lhd_i^\theta, \blacktriangleleft_i^\theta \subseteq \Gamma \times [\![\, \theta \,]\!]$, indexed by natural numbers. Roughly speaking, the index measures the "accuracy" of the approximation: the relation becomes finer as the index increases, starting with the total relation at index 0, where every closure approximates every denotation.

Definition 7.1 (Operational approximation).

$$\begin{array}{l}
\alpha \lhd_0^\theta d \ , \\
(\mathsf{Const}\ m, \eta) \lhd_i^{\mathbf{int}} \iota_\uparrow m \ , \\
(\mathsf{Grab} \rhd c, \eta) \lhd_i^{\theta \to \theta'} f \qquad \text{iff} \qquad \text{for all } k \leqslant i, \text{ if } \alpha \blacktriangleleft_k^\theta d \text{ then } (c, \alpha :: \eta) \blacktriangleleft_k^{\theta'} f\, d \ , \\
(\mathsf{Pair}\ (c_0, c_1), \eta) \lhd_i^{\theta \times \theta'} (d_0, d_1) \qquad \text{iff} \qquad (c_0, \eta) \blacktriangleleft_i^\theta d_0 \text{ and } (c_1, \eta) \blacktriangleleft_i^{\theta'} d_1 \ , \\
\alpha \blacktriangleleft_k^\theta d \qquad \text{iff} \qquad (k, \alpha) \in \Gamma^\theta(d)^{\perp_R \top_R}, \text{ where } \Gamma^\theta(d) = \{\, (k, \alpha) \mid \alpha \lhd_k^\theta d \,\} \ .
\end{array}$$

While in the denotational approximation (Def. 5.1), $\perp$ was strongly related with any closure, now $\perp$ is strongly approximated by any closure only at level 0. As a consequence, we have $(k+1, \alpha) \notin \Gamma^{\mathbf{int}}(\perp)$; from this and from the fact that R_i is step-indexed, is easy to show $\Gamma^{\mathbf{int}}(\perp)^{\perp_R} = \mathbb{N} \times S$. This implies that a closure α which approximates $\perp$ at every level must be a divergent closure.

Lemma 7.2 *Define $R_i = \{\, w \mid w$ can make at least i transition steps$\}$. Let $\alpha \in \Gamma$ such that $\alpha \blacktriangleleft_k^{\mathbf{int}} \perp$ for all $k \in \mathbb{N}$, then $(\alpha,\, s)$ diverges for any stack s.*

Proof. Given that $\Gamma^{\mathbf{int}}(\perp)^{\perp_R} = \mathbb{N} \times S$, we have that for any pair $(N, s) \in \mathbb{N} \times S$ it holds $(\alpha,\, s) \in R_N$. That is, $(\alpha,\, s)$ can make an arbitrarily large number of transition steps. □

The intuitive interpretation of the indices given above suggests that when α is an approximation at index k, it should also be an approximation at a smaller index $j \leqslant k$. In addition, the approximation relation is monotone with respect to the domain order.

Lemma 7.3 *Let $\alpha \in \Gamma$, $d \sqsubseteq d' \in [\![\,\theta\,]\!]$, and $j \leqslant k$. If $\alpha \lhd_k^\theta d$ then $\alpha \lhd_j^\theta d'$. Similarly, if $\alpha \blacktriangleleft_k^\theta d$ then $\alpha \blacktriangleleft_j^\theta d'$.*

From this lemma we deduce that $\Gamma^\theta(d)$ is down-closed and monotone: $d \sqsubseteq d'$ implies $\Gamma^\theta(d) \subseteq \Gamma^\theta(d')$. Moreover the family $\theta_k(\alpha) = \{d \mid \alpha \blacktriangleleft_k^\theta d\}$ is step-indexed over $[\![\,\theta\,]\!]$, and each $\theta_k(\alpha)$ is closed by suprema of chains.

It is not surprising that we can construct tests for compound types by combining tests for simpler types. Recall that this time "tests" are pairs (k, s) where $k \in \mathbb{N}$ and $s \in S$. We only show here one way to obtain tests for product types, and there are other possible combinations that can be examined in the formalization.

Lemma 7.4 *Let $\alpha \in \Gamma$, $s \in S$. If $\alpha \blacktriangleleft_k^{\mathbf{int}} \iota_\uparrow 0$ and $(k, s) \in \Gamma^\theta(d_0)^{\perp_R}$ then it holds $(k, \alpha :: s) \in \Gamma^{\theta \times \theta'}((d_0,\, d_1))^{\perp_R}$. Similarly, if $\alpha \blacktriangleleft_k^{\mathbf{int}} \iota_\uparrow m$ with $m \neq 0$ and $(k, s) \in \Gamma^\theta(d_1)^{\perp_R}$ then $(k, \alpha :: s) \in \Gamma^{\theta \times \theta'}((d_0,\, d_1))^{\perp_R}$.*

The following lemma presents the operational counterpart of Lemma 5.6, showing that approximations compose well with the constructors of the language. We show the proof only for the fixed point operator.

Lemma 7.5 (i) *If $(c, \eta) \blacktriangleleft_k^{\theta \to \theta'} f$ and $(c', \eta) \blacktriangleleft_k^\theta d$, then $(\mathsf{Push}\ c' \rhd c, \eta) \blacktriangleleft_k^{\theta'} f\, d$.*

(ii) *If $\eta \blacktriangleleft_k^\pi \gamma$ and $n < |\pi|$, then $(\mathsf{Access}\ n, \eta) \blacktriangleleft_k^{\pi.n} \gamma \swarrow n$.*

(iii) *If $(c, \eta) \blacktriangleleft_k^{\theta \to \theta} f$ then $(\mathsf{Fix} \rhd c, \eta) \blacktriangleleft_k^\theta \mathbf{Y}_{[\![\,\theta\,]\!]} f$.*

(iv) *If $(c_i, \eta) \blacktriangleleft_k^{\mathbf{int}} d_i$ for all $i \in \{1, \ldots, n\}$, then*
$(\mathsf{Push}\ c_n \rhd \ldots \rhd \mathsf{Push}\ c_1 \rhd \mathsf{Frame}\ \ominus^n, \eta) \blacktriangleleft_k^{\mathbf{int}} \ominus_\perp^n d_1, \ldots, d_n$.

(v) *If $(c, \eta) \blacktriangleleft_k^{\theta \times \theta'} (d_0,\, d_1)$, then $(\mathsf{Push\ Fst} \rhd c, \eta) \blacktriangleleft_k^\theta d_0$ and*
$(\mathsf{Push\ Snd} \rhd c, \eta) \blacktriangleleft_k^{\theta'} d_1$.

(vi) *If $(c, \eta) \blacktriangleleft_k^{\theta \times \theta} (d_0,\, d_1)$ and $(c', \eta) \blacktriangleleft_k^{\mathbf{int}} d$, then*
$(\mathsf{Push}\ c' \rhd c, \eta) \blacktriangleleft_k^\theta (\hat{\lambda}\, z\,.\, \mathit{if}\ z = 0\ \mathit{then}\ d_0\ \mathit{else}\ d_1)_{\perp\!\!\perp}\, d$.

Proof. Let $\alpha = (c, \eta)$, let $\alpha' = (\mathsf{Fix} \rhd c, \eta)$, and $d = \mathbf{Y}_{[\![\,\theta\,]\!]} f$. Our goal is to prove that $\alpha \blacktriangleleft_k^{\theta \to \theta} f$ implies $\alpha' \blacktriangleleft_k^\theta d$, we proceed by induction over k. The case $k = 0$ is trivial since $\blacktriangleleft_0^\theta = \Gamma \times [\![\,\theta\,]\!]$. Now we assume $\alpha \blacktriangleleft_{k+1}^{\theta \to \theta} f$ and prove $\alpha' \blacktriangleleft_{k+1}^\theta d$. We take $(l,\, s) \in \Gamma^\theta(d)^{\perp_R}$ with $l \leqslant k + 1$, and prove $(\alpha',\, s) \in R_l$. We have two cases depending on whether $l \leqslant k$ or $l = k + 1$. In the first case we use our inductive hypothesis $\alpha' \blacktriangleleft_k^\theta d$ to obtain $(\alpha',\, s) \in R_l$. Now assume $l = k + 1$. We have $(k + 1,\, s) \in \Gamma^\theta(d)^{\perp_R}$ and hence $(k,\, s) \in \Gamma^\theta(d)^{\perp_R}$. Since $d = f\, d$ we obtain $(k,\, s) \in \Gamma^\theta(f\, d)^{\perp_R}$. By inductive hypothesis we have $\alpha' \blacktriangleleft_k^\theta d$, and hence $(k,\, \alpha' :: s) \in \Gamma^{\theta \to \theta}(f)^{\perp_R}$. We had $\alpha \blacktriangleleft_{k+1}^{\theta \to \theta} f$ by assumption, and hence $(\alpha,\, \alpha' :: s) \in R_k$. Since $(\alpha',\, s) \longmapsto (\alpha,\, \alpha' :: s)$ we obtain $(\alpha',\, s) \in R_{k+1} = R_l$. □

The fundamental lemma of the logical relation, which states that the compilation of a typing derivation is an approximation of its semantics, is a direct consequence of Lemma 7.5.

Lemma 7.6 (Operational approximation of compiled code) *For all $i \in \mathbb{N}$, if $\eta \blacktriangleleft_i^{\pi} \gamma$, then $(\llparenthesis\, t\, \rrparenthesis_{\pi,\theta}, \eta) \blacktriangleleft_i^{\theta} [\![\, \pi \vdash t : \theta \,]\!]\, \gamma$.*

Here the relation $\blacktriangleleft_i^{\pi}$ is defined as a pointwise extension similarly to 5.3. Lemmas 7.6 and 7.2 lead to the following result.

Lemma 7.7 *If t is a closed term, and $[\![\, [] \vdash t : \mathbf{int} \,]\!]\, () = \bot$, then the configuration $((\llparenthesis\, t\, \rrparenthesis_{[],\mathbf{int}}, []), s)$ diverges for any stack s.*

Finally, as a consequence of lemmas 5.8 and 7.7, we can state a compiler correctness theorem.

Theorem 7.8 *(Compiler correctness) Suppose t is a closed term of type* **int**. *If $[\![\, [] \vdash t : \mathbf{int} \,]\!]\, () = \iota_{\uparrow}\, m$, then $((\llparenthesis\, t\, \rrparenthesis_{[],\mathbf{int}}, []), []) \longmapsto^{*} ((\mathsf{Const}\ m, \eta), [])$, for some $\eta \in H$. Otherwise, if $[\![\, [] \vdash t : \mathbf{int} \,]\!]\, () = \bot$, then $((\llparenthesis\, t\, \rrparenthesis_{[],\mathbf{int}}, []), [])$ diverges.*

8 Formalization

All the results presented in this paper has been completely formalized in the proof assistant Coq (version 8.4pl6 with Ssreflect 1.5). The formalization is constructive, as we do not assume any classical axiom. We invite the curious reader to download [25] and explore the formalization as it complements the content of this article.

Our formal development is based on a domain-theory library by Benton et al. [6] that provided us with the basis to formalize the denotational semantics of the language; our formalization would have taken much more time without that library. As useful as it was, we found some shortcomings that we turned into extensions:

- The original "extension" function, named `kleisli`, of the lifting monad has type `kleisli` $: (P \to Q_{\bot}) \to (P_{\bot} \to Q_{\bot})$. This operator is adequate for a call-by-value language; in our setting, however this is not enough, the semantics of the conditional asks for the following operator `gkleisli` $: (P \to D) \to (P_{\bot} \to D)$ where D is any pointed cpo (not necessarily obtained through lifting).
- A formalization of n-ary morphisms and finite products used to implement the semantics of n-ary operators and to prove some results about them.
- A variety of results regarding Cartesian closed categories, cpos and the computation of least upper bounds.

We also extended a formalization of sequences (finite and infinite) originally written in [18]. Our own development (excluding the domain-library) has 6134 lines of code in total, 2096 of which are specifications and 4038 are proofs.

9 Conclusion and Further Work

We have proved the correctness of a compiler for a call-by-name functional language by means of logical relations defined using biorthogonality and step-indexing. This abstract setting provides a certain degree of flexibility with respect to modifications of the execution environment and it is also modular with respect to the constructors of the language; in particular, the use of step-indexing enabled us to deal with inductive proofs in the presence of recursion.

This approach is similar to [4] but with some important differences due to the order of evaluation (call-by-name instead of call-by-value) and the nature of the abstract machine (KAM rather than the SECD machine). The lack of difference between values and terms in the call-by-name setting turns our logical relations simpler and more intuitive than those in [4]: there is only one kind of approximations on the operational side (closures) and there is no need for a “monadic lifting”. In addition, our definition of the orthogonal operators is simpler since there is no need to parameterize them using environments or any other kind of value.

As future work we plan to extend the source language by enriching the type system and adding new constructors. For example, in [26] we proved the correctness of a compiler for a higher-order imperative language with respect to the big-step operational semantics of the source language; it would be interesting to obtain a relational proof of compiler correctness. We also intend to investigate the application of this technique to lazy functional languages targeting the Sestoft abstract machine [28] or the STG machine [13]. We are also interested in applying the method to other models of execution closer to real assembly code.

References

[1] Ahmed, A., *Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types*, in: *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP’06 (2006), pp. 69–83.

[2] Appel, A. W. and S. Blazy, *Separation Logic for Small-step Cminor*, CoRR **abs/0707.4389** (2007).

[3] Appel, A. W. and D. McAllester, *An Indexed Model of Recursive Types for Foundational Proof-carrying Code*, ACM Transactions on Programing Languages and Systems. **23** (2001), pp. 657–683.

[4] Benton, N. and C.-K. Hur, *Biorthogonality, Step-indexing and Compiler Correctness*, SIGPLAN Not. **44** (2009), pp. 97–108.

[5] Benton, N. and C.-K. Hur, *Realizability and Compositional Compiler Correctness for a Polymorphic Language*, Technical report, MSR-TR 2010-62, Microsoft Research (2010).

[6] Benton, N., A. Kennedy and C. Varming, *Formalizing Domains, Ultrametric Spaces and Semantics of Programming Languages* (2010), unpublished.

[7] Birkhoff, G., “Lattice Theory,” American Mathematical Society Colloquium Publications, vol. 25, 1940.

[8] Chlipala, A., *A Certified Type-preserving Compiler from Lambda Calculus to Assembly Language*, SIGPLAN Not. **42** (2007), pp. 54–65.

[9] Chlipala, A., *A Verified Compiler for an Impure Functional Language*, in: *POPL*, 2010, pp. 93–106.

[10] Diehl, S. and P. Sestoft, *Abstract Machines for Programming Language Implementation*, Future Generation Computer Systems. **16** (2000), pp. 739–751.

[11] Dreyer, D., G. Neis and L. Birkedal, *The Impact of Higher-order State and Control Effects on Local Relational Reasoning*, SIGPLAN Not. **45** (2010), pp. 143–156.

[12] Jaber, G. and N. Tabareau, *The Journey of Biorthogonal Logical Relations to the Realm of Assembly Code*, in: *Workshop LOLA 2011, Syntax and Semantics of Low Level Languages*, Toronto, Canada, 2011, pp. 1–15.

[13] Jones, P. and S. L, *Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine*, Journal of Functional Programming **2** (1992), p. 127 202.

[14] Krivine, J.-L., *Classical Logic, Storage Operators and Second-order Lambda-calculus*, Annals of Pure and Applied Logic **68** (1994), pp. 53 – 78.

[15] Krivine, J.-L., *A Call-by-name Lambda-calculus Machine*, Higher Order Symbolic Computation. **20** (2007), pp. 199–207.

[16] Landin, P. J., *The Mechanical Evaluation of Expressions*, The Computer Journal **6** (1964), pp. 308–320.

[17] Leroy, X., *Formal Verification of a Realistic Compiler*, Communications of the ACM **52** (2009), pp. 107–115.

[18] Leroy, X., *Mechanized Semantics - with Applications to Program Proof and Compiler Verification*, in: *Logics and Languages for Reliability and Security*, IOS Press, 2010 pp. 195–224.

[19] Leroy, X. and H. Grall, *Coinductive Big-step Operational Semantics*, Information and Computation **207** (2009), pp. 284–304.

[20] McCarthy, J. and J. Painter, *Correctness of a Compiler for Arithmetic Expressions*, in: *Mathematical Aspects of Computer Science 1*, 1 **19** (1967), pp. 33–41.

[21] Morris, F. L., *Advice on Structuring Compilers and Proving them Correct*, in: *POPL*, 1973, pp. 144–152.

[22] Ore, O., *Galois Connexions*, Transactions of the American Mathematical Society **55** (1944), pp. 493–513.

[23] Pitts, A. M. and I. D. B. Stark, *Operational Reasoning for Functions with Local State*, in: A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Cambridge University Press, New York, NY, USA, 1998 pp. 227–274.

[24] Reynolds, J. C., *The Coherence of Languages with Intersection Types*, in: *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, TACS '91 (1991), pp. 675–700.

[25] Rodríguez, L., D. Fridlender and M. Pagano, `http://cs.famaf.unc.edu.ar/~leorodriguez/compilercorrectness/`, Coq formalization accompanying this paper.

[26] Rodríguez, L., D. Fridlender and M. Pagano, *A Certified Extension of the Krivine Machine for a Call-by-Name Higher-Order Imperative Language*, in: R. Matthes and A. Schubert, editors, *19th International Conference on Types for Proofs and Programs*, Leibniz International Proceedings in Informatics (LIPIcs) **26** (2014), pp. 230–250.

[27] Selinger, P., *From Continuation Passing Style to Krivine's Abstract Machine*, Manuscript (2003), available in Peter Selinger's web site.

[28] Sestoft, P., *Deriving a Lazy Abstract Machine*, Journal of Functional Programing. **7** (1997), pp. 231–264.