Krivine realizability for compiler correctness

Guilhem Jaber, Nicolas Tabareau

Krivine realizability for compiler correctness

Guilhem Jaber Nicolas Tabareau

Abstract

We propose a semantic type soundness result, formalized in the Coq proof assistant, for a compiler from a simple functional language to SECD machine code. Our result is quite independent from the source language as it uses Krivine's realizability to give a denotational semantics to SECD machine code using only the type system of the source language. We use realizability to prove the correctness of both a call-by-name (CBN) and a call-by-value (CBV) compiler with the same notion of orthogonality. We abstract over the notion of observation (e.g. divergence or termination) and derive an operational correctness result that relates the reduction of a term with the execution of its compiled SECD machine code.

1 Introduction

What kinds of correctness properties do we wish to establish of our compilers? The most obvious answer is that when a high-level source program is compiled to produce a low-level target, the behaviour of the target always agrees with a high-level semantics of the source. But we usually also want to be sure that the target code satisfies certain safety or liveness properties, ensuring that 'bad things' don't happen, or that 'good' ones do. Such properties include memory safety, adherence to information flow policies, termination, and various forms of resource boundedness. For applications in language-based security, the good news is that (fancy) type-like properties of this kind, which at least *seem* less complex to state and check than full functional correctness, are the important ones. On the other hand, for such applications one would really like to certify the code that actually runs, which involves formalizing and verifying type-like properties of machine code. How to do that is the problem we address in this paper.

The approach taken here is essentially a mix between that of earlier works on specifying and verifying memory managers [6] and type preservation for a simple imperative language [9], and of a more recent work on the correctness of a compiler for a functional language to SECD using realizability techniques [8]. In this last paper, Benton and Hur use a realizability relation between SECD machines (code, stack, environment, dump) and (domain theoretic) denotation of the source language to give a denotational semantics to SECD machines. This approach depends heavily on the existence of a proper denotational semantics for the source language. In this paper, we propose to give a denotational semantics to SECD machines more directly using Krivine realizability constructed over types. In this setting, a SECD machine code c will be said to realize a type A

$$c \Vdash A$$

when it passes all the tests π defined in $\|A\|$

$$\forall \pi \in \|A\| \,.\, c \star \pi \in \perp\!\!\!\perp.$$

The $\star$ above defines the interaction of a SECD machine code c with a test π and $\perp\!\!\!\perp$ represents the set of all "good" observations. Now, given a compiler from a typed language to SECD machines code, the idea is to show an adequacy result of the form

"For any term t of type A, the SECD machine code **compile** t realizes the type A."

We want our proof to be sufficiently generic so that we can deal both with CBN and CBV compiler. And we also want to be able to derive an operational correctness result from the adequacy result.

To reason by induction on typing derivation, we have to work with open terms and so realizers must be pieces of SECD machine code together with an environment (c, e). A test is then a dump d and its interaction with a realizer candidate will be defined as the SECD machine

$$(c, e, [], d)$$

that evaluates c in environment e, with dump d and the empty stack.

As we say, we want to derive from adequacy result, a relation between the reduction of a source term t and the execution of the compiled SECD machine code $\mathbf{compile}\, t$, what we call operational correctness. But as types are our unique knowledge on terms, we need a precise type system. Indeed, when a term t reduces to an integer n, we would like to be able type t not only with the type Nat but also with the more precise type

$$\vdash t : \mathrm{Nat}\, P_n$$

saying that t is an integer that reduces to n. So we define a enriched type system where natural numbers can be said to satisfy some property P and with a notion of quantification over integers. With our system, we can type the factorial function `Fac` as

$$\vdash \texttt{Fac} : \forall n.\mathrm{Nat}\, \mathrm{P}_n \to \mathrm{Nat}\, \mathrm{P}_{n!}$$

and type the term `Fac` 5 as

$$\vdash \texttt{Fac}\ 5 : \mathrm{Nat}\, P_{5!}$$

In this way, the adequacy result, that guarantees that the compiled version of factorial $\mathbf{compile}\,\texttt{Fac}$ realizes $\forall n.\mathrm{Nat}\, \mathrm{P}_n \to \mathrm{Nat}\, \mathrm{P}_{n!}$ says a bit more: it says that $\mathbf{compile}\,\texttt{Fac}$ is actually a SECD machine code that transforms an integer n on the stack to the integer $n!$.

Remark that for the reasoning above to be true, we have implicitly worked with termination as notion of observation even in presence of fixpoints! This means that we have to treat fixpoints of our source language with an extreme care. To distinguish between general and well-founded fixpoints, we introduce two separate typing rules: (1) the general fixpoint rule allows divergence and requires to work with step-indexing reasoning [4] and with divergence as notion of observation, (2) the well-founded fixpoint rule guarantees termination and thus allows reasoning with either divergence or termination as notion of observation. By abstracting over the notion of observation, we can prove the adequacy result for both divergence and termination in absence of the general fixpoint rule. Then we can deduce an operational correctness result for terms that can be typed without the general fixpoint rule.

Plan of the paper. We first present the basic ideas of Krivine realizability extended to open terms. After that, we introduce SECD machines and the source language. Then, we describe how to define Krivine realizability for SECD machines and present the adequacy and operational correctness results for a CBN and a CBV compiler, formalized in Coq. The Coq development is available on the web page of the second author.

2 Krivine Realizability

We present in this section the basic ideas of Krivine realizability [11], which has been developed to extend the Curry-Howard isomorphism to classical logics, but also to (some restricted versions of) the axiom of choice and the continuum hypothesis. It is based on a duality between realizers and tests, that can be explained simply:

A term t realizes a type A if it satisfies all the tests of type A.

Here, we present a modified version of Krivine realizability, where we deal with open terms, so realizers are not simply terms but closures, i.e. terms with their environment.

2.1 Krivine Abstract Machine

We use the KAM with environment which evaluate triples formed by a term with its environment and a stack, which are defined by:

- $\texttt{Term} : t, u := x \mid \lambda x.t \mid t\, u$
- $\texttt{Closure} := \texttt{Term} \times \texttt{Env}$
- $\texttt{Env} ::= \text{List } \texttt{Closure}$
- $\texttt{Stack} ::= \text{List } \texttt{Closure}$

A triple (t, e, π), where t is a term, e an environment and π a stack, is called a process. Here are the evaluation rules of processes:

- $(x, e, \pi) \succ (t, e', \pi)$ where $e(x) = (t, e')$.
- $(\lambda x.t, e, c \cdot \pi) \succ (t, e \cdot \{x \mapsto c\}, \pi)$.
- $(t\, u, e, \pi) \succ (t, e, (u, e) \cdot \pi)$.

A real implementation should deal with variable capture. In our Coq formalization we have chosen to works with De Bruijn indexes, so environment are then simply stacks of closures.

2.2 Tests as stacks

Tests have to be thought of as contexts with an hole, that is a place where we can put the term we want to test and evaluate the remaining combination of test and term. But it is not any kind of context: we want the hole to be at the top of the test, since the realizer has to be evaluated before the test. In this case, tests are simply stacks. We will see in Section 5, that, for SECD machine, tests will be dumps, and in more low-level languages like assembly code, it would be program fragments.

As we said, realizers are closures and terms are stacks, but we still need to define what it means for a realizer (t, e) to satisfy a test π, which we note

$$(t, e) \perp \pi.$$

It simply means that their interaction $(t, e) \star \pi = (t, e, \pi)$ is a "good" process, i.e. it belongs to a fixed set $\perp\!\!\!\perp$. Depending on what we want to prove about terms, we can choose different $\perp\!\!\!\perp$, like the set of divergent processes, or the set of terminating processes. In fact, $\perp\!\!\!\perp$ must simply be seen as any choice of observation on processes. One only requires that it satisfies an axiom of stability by anti-reduction:

$$\text{if } p \in \perp\!\!\!\perp \text{ and } p' \succ p \text{ then } p' \in \perp\!\!\!\perp.$$

In the sequel, sets of realizers and tests of A will respectively be noted $|A|$ and $\|A\|$, and realizers will be defined from tests by

$$|A| = \{u \in \texttt{Closure} \mid \forall \pi \in \|A\|\, .u \star \pi \in \perp\!\!\!\perp\}.$$

When $u \in |A|$ we write

$$u \Vdash A.$$

Let us now just recall one of the most famous definition in Krivine realizability, the set of tests of a function

$$\|A \to B\| = \{u \cdot \pi \in \texttt{Stack} \mid u \in |A| \text{ and } \pi \in \|B\|\}.$$

Note, the fact that a test of $A \to B$ can be built thanks to a realizer of A and a test of B is crucial.

2.3 Realizability for open terms

We want to link realizability and typing, showing that if a term is of type A, then it is a realizer of type A. Traditionally in Krivine's work, this link is made for only closed term, assuming that every variable has always been substituted. But since we want to deal with open terms, we need typing contexts: $\Gamma \vdash t : A$. So we define a compatibility relation between an environment e and a typing context Γ, noted $e \triangleright \Gamma$, when each closure in e realizes its corresponding type in Γ:

$$e \triangleright \Gamma \text{ iff for all } c_i \in e, c_i \Vdash T_i.$$

Then, an adequacy lemma can be proved:

$$\text{if } \Gamma \vdash t : A \text{ then for all environments } e \text{ such that } e \triangleright \Gamma, (t, e) \Vdash A.$$

2.4 Compiler correctness

From the fact that $t \Vdash A$, we would like to derive that

$$(t, e, []) \succ^* (v, e', [])$$

where v is a value of type A. We can take $\perp\!\!\!\perp$ to be the set of processes whose evaluation terminates with an empty stack. Then, when we make a closure (t, e) interact with a test π, we want to be sure that the test is evaluated. It means that if $(t, e) \star \pi \in \perp\!\!\!\perp$ then $(t, e, \pi) \succ^* (\lambda x.u, e', \pi)$. With our choice of $\perp\!\!\!\perp$, this is indeed the case, because $(t, e, \pi) \in \perp\!\!\!\perp$ iff $(t, e, \pi) \succ^* (t', e', [])$ and $(t', e', []) \not\succ$, so we can prove that the evaluation goes through $(\lambda x.u, e', \pi)$ if π is not empty.

Notice that if we add to the KAM the usual `callcc` instruction and the stack constants k_π, then this theorem is not true anymore. Indeed $(k_\pi, e', (t, e) \cdot \pi') \succ (t, e, \pi)$ so we can avoid to execute tests thanks to this constant [1].

Now that we are sure that tests are evaluated, we just have to be able to pick up one test that discriminates a term that is a value of type A with a term that is not. This supposes that our source language is expressive. We will see later how this kind of tests can be built for the case of SECD machine, defining discriminating tests for base types and generating tests for functions inductively. .

When we take $\perp\!\!\!\perp$ to be the set of non-terminating terms, which can be useful for example when the strong normalization cannot be proved from our typing rules [2], we can do a similar reasoning on t if we know (by other means) that $(t, e, [])$ evaluates to $(v, e', [])$. Then if $(t, e) \Vdash A$ it can be proved that v is a value of type A.

In what follows, we will adapt this method to the correctness of compilers from PCF to SECD machines.

3 Our target language

We use an expanded version of SECD machines, adding a new value which represent a thunk – a frozen computation – so that a call-by-name reduction strategy can be defined. Here are the basic definitions:

- `Inst` := `App` | `Swap` | `SubstV` | `PushN` | `PushV` | `PushC`r | `PushFC` | `PushRC` | `Sel` | `Op` | `EqTest`.
- `Code` := List `Inst`.
- `Env` := List `Val`.
- `Stack` := List `Val`.
- `Val` := `V`(n) | `FC`(c, e) | `RC`(c, e) | `C`(c, e) where $n \in \mathbb{N}, c \in$ `Code` and $e \in$ `Env`.

[1] This is why in Krivine realizability we focus on *proof-like terms*, i.e. terms which do not contain any stack constants

[2] It is for example the case when we add a typing rule for general recursion. In this case, the usual proof of the adequacy lemma does not work anymore and we need new tools to make it, which are presented in the next part.

$([], e, v :: s, (e', s', c') :: d)$	$\succ$	$(c', e', v :: s', d)$
$(\texttt{Swap} :: c, e, v_1 :: v_2 :: s, d)$	$\succ$	$(c, e, v_2 :: v_1 :: s, d)$
$(\texttt{PushN}\, n :: c, e, s, d)$	$\succ$	$(c, e, \texttt{V}(n) :: s, d)$
$(\texttt{SubstV}\, i :: c, [v_1, \ldots, v_i], s, d)$	$\succ$	$\begin{cases} (c_1, e_1, [], (c, [v_1, \ldots, v_n], s) :: d) & \text{if } v_i = \texttt{C}(c_1, e_1). \\ (c, [v_1, \ldots, v_n], v_i :: s, d) & \text{otherwise.} \end{cases}$
$(\texttt{PushC}r\, c_1 :: c, e, s, d)$	$\succ$	$(c, e, \texttt{C}(c_1, e) :: s, d)$
$(\texttt{PushFC}\, c_1 :: c, e, s, d)$	$\succ$	$(c, e, \texttt{FC}(c_1, e) :: s, d)$
$(\texttt{PushRC}\, c_1 :: c, e, s, d)$	$\succ$	$(c, Ee, \texttt{RC}(c_1, e) :: s, d)$
$(\texttt{App} :: c, e, \texttt{FC}(c_1, e_1) :: t :: s, d)$	$\succ$	$(c_1, t :: e_1, [], (e, s, c) :: d)$
$(\texttt{App} :: c, e, \texttt{RC}(c_1, e_1) :: t :: s, d)$	$\succ$	$(c_1, t :: \texttt{RC}(c_1, e_1) :: e_1, [], (e, s, c) :: d)$
$(\texttt{Op}\odot :: c, e, \texttt{V}(n_1) :: \texttt{V}(n_2) :: s, d)$	$\succ$	$(c, e, \texttt{V}(n_2 \odot n_1) :: s, d)$
$(\texttt{Sel}\, c_1 c_2 :: c, e, \texttt{V}(0) :: s, d)$	$\succ$	$(c_2 \texttt{++} c, e, s, d)$
$(\texttt{Sel}\, c_1 c_2 :: c, e, v :: s, d)$	$\succ$	$(c_1 \texttt{++} c, e, s, d)$
$(\texttt{EqTest} :: c, e, \texttt{V}(n_1) :: \texttt{V}(n_2) :: s, d)$	$\succ$	$(c, e, V(n_1 = n_2) :: s, d)$

Table 1: The SECD machine

- $\texttt{Dump} := \text{List}\,(\texttt{Code} \times \texttt{Env} \times \texttt{Stack})$.

The value $\texttt{C}(c, e)$ represents the thunk of the code c with environment e, while $\texttt{FC}(c, e)$ and $\texttt{RC}(c, e)$ are closures respectively for normal and recursive functions.

Table 1 presents the reduction steps of an SECD machine. To deal with thunks, we need to change the usual `PushV` instruction, which takes the value of a variable in the environment and put it in the stack. Indeed, when this value is a thunk, it shall not be put at the top of the stack, but be evaluated first. That is what the new instruction `SubstV` does. Moreover, we do not keep a `Return` instruction, but we rather assume that the dump is automatically restored when we have no more code to execute. Indeed, to be sure that the interaction of a realizer with a test will evaluate the test, we want to forbid a piece of code to restore the dump at any time, since this is where tests are stored. However, note it is still possible to add some control-flow instruction, as soon as we are able to discriminate between tests and dumps created by those instructions.

4 Our source language

Our source language is PCF with a richer type systems. Terms are defined as

$$t, u := \text{true} \mid \text{false} \mid n \mid x \mid \lambda x.t \mid t\, u \mid \text{Fix}\, t \mid t_1 \odot t_2 \mid t_1 < t2 \mid \text{If}\, t_c\, t_1\, t_2.$$

Our type system, presented in Table 2, is defined in two layers:

- $A, B := \text{Top} \mid \text{Nat}\, \text{P}_{\mathbb{N}} \mid \text{Bool}\, \text{P}_{\mathbb{B}} \mid T \to B$ where $\text{P}_{\mathbb{N}} : \mathbb{N} \to \text{Bool}$ and $P_{\mathbb{B}} : \text{Bool} \to \text{Bool}$ are predicates.
- $T := A \mid \forall n.T(n)$.

It has four main particularities:

- Top is the type of all terms. It is used for the typing rule Rec-Nat.
- Types Nat and Bool come with predicates, to specify more precisely inhabitants of these types. For example, we can use the predicate $P_{even} := n \mapsto \begin{cases} \text{true} & \text{if } n \text{ is even.} \\ \text{false} & \text{otherwise.} \end{cases}$ to define the type $\text{Nat}\, \text{P}_{even}$ of

$$\frac{}{\Gamma \vdash t : \mathrm{Top}} \qquad \frac{\mathrm{P}\, n}{\Gamma \vdash n : \mathrm{Nat}\,\mathrm{P}} \qquad \frac{\mathrm{P}\, b}{\Gamma \vdash b : \mathrm{Bool}\,\mathrm{P}}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \textsc{Rec-Gen}\ \frac{x : A, f : A \to B, \Gamma \vdash t : B}{\Gamma \vdash \mathrm{Fix}\,\lambda f.\lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}$$

$$\frac{\Gamma \vdash t_1 : \mathrm{Nat}\,\mathrm{P}_1 \qquad \Gamma \vdash t_2 : \mathrm{Nat}\,\mathrm{P}_2}{\Gamma \vdash t_1 \odot t_2 : \mathrm{Nat}(\{\odot\}\mathrm{P}_1\mathrm{P}_2)} \qquad \frac{\Gamma \vdash t_1 : \mathrm{Nat}\,\mathrm{P}_1 \qquad \Gamma \vdash t_2 : \mathrm{Nat}\,\mathrm{P}_2}{\Gamma \vdash t_1 < t_2 : \mathrm{Bool}(\{<\}\mathrm{P}_1\mathrm{P}_2)}$$

$$\frac{\Gamma \vdash t_c : \mathrm{Bool}\,\mathrm{P} \qquad \mathrm{P}\,\mathrm{true} \Rightarrow \Gamma \vdash t_1 : A \qquad \mathrm{P}\,\mathrm{false} \Rightarrow \Gamma \vdash t_2 : A}{\Gamma \vdash \mathrm{If}\, t_c\, t_1\, t_2 : A}$$

$$\textsc{Forall}\ \frac{\Gamma \vdash t : T(n)}{\Gamma \vdash t : \forall n.T(n)}\ n\#\Gamma \qquad \frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}$$

$$\textsc{Rec-Nat}\ \frac{x : A(0), f : \mathrm{Top}, \Gamma \vdash t : B(0) \qquad x : A(n+1), f : A(n) \to B(n), \Gamma \vdash t : B(n+1)}{\Gamma \vdash \mathrm{Fix}\,\lambda f.\lambda x.t : \forall n.A(n) \to B(n)}\ n\#\Gamma$$

Table 2: Type system

even numbers. We need such an expressive type system to get an interesting theorem of operational correctness for our compiler.

- The quantification over naturals, which has to be seen as a restricted dependent type. Indeed, considering $\forall n.T(n)$, the bounded variable n can only appear in the predicate of a type Nat or Bool. For example the type of the factorial function can be defined as : $\forall n.\mathrm{Nat}\,\mathrm{P}_n \to \mathrm{Nat}\,\mathrm{P}_{n!}$ where P_k is the predicate $x \mapsto (x = k)$. This enables us to state uniformity results for functions.
- For some technical reasons, we use a stratification on types, inspired by Hindley-Milner system, such that if we consider $T \to B$, we have the guarantee that B does not contain quantified types. Indeed, as we will see later, $\|A\|$ is closed by biorthogonal while it is not the case of $\|\forall n.(T)\|$.

The typing rules presented in the Table 2 are standard. There is one particularity, the presence of two rules for the fixpoint operator, one general, Rec-Gen, and one restricted to the type Nat, Rec-Nat, which gives a quantified term and guarantees that the fixpoint is well-founded.

This means that when the typing proof of a term t does not use Rec-Gen, t normalizes. So we would like to get this crucial information down to **compile** t. To do so, we will see that $\perp\!\!\!\perp$ can be defined as the set of processes which terminates. However, with this definition, it is not possible to deduce directly, from that fact that t is of type A, that **compile** t realizes A, as soon the typing of A use Rec-Gen. We will see later that step-indexing and divergence have to be used to solve this problem.

Finally, we use an usual coercion of types,

$$A <: B,$$

which allows to instantiate quantifiers thanks to the following rule:

$$A(k) <: \forall n.A(n) \text{ for all } k \in \mathrm{Nat}.$$

Notice that polymorphic types can be handle in the same way than quantification on naturals. The compiler, presented in Table 3, is based on a modified version of this source language where variables have been replaced by De Bruijn indexes. Note that there are two definitions for the compilation of an application, depending on if we are in CBN or CBV.

$$
\begin{array}{lcl}
\textbf{compile}\, n & = & \texttt{PushN}\, n \\
\textbf{compile}\, \text{false} & = & \texttt{PushN}\, 0 \\
\textbf{compile}\, \text{true} & = & \texttt{PushN}\, 1 \\
\textbf{compile}\, \text{Var}_i & = & \texttt{SubstV}\, i \\
\textbf{compile}\, \lambda.t & = & \texttt{PushFC}\, (\textbf{compile}\, t) \\
\textbf{compile}\, (t\, u) & = & \begin{cases} \texttt{PushC}r\, (\textbf{compile}\, u) :: (\textbf{compile}\, t) :: [\texttt{App}] & \text{in CBN} \\ (\textbf{compile}\, t)\texttt{++}(\textbf{compile}\, u)\texttt{++}[\texttt{Swap} :: \texttt{App}] & \text{in CBV} \end{cases} \\
\textbf{compile}\, (e_1 \odot e_2) & = & (\textbf{compile}\, e_1)\texttt{++}(\textbf{compile}\, e_2)\texttt{++}[\texttt{Op}\, \odot] \\
\textbf{compile}\, (e_1 > e_2) & = & (\textbf{compile}\, e_1)\texttt{++}(\textbf{compile}\, e_2)\texttt{++}[\texttt{Swap} :: (\texttt{Op}\ <)] \\
\textbf{compile}\, (\text{If}\, t_c\, t_1\, t_2) & = & (\textbf{compile}\, t_c)\texttt{++}[\texttt{Sel}\, (\textbf{compile}\, t_1)(\textbf{compile}\, t_2)] \\
\textbf{compile}\, (\text{Fix}\, \lambda t) & = & \texttt{PushRC}\, (\textbf{compile}\, t)
\end{array}
$$

Table 3: The compiler

5 Krivine realizability for SECD

The interaction $\star$ between a value v and a dump d is defined in the following way: If $v = \texttt{C}(c, e)$ (i.e. it is a thunk), then $v \star d = (c, e, [], d)$, otherwise $v \star d = ([], [], [v], d)$.

5.1 Step indexing

To deal with general recursion, we use the common technique of step indexing [4].ù Indeed, when we want to prove the adequacy of the compiler by induction on the proof of $\Gamma \vdash t : A$, the case of Rec-Gen is an issue, because evaluating **compile** $\lambda.t$ interacting with a test of $A \to B$, we get again **compile** $\lambda.t$ interacting with a new test of $A \to B$, so there seems to be no way to conclude. In the case of the well-founded rule Rec-Nat, we can make an induction on n, but for Rec-Gen there is nothing decreasing during the evaluation.

That is why realizers, tests and processes have to come with an index. But it is also the case of $\perp\!\!\!\perp$: we consider a family of sets $(\perp\!\!\!\perp_i)_{i\in\mathbb{N}}$ which have to be considered as approximations of $\perp\!\!\!\perp$.

And to deal with the basic case of the induction for Rec-Gen, we need to add a new axiom about $\perp\!\!\!\perp$:

$$\perp\!\!\!\perp_0 \text{ contains all the processes.}$$

For example, if we want to take for $\perp\!\!\!\perp$ the set of divergence terms, $\perp\!\!\!\perp_i$ is simply the processes which can reduce at least i times, and we see that $\perp\!\!\!\perp_0$ contains trivially all the processes.

When we forget about the Rec-Gen typing rule, we can simply take $\perp\!\!\!\perp_i = \perp\!\!\!\perp$ for all i and forget the axiom about $\perp\!\!\!\perp_0$.

5.2 Definition of tests

Realizers are couples formed by a value and an index, and the tests are couples formed by a dump and an index. Orthogonality relations between realizers and tests are defined as:

- $(v, k) \perp (d, j)$ iff $j \leq k \to v \star d \in \perp\!\!\!\perp_j$.
- $(d, k) \perp (v, j)$ iff $j \leq k \to v \star d \in \perp\!\!\!\perp_j$.

We will define the set of realizers of type A from the test of type A, by orthogonality:

$$|A| = \|A\|^{\perp} = \{(v, k) \in \texttt{Val} \times \mathbb{N} \mid \forall (d, j) \in \|A\|\,, (v, k) \perp (d, j)\}.$$

We write $v \Vdash A$ if for all $k \in \mathbb{N}, (v, k) \Vdash A$, and we will often abbreviate $\texttt{C}(c, e) \Vdash A$ to $(c, e) \Vdash A$.

To define $\|A\|$, the set of tests of type A, we first need to define $[\![A]\!]$, the set of *primitive* tests of type A :

- $[\![\mathrm{Top}]\!] = \{(d,k) \in \mathtt{Dump} \times \mathbb{N} \,|\, \forall v \in \mathtt{Val}, v \star d \in \perp\!\!\!\perp_k\}$.
- $[\![\mathrm{Nat\,P}]\!] = \{(d,k) \in \mathtt{Dump} \times \mathbb{N} \,|\, \forall n \in \mathbb{N} \text{ s.t. } \mathrm{P}\, n, (\mathtt{V}\, n) \star d \in \perp\!\!\!\perp_k\}$.
- $[\![\mathrm{Bool\,P}]\!] = \{(d,k) \in \mathtt{Dump} \times \mathbb{N} \,|\, \forall b \in \mathbb{B} \text{ s.t. } \mathrm{P}\, b, (\mathtt{V}\, \hat{b}) \star d \in \perp\!\!\!\perp_k\}$.
- $[\![A \to B]\!] = \{((\mathtt{App} :: c, u :: s, e) :: d, k) \in \mathtt{Dump} \times \mathbb{N} \,|\, (u,k) \in |A| \text{ and } \forall j < k, (d,j) \in \|B\|\}$. The quantification $\forall j < k$ is a reminiscence of the Löb rule and the later modality, just like in [5]

But the definition of these primitive tests is too rigid, we would like to be able to define tests which "behave" extensionally like them. More precisely, if d is a test of T and for all value u, $d \star u \in \perp\!\!\!\perp$ implies $d' \star u \in \perp\!\!\!\perp$, we would like d' be also a test of T. That is why one has to close these sets by biorthogonality, to get more tests:

$$\|A\| = [\![A]\!]^{\perp\perp}.$$

Notice that we close the sets only for non quantified types. Indeed, we define tests of $\forall n.T(n)$ in the following way:

$$\|\forall n.T(n)\| = \bigcup_{n \in \mathbb{N}} \|T(n)\|.$$

This is where are stratified definition of types comes into the picture. We know that every type defined in the first layer is closed by biorthogonality.

Unwinding the biorthogonality (and forgetting for one moment step-indexing), we get that d is a test of type A if it goes well with all the values v which go well with the primitives tests of type A.

However, since we have expended SECD machine with thunks, we must keep in mind that we also expended tests, which could use its. This could be a problem when we want to prove the correction of the CBV compiler, so in this case we need to restrict the definition of biorthogonality to values which are not thunks: d is a test of type A if it goes well with all the values v which are not thunks and go well with the primitives tests of type A. This definition is also fine to prove the correction of the CBN compiler. Notice that we do not change the definition of orthogonality used between realizers and tests. Indeed, to prove the adequacy lemma, we use the fact that if t is of type A then $\mathtt{C}(\mathbf{compile}\, t, []) \Vdash A$.

6 Results of correctness

Finally, we can prove the adequacy lemma between type system and realizability:

Theorem 1 (Adequacy)**.** *if $\Gamma \vdash t : T$ then for all environments e such that $e \triangleright \Gamma$, $(\mathbf{compile}\, t, e) \Vdash T$.*

Once we have proved that $(c,e) \Vdash T$, properties on c can be deduced, in the same way we have done with the KAM. We first restrict to the case where $\Gamma \vdash t : T$ is proved without Rec-Gen. Then we are sure that t normalizes, and we can work with $\perp\!\!\!\perp$ as the set of the processes reducing to $([], e, [v], [])$ for some $v \in \mathtt{Val}$. In this case, we can forget everything about step-indexing.

In what follows, we will suppose that predicates P which come from types Nat and Bool are expressible with SECD machines. This is of course the case of P_n. Then we can prove the following correctness theorem:

Theorem 2 (Operational correctness)**.** *If $\Gamma \vdash t : T$ and $e \triangleright \Gamma$ then $(\mathbf{compile}\ t, e, [], []) \succ^* ([], e', [v], [])$ and v is a value of type T :*

- *If T is $\mathrm{Nat\,P}$ then $v = \mathtt{V}(n)$ and $\mathrm{P}(n)$.*
- *If T is $\mathrm{Bool\,P}$ then $v = \mathtt{V}(b)$ with $b = 0$ or $b = 1$ and $\mathrm{P}(b)$*
- *If T is $S \to B$ then $v = \mathtt{FC}(c,e)$ or $v = \mathtt{RC}(c,e)$ and for all values $u \vdash S$, $(\mathtt{App}, [], v :: [u], []) \succ^* ([], e', [w], [])$ with w of type B.*
- *If T is $\forall n.S(n)$ then for all $n \in \mathbb{N}$, v is a value of type $S(n)$.*

This theorem gives us operational results for types $\mathrm{Nat}\,\mathrm{P}_n$ or $\mathrm{Bool}\,\mathrm{P}_b$. Indeed, in these cases, $t \to^* n$ so the reduction of t corresponds to the evaluation of **compile** t.

We will prove a more general result about closure $(c, e) \Vdash T$, then operational correctness follows from adequacy lemma. First, as what we did for the KAM, considering the interaction $u \star d$ (in particular for $u = \mathtt{C}(c, e)$), we prove that the test d is evaluated:

$$\text{if } u \star d \succ^n ([], e, [v], []) \text{ then there exists } k \leq n \text{ s.t. } u \star d \succ^k ([], e', [v'], d).$$

Then, we make an induction on T:

- if $T = \mathrm{Nat}\,\mathrm{P}$, one has to prove that $(c, e, [], []) \succ^* ([], e', [\mathtt{V}(m)], [])$ with m satisfying P. Taking $d \in \|\mathrm{Nat}\,\mathrm{P}\|$, we already know that:
  - $(c, e, [], d) \succ^* ([], e, [v'], d)$.
  - $([], [], [\mathtt{V}(n)], d) \succ^* ([], e', [w], [])$ for all n satisfying P.

  So we need to be able to pick up a test d which could discriminate between $\mathtt{V}(n)$ and other values. To do so, the predicate P needs to be expressible in our language. For example, taking P as P_n, d can be $(\mathtt{PushV}\, n :: \mathtt{EqTest} :: [\mathtt{Sel}\, [\mathtt{PushV}\, 1]\, \Omega], [], [])$ where Ω is a diverging term. Then

  $$\begin{array}{lll} ([], e, [v'], d) & \succ^2 & (\mathtt{EqTest} :: [\mathtt{Sel}\, [\mathtt{PushV}\, 1]\, \Omega], [], \mathtt{V}(n) :: [v'], []) \\ & \succ^* & \begin{cases} ([], [], [\mathtt{V}(1)], []) \in \perp\!\!\!\perp & \text{if } v' = \mathtt{V}(n). \\ (\Omega, [], s, []) \notin \perp\!\!\!\perp & \text{otherwise.} \end{cases} \end{array}$$

  Using the fact that $([], e, [v'], d) \in \perp\!\!\!\perp$,it cannot reduce to a process not in $\perp\!\!\!\perp$, so $v' = \mathtt{V}(n)$. The proof for $\mathrm{Bool}\,\mathrm{P}$ is made in the same way.

- If $(c, e) \Vdash S \to B$ and $u \Vdash S$, we first prove that $(c, e, [], [] \succ^* ([], e, [v], [])$ using the fact that the empty dump is a valid test. Then for all $u \Vdash S$ one has to prove that $([\mathtt{App}], e, v :: [u], []) \succ^* ([], e', [w], [])$ and $w \Vdash B$.

  To do so, one takes a test d in $\|B\|$, then $([\mathtt{App}], e, [u]) :: d$ is in $\|A \to B\|$ so

  $$([], [], [v], ([\mathtt{App}], e, [u]) :: d) \succ ([\mathtt{App}], e, v :: [u], d) \succ^* ([], e', [w], d)$$

  So v has to be $\mathtt{FC}(c, env)$ or $\mathtt{RC}(c, env)$, and $w \perp d$ for all $d \in \|B\|$, i.e. $w \Vdash B$.

- Finally, if $(c, e) \Vdash \forall n.S(n)$, it satisfies for all $n \in \mathbb{N}$ the tests in $\|S(n)\|$, i.e. $(c, e) \Vdash S(n)$.

  Note when $S(n)$ is $U(n) \to B(n)$, we get an uniform result of correctness : for all $n \in \mathbb{N}$ and $u \Vdash A(n)$, $(c\mathtt{++}[\mathtt{App}], e, [u], []) \succ^* ([], e', [v], [])$ with $v \Vdash B$.

Notice that the informations we get on **compile** t are strictly those we get from the typing of A. So if we cannot deduce that t normalizes from its type, for example if the proof of its typing use Rec-Gen, then their is no way we can get with this method that the evaluation of **compile** t terminates. But in this case, we can still get some information on the result of the evaluation of **compile** t, if we suppose it terminates. Indeed, taking $\perp\!\!\!\perp$ as the diverging processes and using step-indexing just as explain above, tests interacting with **compile** t must be evaluated since **compile** t terminates. And it is still possible to build tests which discriminate between a natural and an other value, but here the case where it must diverge is swapped.

6.1 The Factorial example

Consider the term $\mathtt{Fac} = \mathrm{Fix}\,(\lambda f.\lambda n.(\mathrm{If}\ (n > 0)\ (n * f(n-1))\ 1))$ which computes the factorial of a natural number. Then, as shown in the table 4, it can be typed using Rec-Nat. The type we get is

$$\forall n.\mathrm{Nat}\,\mathrm{P}_n \to \mathrm{Nat}\,\mathrm{P}_{n!}$$

$$\dfrac{\dfrac{}{\Gamma_0 \vdash (x>0) : \mathrm{Bool}\,\mathrm{P}_{(x>0)}} \qquad \dfrac{}{(x>0) \Rightarrow \Gamma_0 \vdash (x * f(x-1)) : \mathrm{Nat}\,\mathrm{P}_1} \qquad \dfrac{}{(x=0) \Rightarrow \Gamma_0 \vdash 1 : \mathrm{Nat}\,\mathrm{P}_1}}{n : \mathrm{Nat}\,\mathrm{P}_0, f : \mathrm{Top} \vdash \mathrm{If}\ (x>0)\,(n * f(x-1))\,1 : \mathrm{Nat}\,\mathrm{P}_1}$$

(a) Basic case

$$\dfrac{\dfrac{}{\Gamma_n \vdash (n>0) : \mathrm{Bool}\,\mathrm{P}_{(n>0)}} \qquad \dfrac{\dfrac{}{\Gamma_n \vdash x : \mathrm{Nat}\,\mathrm{P}_{(n+1)}} \qquad \dfrac{\dfrac{}{\Gamma_n \vdash f : \mathrm{Nat}\,\mathrm{P}_n \to \mathrm{Nat}\,\mathrm{P}_{n!}} \qquad \dfrac{}{\Gamma_n \vdash x : \mathrm{Nat}\,\mathrm{P}_n}}{\Gamma_n \vdash f(x) : \mathrm{Nat}\,\mathrm{P}_{n!}}}{(x>0) \Rightarrow \Gamma_n \vdash x * f(x-1) : \mathrm{Nat}\,\mathrm{P}_{(n+1)!}} \qquad \dfrac{\mathit{False}}{(x=0) \Rightarrow \Gamma_n \vdash 1 : \mathrm{Nat}\,\mathrm{P}_{(n+1)!}}}{x : \mathrm{Nat}\,\mathrm{P}_{(n+1)}, f : \mathrm{Nat}\,\mathrm{P}_n \to \mathrm{Nat}\,\mathrm{P}_{n!} \vdash \mathrm{If}\ (x>0)\,(x * f(x-1))\,1 : \mathrm{Nat}\,\mathrm{P}_{(n+1)!}}$$

(b) Induction case

Table 4: Typing of Factorial

which is the more precise we can hope. So for all values v which realize $\mathrm{Nat}\,\mathrm{P}_n$, $(\mathbf{compile}\,\mathtt{Fac}, [], [v], [])$ realizes $\mathrm{Nat}\,\mathrm{P}_{n!}$. But then, for all $n \in \mathrm{Nat}$,

$$(\mathbf{compile}\,\mathtt{Fac}, [], [\mathtt{V}(n)], []) \succ^* ([], e, [\mathtt{V}(n)], []),$$

which is a special case of our operational correctness of our compiler.

7 Discussion

We have seen how the correctness of our compiler can be deduced from the usual adequacy lemma of realizability. But this result of correctness has to be seen as a result of conservation of typing between our high-level term and its compiled program.

It does not link the reduction of a term in the source language with the evaluation of it compiled term. Indeed, we haven't even defined the reduction of our source language. But it is still possible to get some operational results, as soon as our type system is rich enough to give precise information on the reduction of high-level terms.

SECD machine is a first step to assembly language. Indeed, dumps can be seen as a way to save the code pointer plus the current environment and stack. But it is not straight to extend this method so that realizers could be assembly terms, for many reasons: (1) an assembly program is almighty and can cheat when interacting with a test, simply erasing it and faking it has succeeded it, (2) we need to deal with memory allocation. (3) There is no more `App` instruction, tests of $A \to B$ cannot be defined directly anymore. But it seems doable to adapt our framework to the work of [9] .

We also want to look at various forms of compiler correctness for high-level languages with references or effects. We and others have successfully applied essentially the same mathematical machinery to reasoning about higher-order functions with encapsulated local state [1, 7, 15, 2], so moving those ideas down to the low level looks very doable.

There is a great deal of related work on program logics, compiler correctness and the semantics of types, much of which we have plundered here. Formal verification of compilers goes back over four decades [13, 10] and has recently received increased attention, with notable automated efforts including Leroy's verification of an optimizing compiler for a C-like language [12]. Appel's Foundational Proof Carrying Code project [3] at Princeton has very similar goals to this work, and many of the techniques we use were introduced by Appel and his coauthors. We mention in particular the use of step-indexing [4, 16] and its modal refinement [5]. Shao's group at Yale have also done impressive work on formalizing and verifying specifications for low-level code in Coq, including memory managers, garbage collectors and other challenging pieces of systems code [14, 17]. It will be interesting to see if we can do similar things in our relational style.

References

[1] A. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, Princeton, NJ, USA, 2004.

[2] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*, 2009.

[3] A. Appel. Foundational proof-carrying code. In *Proc. 16th IEEE Symposium on Logic in Computer Science (LICS)*, 2001.

[4] A. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5), 2001.

[5] A.W. Appel, P.-A. Melliès, C.D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–122, 2007.

[6] N. Benton. Abstracting allocation: The new new thing. In *CSL '06*, volume 4207 of *LNCS*. Springer-Verlag, September 2006.

[7] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *Proc. 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *Lecture Notes in Computer Science*, 2005.

[8] Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 97–108, New York, NY, USA, 2009. ACM.

[9] Nick Benton and Nicolas Tabareau. Compiling functional types to relational specifications for low level imperative code. In *TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation*, pages 3–14, New York, NY, USA, 2009. ACM.

[10] M. Dave. Compiler verification: a bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6), 2003.

[11] JL Krivine. Realizability in classical logic. Course notes of a series of lectures given in the University of Marseille, may 2004 (last revision: july 2005). *Panoramas et syntheses, Société Mathématique de France*.

[12] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. 33rd Symposium on Principles of Programming Languages (POPL)*, 2006.

[13] J. McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. *Proceedings Symposium in Applied Mathematics*, 19:33–41, 1967.

[14] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd ACM Symposium on Principles of Programming Languages (POPL)*, 2006.

[15] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*. CUP, 1998.

[16] G. Tan, A. Appel, K. Swadi, and D. Wu. Construction of a semantic model for a typed assembly language. In *Proc. 5th Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2004.

[17] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50, 2004.