Compositional Compiler Correctness with Quantified Types

Nick Benton[1] and Chung-Kil Hur[2]

[1] Microsoft Research [nick@microsoft.com]
[2] University of Cambridge [ckh25@cam.ac.uk]

Abstract. We define operational logical relations between terms of a polymorphically typed functional language and low-level programs for a variant SECD machine. The relations, defined using relational parametricity, biorthogonality and step-indexing, give extensional and compositional specifications that capture what it means for low-level code and machine values to realize typed source-level terms. We show how the relations can be used to establish not only a traditional form of compiler correctness, but also to justify source-level transformations and, by giving compiler-independent specifications of how source-level abstractions are mapped to the target level, to justify linking compiled code with hand-optimized code fragments that exploit non-parametric and non-functional low-level operations, yet are extensionally well-behaved. The definitions and results have been fully formalized using the Coq proof assistant.

1 Introduction

Statements and proofs of compiler correctness concern relations between source and target languages. The minimal requirements of such a relation are (1) that any executable source program (e.g. closed expression of ground type) is related to its compiled version; (2) that the relation respects the semantics of the two languages (e.g. by being some kind of bisimulation); (3) that the relation is adequate, in that it does the right thing (e.g. entails cotermination) at observable types. There is a whole range of possible correspondence relations that satisfy these requirements, and the choice is not just one of proof technique, but significantly affects the strength and utility of theorem one proves.

The straightforward approach is to take the 'least' extension of the relation defined by the compilation function itself, subject to the requirement that an inductive proof over programs goes through. But the straightforward approach is not compositional. The notion of correspondence it gives for *open* terms – the induction hypothesis of the main proof – essentially only says anything about what happens in closing contexts compiled by this particular compiler. It cannot be used to justify or reason about linking low-level code produced by the compiler with code from elsewhere, such as a previous version of the compiler, foreign code from another language, or handcrafted machine code implementing a library.

This is a significant shortcoming, since no real compiler produces machine code with no external dependencies.

Similarly, at the high level we do not want the correctness relation to be tied to the specific optimizations that the compiler performs; rather, it should encompass a sufficiently wide range of equations in the first place, for example by being closed under contextual equivalence of high-level programs. Defining a sound, compositional notion of what it means for a piece of low-level code to implement, or *realize*, a particular high-level term that is maximally permissive in terms of just how the low-level code goes about its computation is a surprisingly subtle matter, which we have previously addressed in the context of compiling a simply-typed functional language [3]. In that work, we related a denotational semantics of the source language to the behaviour of low-level code via logical relations defined using two main techniques: step-indexing, in the style of Appel and McAllester [2], Ahmed [1], and others, and biorthogonality, as introduced by Pitts and Stark [10] and Krivine [7], and further applied by, for example, Vouillon and Melliès [11].

This paper follows the broad outline of our previous one [3], but makes a number of new contributions. The main one is that we now treat a language with impredicative universal and existential types. This is a non-trivial technical extension, and, as one of the examples will demonstrate, our realizability relation captures the requirement on low-level code realizing quantified types to behave parametrically from an extensional perspective, whilst allowing sufficient freedom for it to implement that behaviour in a decidedly non-parametric manner. We compositionally prove full functional correctness for a compiler for the polymorphic language, which also now performs tail-call optimizations. The second main difference from our previous work is that we work with an operational, rather than a denotational, semantics for the high-level language, and that rather than bake in a particular high-level notion of equivalence that should be respected, we parameterize our definitions and results by an adequate congruence relation on the source. In particular, if one takes this parameter to be the contextual equivalence of source language, the resulting realizability relation is fully-abstract. The operational relations make use of a novel form of converging sequence of approximations in place of an admissible closure operation appearing in our earlier work.

All the metatheory and examples have been formally verified in the Coq proof assistant, and the proof script is available from the authors' web pages.

2 Source Language

F_v is a polymorphically-typed call-by-value functional language with recursion. The grammar of types is

$$\tau := X \mid \texttt{Int} \mid \texttt{Bool} \mid \tau \to \tau' \mid \tau \times \tau' \mid \forall X.\tau \mid \exists X.\tau$$

Type variable contexts Θ are lists of type variables, whilst type contexts Γ map term variables to types in the usual way.

Values:

$$\frac{}{\Theta; \Gamma, x : \tau \vdash x : \tau} \qquad \frac{}{\Theta; \Gamma \vdash b : \texttt{Bool}}\ (b \in \mathbb{B}) \qquad \frac{}{\Theta; \Gamma \vdash n : \texttt{Int}}\ (n \in \mathbb{N})$$

$$\frac{\Theta; \Gamma, x : \tau \vdash M : \tau'}{\Theta; \Gamma \vdash \lambda x.\, M : \tau \to \tau'} \qquad \frac{\Theta; \Gamma, f : \tau \to \tau', x : \tau \vdash M : \tau'}{\Theta; \Gamma \vdash \mathrm{Rec}\ f\, x.\, M : \tau \to \tau'} \qquad \frac{\Theta; \Gamma \vdash V_i : \tau_i\ (i = 1, 2)}{\Theta; \Gamma \vdash \langle V_1, V_2 \rangle : \tau_1 \times \tau_2}$$

$$\frac{\Theta \vdash \Gamma \quad \Theta, X \vdash \tau \quad \Theta, X; \Gamma \vdash V : \tau}{\Theta; \Gamma \vdash \Lambda X.V : \forall X.\tau} \qquad \frac{\Theta \vdash \Gamma \quad \Theta, X \vdash \tau \quad \Theta \vdash \tau' \quad \Theta; \Gamma \vdash V : \tau[\tau'/X]}{\Theta; \Gamma \vdash \mathrm{Pack}\{\tau', V\} : \exists X.\tau}$$

Expressions:

$$\frac{\Theta; \Gamma \vdash V : \tau}{\Theta; \Gamma \vdash [V] : \tau} \qquad \frac{\Theta; \Gamma \vdash M : \tau \quad \Theta; \Gamma, x : \tau \vdash N : \tau'}{\Theta; \Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : \tau'} \qquad \frac{\Theta; \Gamma \vdash V_1 : \tau \to \tau' \quad \Theta; \Gamma \vdash V_2 : \tau}{\Theta; \Gamma \vdash V_1\, V_2 : \tau'}$$

$$\frac{\Theta; \Gamma \vdash V : \texttt{Bool} \quad \Theta; \Gamma \vdash M_1 : \tau \quad \Theta; \Gamma \vdash M_2 : \tau}{\Theta; \Gamma \vdash \texttt{if}\ V\ \texttt{then}\ M_1\ \texttt{else}\ M_2 : \tau} \qquad \frac{\Theta; \Gamma \vdash V_1 : \texttt{Int} \quad \Theta; \Gamma \vdash V_2 : \texttt{Int}}{\Theta; \Gamma \vdash V_1 \star V_2 : \texttt{Int}}$$

$$\frac{\Theta; \Gamma \vdash V_1 : \texttt{Int} \quad \Theta; \Gamma \vdash V_2 : \texttt{Int}}{\Theta; \Gamma \vdash V_1 > V_2 : \texttt{Bool}} \qquad \frac{\Theta; \Gamma \vdash V : \tau_1 \times \tau_2}{\Theta; \Gamma \vdash \pi_i(V) : \tau_i\ (i = 1, 2)}$$

$$\frac{\Theta \vdash \Gamma \quad \Theta, X \vdash \tau \quad \Theta; \Gamma \vdash V : \forall X.\tau \quad \Theta \vdash \tau'}{\Theta; \Gamma \vdash V\, \tau' : \tau[\tau'/X]}$$

$$\frac{\Theta \vdash \Gamma \quad \Theta \vdash \tau \quad \Theta, X \vdash \tau' \quad \Theta; \Gamma \vdash V : \exists X.\tau' \quad \Theta, X; \Gamma, x : \tau' \vdash M : \tau}{\Theta; \Gamma \vdash \mathrm{Unpack}\, V\ \mathrm{as}\, \{X, x\}\ \mathrm{in}\, M : \tau}$$

Fig. 1. Typing rules for F_v

We separate values, V, from expressions, M, and restrict the syntax to ANF, with explicit sequencing by `let` and inclusion of values into expressions by $[\cdot]$. The typing rules for F_v are shown in Figure 1; we omit the rules for well-formedness of types and contexts and most of the obvious well-formedness hypotheses in the type rules. The $\star$ and $>$ symbols stand for integer- and boolean-valued primitive operation. Note the 'value restriction': type abstraction and existential package formation are applied to (and yield) syntactic values.

For $\Theta \vdash \Gamma$ and $\Theta; \Gamma \vdash \tau$, we define the following sets:

$$\begin{aligned}
\mathit{Value}\ \Theta\ \Gamma\ \tau &= \{\, V \mid \Theta; \Gamma \vdash V : \tau \,\} \\
\mathit{CValue}\ \tau &= \mathit{Value}\ []\ []\ \tau \\
\mathit{Exp}\ \Theta\ \Gamma\ \tau &= \{\, M \mid \Theta; \Gamma \vdash M : \tau \,\} \\
\mathit{CExp}\ \tau &= \mathit{Exp}\ []\ []\ \tau \\
\mathit{EValue}\ \Theta\ \Gamma &= \textstyle\prod_{i=1\ldots n} \mathit{Value}\ \Theta\ []\ \tau_i \qquad \text{where } \Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n\ .
\end{aligned}$$

F_v has a standard deterministic call-by-value semantics, defined over closed terms of closed types. The interesting small-step transitions include:

$$\begin{aligned}
\texttt{let}\ x = [V]\ \texttt{in}\ N &\mapsto N[V/x] \\
(\mathrm{Rec}\ f\, x.\, M)\, V &\mapsto M[(\mathrm{Rec}\ f\, x.\, M)/f\,,\ V/x] \\
(\Lambda X.V)\, t &\mapsto V[t/X] \\
\mathrm{Unpack}\, (\mathrm{Pack}\{\tau, V\})\ \mathrm{as}\, \{X, x\}\ \mathrm{in}\, M &\mapsto M[\tau/X][V/x]
\end{aligned}$$

There is an equivalent big-step style semantics. Write $M \Mapsto V$ (resp. $M \Mapsto_k V$) to mean that the closed expression M, of some closed type, evaluates to the closed value V of the same type (resp. in k steps).

3 Target Machine

The low-level target is a variant of the SECD machine [8], extending the one we used in a previous paper [3]. We have previously looked at compilation to a lower-level assembly language [4], but have here chosen a slightly higher-level target so as to keep the presentation less cluttered with low-level detail. Although the SECD machine was originally designed as a target for compiling functional languages, the kind of relations we construct between source and target will work for lower-level targets too. The substantial independence of our definitions from the fine detail of exactly what compiled code looks like is part of the point of the compositional, extensional approach we are espousing, and we have added several new, non-functional, operations to the original SECD machine so that we can express more interesting and realistic low-level optimizations. The set *Instruction*, ranged over by *inst*, is defined by

$$\begin{aligned} \mathit{inst} := {} & \mathtt{Pop} \mid \mathtt{Push}\, i \mid \mathtt{PushE} \mid \mathtt{PopE} \mid \mathtt{PushN}\, n \mid \mathtt{Op}\, \star \mid \mathtt{PushC}\, c \mid \mathtt{PushRC}\, c \mid \\ & \mathtt{App} \mid \mathtt{AppNoDump} \mid \mathtt{Ret} \mid \mathtt{Sel}\, (c_1, c_2) \mid \mathtt{SelNoDump}\, (c_1, c_2) \mid \mathtt{Join} \mid \\ & \mathtt{MkPair} \mid \mathtt{Fst} \mid \mathtt{Snd} \mid \mathtt{Eq} \mid \mathtt{IsNum} \end{aligned}$$

where c ranges over $\mathit{Code} \stackrel{\text{def}}{=} \mathit{list}\, \mathit{Instruction}$, n and i range over naturals, and $\star$ over binary operations on naturals. The set *MValue* of machine values, ranged over by v is defined by

$$v := \underline{n} \mid \mathtt{CL}\,(e, c) \mid \mathtt{RCL}\,(e, c) \mid \mathtt{PR}\,(v_1, v_2)$$

where e ranges over $\mathit{MEnv} \stackrel{\text{def}}{=} \mathit{list}\, \mathit{MValue}$. So an *MValue* is either a natural, a closure containing an environment and some code, a recursive closure, or a pair of values. We also define

$$\begin{aligned} \mathit{Stack} &= \mathit{list}\, \mathit{MValue} \\ \mathit{Dump} &= \mathit{list}\, (\mathit{Code} \times \mathit{MEnv} \times \mathit{Stack}) \\ \mathit{CESD} &= \mathit{Code} \times \mathit{MEnv} \times \mathit{Stack} \times \mathit{Dump} \end{aligned}$$

A configuration $\langle c,\ e,\ s,\ d\rangle \in \mathit{CESD}$ comprises the code to be executed, c, the current environment, e, an evaluation stack s and a call stack, or dump, d.

The deterministic one-step transition relation $\mapsto$ between configurations is defined in Figure 2. Note the non-standard 'no dump' forms of application and selection, which do not save a continuation on the dump and which we will use to compile tail-call optimizations; the `PushE` and `PopE` instructions, allowing the environment to be modified; the `IsNum` test for numberhood, which we will use in one of our examples; and the equality test `Eq`. Our previous paper [3] explains how the presence of non-functional 'reflective' operations, such as `Eq`, in the target breaks a straightforward realizability interpretation of types in the presence of term-level recursion, requiring step-indexing (or similar) in defining low-level interpretations of high-level types and values.

Configurations with no successor are said to be *stuck* or *terminated.* Write $\mathit{cesd} \mapsto^k$ (resp. $\mathit{cesd} \mapsto^k \mathit{cesd}'$) if cesd takes at least k steps without getting stuck (resp. and reduces to cesd'), and say it diverges, written $\mathit{cesd} \mapsto^\omega$ if it can always take a step:

$$\mathit{cesd} \mapsto^\omega \stackrel{\text{def}}{\iff} (\forall k, \mathit{cesd} \mapsto^k)$$

$$\begin{array}{rcll}
\langle \texttt{Pop} :: c,\, e,\, v :: s,\, d\rangle & \mapsto & \langle c,\, e,\, s,\, d\rangle & \\
\langle \texttt{Push}\, i :: c,\, [v_1,\ldots,v_k],\, s,\, d\rangle & \mapsto & \langle c,\, [v_1,\ldots,v_k],\, v_i :: s,\, d\rangle & \\
\langle \texttt{PushE} :: c,\, e,\, v :: s,\, d\rangle & \mapsto & \langle c,\, v :: e,\, s,\, d\rangle & \\
\langle \texttt{PopE} :: c,\, v :: e,\, s,\, d\rangle & \mapsto & \langle c,\, e,\, v :: s,\, d\rangle & \\
\langle \texttt{PushN}\, n :: c,\, e,\, s,\, d\rangle & \mapsto & \langle c,\, e,\, \underline{n} :: s,\, d\rangle & \\
\langle \texttt{PushC}\, \mathit{bod} :: c,\, e,\, s,\, d\rangle & \mapsto & \langle c,\, e,\, \texttt{CL}\,(e, \mathit{bod}) :: s,\, d\rangle & \\
\langle \texttt{PushRC}\, \mathit{bod} :: c,\, e,\, s,\, d\rangle & \mapsto & \langle c,\, e,\, \texttt{RCL}\,(e, \mathit{bod}) :: s,\, d\rangle & \\
\langle \texttt{App} :: c,\, e,\, v :: \texttt{CL}\,(e', \mathit{bod}) :: s,\, d\rangle & \mapsto & \langle \mathit{bod},\, v :: e',\, [],\, (c,e,s) :: d\rangle & \\
\langle \texttt{App} :: c,\, e,\, v :: \texttt{RCL}\,(e', \mathit{bod}) :: s,\, d\rangle & \mapsto & \langle \mathit{bod},\, v :: \texttt{RCL}\,(e', \mathit{bod}) :: e',\, [],\, (c,e,s) :: d\rangle & \\
\langle \texttt{AppNoDump} :: c,\, e,\, v :: \texttt{CL}\,(e', \mathit{bod}) :: s,\, d\rangle & \mapsto & \langle \mathit{bod},\, v :: e',\, [],\, d\rangle & \\
\langle \texttt{AppNoDump} :: c,\, e,\, v :: \texttt{RCL}\,(e', \mathit{bod}) :: s,\, d\rangle & \mapsto & \langle \mathit{bod},\, v :: \texttt{RCL}\,(e', \mathit{bod}) :: e',\, [],\, d\rangle & \\
\langle \texttt{Op}\star :: c,\, e,\, \underline{n_2} :: \underline{n_1} :: s,\, d\rangle & \mapsto & \langle c,\, e,\, \underline{n_1 \star n_2} :: s,\, d\rangle & \\
\langle \texttt{Ret} :: c,\, e,\, v :: s,\, (c', e', s') :: d\rangle & \mapsto & \langle c',\, e',\, v :: s',\, d\rangle & \\
\langle \texttt{Sel}\,(c_1, c_2) :: c,\, e,\, v :: s,\, d\rangle & \mapsto & \langle c_1,\, e,\, s,\, (c, [], []) :: d\rangle & (\text{if } v \neq \underline{0}) \\
\langle \texttt{Sel}\,(c_1, c_2) :: c,\, e,\, \underline{0} :: s,\, d\rangle & \mapsto & \langle c_2,\, e,\, s,\, (c, [], []) :: d\rangle & \\
\langle \texttt{SelNoDump}\,(c_1, c_2) :: c,\, e,\, v :: s,\, d\rangle & \mapsto & \langle c_1,\, e,\, s,\, d\rangle & (\text{if } v \neq \underline{0}) \\
\langle \texttt{SelNoDump}\,(c_1, c_2) :: c,\, e,\, \underline{0} :: s,\, d\rangle & \mapsto & \langle c_2,\, e,\, s,\, d\rangle & \\
\langle \texttt{Join} :: c,\, e,\, s,\, (c', e', s') :: d\rangle & \mapsto & \langle c',\, e,\, s,\, d\rangle & \\
\langle \texttt{MkPair} :: c,\, e,\, v_1 :: v_2 :: s,\, d\rangle & \mapsto & \langle c,\, e,\, \texttt{PR}\,(v_2, v_1) :: s,\, d\rangle & \\
\langle \texttt{Fst} :: c,\, e,\, \texttt{PR}\,(v_1, v_2) :: s,\, d\rangle & \mapsto & \langle c,\, e,\, v_1 :: s,\, d\rangle & \\
\langle \texttt{Snd} :: c,\, e,\, \texttt{PR}\,(v_1, v_2) :: s,\, d\rangle & \mapsto & \langle c,\, e,\, v_2 :: s,\, d\rangle & \\
\langle \texttt{Eq} :: c,\, e,\, v_1 :: v_2 :: s,\, d\rangle & \mapsto & \langle c,\, e,\, \underline{1} :: s,\, d\rangle & (\text{if } v_1 = v_2) \\
\langle \texttt{Eq} :: c,\, e,\, v_1 :: v_2 :: s,\, d\rangle & \mapsto & \langle c,\, e,\, \underline{0} :: s,\, d\rangle & (\text{if } v_1 \neq v_2) \\
\langle \texttt{IsNum} :: c,\, e,\, \underline{n} :: s,\, d\rangle & \mapsto & \langle c,\, e,\, \underline{1} :: s,\, d\rangle & \\
\langle \texttt{IsNum} :: c,\, e,\, v :: s,\, d\rangle & \mapsto & \langle c,\, e,\, \underline{0} :: s,\, d\rangle & (\text{if } v \text{ is not } \underline{n} \text{ for any } n)
\end{array}$$

Fig. 2. Operational Semantics of Extended SECD Machine

Conversely, we say *cesd* terminates, and write $\mathit{cesd} \mapsto^* \not\lightning$, if it does not diverge.

4 Compiling $\mathbf{F}_v$ to SECD

The compiler is shown in Figure 3 and comprises mutually-recursive functions, both written $(\!|\cdot|\!)$, mapping typed F_v values and expressions into *Code*. The compilation of expressions is parameterized by a boolean flag *ret* that identifies expressions that are in 'tail position' and hence expect to be immediately followed by a return instruction. Applications in tail position are compiled with the `AppNoDump` instruction, which does *not* push the calling context to the dump, so allowing the called function to return directly to the caller's caller. Similarly, conditionals normally push a common continuation to the dump, and each branch ends with a `Join`; when the conditional is in tail position, however, the pushing of the context is elided and each branch compiled in tail position.

5 Logical Relations

In this section we define logical relations between components of the low-level SECD machine and terms of the high-level language F_v, with the intention of capturing just when a piece of low-level code realizes a particular source term, relative to the interfaces implicit in the calling conventions of the compiler. In fact, there will be two relations: $\preceq$ (in Section 5.1) defining when a low-level

Values:

$$
\begin{aligned}
\llparenthesis \Theta; x_1 : \tau_1, \ldots, x_n : \tau_n \vdash x_i : \tau_i \rrparenthesis &= [\texttt{Push}\, i] \\
\llparenthesis \Theta; \Gamma \vdash \mathit{true} : \texttt{Bool} \rrparenthesis &= [\texttt{PushN}\, 1] \\
\llparenthesis \Theta; \Gamma \vdash \mathit{false} : \texttt{Bool} \rrparenthesis &= [\texttt{PushN}\, 0] \\
\llparenthesis \Theta; \Gamma \vdash n : \texttt{Int} \rrparenthesis &= [\texttt{PushN}\, n] \\
\llparenthesis \Theta; \Gamma \vdash \langle V_1, V_2 \rangle : \tau_1 \times \tau_2 \rrparenthesis &= \llparenthesis \Theta; \Gamma \vdash V_1 : \tau_1 \rrparenthesis + + \llparenthesis \Theta; \Gamma \vdash V_2 : \tau_2 \rrparenthesis + + [\texttt{MkPair}] \\
\llparenthesis \Theta; \Gamma \vdash \lambda x.\, M : \tau \to \tau' \rrparenthesis &= [\texttt{PushC}\, (\llparenthesis \Theta; \Gamma, x : \tau \vdash M : \tau' \rrparenthesis_{\mathsf{true}})] \\
\llparenthesis \Theta; \Gamma \vdash \mathrm{Rec}\ f\, x = M : \tau \to \tau' \rrparenthesis &= [\texttt{PushRC}\, (\llparenthesis \Theta; \Gamma, f : \tau \to \tau', x : \tau \vdash M : \tau' \rrparenthesis_{\mathsf{true}})] \\
\llparenthesis \Theta; \Gamma \vdash \Lambda X.\, V : \forall X.\tau \rrparenthesis &= \llparenthesis \Theta, X; \Gamma \vdash V : \tau \rrparenthesis \\
\llparenthesis \Theta; \Gamma \vdash \mathrm{Pack}\{\tau', V\} : \exists X.\tau \rrparenthesis &= \llparenthesis \Theta; \Gamma \vdash V : \tau[\tau'/X] \rrparenthesis
\end{aligned}
$$

Expressions:

$$
\begin{aligned}
\llparenthesis \mathit{ret} \rrparenthesis &= \text{if } \ \mathit{ret} = \mathsf{true} \ \text{ then } \ [\texttt{Ret}] \ \text{ else } \ [] \\
\llparenthesis \Theta; \Gamma \vdash [V] : t \rrparenthesis_{ret} &= \llparenthesis \Theta; \Gamma \vdash V : t \rrparenthesis + + \llparenthesis \mathit{ret} \rrparenthesis \\
\llparenthesis \Theta; \Gamma \vdash V_1 \star V_2 : \texttt{Int} \rrparenthesis_{ret} &= \llparenthesis \Theta; \Gamma \vdash V_1 : \texttt{Int} \rrparenthesis + + \llparenthesis \Theta; \Gamma \vdash V_2 : \texttt{Int} \rrparenthesis + + [\texttt{Op}\, \star] + + \llparenthesis \mathit{ret} \rrparenthesis \\
\llparenthesis \Theta; \Gamma \vdash V_1 > V_2 : \texttt{Bool} \rrparenthesis_{ret} &= \llparenthesis \Theta; \Gamma \vdash V_1 : \texttt{Int} \rrparenthesis + + \llparenthesis \Theta; \Gamma \vdash V_2 : \texttt{Int} \rrparenthesis + + \\
&\qquad [\texttt{Op}\, (\lambda(n_1, n_2).n_1 > n_2 \supset 1 \mid 0)] + + \llparenthesis \mathit{ret} \rrparenthesis \\
\llparenthesis \Theta; \Gamma \vdash \pi_1(V) : \tau_1 \rrparenthesis_{ret} &= \llparenthesis \Theta; \Gamma \vdash V : \tau_1 \times \tau_2 \rrparenthesis + + [\texttt{Fst}] + + \llparenthesis \mathit{ret} \rrparenthesis \\
\llparenthesis \Theta; \Gamma \vdash \pi_2(V) : \tau_2 \rrparenthesis_{ret} &= \llparenthesis \Theta; \Gamma \vdash V : \tau_1 \times \tau_2 \rrparenthesis + + [\texttt{Snd}] + + \llparenthesis \mathit{ret} \rrparenthesis \\
\llparenthesis \Theta; \Gamma \vdash \texttt{if}\ V\ \texttt{then}\ M_1\ \texttt{else}\ M_2 : \tau \rrparenthesis_{\mathsf{true}} &= \llparenthesis \Theta; \Gamma \vdash V : \texttt{Bool} \rrparenthesis + + \\
&\qquad [\texttt{SelNoDump}\, (\llparenthesis \Theta; \Gamma \vdash M_1 : \tau \rrparenthesis_{\mathsf{true}}, \llparenthesis \Theta; \Gamma \vdash M_2 : \tau \rrparenthesis_{\mathsf{true}})] \\
\llparenthesis \Theta; \Gamma \vdash \texttt{if}\ V\ \texttt{then}\ M_1\ \texttt{else}\ M_2 : \tau \rrparenthesis_{\mathsf{false}} &= \llparenthesis \Theta; \Gamma \vdash V : \texttt{Bool} \rrparenthesis + + \\
&\qquad [\texttt{Sel}\, (\llparenthesis \Theta; \Gamma \vdash M_1 : \tau \rrparenthesis_{\mathsf{false}} + + [\texttt{Join}], \llparenthesis \Theta; \Gamma \vdash M_2 : \tau \rrparenthesis_{\mathsf{false}} + + [\texttt{Join}])] \\
\llparenthesis \Theta; \Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : \tau' \rrparenthesis_{ret} &= \llparenthesis \Theta; \Gamma \vdash M : \tau \rrparenthesis_{\mathsf{false}} + + \\
&\qquad [\texttt{PushE}] + + \llparenthesis \Theta; \Gamma, x : \tau \vdash N : \tau' \rrparenthesis_{ret} + + [\texttt{PopE}] \\
\llparenthesis \Theta; \Gamma \vdash V_1\, V_2 : \tau' \rrparenthesis_{\mathsf{true}} &= \llparenthesis \Theta; \Gamma \vdash V_1 : \tau \to \tau' \rrparenthesis + + \llparenthesis \Theta; \Gamma \vdash V_2 : \tau \rrparenthesis + + [\texttt{AppNoDump}] \\
\llparenthesis \Theta; \Gamma \vdash V_1\, V_2 : \tau' \rrparenthesis_{\mathsf{false}} &= \llparenthesis \Theta; \Gamma \vdash V_1 : \tau \to \tau' \rrparenthesis + + \llparenthesis \Theta; \Gamma \vdash V_2 : \tau \rrparenthesis + + [\texttt{App}] \\
\llparenthesis \Theta; \Gamma \vdash V\, \tau' : \tau[\tau'/X] \rrparenthesis_{ret} &= \llparenthesis \Theta; \Gamma \vdash V : \forall X.\tau \rrparenthesis + + \llparenthesis \mathit{ret} \rrparenthesis \\
\llparenthesis \Theta; \Gamma \vdash \mathrm{Unpack}\, V\, \mathrm{as}\, \{X, x\}\, \mathrm{in}\, M : \tau \rrparenthesis_{ret} &= \llparenthesis \Theta; \Gamma \vdash V : \exists X.\tau' \rrparenthesis + + \\
&\qquad [\texttt{PushE}] + + \llparenthesis \Theta, X; \Gamma, x : \tau' \vdash M : \tau \rrparenthesis_{ret} + + [\texttt{PopE}]
\end{aligned}
$$

Fig. 3. Compiler for F_v

component approximates a source term, and $\succeq$ (Section 5.2) expressing the converse; the two directions correspond roughly to the traditional soundness and adequacy theorems used to show correspondence between an operational and a denotational semantics.

The detailed definitions of the relations are slightly complex, so it seems worth starting with a high-level overview. Firstly, as indicated in the introduction, each of the relations is parameterized by a notion of contextual approximation, $\sqsubseteq$, on the source language. $\sqsubseteq$ can be taken to be the contextual preorder for F_v, but may be something weaker, such as the order of some non-fully abstract denotational model. The factorization separates concerns and provides some 'tuneability' in the degree to which the realizability relation is required to respect source-level equivalence. Both relations also involve (at least intuitively) operational analogues of the denotational notion of ω-chain: in the case of $\preceq$, this is given by step-indexing on the low-level machine, whilst for $\succeq$ the 'chains' are on the source side, and we use an interesting new construction based on an $\mathbb{N}$-indexed family of precongruences $\precsim_i^{\tau}$. The other important part of the constructions is the use of certain closure operators to 'extensionalize' the sets of low-level values that are related to particular F_v terms. These operators are defined using biorthogonality, which has the flavour of double-negation: one starts with an over-intensional set of low-level values or computations, constructs the set of all contexts that yield a particular observation when combined with (plugged into) any element of the initial set, and then comes back to the set of all low-level values or computations that yield the observation when combined with any of those contexts. In the case of $\preceq$, the obervation is divergence (actually, stepping for at least some number of steps), whilst for $\succeq$ the observation is termination. Each of the relations is essentially logical, with universals and existentials interpreted by quantification over relations between source and target (actually, over certain indexed families of relations in the case of $\preceq$). We now make these intuitions precise.

Precongruence relations on $\mathbf{F}_v$. The formal requirement on the source relation $\sqsubseteq$, by which we parameterize our logical relations, is that it be an *adequate precongruence*, by which we mean:

1. $\sqsubseteq$ is reflexive and transitive.
2. $\sqsubseteq$ is preserved by all constructs of F_v. For example:
$$\frac{\Theta; \Gamma \vdash V \sqsubseteq V' : \tau_1[\tau_2/X]}{\Theta; \Gamma \vdash \mathrm{Pack}\{\tau_2, V\} \sqsubseteq \mathrm{Pack}\{\tau_2, V'\} : \exists X.\tau_1}$$
$$\frac{\Theta; \Gamma \vdash M \sqsubseteq M' : \tau_1 \quad \Theta; \Gamma, x : \tau_1 \vdash N \sqsubseteq N' : \tau_2}{\Theta; \Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N \sqsubseteq \texttt{let}\ x = M'\ \texttt{in}\ N' : \tau_2}$$
3. $\sqsubseteq$ is preserved by both type and value substitution, e.g.
$$\frac{X_1, \ldots, X_n; \Gamma \vdash V \sqsubseteq V' : \tau \quad \{\Theta_2 \vdash \tau_i\}_{1 \le i \le n}}{\Theta_2; \Gamma[\tau_i/X_i] \vdash V[\tau_i/X_i] \sqsubseteq V'[\tau_i/X_i] : \tau[\tau_i/X_i]}$$
$$\frac{\Theta; x_1 : \tau_1, \ldots, x_n : \tau_n \vdash V \sqsubseteq V' : \tau \quad \{\Theta; \Gamma_2 \vdash W_i \sqsubseteq W_i' : \tau_i\}_{1 \le i \le n}}{\Theta; \Gamma_2 \vdash V[W_i/x_i] \sqsubseteq V'[W_i'/x_i] : \tau}$$
4. $\sqsubseteq$ is adequate, reflecting ground observations on closed programs. So if $[]; [] \vdash M \sqsubseteq M' : \texttt{Int}$ and $M \models\!\!\!\Rightarrow n$ then $M' \models\!\!\!\Rightarrow n$, and similarly for booleans.

The 'true' observational preorder for F_v is the *largest* such adequate precongruence, but we do not fix that choice in defining the relations between high-level and low-level. In particular, the conditions on $\sqsubseteq$ only require existential packages to be related when the witness type τ_2 is the same, but the largest $\sqsubseteq$ will also relate packages with different witness types.

Approximation on $\mathbf{F}_v$. For one of the logical relations we shall define between low-level and high-level, we will also make use of a particular $\mathbb{N}$-indexed family of (non-adequate) precongruences, $\precsim_i^\tau$. These relations give a notion of approximation on F_v terms: intuitively, $M \precsim_i^\tau M'$ says that M is less than or equal to M' up to i steps of execution. Indeed, these relations are approximately adequate in the the sense that if $[]; [] \vdash M \precsim_i^{\texttt{Int}} M' : \texttt{Int}$ and $M \models\!\!\!\Rightarrow_j n$ for $j < i$ then $M' \models\!\!\!\Rightarrow n$, and similarly for booleans.

If $RV_i \subseteq (CValue\ \tau) \times (CValue\ \tau)$ is an indexed relation between closed values of a closed type τ, then let the indexed relation $RV^\bullet$ be defined by

$$\begin{aligned}&RV^\bullet_i \subseteq (CExp\ \tau) \times (CExp\ \tau)\\ &= \{(M, M') \mid \forall j < i, \forall V, M \Mapsto_j V \rightarrow \exists j' \leq j, \exists V', M' \Mapsto_{j'} V' \wedge (V, V') \in RV_{i-j}\}\end{aligned}$$

and note that $RV^\bullet$ at index i only depends on RV at strictly smaller indices if M takes at least one step. We then define the indexed relations $\precsim^\tau_i \subseteq (CValue\ \tau) \times (CValue\ \tau)$ by well-founded induction on the indices:

$$\begin{aligned}
\precsim^{\texttt{Int}}_i &= \{(n, n) \mid n \in \mathbb{N}\}\\
\precsim^{\texttt{Bool}}_i &= \{(b, b) \mid b \in \mathbb{B}\}\\
\precsim^{\tau_1 \times \tau_2}_{i+1} &= \{((V_1, V_2), (V_1', V_2')) \mid (V_1, V_1') \in \precsim^{\tau_1}_i \wedge (V_2, V_2') \in \precsim^{\tau_2}_i\}\\
\precsim^{\tau_1 \rightarrow \tau_2}_i &= \{(F, F') \mid \forall V : \tau_1, (F\,V,\ F'\,V) \in (\precsim^{\tau_2})^\bullet_i\}\\
\precsim^{\forall X.\tau}_i &= \{(V, V') \mid \forall \tau', (V\,\tau',\ V'\,\tau') \in \left(\precsim^{\tau[\tau'/X]}\right)^\bullet_i\}\\
\precsim^{\exists X.\tau}_{i+1} &= \{(\mathrm{Pack}\{\tau', V\}, \mathrm{Pack}\{\tau', V'\}) \mid (V, V') \in \precsim^{\tau[\tau'/X]}_i\}
\end{aligned}$$

where in the pair and existential cases, we take $\precsim^\tau_0$ to be the total relation on values of type τ. This is an 'applicative' rather than a 'logical' definition, in that the case for functions involves identical (rather than related) arguments.

We extend to open values and types by quantifying over type and term substitutions. If $\Theta = X_1, \ldots, X_u$, $\Gamma = x_1 : \sigma_1, \ldots, x_m : \sigma_m$, $\Theta \vdash \Gamma$, then:

$$\begin{aligned}
\precsim^{\Theta;\Gamma\vdash\sigma}_i &\subseteq (Value\ \Theta\ \Gamma\ \sigma) \times (Value\ \Theta\ \Gamma\ \sigma)\\
&= \{(V, V') \mid \forall \boldsymbol{\tau} : \Theta, \forall \boldsymbol{W} : \Gamma[\tau_k/X_k],\\
&\qquad (V[\tau_k/X_k][W_j/x_j],\ V'[\tau_k/X_k][W_j/x_j]) \in \precsim^{\sigma[\tau_k/X_k]}_i\}
\end{aligned}$$

We overload the notation for general open expressions:

$$\begin{aligned}
\precsim^{\Theta;\Gamma\vdash\sigma}_i &\subseteq (Exp\ \Theta\ \Gamma\ \sigma) \times (Exp\ \Theta\ \Gamma\ \sigma)\\
&= \{(M, M') \mid \forall \boldsymbol{\tau} : \Theta, \forall \boldsymbol{W} : \Gamma[\tau_k/X_k],\\
&\qquad (M[\tau_k/X_k][W_j/x_j],\ M'[\tau_k/X_k][W_j/x_j]) \in (\precsim^{\sigma[\tau_k/X_k]})^\bullet_i\}
\end{aligned}$$

For term environments, $\precsim^{\Theta\vdash\Gamma}_i \subseteq (EValue\ \Theta\ \Gamma) \times (EValue\ \Theta\ \Gamma)$ is defined pointwise from the value relation.

Biorthogonality on SECD. On the SECD machine side, we will follow the general pattern of biorthogonality [3], and work with (predicates on) substructures of machine configurations. Complete configurations are elements of *CESD*, whilst the substructures will be elements of *MValue* and of *MComp*, which is defined to be *Code* × *Stack*. If v : *MValue* then we define $\widehat{v}$: *MComp* to be $([], [v])$, the computation comprising an empty instruction sequence and a singleton stack with v on. Similarly, if c : *Code* then $\widehat{c}$: *MComp* is $(c, [])$.

The basic plugging operation on the low-level side is $\cdot \bowtie \cdot$, taking an element of *MComp*, the computation under test, and an element of *CESD*, thought of as a context, and combining them to yield a runnable configuration in *CESD*:

$$(c, s) \bowtie \langle c', e', s', d'\rangle \ = \ \langle c \mathbin{+\!\!+} c', e', s \mathbin{+\!\!+} s', d'\rangle$$

We also have an operation $\cdot \smile \cdot$ that appends an element of *MEnv* onto the environment component of a configuration:

$$e \smile \langle c',\, e',\, s',\, d'\rangle \;=\; \langle c',\, e +\!\!+\, e',\, s',\, d'\rangle$$

5.1 Approximating High-level By Low-level

The logical relation expressing what it means for low-level computations to approximate high-level terms works with step-indexed entities. We write *iMValue* for $\mathbb{N} \times \mathit{MValue}$, *iMComp* for $\mathbb{N} \times \mathit{MComp}$ and *iCESD* for $\mathbb{N} \times \mathit{CESD}$.

Now, for each machine environment $e \in \mathit{MEnv}$ and set $P \subseteq \mathit{iMValue}$ of indexed values, we define the set of indexed contexts $\downarrow^{e}(P) \subseteq \mathit{iCESD}$ to be

$$\{(j, \mathit{cesd}) \mid \forall (i,v) \in P, (\widehat{v} \bowtie (e \smile \mathit{cesd})) \mapsto^{\min(i,j)}\}$$

So an indexed context is in $\downarrow^{e}(P)$ if whenever we plug it together with an indexed value from P and the environment e, the resulting configuration takes a number of steps that is at least the minimum of the two indices.

Coming back the other way, if $e \in \mathit{MEnv}$ and $Q \subseteq \mathit{iCESD}$, we define the set of indexed machine computations $\Uparrow^{e}(Q) \subseteq \mathit{iMComp}$ to be

$$\{(i, \mathit{cs}) \mid \forall (j, \mathit{cesd}) \in Q, (\mathit{cs} \bowtie (e \smile \mathit{cesd})) \mapsto^{\min(i,j)}\}$$

Each of these operations is half of a contravariant adjunction (Galois connection), between the lattice of subsets of *iCESD* and those of subsets of *iMValue* and of *iMComp*, respectively.

Now we can start relating the low-level machine to the high-level language. If $e \in \mathit{MEnv}$ is a machine environment, τ is a closed type and $RT_i \subseteq \mathit{MValue} \times (\mathit{CValue}\ \tau)$ is a $\mathbb{N}$-indexed relation between machine values and closed F_v values of type τ, then define the indexed relation $\langle RT^{e}_{\perp}\rangle_i \subseteq \mathit{MComp} \times (\mathit{CExp}\ \tau)$ by

$$\begin{aligned}\langle RT^{e}_{\perp}\rangle_i = \{&(\mathit{comp}, M) \mid \\ &(i, \mathit{comp}) \in \Uparrow^{e} (\downarrow^{e} (\{(j,v) \mid \exists V : \tau, M \Mapsto V \wedge (v,V) \in RT_j\}))\}\end{aligned}$$

which is a kind of relational action of the lift monad, taking an indexed relation between low-level and high-level values to one between low-level and high-level computations. Notice that although this relation talks about high-level expressions evaluating to values, it does not directly say anything about related low-level computations 'yielding' values. Instead, we start with an initial set of low-level values, and then take the set of computations that are orthogonal to the set of contexts orthogonal to that initial set. This is what makes our relations 'extensional': we avoid overspecifying the intensional details of non-observable intermediate configurations, referring only to the observable behaviour of computations in context.

If τ is a closed source type, then let iRel_τ be the set of $\mathbb{N}$-indexed relations $R_i \subseteq \mathit{MValue} \times (\mathit{CValue}\ \tau)$. We say such a relation is *decreasing* when $R_0 \supseteq R_1 \supseteq \ldots$. The intuition of step-indexing is that a relation R is approximated by relations of the form 'not detectably un-related within i-steps', so the relations should get finer as more steps are available for testing. If $\Theta = X_1, \ldots, X_n$ is a type variable environment, then a relation environment for Θ is a vector $\boldsymbol{\tau R}$ of pairs $\tau_1 R_1, \ldots, \tau_n R_n$ where each τ_k is a closed type and $R_k \in \mathit{iRel}_{\tau_k}$.

Now for each Θ, matching relation environment $\boldsymbol{\tau R}$ and type σ such that $\Theta \vdash \sigma$, we define the indexed relation

$$\trianglelefteq_i^{\boldsymbol{\tau R},\sigma} \subseteq \mathit{MValue} \times (\mathit{CValue}\,(\sigma[\tau_k/X_k]))$$

by induction on σ as follows:

$$\begin{aligned}
\trianglelefteq_i^{\boldsymbol{\tau R},X_k} &= (R_k)_i \\
\trianglelefteq_i^{\boldsymbol{\tau R},\texttt{Int}} &= \{(\underline{n}, n) \mid n \in \mathbb{N}\} \\
\trianglelefteq_i^{\boldsymbol{\tau R},\texttt{Bool}} &= \{(\underline{0}, \mathit{false})\} \cup \{(\underline{n+1}, \mathit{true}) \mid n \in \mathbb{N}\} \\
\trianglelefteq_i^{\boldsymbol{\tau R},\sigma_1 \times \sigma_2} &= \{(\texttt{PR}\,(v_1, v_2), (V_1, V_2)) \mid (v_1, V_1) \in \trianglelefteq_i^{\boldsymbol{\tau R},\sigma_1} \wedge\ (v_2, V_2) \in \trianglelefteq_i^{\boldsymbol{\tau R},\sigma_2}\} \\
\trianglelefteq_i^{\boldsymbol{\tau R},\sigma_1 \to \sigma_2} &= \{(f, F) \mid \forall k \leq i, \forall (v, V) \in \trianglelefteq_k^{\boldsymbol{\tau R},\sigma_1}, \\
&\qquad (([\texttt{App}], [v, f]),\ (F\,V)) \in \langle (\trianglelefteq^{\boldsymbol{\tau R},\sigma_2})_{\bot}^{[]} \rangle_k\} \\
\trianglelefteq_i^{\boldsymbol{\tau R},\forall X.\sigma} &= \{(v, V) \mid \forall \tau', \forall R' \in \mathit{iRel}_{\tau'}, \\
&\qquad \text{decreasing } R' \to (\widehat{v},\ (V\,\tau')) \in \langle (\trianglelefteq^{\boldsymbol{\tau R},\tau' R',\sigma})_{\bot}^{[]} \rangle_i\} \\
\trianglelefteq_i^{\boldsymbol{\tau R},\exists X.\sigma} &= \{(v, \mathrm{Pack}\{\tau', V\}) \mid \exists R' \in \mathit{iRel}_{\tau'}, \\
&\qquad \text{decreasing } R' \wedge (v, V) \in \trianglelefteq_i^{\boldsymbol{\tau R},\tau' R',\sigma}\}
\end{aligned}$$

So the relation associated with a type variable is looked up in the relation environment, machine integers approximate the corresponding high-level value, the machine zero approximates false, and all non-zero machine integers approximate the true value of F_v. Pair values on the machine approximate high-level pairs pointwise.

The case for functions follows the usual pattern of monadic Kripke logical relations: at all future worlds (smaller indices) take related arguments to results related by the monadic lifting of the relational interpretation of the result type. This is where the low-level interface for function types, the calling convention, is specified: a machine value approximates a high-level function when putting it on the stack with a value approximating a high-level argument and executing an `App` instruction yields behaviour that approximates that of the high-level application.

Universally quantified types are intepreted using relational parametricity, where we quantify over all decreasing indexed relations between machine values and values of some F_v type. Similarly, a machine value v is related to a high-level existential package if there is *some* decreasing relation between low-level values and values of the witnessing source type such that v is related to the packed value.

Having defined the relation for values, we lift it to environments. If $\Theta \vdash \Gamma$ where Γ is $x_1 : \sigma_1, \ldots, x_m : \sigma_m$, and $\boldsymbol{\tau R}$ is a relation environment for Θ, then the indexed relation

$$\trianglelefteq_i^{\boldsymbol{\tau R},\Gamma} \ \subseteq\ \mathit{MEnv} \times (\mathit{EValue}\,[]\,(\Gamma[\tau_k/X_k]))$$

is the set of pairs $([v_1,\ldots,v_m],[V_1,\ldots,V_m])$ such that $v_j \trianglelefteq_i^{\boldsymbol{\tau R},\sigma_j} V_j$ for all $1 \leq j \leq m$. Then for general expressions in context:

$$\begin{aligned}&\trianglelefteq_i^{\boldsymbol{\tau R},\Gamma\vdash\sigma} \subseteq \mathit{MComp} \times (\mathit{Exp}\,[]\,(\Gamma[\tau_k/X_k])\,(\sigma[\tau_k/X_k]))\\ &\quad = \{(\mathit{comp}, M) \mid \forall i' \leq i, \forall (e,\boldsymbol{V}) \in \trianglelefteq_{i'}^{\boldsymbol{\tau R},\Gamma}, (\mathit{comp}, M[V_j/x_j]) \in \langle (\trianglelefteq^{\boldsymbol{\tau R},\sigma})_\perp^e \rangle_{i'}\}\end{aligned}$$

which is again 'logical', taking related environments to related computations for all smaller indices.

All that was for closed types. For closed values of *open* types $\Theta \vdash \sigma$, we define

$$\begin{aligned}\trianglelefteq_i^{\Theta\vdash\sigma} &\subseteq \mathit{MValue} \times (\mathit{Value}\,\Theta\,[]\,\sigma)\\ &= \{(v,V) \mid \forall \boldsymbol{\tau R} : \Theta, \text{decreasing } \boldsymbol{R} \to v \trianglelefteq_i^{\boldsymbol{\tau R},\sigma} (V[\tau_k/X_k])\}\end{aligned}$$

by simply quantifying over all decreasing relation environments (i.e. those for which every R_k is decreasing) for Θ. The definitions of $\trianglelefteq^{\Theta\vdash\Gamma}$, for environments with open types, and $\trianglelefteq^{\Theta;\Gamma\vdash\sigma}$, for open expressions with open types, follow just the same pattern. We also need to define a version of the order that relates machine computations to open values when the computation immediately constructs a related closed value:

$$\begin{aligned}&\trianglelefteq_i^{\Theta;\Gamma\vdash\sigma} \subseteq \mathit{MComp} \times (\mathit{Value}\,\Theta\,\Gamma\,\sigma)\\ &= \{\,(\mathit{comp}, V) \mid \forall e, \forall \mathit{cesd}, |e| = |\Gamma| \to \exists j, \exists v,\\ &\qquad \mathit{comp} \bowtie (e \smile \mathit{cesd}) \mapsto^j \widehat{v} \bowtie (e \smile \mathit{cesd}) \wedge\\ &\qquad \forall \boldsymbol{\tau R} : \Theta, \text{decreasing } \boldsymbol{R} \to \forall i' \leq i, \forall \boldsymbol{W},\\ &\qquad e \trianglelefteq_{i'}^{\boldsymbol{\tau R},\Gamma} \boldsymbol{W} \to v \trianglelefteq_{i'}^{\boldsymbol{\tau R},\Gamma\vdash\sigma} V[\tau_k/X_k][W_k/x_k]\,\}\end{aligned}$$

Finally, we can define the 'less than or equal' relations between the machine and the source language by quantifying over all step indices and combining with the adequate precongruence on the source language:

$$\begin{aligned}v \preceq^{\Theta\vdash\sigma} V \in \mathit{Value}\,\Theta\,[]\,\sigma &\overset{\text{def}}{\Longleftrightarrow} \exists V', \Theta \vdash V' \sqsubseteq V : \sigma \wedge \forall i, v \trianglelefteq_i^{\Theta\vdash\sigma} V'\\ e \preceq^{\Theta\vdash\Gamma} \boldsymbol{V} \in \mathit{EValue}\,\Theta\,\Gamma &\overset{\text{def}}{\Longleftrightarrow} \exists \boldsymbol{V'}, \Theta \vdash \boldsymbol{V'} \sqsubseteq \boldsymbol{V} : \Gamma \wedge \forall i, e \trianglelefteq_i^{\Theta\vdash\Gamma} \boldsymbol{V'}\\ \mathit{comp} \preceq^{\Theta;\Gamma\vdash\sigma} V \in \mathit{Value}\,\Theta\,\Gamma\,\sigma &\overset{\text{def}}{\Longleftrightarrow} \exists V', \Theta;\Gamma \vdash V' \sqsubseteq V : \sigma \wedge \forall i, \mathit{comp} \trianglelefteq_i^{\Theta;\Gamma\vdash\sigma} V'\\ \mathit{comp} \preceq^{\Theta;\Gamma\vdash\sigma} M \in \mathit{Exp}\,\Theta\,\Gamma\,\sigma &\overset{\text{def}}{\Longleftrightarrow} \exists M', \Theta;\Gamma \vdash M' \sqsubseteq M : \sigma \wedge \forall i, \mathit{comp} \trianglelefteq_i^{\Theta;\Gamma\vdash\sigma} M'\end{aligned}$$

By definition, these relations preserve the given adequate precongruence $\sqsubseteq$ on F_v. For instance, $\mathit{comp} \preceq^{\Theta;\Gamma\vdash\sigma} M,\ M \sqsubseteq M' \ \Rightarrow\ \mathit{comp} \preceq^{\Theta;\Gamma\vdash\sigma} M'$.

5.2 Approximating Low-Level By High-Level

The second logical relation captures what it means for a machine computation to be 'greater than or equal to' an F_v term. We again use biorthogonality on the machine side, but this time with respect to the observation of termination. The property of being less than some fixed high-level term is a safety property of machine computations, for which step indexing is well-suited, whereas being greater is a liveness property: we will be establishing that certain machine configurations terminate. One might reasonably use step-indexing on the high-level language for defining the 'greater than' relation, but the operational semantics of F_v is sufficiently regular that we can avoid working with steps until the end, when we impose a novel operational form of closure by exploiting the family of relations $\precsim_i^\tau$ that we defined earlier.

We again start by defining a relational action for the lift monad by biorthogonality. For $e \in \mathit{MEnv}$, let

$$\begin{aligned} \underline{\downarrow}^{e}(\cdot) &\;:\; \mathbb{P}(\mathit{MValue}) \to \mathbb{P}(\mathit{CESD}) \\ \underline{\downarrow}^{e}(P) &= \{\mathit{cesd} \mid \forall v \in P, (\widehat{v} \bowtie (e \smile \mathit{cesd})) \mapsto^{*} \lightning\} \\ \overline{\Uparrow}^{e}(\cdot) &\;:\; \mathbb{P}(\mathit{CESD}) \to \mathbb{P}(\mathit{MComp}) \\ \overline{\Uparrow}^{e}(Q) &= \{\mathit{comp} \mid \forall \mathit{cesd} \in Q, (\mathit{comp} \bowtie (e \smile \mathit{cesd})) \mapsto^{*} \lightning\} \end{aligned}$$

and now, for a closed type τ and a relation $\mathit{RT} \subseteq \mathit{MValue} \times (\mathit{CValue}\,\tau)$, define

$$\begin{aligned} &[\mathit{RT}^{e}_{\perp}] \subseteq \mathit{MComp} \times (\mathit{CExp}\,\tau) \\ &\;= \{(\mathit{comp}, M) \mid \forall V, M \models\!\!\!\Rightarrow V \to \mathit{comp} \in \overline{\Uparrow}^{e}(\underline{\downarrow}^{e}(\{v \mid v\,\mathit{RT}\,V\}))\} \end{aligned}$$

So, given a low-high relation RT on values, we relate a low computation to a high one if whenever the high computation yields a value V , the low computation terminates in all contexts that terminate whenever they are fed values v that are related to V by RT.

For a closed type τ, let $\mathit{Rel}_{\tau} = \mathit{MValue} \times (\mathit{CValue}\ \tau)$. For a type variable environment $\Theta = X_1, \ldots, X_m$, a relation environment for Θ is now just a vector $\boldsymbol{\tau R}$ of pairs of closed types and relations, where $R_k \subseteq \mathit{Rel}_{\tau_k}$ – note that there are no step-indices in this case. For each $\boldsymbol{\tau R} : \Theta$ and $\Theta \vdash \sigma$, we now define the relation

$$\trianglerighteq^{\boldsymbol{\tau R},\sigma} \subseteq \mathit{MValue} \times (\mathit{CValue}\,(\sigma[\tau_k/X_k]))$$

by induction on σ:

$$\begin{aligned} \trianglerighteq^{\boldsymbol{\tau R},X_k} &= R_k \\ \trianglerighteq^{\boldsymbol{\tau R},\mathtt{Int}} &= \{(\underline{n}, n) \mid n \in \mathbb{N}\} \\ \trianglerighteq^{\boldsymbol{\tau R},\mathtt{Bool}} &= \{(\underline{0}, \mathit{false})\} \cup \{(\underline{n+1}, \mathit{true}) \mid n \in \mathbb{N}\} \\ \trianglerighteq^{\boldsymbol{\tau R},\tau_1 \times \tau_2} &= \{(\mathtt{PR}\,(v_1, v_2), (V_1, V_2)) \mid (v_1, V_1) \in \trianglerighteq^{\boldsymbol{\tau R},\tau_1} \wedge (v_2, V_2) \in \trianglerighteq^{\boldsymbol{\tau R},\tau_2}\} \\ \trianglerighteq^{\boldsymbol{\tau R},\sigma_1 \to \sigma_2} &= \{(f, F) \mid \forall (v, V) \in \trianglerighteq^{\boldsymbol{\tau R},\sigma_1}, (([\mathtt{App}], [v, f]),\, (F\,V)) \in \left[(\trianglerighteq^{\boldsymbol{\tau R},\sigma_2})^{[]}_{\perp}\right]\} \\ \trianglerighteq^{\boldsymbol{\tau R},\forall X.\sigma} &= \{(v, V) \mid \forall \tau', \forall R' \in \mathit{Rel}_{\tau'}, (\widehat{v}, (V\,\tau')) \in \left[(\trianglerighteq^{\boldsymbol{\tau R},\tau' R',\sigma})^{[]}_{\perp}\right]\} \\ \trianglerighteq^{\boldsymbol{\tau R},\exists X.\sigma} &= \{(v, \mathrm{Pack}\{\tau', V\}) \mid \exists R' \in \mathit{Rel}_{\tau'}, (v, V) \in \trianglerighteq^{\boldsymbol{\tau R},\tau' R',\sigma}\} \end{aligned}$$

This relation also lifts pointwise to environments. For $\mathit{\Gamma} = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ and $\Theta \vdash \mathit{\Gamma}$ and $\boldsymbol{\tau R} : \Theta$, define $\trianglerighteq^{\boldsymbol{\tau R},\mathit{\Gamma}} \subseteq \mathit{MEnv} \times (\mathit{EValue}\,[]\,(\mathit{\Gamma}[\tau_k/X_k]))$ by

$$\trianglerighteq^{\boldsymbol{\tau R},\mathit{\Gamma}} = \{([v_1, \ldots, v_n], (V_1, \ldots, V_n)) \mid \forall j, (v_j, V_j) \in \trianglerighteq^{\boldsymbol{\tau R},\sigma_j}\}$$

and then for computations in context,

$$\begin{aligned} &\trianglerighteq^{\boldsymbol{\tau R},\mathit{\Gamma} \vdash \sigma} \subseteq \mathit{MComp} \times (\mathit{Exp}\,[]\,(\mathit{\Gamma}[\tau_k/X_k])\,(\sigma[\tau_k/X_k])) \\ &\;= \{(\mathit{comp}, M) \mid \forall (e, \boldsymbol{V}) \in \trianglerighteq^{\boldsymbol{\tau R},\mathit{\Gamma}}, (\mathit{comp}, M[V_j/x_j]) \in \left[(\trianglerighteq^{\boldsymbol{\tau R},\sigma})^{e}_{\perp}\right]\} \end{aligned}$$

For open types, we then define $\trianglerighteq^{\Theta \vdash \sigma}$ by universally quantifying over all relation environments $\boldsymbol{\tau R} : \Theta$, just as we did for the $\trianglelefteq_{i}^{\Theta \vdash \sigma}$. There are similar definitions of $\trianglerighteq^{\Theta \vdash \mathit{\Gamma}}$, for open environments, $\trianglerighteq^{\Theta;\mathit{\Gamma} \vdash \sigma}$, for open expressions, and $\trianglerighteq^{\Theta;\mathit{\Gamma} \vdash \sigma}$ between low-level computations and high-level open values.

Following the same pattern as for the 'less than' relation, we will define the real 'greater than' relation between machine computations and high-level terms by combining with the adequate precongruence parameter $\sqsubseteq$. However, we want the set of all F_v terms that are less than some fixed piece of low-level code to be admissible, i.e. to satisfy some operational analogue of being closed under limits of ω-chains. Without such a property, we will, for example, be unable to establish one direction of compiler correctness in the case of recursive functions. This is where we use the $\stackrel{\sqsubset}{\sim}_i^{\tau}$ relations that we defined earlier: we use them to explicitly close up the high-level side of the 'greater than' relations under limits. Formally

$$\begin{aligned}&\succeq^{\Theta\vdash\tau}\subseteq \mathit{MValue}\times(\mathit{Value}\ \Theta\ []\ \tau)\\ &\quad=\{(v,V)\mid \exists V',\exists \boldsymbol{W}, \Theta\vdash V\sqsubseteq V':\tau\wedge(\forall i, V'\stackrel{\sqsubset}{\sim}_i^{\Theta\vdash\tau} W_i)\wedge(\forall i, v\trianglerighteq^{\Theta\vdash\tau} W_i)\}\end{aligned}$$

Intuitively, $V\stackrel{\sqsubset}{\sim}_i^{\Theta\vdash\tau} W$ means that W is truly greater than or equal to V, or W is actually less than V but is equal to V up to i steps. So, sequences of values $\{W_i\}_{i\in\mathbb{N}}$ such that $V\stackrel{\sqsubset}{\sim}_i^{\Theta\vdash\tau} W_i$ include some sequences whose components are less than V, but approaching V as the index increases. In particular, for a F_v expression $\Theta; f:\tau\to\tau', x:\tau\vdash M:\tau'$, it holds that

$$\mathrm{Rec}\, f\, x.\, M \stackrel{\sqsubset}{\sim}_i^{\Theta\vdash\tau\to\tau'} \mathrm{Recn}\, i\, f\, x.\, M \quad \text{for all } i \tag{1}$$

where $\mathrm{Recn}\, i\, f\, x.\, M$ denotes the value that unfolds its body i times and then diverges. Note that the property (1) plays a crucial role in the proof of the correctness of our compiler.

Similarly, we define the 'greater than' relation for environments and expressions in context

$$\begin{aligned}
e\succeq^{\Theta\vdash\Gamma}\boldsymbol{V}\in \mathit{EValue}\ \Theta\ \Gamma &\overset{\text{def}}{\iff} \exists\boldsymbol{V'},\exists\overrightarrow{\boldsymbol{W}},\Theta\vdash\boldsymbol{V}\sqsubseteq\boldsymbol{V'}:\Gamma\ \wedge\\
&\qquad(\forall i,\boldsymbol{V'}\stackrel{\sqsubset}{\sim}_i^{\Theta\vdash\Gamma}(\boldsymbol{W})_i)\wedge(\forall i, e\trianglerighteq^{\Theta\vdash\Gamma}(\boldsymbol{W})_i)\\
\mathit{comp}\succeq^{\Theta;\Gamma\vdash\sigma} V\in \mathit{Value}\ \Theta\ \Gamma\ \sigma &\overset{\text{def}}{\iff} \exists V',\exists\boldsymbol{W},\Theta;\Gamma\vdash V\sqsubseteq V':\sigma\ \wedge\\
&\qquad(\forall i, V'\stackrel{\sqsubset}{\sim}_i^{\Theta;\Gamma\vdash\sigma} W_i)\wedge(\forall i,\mathit{comp}\trianglerighteq^{\Theta;\Gamma\vdash\sigma} W_i)\\
\mathit{comp}\succeq^{\Theta;\Gamma\vdash\sigma} M\in \mathit{Exp}\ \Theta\ \Gamma\ \sigma &\overset{\text{def}}{\iff} \exists M',\exists\boldsymbol{N},\Theta;\Gamma\vdash M\sqsubseteq M':\sigma\ \wedge\\
&\qquad(\forall i, M'\stackrel{\sqsubset}{\sim}_i^{\Theta;\Gamma\vdash\sigma} N_i)\wedge(\forall i,\mathit{comp}\trianglerighteq^{\Theta;\Gamma\vdash\sigma} N_i)
\end{aligned}$$

These relations preserve the adequate precongruence $\sqsubseteq$ on F_v. For instance, $\mathit{comp}\succeq^{\Theta;\Gamma\vdash\sigma} M, M'\sqsubseteq M \ \Rightarrow\ \mathit{comp}\succeq^{\Theta;\Gamma\vdash\sigma} M'$.

5.3 Realizability and Equivalence

Having defined the two logical relations, we can put them together to define relations expressing that a machine value, environment or computation realizes a F_v term:

$$\begin{aligned}
v\models^{\Theta\vdash\sigma} V &\overset{\text{def}}{\iff} v\preceq^{\Theta\vdash\sigma} V\ \wedge\ v\succeq^{\Theta\vdash\sigma} V\\
e\models^{\Theta\vdash\Gamma}\boldsymbol{V} &\overset{\text{def}}{\iff} e\preceq^{\Theta\vdash\Gamma}\boldsymbol{V}\ \wedge\ e\succeq^{\Theta\vdash\Gamma}\boldsymbol{V}\\
\mathit{comp}\models^{\Theta;\Gamma\vdash\sigma} V &\overset{\text{def}}{\iff} \mathit{comp}\preceq^{\Theta;\Gamma\vdash\sigma} V\ \wedge\ \mathit{comp}\succeq^{\Theta;\Gamma\vdash\sigma} V\\
\mathit{comp}\models^{\Theta;\Gamma\vdash\sigma} M &\overset{\text{def}}{\iff} \mathit{comp}\preceq^{\Theta;\Gamma\vdash\sigma} M\ \wedge\ \mathit{comp}\succeq^{\Theta;\Gamma\vdash\sigma} M
\end{aligned}$$

One can see from the definitions that $\preceq$ is closed under composition with $\sqsubseteq$ on the right, and that $\succeq$ is closed under composition with $\sqsupseteq$. Together, these imply that our realizability relations respect the chosen notion of equivalence for F_v:

Lemma 1.

1. *If* $v \models^{\Theta \vdash \sigma} V$, $V == V'$ *then* $v \models^{\Theta \vdash \sigma} V'$.
2. *If* $e \models^{\Theta \vdash \Gamma} \boldsymbol{V}$, $\boldsymbol{V} == \boldsymbol{V'}$ *then* $e \models^{\Theta \vdash \Gamma} \boldsymbol{V'}$.
3. *If* $comp \models^{\Theta;\Gamma \vdash \sigma} V$, $V == V'$ *then* $comp \models^{\Theta;\Gamma \vdash \sigma} V'$.
4. *If* $comp \models^{\Theta;\Gamma \vdash \sigma} M$, $M == M'$ *then* $comp \models^{\Theta;\Gamma \vdash \sigma} M'$.

where $X == X'$ *denotes* $X \sqsubseteq X' \wedge X' \sqsubseteq X$.

These relations naturally induce typed relations between machine components. We just give the case for computations

$$\begin{array}{l} \Theta; \Gamma \vdash comp_1 \mathrel{\dot{\sim}} comp_2 : \sigma \\ \overset{\text{def}}{\iff} \exists M \in Exp\, \Theta\, \Gamma\, \sigma, comp_1 \models^{\Theta;\Gamma \vdash \sigma} M \;\wedge comp_2 \models^{\Theta;\Gamma \vdash \sigma} M \end{array}$$

We conjecture that the $\dot{\sim}$ relations induced by taking the observational preorder of F_v for the relation $\sqsubseteq$ are transitive,[3] but in any case, we can apply an explicit transitive closure operation to obtain a low-level notion of typed equivalence $\sim$:

$$\Theta; \Gamma \vdash comp_1 \sim comp_2 : \sigma \overset{\text{def}}{\iff} \Theta; \Gamma \vdash comp_1 \mathrel{\dot{\sim}^{+}} comp_2 : \sigma$$

The characterization we have given of what it means for a piece of SECD machine code to be a valid compilation of a source language expression makes surprisingly little reference to actual low-level code and values. We specify the encoding for base types and pairs and otherwise the only real piece of code that shows up is the application instruction in the definition of the relations at function types. That calling convention is the interface across which linked components communicate. Furthermore, the specifications all refer, ultimately, to whether particular larger configurations resulting from linking the object of the specification into a larger context terminate or diverge: so long as that observable behaviour is preserved, the code can compute in any way it likes to get there.

If $comp \in MComp$, we say $comp$ diverges unconditionally if for any $cesd$, $(comp \bowtie cesd) \mapsto^{\omega}$.

Lemma 2 (Ground divergence adequacy). *For any* $comp \in MComp$, *type* $\tau =$ `Int` *or* `Bool`, *and* $M \in CExp\ \tau$ *if* $comp \models^{[];[] \vdash \tau} M$ *and the* F_v *term* M *diverges, then* $comp$ *diverges unconditionally.*

We say a computation $comp$ converges to a natural n if plugging it into an arbitrary context equiterminates with plugging n into that context:

$$\begin{array}{c} \forall cesd,\ \widehat{\underline{n}} \bowtie cesd \mapsto^{*} \lightning \implies comp \bowtie cesd \mapsto^{*} \lightning \\ \wedge\ \widehat{\underline{n}} \bowtie cesd \mapsto^{\omega} \implies comp \bowtie cesd \mapsto^{\omega}\ . \end{array}$$

And we then show that if a computation realizes a closed F_v term that evaluates to an integer n, then it converges to n:

[3] The reason transitivity seemed likely to fail for our previous denotational-based realizability relations came down to the lack of full abstraction of the domain-theoretic semantics, which is not an issue with the operationally based relations.

Lemma 3 (Ground convergence adequacy). *For any comp* $\in$ *MComp*, $M \in$ *CExp* `Int` *and* $n \in \mathbb{N}$, *if comp* $\models^{[];[]\vdash \texttt{Int}} M$ *and* $M \models\!\!\!\Rightarrow n$ *then comp converges to* n.

Convergence adequacy also holds for observation at the boolean type, with a definition of convergence to a value b that quantifies over those test contexts *cesd* that terminate or diverge uniformly for all machine values representing b.

What allows our realizability semantics to be used for modular reasoning about what happens when one links code obtained from different places and proved by different means, is that it is compositional. An important case is that of function application:

Lemma 4 (Compositionality for application). *For any* $cf, cx \in$ *Code and* $V_f \in$ *Value* $\Theta\, \Gamma\, (\tau \to \tau')$, $V_x \in$ *Value* $\Theta\, \Gamma\, \tau$,

$$\widehat{cf} \models^{\Theta;\Gamma\vdash\tau\to\tau'} V_f \quad\wedge\quad \widehat{cx} \models^{\Theta;\Gamma\vdash\tau} V_x \quad\Longrightarrow\quad cf +\!\!+ \widehat{cx +\!\!+} [\texttt{App}] \models^{\Theta;\Gamma\vdash\tau'} V_f\, V_x$$

6 Applications

6.1 Compiler Correctness

Our motivating application was establishing the functional correctness of the compiler that we presented earlier.

Theorem 1 (Compiled code approximates source).

1. *For all* Θ, Γ, V, τ, *if* $\Theta; \Gamma \vdash V : \tau$ *then* $(\!|V|\!) \preceq^{\Theta;\Gamma\vdash\tau} V$.
2. *For all* Θ, Γ, M, τ, *if* $\Theta; \Gamma \vdash M : \tau$ *then* $(\!|M|\!)_{\mathsf{false}} \preceq^{\Theta;\Gamma\vdash\tau} M$.

As compilations with the *ret* flag on and off are mutually dependent, it is hard to prove the above theorem directly. So we define a new relation $\preceq^{\Theta;\Gamma\vdash\tau}_{\texttt{ret}}$ to state the relationship between $(\!|M|\!)_{\mathsf{true}}$ and M, and prove the above theorem extended with $(\!|M|\!)_{\mathsf{true}} \preceq^{\Theta;\Gamma\vdash\tau}_{\texttt{ret}} M$. The three parts are proved simultaneously by structural induction on V and M. In the case for recursive functions, there is a nested induction over the step indices.

Observe that, although the proof of the compiler has to account for the tail-call optimizations we perform, the notion of tail-call doesn't appear anywhere in the definition of the realiability relation itself.

Theorem 2 (Source approximates compiled code).

1. *For all* Θ, Γ, V, τ, *if* $\Theta; \Gamma \vdash V : \tau$ *then* $(\!|V|\!) \succeq^{\Theta;\Gamma\vdash\tau} V$.
2. *For all* Θ, Γ, M, τ, *if* $\Theta; \Gamma \vdash M : \tau$ *then* $(\!|M|\!)_{\mathsf{false}} \succeq^{\Theta;\Gamma\vdash\tau} M$.

This proof is also by simultaneous structural induction, with a strengthened hypothesis that relates $(\!|M|\!)_{\mathsf{true}}$ and M. This time, the proof for recursive functions involves showing that $\mathrm{Recn}\, i\, f\, x.\, M$ for all i are in the relation and then concluding that $\mathrm{Rec}\, f\, x.\, M$ is in the relation by the property (1).

Corollary 1 (Compiled code realizes source).

1. *For all* Θ, Γ, V, τ*, if* $\Theta; \Gamma \vdash V : \tau$ *then* $(\!|V|\!) \models^{\Theta;\Gamma\vdash\tau} V$.
2. *For all* Θ, Γ, M, τ*, if* $\Theta; \Gamma \vdash M : \tau$ *then* $(\!|M|\!)_{\mathsf{false}} \models^{\Theta;\Gamma\vdash\tau} M$.

So the compiled code of a well-typed term always realizes that term. A consequence is that compiled code for whole programs has the correct operational behaviour according to the operational semantics of the programs:

Corollary 2 (Correctness for whole programs). *For any* M *with* $[]; [] \vdash M : \mathtt{Int}$*,*

– *If* M *diverges, then* $(\!|M|\!)_{\mathsf{false}}$ *diverges unconditionally.*
– *If* $M \models\!\!\!> n$*, then* $(\!|M|\!)_{\mathsf{false}}$ *converges to* n*.*

which follows by the adequacy lemmas above. It is this last corollary that is normally thought of as compiler correctness, but it is Corollary 1 that is really interesting, as it is that which allows us to reason about the combination of compiled code with code from elsewhere.

6.2 Hand-written low-level code

In this section we give interesting examples of hand-written low-level code. To increase readability, we write high-level terms of F_v in non-ANF form.

Example: Fixed-point combinator. One can use the Rec construct to write a CBV fixed-point combinator in F_v:

$$\mathsf{FixC} = \Lambda X.\, \Lambda Y.\, \lambda F : (X \to Y) \to (X \to Y).\, \mathrm{Rec}\; f\, x.\, F\, f\, x$$

which compiles to code using SECD's recursive closures:

$[\mathtt{PushC}\, [\mathtt{PushRC}\, [\mathtt{Push}\, 2, \mathtt{Push}\, 1, \mathtt{App}, \mathtt{PushE}, \mathtt{Push}\, 0, \mathtt{Push}\, 1, \mathtt{AppNoDump}, \mathtt{PopE}], \mathtt{Ret}]]$

But one can also realize the term FixC without recursive closures, by hand-encoding a fixed-point combinator

$$\lambda F.\, \lambda x.\, (\lambda y.\, F(\lambda z.\, y\, y\, z))\, (\lambda y.\, F(\lambda z.\, y\, y\, z))\, x$$

of untyped CBV lambda calculus in SECD:

$\mathsf{YCombinator} =$
$[\mathtt{PushC}\, [\mathtt{PushC}\, [\mathtt{PushC}\, [\mathtt{Push}\, 2, \mathtt{PushC}\, [\mathtt{Push}\, 1, \mathtt{Push}\, 1, \mathtt{App}, \mathtt{Push}\, 0, \mathtt{App}, \mathtt{Ret}], \mathtt{App}, \mathtt{Ret}],$
$\mathtt{PushC}\, [\mathtt{Push}\, 2, \mathtt{PushC}\, [\mathtt{Push}\, 1, \mathtt{Push}\, 1, \mathtt{App}, \mathtt{Push}\, 0, \mathtt{App}, \mathtt{Ret}], \mathtt{App}, \mathtt{Ret}],$
$\mathtt{App}, \mathtt{Push}\, 0, \mathtt{App}, \mathtt{Ret}], \mathtt{Ret}]]$

One can then prove, by direct reasoning about the operational semantics of low-level code and of F_v, but *not* involving anything to do with the compiler, the following:

Lemma 5.

$$\widehat{\mathsf{YCombinator}} \models^{\vdash \forall X.\, \forall Y.\, ((X\to Y)\to X\to Y)\to X\to Y} \mathsf{FixC}.$$

Taken together with the compiler correctness result for open programs and the compositionality of the realizability relation, the above implies that linking the handcrafted code with code produced by the compiler for any term with an appropriately typed free variable will be observationally indistinguishable from linking in the compiled code for any source-level program that is == to the explicitly recursive FixC.

Example: Polymorphic list module. We now give an example that involves relational parametricity, for both universal (polymorphic) and existential (abstract) types. Consider the following signature for a polymorphic list module:

$$\texttt{SigPolList} = \forall X.\, \exists LX.\, LX \times (X \times LX \to LX) \times (LX \to \texttt{Option}\,(X \times LX))$$

where

$$\texttt{Option}\,\tau = \forall Y.\, Y \times (\tau \to Y) \to Y$$

for any type τ and a fresh type variable Y. The signature says that an implementation of polymorphic lists should have some private representing type LX, equipped with three standard list manipulation operations:

$$\begin{aligned} \mathit{nil} &: LX \\ \mathit{cons} &: X \times LX \to LX \\ \mathit{split} &: LX \to \texttt{Option}\,(X \times LX) \end{aligned}$$

where the option type is the usual Church-encoding, with operations:

$$\begin{aligned} \mathrm{none}_\tau : \texttt{Option}\,\tau &= \Lambda X.\, \lambda(n,s).\, n \\ \mathrm{some}_\tau\,(v:\tau) : \texttt{Option}\,\tau &= \Lambda X.\, \lambda(n,s).\, s\,v \end{aligned}$$

Now one concrete implementation of the signature is given by the well-known Church-encoding of lists:

$$\mathsf{PList} : \texttt{SigPolList} = \Lambda X.\, \mathrm{Pack}\{\texttt{List}\,X, (\mathrm{nil}_X, \lambda(hd,tl).\, \mathrm{cons}_X\, hd\, tl, \lambda\, l.\, \mathrm{split}_X\, l)\}$$

where

$$\texttt{List}\,\tau \;=\; \forall Y.\, Y \times (\tau \times Y \to Y) \to Y$$

for any type τ and a fresh type variable Y, and where

$$\begin{aligned} &\mathrm{nil}_\tau : \texttt{List}\,\tau \;=\; \Lambda Y.\, \lambda(n,c).\, n \\ &\mathrm{cons}_\tau\,(hd:\tau)\,(tl:\texttt{List}\,\tau) : \texttt{List}\,\tau \;=\; \Lambda Y.\, \lambda(n,c).\, c\,(hd, tl\,Y\,(n,c)) \\ &\mathrm{split}_\tau\,(l:\texttt{List}\,\tau) : \texttt{Option}\,(\tau \times \texttt{List}\,\tau) \;= \\ &\quad l\,(\texttt{Option}\,(\tau \times \texttt{List}\,\tau))\,(\mathrm{none}_{\tau \times \texttt{List}\,\tau}, \\ &\qquad \lambda(hd, htl).\, htl\,(\texttt{Option}\,(\tau \times \texttt{List}\,\tau))\,(\mathrm{some}_{\tau \times \texttt{List}\,\tau}\,(hd, \mathrm{nil}_\tau), \\ &\qquad\quad \lambda(hd', tl).\, \mathrm{some}_{\tau \times \texttt{List}\,\tau}\,(hd, \mathrm{cons}_\tau\, hd'\, tl))) \end{aligned}$$

But this implementation is rather inefficient. For instance, the list splitting operation takes $\mathcal{O}(n)$ time for lists of length n, rather than constant time. But, lacking recursive types, this is the best we can do in the F_v source language.

Using our realizability relations, we can treat the inefficient encoding of lists as a specification, and instead write some cunning, hand-optimized SECD machine code that provably realizes, i.e. behaves exactly the same as from the point of view of well-behaved clients, PList. There are two tricks in our implementation. First, we represent lists as nested tuples of elements, which is something we could not type in F_v. We then further optimize by playing a highly non-parametric low-level representation trick: when the representation of list elements is (dynamically!) observed to be a natural number, we further compress the representation by an encoding of sequences of natural numbers as a single natural number. The implementation of PList in SECD is parameterized by pairing and projection functions on natural numbers:

$$\mathsf{npair} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}, \quad \mathsf{nfst} : \mathbb{N} \to \mathbb{N}, \quad \mathsf{nsnd} : \mathbb{N} \to \mathbb{N}$$

such that, forall $n, m \in \mathbb{N}$,

$$\mathsf{npair}\, n\, m > 0, \quad \mathsf{nfst}\,(\mathsf{npair}\, n\, m) = n, \quad \mathsf{nsnd}\,(\mathsf{npair}\, n\, m) = m;$$

and an implementation of the functions in SECD, $\mathsf{NPair}, \mathsf{NSplit} \in \mathit{Code}$ such that

$$\forall \mathit{cesd}\, n\, m\, \exists k\, (\mathsf{NPair}, [\underline{m}, \underline{n}]) \bowtie \mathit{cesd} \mapsto^k ([], [\underline{\mathsf{npair}\, n\, m}]) \bowtie \mathit{cesd}$$
$$\forall \mathit{cesd}\, n\, \exists k\, (\mathsf{NSplit}, [\underline{n}]) \bowtie \mathit{cesd} \mapsto^k ([], [\mathtt{PR}\,(\underline{\mathsf{nfst}\, n}, \underline{\mathsf{nsnd}\, n})]) \bowtie \mathit{cesd}$$

The function $\lambda(n, m).2^n \times 3^m$ meets this spec, for example.

Exploiting the instruction `IsNum` that checks if a given machine value is a number or not, one can compactly represent a list of machine values in SECD as follows:

$$\mathsf{encodeM}\,(vs : \mathit{list}\, \mathit{MValue}) : \mathit{MValue}$$
$$= \begin{cases} \underline{0} & \text{if } vs = [] \\ \underline{\mathsf{npair}\, n\, m} & \text{if } vs = \underline{n} :: tl \wedge \mathsf{encodeM}\, tl = \underline{m} \\ \mathtt{PR}\,(hd, tl) & \text{otherwise } (vs = hd :: tl) \end{cases}$$

The following are implementations of the three functions of `SigPolList` in SECD according to the above representation.

$\mathsf{NilM} = [\mathtt{PushN}\, 0]$

```
ConsM = [PushC [Push 0, Fst, IsNum,
                Sel ([Push 0, Snd, IsNum, Join], [Push 0, Snd, PushN 0, Join]),
                Sel (NPair ++ [Join], [MkPair, Join]), Ret]]
```

```
SplitM = [PushC [
  Push 0,
  Sel ([PushC [Push 0, Snd, Push 1, IsNum, Sel (NSplit ++ [Join], [Join]), App, Ret],
        Join],
       [PushC [Push 0, Fst, Ret], Join]),
  Ret]]
```

To help understanding, we give pseudo lambda terms interpreting the above code:

$\mathsf{NilM} \approx \underline{0}$

$\mathsf{ConsM} \approx \lambda(hd, tl).\, \text{if } hd = \underline{n} \wedge tl = \underline{m} \text{ then } \underline{\mathsf{npair}\, n\, m} \text{ else } \mathtt{PR}\,(hd, tl)$

$\mathsf{SplitM} \approx \lambda\, l.\, \text{if } l \neq \underline{0} \text{ then } \lambda(n, s).\, \text{if } l = \underline{n} \text{ then } s\,(\mathsf{nfst}\, \underline{n}, \mathsf{nsnd}\, \underline{n}) \text{ else } s\, l$
$\qquad\qquad\qquad\qquad \text{else } \lambda(n, s).\, n$

We can formally prove that this optimized low-level implementation realizes, and is therefore substitutable for (in any related context), the module PList:

Lemma 6.

$$\mathsf{NilM} +\!\!+ \mathsf{ConsM} +\!\!+ \widehat{[\mathtt{MkPair}]} +\!\!+ \mathsf{SplitM} +\!\!+ [\mathtt{MkPair}] \models^{\vdash \mathtt{SigPolList}} \mathsf{PList}.$$

7 Discussion

We have presented a compositional and extensional operational realizability relation between a parametrically polymorphic source language and a low-level untyped abstract machine. The relation was used to establish both full functional correctness for an optimizing compiler and to justify the linking of code from any compiler meeting the specification with hand-optimized fragments of low-level code. The relations, which are parameterized on a notion of source language equivalence, make crucial use of both biorthogonality and step-indexing, as well as introducing an interesting new form of operationally converging sequence.

Compared with our earlier work [3], the biggest differences are that the source language is now polymorphic, with a relationally parametric interpretation, and we work with an operational semantics for the source language, rather than a denotational one. The compiler now does some optimization and as well as the new $\precsim_i^{\tau}$ relations, the new factorization of our logical introducing by the $\sqsubseteq$ precongruence deserves comment. One way of looking at this is that it allows one to incorporate into the relation any well-behaved (but possibly incomplete with respect to contextual equivalence) system of source-level transformation rules, as might be used in an optimizing compiler, whilst still maintaining compositionality.

We have formalized and verified all the results, including the examples, in the Coq proof assistant. One interesting feature of the formalization, which we unfortunately do not have space to discuss here, is that it is based on a new 'strongly-typed' representation of polymorphic terms and substitutions, using dependent types to define F_v terms and substitutions that are well-typed by construction. The complete Coq script runs to about 14,000 lines. Of these, around 2000 are taken up with the representation of F_v and the various 'bread and butter' lemmas about the different forms of substitution thereon. Proving the examples is a non-trivial task in itself: the verification of the somewhat sophisticated non-parametric implementation of the parametric list module alone is nearly 3000 lines, though that could be immensely shortened with better tactical or reflective support for working with the two operational semantics (one doesn't really need that for metatheoretic results, but for working with examples it's very desirable). The strongly-typed term representation really paid off in practice, and we plan to further streamline it. By comparison, the script associated with our earlier work [3] was around 4,000 lines, though that relied on a separate formalization of domain theory and denotational semantics and was for a simpler language and compiler. Treating impredicative polymorphism denotationally would involve working with PERs on a universal domain or impredicative-Set-based cpos; the operational approach is more elementary likely to scale to other language features (though some 'boring' repeated manipulations in operational proofs are encapsulated once and for all in denotational ones).

Apart from our own recent work, there have been dozens of other compiler correctness proofs done in the last 35 years or so, amongst which we particularly mention the classic work on the VLISP verified Scheme compiler [6], which relates a denotational semantics to, ultimately, real machine code; the Coq formalization of compiler correctness for a total functional language by Chlipala [5]; and the work of Leroy [9] on mechanically verifying a realistic compiler for a C-like language. A distinguishing feature of our work is the focus on compositional specifications that are independent of any particular compiler and can be used to independently verify foreign code. We have also looked at low-level semantic type soundness in a similar style [4].

Other avenues for future work include looking at recursive types and references, and transferring our results to a lower level target machine. We would also like to make our specifications even more independent of the source language, by expressing them in a logic that talks only about the low-level machine.

References

[1] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *15th European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, 2006.

[2] 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.

[3] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ACM International Conference on Functional Programming*, 2009.

[4] N. Benton and N. Tabareau. Compiling functional types to relational specifications for low level imperative code. In *4th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, 2009.

[5] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[6] J. Guttman, J. Ramsdell, and M. Wand. VLISP: A verified implementation of scheme. *Lisp and Symbolic Computation*, 8(1/2), 1995.

[7] J. L. Krivine. Classical logic, storage operators and second-order lambda calculus. *Annals of Pure and Applied Logic*, 1994.

[8] P. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4), 1964.

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

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

[11] J. Vouillon and P.-A. Melliès. Semantic types: A fresh look at the ideal model for types. In *31st ACM Symposium on Principles of Programming Languages (POPL)*, 2004.