Realizability and Compositional Compiler Correctness for a Polymorphic Language

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

[1] Microsoft Research [nick@microsoft.com]
[2] PPS, Université Paris Diderot [gil@pps.jussieu.fr]

Abstract. We construct operationally-based realizability relations between phrases in a language with both universal and existential types and programs for a variant SECD machine. The relations, defined using parametricity, biorthogonality and step-indexing, give extensional and compositional specifications of when low-level code and values realize typed source-level terms. We prove full functional correctness of a compiler in terms of these relations and show how they also justify both source-level transformations and the linking of compiled code with hand-optimized code fragments that exploit non-parametric and non-functional low-level operations whilst being extensionally well-behaved. The definitions and results have been fully formalized in Coq.

1 Introduction

As the title suggests, this paper will describe a (mechanized) proof that a particular compiler is correct. The real subject, however, is how to define a good specification of when a low-level code fragment should be said to 'correspond to' a phrase in a high-level language.

A straightforward compiler correctness theorem says that for every closed, ground type source program P, the result $C(P)$ of running compiler C on P is a target program whose observable behaviour (termination, final result, IO behaviour) 'matches' that of C. Proving such a theorem involves a strengthened induction hypothesis, relating *open* source phrases of higher types to target code (and values). This richer relation, typically some kind of (bi)simulation, is often essentially the simplest extension of the function C itself that suffices to establish the 'big' theorem about complete programs, which are, after all, the only ones we can run according to the semantics of our source language. But the 'closed' systems we really run are not the result of a single compilation: they are composed by linking code from many places, including libraries, the operating system, the runtime system and foreign functions, which may be compiled with different compilers and written in many languages, including 'cleverly' handcrafted machine code. To reason modularly about all these components, we need a clean specification of the interface between compiled code and its environment, a job for which a naive induction hypothesis is inappropriate. The kind of specification we want should constrain only the observable behaviour of code, rather than intensional details of just how it executes, and make no reference to details of a particular compiler beyond those aspects of data representation and calling conventions that *have* to be agreed upon for interoperability. Similarly, the correctness relation should not be tweaked to admit individual source-level optimizations, but should rather be closed under a rich set of high-level equations (possibly even source contextual equivalence) by construction.

In a previous paper, we addressed the question of when low-level code correctly realizes a source term by defining relations between the denotational semantics of a simply-typed source language and the operational behaviour of a lower-level virtual machine [3]. Here we take another step forward, treating a source language with impredicative universal and existential types and defining relations that express how parametricity and data abstraction principles from the typed source translate to an untyped target. 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 performs tail-call optimizations. Another major difference from our previous work is that we work with an operational, rather than a denotational, semantics for the high-level language. Rather than fix a high-level notion of equivalence that should be respected, we parameterize our definitions and results by a novel form of adequate precongruence relation on the source, incorporating an abstract notion of chain to capture analogues of domain-theoretic admissibility.

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 High- and Low-Level Languages

High-Level Language. F_v is a conventional call-by-value functional language with recursion and impredicative universal and existential types. The types are:

$$\tau := X \mid \mathtt{Int} \mid \tau \to \tau' \mid 1 \mid \tau \times \tau' \mid \tau + \tau' \mid \forall X.\tau \mid \exists X.\tau$$

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 $[\cdot]$. Selected typing rules for F_v are shown in Figure 1, where Θ and Γ are contexts for type and term variables respectively. Write $\mathit{Value}\,\Theta\,\Gamma\,\tau$ for the set of values of type τ in contexts $\Theta;\Gamma$ (where $\Theta;\Gamma \vdash \tau$), $\mathit{CValue}\,\tau$ for the closed values of (closed) type τ, and similarly for expressions. If $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ then the set of environments $\mathit{EValue}\,\Theta\,\Gamma$ is $\prod_{i=1\ldots n} \mathit{Value}\,\Theta\,[]\,\tau_i$.

Selected transitions from the standard CBV semantics of F_v are also shown in Figure 1. There is an equivalent big-step semantics and we write $M \mathrel{|}\!\!\Rightarrow$ (resp. $M \mathrel{|}\!\!\Rightarrow V$) when the closed expression M converges (to the closed value V).

Our realizability relations will be parameterized by a notion of observational approximation $\sqsubseteq$ on F_v. To cope abstractly with recursion, we take as basic

Values:

$$\frac{\Theta; \Gamma, f : \tau \to \tau', x : \tau \vdash M : \tau'}{\Theta; \Gamma \vdash \texttt{rec}\ f\,(x : \tau) : \tau' = M : \tau \to \tau'} \qquad \frac{\Theta \vdash \Gamma \quad \Theta, X \vdash \tau \quad \Theta, X; \Gamma \vdash V : \tau}{\Theta; \Gamma \vdash \Lambda X.V : \forall X.\tau}$$

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

Expressions:

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

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

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

Transition semantics:

$$\texttt{let}\ x = [V]\ \texttt{in}\ N \ \mapsto\ N[V/x] \qquad (\Lambda X.V)\,\tau \ \mapsto\ V[\tau/X]$$

$$(\texttt{rec}\ f\,(x : \tau) : \tau' = M)\,V \mapsto M[(\texttt{rec}\ f\,(x : \tau) : \tau' = M)/f\ ,\ V/x]$$

$$\texttt{unpack}\,(\texttt{pack}\,\tau, V\,\texttt{to}\,\exists X.\tau')\,\texttt{as}\,X, x\,\texttt{in}\,M \mapsto M[\tau/X][V/x]$$

Fig. 1. Selected typing and transition rules for F_v

$\hat{\sqsubseteq}_{\Theta\Gamma\tau} \subseteq (\mathit{Value}\ \Theta\,\Gamma\,\tau) \times (\mathit{Value}\ \Theta\,\Gamma\,\tau)^\omega$, a slightly unusual (type- & context-indexed, but we usually omit indices) relation between values and *sequences* of values, and derive the associated order on values via constant sequences: $V \sqsubseteq V'$ iff $V \hat{\sqsubseteq} (\lambda i \in \omega.V')$. There are homonymous orders on expressions and environments. A sequence $\langle V_i \rangle_i$ is a *chain* if $V_i \sqsubseteq V_j$ for all $i \leq j$. One should think of $V \hat{\sqsubseteq} \langle W_i \rangle_i$ meaning $V \sqsubseteq \sqcup_i W_i$ in a domain-theoretic sense, but without requiring lubs to exist.

The conditions on $\hat{\sqsubseteq}$ and $\sqsubseteq$, eliding types and with the same conditions applied, *mutatis mutandis*, to the order on expressions, are: *[Chain]*: if $V \hat{\sqsubseteq} \langle W_i \rangle_i$ then $\langle W_i \rangle_i$ is a chain; *[Elem]*: if $\langle W_i \rangle_i$ is a chain, $W_j \hat{\sqsubseteq} \langle W_i \rangle_i$ for all j; *[Refl]*: $V \sqsubseteq V$ for all V of the appropriate type; *[Trans]*: if $U \hat{\sqsubseteq} \langle V_i \rangle_i$ and $V_j \hat{\sqsubseteq} \langle W_i \rangle_i$ for all j, then $U \hat{\sqsubseteq} \langle W_i \rangle_i$; *[Subst]*: if $V \hat{\sqsubseteq} \langle W_i \rangle_i$ then $V[U/x] \hat{\sqsubseteq} \langle W_i[U/x] \rangle_i$ and similarly for type substitutions; *[Compat]*: all constructs of F_v preserve $\hat{\sqsubseteq}$, e.g.

$$\frac{\Theta; \Gamma \vdash V \hat{\sqsubseteq} \langle W_i \rangle_i : \tau[\tau'/X]}{\Theta; \Gamma \vdash \texttt{pack}\,\tau', V\,\texttt{to}\,\exists X.\tau \hat{\sqsubseteq} \langle \texttt{pack}\,\tau', W_i\,\texttt{to}\,\exists X.\tau \rangle_i : \exists X.\tau}$$

[Beta]: $\texttt{let}\ x = [V]\ \texttt{in}\ N \sqsubseteq N[V/x]$ and vice versa; *[Adeq]*: If $M \hat{\sqsubseteq} \langle N_i \rangle_i : \texttt{Int}$ and $M \mathrel{|\!\!\Rightarrow} n$ for some n, then there exists j such that $N_j \mathrel{|\!\!\Rightarrow} n$, and similarly for the unit type; *[Unfold]*: $\texttt{rec}\ f\,x.\,M \hat{\sqsubseteq} \langle \texttt{recn}_i\ f\,x.\,M \rangle_i$ where $\texttt{recn}_0\ f\,x.\,M = \texttt{rec}\ f\,x.\,f\,x$ and $\texttt{recn}_{i+1}\ f\,x.\,M = \lambda x.M[\texttt{recn}_i\ f\,x.\,M/f]$. These conditions imply that $\sqsubseteq$ is an adequate precongruence satisfying an 'unwinding theorem' [12]. An important example is given by defining $V \hat{\sqsubseteq} \langle W_i \rangle_i$ iff $\forall C[\cdot], C[V] \mathrel{|\!\!\Rightarrow} \implies \exists j, \forall i \geq j, C[W_i] \mathrel{|\!\!\Rightarrow}$; another is generated by a (non fully-abstract) step-indexed logical relation like that of Ahmed [1].The extension of $\hat{\sqsubseteq}$ to environments, which are typed lists of closed values, is pointwise.

Low-Level Machine. Our target is a variant SECD machine [9]. Although the SECD machine was designed as a target for compiling functional languages, the kind of relations we construct will work for lower-level targets too. The

$$\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' {+\!\!+} 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. 3. Operational Semantics of Extended SECD Machine

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 new, non-functional, operations to the original SECD machine to express more interesting and realistic low-level optimizations.

$$\begin{array}{l}
\mathit{CESD} \overset{\text{def}}{=} \mathit{Code} \times \mathit{MEnv} \times \mathit{Stack} \times \mathit{Dump} \\
c \in \mathit{Code} \overset{\text{def}}{=} \mathit{list\,Instruction} \\
e \in \mathit{MEnv} \overset{\text{def}}{=} \mathit{list\,MVal} \\
s \in \mathit{Stack} \overset{\text{def}}{=} \mathit{list\,MVal} \\
d \in \mathit{Dump} \overset{\text{def}}{=} \mathit{list}\,(\mathit{Code} \times \mathit{MEnv} \times \mathit{Stack}) \\
\\
\mathit{Instruction} \ni \mathit{inst} := \texttt{Pop} \mid \texttt{Push}\, i \mid \texttt{PushE} \\
\quad \mid \texttt{PopE} \mid \texttt{PushN}\, n \mid \texttt{Op}\star \mid \texttt{PushC}\, c \mid \texttt{PushRC}\, c \\
\quad \mid \texttt{App} \mid \texttt{AppNoDump} \mid \texttt{Ret} \mid \texttt{Sel}\,(c_1, c_2) \\
\quad \mid \texttt{SelNoDump}\,(c_1, c_2) \mid \texttt{Join} \\
\quad \mid \texttt{MkPair} \mid \texttt{Fst} \mid \texttt{Snd} \mid \texttt{Eq} \mid \texttt{IsNum} \\
n, i \in \mathbb{N} \qquad \star \in \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
\mathit{MVal} \ni v := \underline{n} \mid \texttt{CL}\,(e, c) \mid \texttt{RCL}\,(e, c) \mid \texttt{PR}\,(v_1, v_2)
\end{array}$$

Fig. 2. Extended SECD machine

A configuration is a quadruple $\langle c,\ e,\ s,\ d\rangle \in \mathit{CESD}$, as defined in Figure 2. An *MVal* is either a natural, a closure, a recursive closure, or a pair of values. The deterministic transition relation $\mapsto$ between configurations is defined in Figure 3. The non-standard 'no dump' forms of application and selection do not save a continuation on the dump and are used for tail-call optimizations. PushE and PopE allow the environment to be modified. IsNum tests for numberhood, and Eq for intensional equality. Our previous paper [3] explains how non-functional 'reflective' operations, such as

Values:

$$\begin{aligned}
\llparenthesis \Theta; \boldsymbol{x_j} : \boldsymbol{\tau_j} \vdash x_i : \tau_i \rrparenthesis &= [\mathtt{Push}\, i] \\
\llparenthesis () \rrparenthesis &= [\mathtt{PushN}\, 0] \\
\llparenthesis n \rrparenthesis &= [\mathtt{PushN}\, n] \\
\llparenthesis \langle V_1, V_2 \rangle \rrparenthesis &= \llparenthesis V_1 \rrparenthesis +\!\!+ \llparenthesis V_2 \rrparenthesis +\!\!+ [\mathtt{MkPair}] \\
\llparenthesis \mathtt{inl}\, V \rrparenthesis &= [\mathtt{PushN}\, 1] +\!\!+ \llparenthesis V \rrparenthesis +\!\!+ [\mathtt{MkPair}] \\
\llparenthesis \mathtt{inr}\, V \rrparenthesis &= [\mathtt{PushN}\, 0] +\!\!+ \llparenthesis V \rrparenthesis +\!\!+ [\mathtt{MkPair}] \\
\llparenthesis \lambda x.\, M \rrparenthesis &= [\mathtt{PushC}\, (\llparenthesis M \rrparenthesis_{\mathsf{true}})] \\
\llparenthesis \mathtt{rec}\ f\, x\ =\ M \rrparenthesis &= [\mathtt{PushRC}\, (\llparenthesis M \rrparenthesis_{\mathsf{true}})] \\
\llparenthesis \Lambda X.\, V \rrparenthesis &= \llparenthesis V \rrparenthesis \\
\llparenthesis \mathtt{pack}\, \tau', V\ \mathtt{to}\, \exists X.\tau \rrparenthesis &= \llparenthesis V \rrparenthesis
\end{aligned}$$

Expressions:

$$\begin{aligned}
\llparenthesis ret \rrparenthesis &= \text{if}\ \ ret = \mathtt{true}\ \ \text{then}\ \ [\mathtt{Ret}]\ \ \text{else}\ \ [] \\
\llparenthesis [V] \rrparenthesis_{ret} &= \llparenthesis V \rrparenthesis +\!\!+ \llparenthesis ret \rrparenthesis \\
\llparenthesis V_1 \star V_2 \rrparenthesis_{ret} &= \llparenthesis V_1 \rrparenthesis +\!\!+ \llparenthesis V_2 \rrparenthesis +\!\!+ [\mathtt{Op}\, \star] +\!\!+ \llparenthesis ret \rrparenthesis \\
\llparenthesis V_1 > V_2 \rrparenthesis_{ret} &= \llparenthesis V_1 \rrparenthesis +\!\!+ \llparenthesis V_2 \rrparenthesis +\!\!+ [\mathtt{Op}\, (\lambda(n_1, n_2).n_1 > n_2 \supset 1 \mid 0), \mathtt{PushN}\, 0, \mathtt{MkPair}] +\!\!+ \llparenthesis ret \rrparenthesis \\
\llparenthesis \pi_1(V) \rrparenthesis_{ret} &= \llparenthesis V \rrparenthesis +\!\!+ [\mathtt{Fst}] +\!\!+ \llparenthesis ret \rrparenthesis \\
\llparenthesis \pi_2(V) \rrparenthesis_{ret} &= \llparenthesis V \rrparenthesis +\!\!+ [\mathtt{Snd}] +\!\!+ \llparenthesis ret \rrparenthesis
\end{aligned}$$

$$\begin{aligned}
&\llparenthesis \mathtt{case}\, V\ \mathtt{of}\ \mathtt{inl}\, x.M_1 \mid \mathtt{inr}\, y.M_2 \rrparenthesis_{\mathsf{true}} \\
&\quad = \llparenthesis V \rrparenthesis +\!\!+ [\mathtt{Dup}, \mathtt{Snd}, \mathtt{PushE}, \mathtt{Fst}, \mathtt{SelNoDump}\, (\llparenthesis M_1 \rrparenthesis_{\mathsf{true}}, \llparenthesis M_2 \rrparenthesis_{\mathsf{true}}), \mathtt{PopE}] \\
&\llparenthesis \mathtt{case}\, V\ \mathtt{of}\ \mathtt{inl}\, x.M_1 \mid \mathtt{inr}\, y.M_2 \rrparenthesis_{\mathsf{false}} \\
&\quad = \llparenthesis V \rrparenthesis +\!\!+ [\mathtt{Dup}, \mathtt{Snd}, \mathtt{PushE}, \mathtt{Fst}, \mathtt{Sel}\, (\llparenthesis M_1 \rrparenthesis_{\mathsf{false}} +\!\!+ [\mathtt{Join}], \llparenthesis M_2 \rrparenthesis_{\mathsf{false}} +\!\!+ [\mathtt{Join}]), \mathtt{PopE}] \\
&\llparenthesis \mathtt{let}\ x = M\ \mathtt{in}\ N \rrparenthesis_{ret} = \llparenthesis M \rrparenthesis_{\mathsf{false}} +\!\!+ [\mathtt{PushE}] +\!\!+ \llparenthesis N \rrparenthesis_{ret} +\!\!+ [\mathtt{PopE}]
\end{aligned}$$

$$\begin{aligned}
\llparenthesis V_1\ V_2\ : \rrparenthesis_{\mathsf{true}} &= \llparenthesis V_1 \rrparenthesis +\!\!+ \llparenthesis V_2 \rrparenthesis +\!\!+ [\mathtt{AppNoDump}] \\
\llparenthesis V_1\ V_2 \rrparenthesis_{\mathsf{false}} &= \llparenthesis V_1 \rrparenthesis +\!\!+ \llparenthesis V_2 \rrparenthesis +\!\!+ [\mathtt{App}] \\
\llparenthesis V\ \tau' \rrparenthesis_{ret} &= \llparenthesis V \rrparenthesis +\!\!+ \llparenthesis ret \rrparenthesis
\end{aligned}$$

$$\llparenthesis \mathtt{unpack}\, V\ \mathtt{as}\, X, x\ \mathtt{in}\, M \rrparenthesis_{ret} = \llparenthesis V \rrparenthesis +\!\!+ [\mathtt{PushE}] +\!\!+ \llparenthesis M \rrparenthesis_{ret} +\!\!+ [\mathtt{PopE}]$$

Fig. 4. Compiler for F_v

Eq, in the target break 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 terms.

Configurations with no successor are *terminated*. Write $cesd \mapsto^k$ if $cesd$ takes at least k steps without having terminated, and say it diverges, written $cesd \mapsto^\omega$, if it can always take a step. We say $cesd$ terminates, and write $cesd \mapsto^* \lightning$, if it does not diverge.

Compiling $\mathbf{F}_v$ to SECD. The compiler is shown in Figure 4 and comprises mutually-recursive functions, both written $\llparenthesis \cdot \rrparenthesis$, 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.

3 Logical Relations

We will now define two logical relations between components of the SECD machine and terms of F_v. $\preceq$ specifies when a low-level component approximates (in the observational sense of 'diverging in more contexts than') a source term at a particular type, whilst $\succeq$ expresses the converse. These two relations can be seen as corresponding to the traditional soundness and adequacy theorems used to show correspondence between an operational and a denotational semantics.

We start by giving a broad overview of the constructions used in defining the relations. Firstly, both relations are parameterized by $\hat{\sqsubseteq}$, an approximation relation on the source satisfying the conditions we gave in Section 2. $\sqsubseteq$ 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 preserve source-level equivalence. Secondly, the $\preceq$ relation, which is intuitively about specifying that low-level code should diverge in certain contexts, involves step-indexing [2] on the low-level side. Divergence arises as the limit as k increases of 'takes at least k steps without terminating'.

A third important construction is the use of biorthogonality to 'extensionalize' the sets of low-level values that are related to particular F_v terms. We are trying to define compositional specifications for *components* of SECD configurations, in particular instruction sequences $c \in \mathit{Code}$, but we want those specifications to ultimately depend only on the observable behaviour of complete, runnable configurations. This is achieved, building on ideas of Pitts and of Krivine, by making our specifications '$\top\top$-closed' [12, 13]. The rough idea here is that one starts with an over-intensional set of computations, constructs the set of all contexts that yield some particular observation when linked with any element of the initial set, and then constructs the set – larger than that with which one started – of those computations that yield the observation when combined with any of those contexts. In the case of $\preceq$, the observation will be divergence (actually, stepping for at least some number of steps), whilst for $\succeq$ the observation will be termination.

We will clearly want to relate source terms both to constructed machine values and to instruction sequences, but it is convenient to work with pairs $(c, s) \in \mathit{MComp} \stackrel{\mathrm{def}}{=} \mathit{Code} \times \mathit{Stack}$ instead of isolated bits of code. If $c \in \mathit{Code}$, write $\widehat{c} \in \mathit{MComp}$ for $(c, [])$, and if $v \in \mathit{MVal}$, write $\widehat{v} \in \mathit{MComp}$ for $([], [v])$. We will use concatenation to link elements of MComp with contexts, which are themsleves elements of CESD.

Approximating High-level By Low-level. The $\preceq$ relation works with step-indexed entities. We write $\mathit{iMValue}$ for $\mathbb{N} \times \mathit{MVal}$, iMComp for $\mathbb{N} \times \mathit{MComp}$ and iCESD for $\mathbb{N} \times \mathit{CESD}$. The relation is 'logical', with each type constructor having an associated relational action. Biorthogonality is used in defining the action of the lifting monad that, whilst not reflected explicitly in F_v types, is morally there in the distinction between expressions and values. Given a relation R between F_v values and machine values, we'll want a relation between F_v expressions and machine computations. A direct approach would say something about the expression and the computation evaluating to R-related values, but that would overspecify just *how* the low-level code must compute, requiring a non-observable intermediate configuration of a particular shape. We instead exploit two maps, each of which is half of a contravariant Galois connection, between the lattice of subsets of *iCESD* and those of subsets of *iMValue* and of *iMComp*, respectively. If $pe \in \mathit{MEnv}$ and $P \subseteq \mathit{iMValue}$, define

$$\begin{aligned}\downarrow^{pe}(P) &\stackrel{\text{def}}{=} \{(j, \langle c,\ e,\ s,\ d\rangle) \mid \forall(i,v) \in P, \langle c,\ pe{+\!\!+}e,\ v :: s,\ d\rangle \mapsto^{\min(i,j)}\}\\ &\subseteq\ \mathit{iCESD}.\end{aligned}$$

So an indexed context is in $\downarrow^{pe}(P)$ if whenever we link it with an indexed value from P and the machine environment pe, the resulting configuration takes a number of steps that is at least the minimum of the two indices. Coming back the other way, if $pe \in \mathit{MEnv}$ and $Q \subseteq \mathit{iCESD}$, define

$$\begin{aligned}\Uparrow^{pe}(Q) &\stackrel{\text{def}}{=} \{(i, (c', s')) \mid \forall(j, \langle c,\ e,\ s,\ d\rangle) \in Q, \langle c'{+\!\!+}c,\ pe{+\!\!+}e,\ s'{+\!\!+}s,\ d\rangle \mapsto^{\min(i,j)}\}\\ &\subseteq\ \mathit{iMComp}.\end{aligned}$$

The maps are indexed by an (an extension of) the environment as a way of sharing the environment between the computation and the context.

If $e \in \mathit{MEnv}$, τ is a closed type and $RT_i \subseteq \mathit{MVal} \times (\mathit{CValue}\,\tau)$ 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[3]

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

If τ is a closed source type, then let iRel_τ be the set of $\mathbb{N}$-indexed relations $R_i \subseteq \mathit{MVal} \times (\mathit{CValue}\ \tau)$. 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 $\Theta \vdash \sigma$, the indexed relation

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

is defined by induction on σ, as shown in Figure 3. The relational interpretation of a type variable is looked up in the relation environment, machine integers approximate the corresponding high-level value, any machine value approximates the unit value, and machine pairs approximate F_v pairs pointwise. High-level sum values are approximated by tagged pairs on the machine. 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

[3] The lifting is a form of possibility modality, in the sense of Evaluation Logic [11], whence the notation $\langle R\rangle$.

$$
\begin{aligned}
\trianglelefteq_i^{\boldsymbol{\tau R},X_k} &= (R_k)_i \\
\trianglelefteq_i^{\boldsymbol{\tau R},\mathtt{Int}} &= \{(\underline{n}, n) \mid n \in \mathbb{N}\} \\
\trianglelefteq_i^{\boldsymbol{\tau R},1} &= \{(v, ()) \mid v \in \mathit{MVal}\} \\
\trianglelefteq_i^{\boldsymbol{\tau R},\sigma_1 \times \sigma_2} &= \{(\mathtt{PR}\,(v_1, v_2), \langle V_1, V_2\rangle) \mid (v_1, V_1) \in \trianglelefteq_i^{\boldsymbol{\tau R},\sigma_1} \wedge\ (v_2, V_2) \in \trianglelefteq_i^{\boldsymbol{\tau R},\sigma_2}\} \\
\trianglelefteq_i^{\boldsymbol{\tau R},\sigma_1 + \sigma_2} &= \{(\mathtt{PR}\,(\underline{n+1}, v), \mathtt{inl}\,V) \mid (v, V) \in \trianglelefteq_i^{\boldsymbol{\tau R},\sigma_1}, n \in \mathbb{N}\} \\
&\quad \cup \{(\mathtt{PR}\,(\underline{0}, v), \mathtt{inr}\,V) \mid (v, V) \in \trianglelefteq_i^{\boldsymbol{\tau R},\sigma_2}\} \\
\trianglelefteq_i^{\boldsymbol{\tau R},\sigma_1 \to \sigma_2} &= \{(f, F) \mid \forall k \le i, \forall (v, V) \in \trianglelefteq_k^{\boldsymbol{\tau R},\sigma_1}, \\
&\quad (([\mathtt{App}], [v, f]),\ (F\,V)) \in \langle(\trianglelefteq^{\boldsymbol{\tau R},\sigma_2})_\perp^{[]}\rangle_k\} \\
\trianglelefteq_i^{\boldsymbol{\tau R},\forall X.\sigma} &= \{(v, V) \mid \forall \tau', \forall R' \in \mathit{iRel}_{\tau'}, \\
&\quad \text{decreasing } R' \to (\widehat{v},\ (V\,\tau')) \in \langle(\trianglelefteq^{\boldsymbol{\tau R},\tau' R',\sigma})_\perp^{[]}\rangle_i\} \\
\trianglelefteq_i^{\boldsymbol{\tau R},\exists X.\sigma} &= \{(v, \mathtt{pack}\,\tau', V\,\mathtt{to}\,\exists X.\sigma) \mid \exists R' \in \mathit{iRel}_{\tau'}, \\
&\quad \text{decreasing } R' \wedge (v, V) \in \trianglelefteq_i^{\boldsymbol{\tau R},\tau' R',\sigma}\}
\end{aligned}
$$

Fig. 5. The relation $\trianglelefteq_i^{\boldsymbol{\tau R},\sigma}$

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 F_v type such that v is related to the packed value.

We lift $\trianglelefteq$ to environments pointwise. 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 $1 \le j \le m$. Then for general expressions in context:

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

which is again 'logical', taking related environments to related computations for all smaller indices. That was for closed types. For closed values of *open* types $\Theta \vdash \sigma$, we define

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

by 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. Finally, we can define the true $\preceq$ relations between the machine and F_v by quantifying over all step indices and combining with the $\sqsubseteq$ relation:

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

Approximating Low-Level By High-Level. The $\succeq$ relation expresses when a machine computation is approximated by 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 appropriate, whereas being greater is a liveness property: we will be specifying that certain machine configurations terminate. We again start with maps between sets of contexts and sets of values and of computations, defining a notion of orthogonality. For $pe \in \mathit{MEnv}$, $P \subseteq \mathit{MVal}$, $Q \subseteq \mathit{CESD}$, define

$$\begin{aligned}
\underline{\downarrow}^{pe}(P) &= \{\langle c,\, e,\, s,\, d\rangle \mid \forall v \in P, \langle c,\, pe{+\!\!+}e,\, v :: s,\, d\rangle \mapsto^* \lightning\} \quad \subseteq \mathit{CESD} \\
\overline{\overline{\Uparrow}}^{pe}(Q) &= \{(c', s') \mid \forall \langle c,\, e,\, s,\, d\rangle \in Q, \langle c'{+\!\!+}c,\, pe{+\!\!+}e,\, s'{+\!\!+}s,\, d\rangle \mapsto^* \lightning\} \subseteq \mathit{MComp}
\end{aligned}$$

and now, for a closed type τ and a relation $\mathit{RT} \subseteq \mathit{MVal} \times (\mathit{CValue}\,\tau)$, define a relational lifting modality by

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

That is, given a low-high relation RT on values, the lifted relation holds between 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 closed τ, let Rel_τ be $\mathbb{P}(\mathit{MVal}\times(\mathit{CValue}\,\tau))$. If $\Theta = X_1, \ldots, X_m$, a relation environment for Θ is now a vector $\boldsymbol{\tau R}$ of pairs of closed types and relations, where $R_k \in \mathit{Rel}_{\tau_k}$. For each $\boldsymbol{\tau R} : \Theta$ and $\Theta \vdash \sigma$, we now define the relation

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

by induction on σ, as shown in Figure 3. This relation also lifts pointwise to environments. For $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ and $\Theta \vdash \Gamma$ and $\boldsymbol{\tau R} : \Theta$, define $\trianglerighteq^{\boldsymbol{\tau R},\Gamma} \subseteq \mathit{MEnv} \times (\mathit{EValue}\,[]\,(\Gamma[\tau_k/X_k]))$ to be the set of pairs $([v_1, \ldots, v_n], (V_1, \ldots, V_n))$ such that $(v_j, V_j) \in \trianglerighteq^{\boldsymbol{\tau R},\sigma_j}$ for all j. Then for computations in context,

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

For open types, we 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 $\trianglerighteq^{\Theta\vdash\Gamma}$, for open environments, and $\trianglerighteq^{\Theta;\Gamma\vdash\sigma}$, for open expressions. The true $\succeq$ relations are then given by combining with $\hat{\sqsubseteq}$ as follows:

$$\begin{aligned}
v \succeq^{\Theta\vdash\tau} V \in \mathit{Value}\,\Theta\,[]\,\tau \quad &\overset{\text{def}}{\iff} \quad \exists\langle W_i\rangle_i, \Theta; - \vdash V \hat{\sqsubseteq} \langle W_i\rangle_i : \tau \wedge \forall j, v \trianglerighteq^{\Theta\vdash\tau} W_j \\
e \succeq^{\Theta\vdash\Gamma} \boldsymbol{V}\in \mathit{EValue}\,\Theta\,\Gamma \quad &\overset{\text{def}}{\iff} \quad \exists\langle \boldsymbol{W}_i\rangle_i, \Theta \vdash \boldsymbol{V} \hat{\sqsubseteq} \langle \boldsymbol{W}_i\rangle_i : \Gamma \wedge \forall j, e \trianglerighteq^{\Theta\vdash\Gamma} \boldsymbol{W}_j \\
\mathit{comp} \succeq^{\Theta;\Gamma\vdash\sigma} M\in \mathit{Exp}\,\Theta\,\Gamma\,\sigma \quad &\overset{\text{def}}{\iff} \quad \exists\langle N_i\rangle_i, \Theta;\Gamma \vdash M \hat{\sqsubseteq} \langle N_i\rangle_i : \sigma \wedge \forall j, \mathit{comp} \trianglerighteq^{\Theta;\Gamma\vdash\sigma} N_j.
\end{aligned}$$

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

Fig. 6. The relation $\trianglerighteq^{\boldsymbol{\tau R},\sigma}$

Realizability. The conjunction of the two approximation relations is our notion of when a machine value, environment or computation realizes an F_v term:

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

It is immediate from the definitions that the $\models$ relations are closed on the right under $==$, the symmetric closure of $\sqsubseteq$. We note again that these definitions make surprisingly little reference to actual low-level code and values. We specify the encoding for base types, pairs and sums 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.

If $(c', s') \in \mathit{MComp}$, we say (c', s') diverges unconditionally if for any c, e, s, d, $\langle c' {+\!\!+} c,\ e,\ s' {+\!\!+} s,\ d\rangle \mapsto^{\omega}$.

Lemma 1 (Ground divergence adequacy). *For any* $\mathit{comp} \in \mathit{MComp}$*, type* $\tau = \mathtt{Int}$ *or* 1*, and* $M \in \mathit{CExp}\,\tau$ *if* $\mathit{comp} \models^{[];[]\vdash\tau} M$ *and the* F_v *term* M *diverges, then* comp *diverges unconditionally.*

Say a computation (c', s') converges to a natural n if plugging it into an arbitrary context equiterminates with plugging n into that context:

$$
\begin{aligned}
\forall c, e, s, d,\ \langle c,\ e,\ \underline{n} :: s,\ d\rangle \mapsto^{*} \lightning \;&\Longrightarrow\; \langle c' {+\!\!+} c,\ e,\ s' {+\!\!+} s,\ d\rangle \mapsto^{*} \lightning \\
\wedge\ \langle c,\ e,\ \underline{n} :: s,\ d\rangle \mapsto^{\omega} \;&\Longrightarrow\; \langle c' {+\!\!+} c,\ e,\ s' {+\!\!+} s,\ d\rangle \mapsto^{\omega}\ .
\end{aligned}
$$

If a computation realizes a closed F_v term that evaluates to an integer n, then it converges to n:

Lemma 2 (Ground convergence adequacy). *For any* $\mathit{comp} \in \mathit{MComp}$*,* $M \in \mathit{CExp}\ \mathtt{Int}$ *and* $n \in \mathbb{N}$*, if* $\mathit{comp} \models^{[];[]\vdash\mathtt{Int}} M$ *and* $M \Mapsto n$ *then* comp *converges to* n*.*

Convergence adequacy also holds for observation at the unit type, with a definition of convergence that quantifies over test contexts *cesd* whose termination is independent of the value on the top of the stack (since any value realizes ()).

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

Lemma 3 (Compositionality for application). *For any $cf, cx \in Code$ and $V_f \in Value\,\Theta\,\Gamma\,(\tau \to \tau')$, $V_x \in Value\,\Theta\,\Gamma\,\tau$,*

$$(cf, []) \models^{\Theta;\Gamma\vdash\tau\to\tau'} [V_f] \;\wedge\; (cx, []) \models^{\Theta;\Gamma\vdash\tau} [V_x] \implies (cf{+}{+}cx{+}{+}[\mathtt{App}], []) \models^{\Theta;\Gamma\vdash\tau'} V_f\,V_x.$$

4 Examples

Compiler Correctness. Whilst the realizability relation is defined without reference to our compiler, we do of course intend that the compiler is correct, in the sense that compiled code always realizes the original source term:

Theorem 1 (Compositional Compiler Correctness).

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 $(\!|M|\!)_{\mathsf{false}} \models^{\Theta;\Gamma\vdash\tau} M$.*

Both directions of the above are proved by simultaneous structural induction, with a strengthened induction hypothesis to account for compilation of expressions with the *ret* flag set to true. In the $\preceq$ direction, the case for recursive functions involves a nested induction over step indices, whilst in the the $\succeq$ direction, we appeal to the unwinding property of the $\sqsubseteq$ parameter relation.

A consequence is that compiled code for whole programs has the correct operational behaviour according to the operational semantics of the programs:

Corollary 1 (Correctness for whole programs). *For any $M \in CExp$ `Int`, if M diverges, then $(\!|M|\!)_{\mathsf{false}}$ diverges unconditionally and if $M \mapsto\!\!\!\Rightarrow n$, then $(\!|M|\!)_{\mathsf{false}}$ converges to n.*

The corollary is normally thought of as compiler correctness, but it is Theorem 1 that lets us reason about combining compiled code with code from elsewhere.

Hand-written fixed-point combinator. One can use the `rec` construct to write a CBV fixed-point combinator in F_v:

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

which compiles to code using SECD's recursive closures:

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

Alternatively, we can hand-encode $\lambda F.\, \lambda x.\, (\lambda y.\, F(\lambda z.\, y\, y\, z))\, (\lambda y.\, F(\lambda z.\, y\, y\, z))\, x$, which is an *untyped* CBV fixpoint combinator, as the following SECD code:

```
YCombinator =
[PushC [PushC [
    PushC [Push 2, PushC [Push 1, Push 1, App, Push 0, App, Ret], App, Ret],
    PushC [Push 2, PushC [Push 1, Push 1, App, Push 0, App, Ret], App, Ret],
    App, Push 0, App, Ret], Ret]]
```

Direct reasoning about the operational semantics of low-level code and of F_v, *not* involving anything to do with the compiler, shows the following:

Lemma 4. $\mathsf{YCombinator}, [] \models^{\vdash \forall X.\,\forall Y.\,((X \to Y) \to X \to Y) \to X \to Y} \mathsf{FixC}$.

Together with Theorem 1 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 term that is == to the explicitly recursive FixC.

Context manipulation. For this machine, most interesting examples of the difference between intensional and extensional specifications involve higher-order functions and cunning use of the intensional equality test. But one simple first-order example is that for any $n \in \mathbb{N}$,

$$([\mathtt{PushN}\,n, \mathtt{PushN}\,1, \mathtt{Sel}\,([\mathtt{Join}, \mathtt{PushN}\,0, \mathtt{Pop}], [])], []) \models^{\vdash \mathtt{Int}} [n]$$

This code would not be in a simple direct-style realizability relation because as well as pushing n onto the stack, it modifies its context (non-observably), appending push and pop instructions to the end of the current continuation.

Hand-optimized polymorphic list module.. We now give an example that involves relational parametricity, for both universal (polymorphic) and existential (abstract) types. Consider a signature for a polymorphic list module:

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

where $\mathtt{Option}\,\tau = 1{+}\tau$ and we write *none* and *some* for the constructors as usual. The signature says that an implementation of polymorphic lists should have some private representing type LX, equipped with three standard list manipulation operations: constructors $\mathit{nil} : LX$, $\mathit{cons} : X \times LX \to LX$ and a destructor $\mathit{split} : LX \to \mathtt{Option}\,(X \times LX)$.

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

$$\mathsf{PList} : \mathtt{SigPolList} = \Lambda X.\,\mathtt{pack}\,\mathtt{List}\,X, (\mathrm{nil}_X,\ \mathrm{cons}_X,\ \mathrm{split}_X)\,\mathtt{to}\,\ldots$$

where $\mathtt{List}\,\tau \;=\; \forall Y.\,Y \times (\tau \times Y \to Y) \to Y$ for any type τ and a fresh type variable Y, and where

$$\begin{array}{l}
\mathrm{nil}_\tau : \mathtt{List}\,\tau \;=\; \Lambda Y.\,\lambda(n,c).\,n \\
\mathrm{cons}_\tau\,(\mathit{hd} : \tau)\,(\mathit{tl} : \mathtt{List}\,\tau) : \mathtt{List}\,\tau \;=\; \Lambda Y.\,\lambda(n,c).\,c\,(\mathit{hd}, \mathit{tl}\,Y\,(n,c)) \\
\mathrm{split}_\tau\,(l : \mathtt{List}\,\tau) : \mathtt{Option}\,(\tau \times \mathtt{List}\,\tau) \;= \\
\quad l\,(\mathtt{Option}\,(\tau \times \mathtt{List}\,\tau))\,(\mathrm{none}_{\tau \times \mathtt{List}\,\tau}, \\
\qquad \lambda(\mathit{hd}, \mathit{htl}).\,\mathtt{case}\,\mathit{htl}\,\mathtt{of}\,\mathtt{inl}\,().\,\mathrm{some}_{\tau \times \mathtt{List}\,\tau}\,(\mathit{hd}, \mathrm{nil}_\tau) \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathtt{inr}\,(\mathit{hd}', \mathit{tl}).\,\mathrm{some}_{\tau \times \mathtt{List}\,\tau}\,(\mathit{hd}, \mathrm{cons}_\tau\,\mathit{hd}'\,\mathit{tl}))
\end{array}$$

The Church encoding is elegant, but inefficient: the list splitting operation is $\mathcal{O}(n)$, rather than $\mathcal{O}(1)$, for example. But without recursive types, we can do no better in the F_v source language. However, one can treat the inefficient encoding of lists as a specification, and 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 would not be typeable 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 via a (potentially iterated) Gödel numbering of sequences of natural numbers. The optimized implementation of the list module in SECD code 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 $n, m \in \mathbb{N}$, $\mathsf{npair}\, n\, m > 0$, $\mathsf{nfst}\,(\mathsf{npair}\, n\, m) = n$ and $\mathsf{nsnd}\,(\mathsf{npair}\, n\, m) = m$. One could take $\mathsf{npair}\, n\, m$ to be $2^n \times 3^m$, with matching projection functions, for example. We assume low-level implementations $\mathsf{NPair}, \mathsf{NSplit} \in \mathit{Code}$ of these pairing operations, such that

$$\begin{array}{l}
\forall c\, e\, s\, d\, n\, m\, \exists k\, \langle \mathsf{NPair}{+\!\!+}c,\ e,\ \underline{m} :: \underline{n} :: s,\ d\rangle \mapsto^k \langle c,\ e,\ \underline{\mathsf{npair}\, n\, m} :: s,\ d\rangle \\
\forall c\, e\, s\, d\, n\, \exists k\, \langle \mathsf{NSplit}{+\!\!+}c,\ e,\ \underline{n} :: s,\ d\rangle \mapsto^k \langle c,\ e,\ \mathtt{PR}\,(\underline{\mathsf{nfst}\, n}, \underline{\mathsf{nsnd}\, n}) :: s,\ d\rangle
\end{array}$$

Exploiting the instruction `IsNum` that checks if a given machine value is a number, one can compactly represent a list of machine values as follows:

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

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

```
  NilM=[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 ([PushN 0, Push 0, IsNum, Sel (NSplit++[Join], [Join]),
                    MkPair, Join],
                  [PushN 1, PushN 1, MkPair, Join]),
              Ret]].
```

To aid understanding, we give pseudo-F_v terms interpreting the above code:

$$\begin{array}{l}
\mathsf{NilM} \approx \underline{0} \\
\mathsf{ConsM} \approx \lambda(hd, tl).\,\text{if } hd = \underline{n} \wedge tl = \underline{m} \text{ then } \underline{\mathsf{npair}\, n\, m} \text{ else } \mathtt{PR}\,(hd, tl) \\
\mathsf{SplitM} \approx \lambda\, l.\,\text{if } l \neq \underline{0} \text{ then if } l = \underline{n} \text{ then } \mathit{some}\,(\mathsf{nfst}\,\underline{n}, \mathsf{nsnd}\,\underline{n}) \text{ else } \mathit{some}\, l \\
\qquad\qquad\qquad\qquad \text{else } \mathit{none}
\end{array}$$

One can then verify that the above optimized low-level implementation realizes, and is therefore substitutable for (in any related context) the code compiled from, the Church-encoded F_v module PList:

Lemma 5. $(\mathsf{NilM}{+\!\!+}\mathsf{ConsM}{+\!\!+}\mathsf{SplitM}{+\!\!+}[\mathtt{MkPair}, \mathtt{MkPair}], []) \models^{\vdash \mathtt{SigPolList}} \mathsf{PList}$.

Whilst frivolous, this shows a non-trivial extensionally parametric specification being realized by an implementation that is intensionally unequivocally non-parametric.

5 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 relation involves relational parametricity, biorthogonality and step-indexing, as well as being parameterized by a novel form of adequate precongruence, amounting to a kind of ideal closure operation, on the source language.

The results and examples are all formalized in the Coq proof assistant, using a strongly-typed representation of polymorphic terms and substitutions that we describe in a companion paper [4]. Using the strongly-typed representation involves some occasionally-vexing manipulation of casts and heterogeneous ('John Major') equality; this led the second author to develop a Coq library for rewriting with heterogenous equality [8], which is exploited in the latest version of the proofs for the present paper. The full formalization of the source language, machine, logical relations, compiler correctness proof and examples is around 6000 lines, which seems very reasonable, though this is now at least our third completely fresh version of a mechanized compiler correctness theorem. Our initial formalization for this language was over twice as long; the improvement is partly in the formulation of the logical relations and partly in the details of the mechanization. The reasons for mechanizing at all are twofold. Firstly, the reasoning is already sufficiently complex that we really would have low confidence in (and would find it hard to manage) paper proofs, especially for non-trivial examples such as the polymorphic list module. Secondly, mechanization would be an integral part of any realistic infrastructure based on certified code, so we simultaneously want to establish the feasibility of (and extend (or encourage others to extend) the state of the art in) doing mechanized proofs in this area.

The closest related work is our own paper on relating the denotational semantics of a simply-typed language to a similar machine [3]. There have been many 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 [7], which relates a denotational semantics to, ultimately, real machine code; the Coq formalization of compiler correctness for a total functional language by Chlipala [6]; and the work of Leroy [10] 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 [5] and the present work arose naturally from an attempt to understand the way in which that semantics failed to be sufficiently abstract.

Step-indexing has been widely used to tame recursive phenomena in operational semantics in the last decade. During the course of this research, however, we have found that it does inhibit some aggresive low-level program transformations that we believe *should* be legal. Intuitively, non-functional operations such as comparing function pointers allow one to look 'too far' into the future, so clever optimizations make misbehaving computations misbehave 'too soon' to respect current forms of step-indexed relations. Finding a way around this problem will be interesting and challenging. Other avenues for further 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, C.-K. Hur, A. Kennedy, and C. McBride. Strongly typed term representations in Coq. Submitted, November 2009.

[5] 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.

[6] 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.

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

[8] C. K. Hur. Heq: A Coq library for heterogeneous equality. `http://www.pps.jussieu.fr/~gil/Heq/`, May 2010.

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

[10] 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.

[11] A. M. Pitts. Evaluation logic. In *IVth Higher Order Workshop, Banff 1990*, Workshops in Computing, 1991.

[12] A. M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7. 2005.

[13] 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.