Biorthogonality, Step-Indexing and Compiler Correctness

Nick Benton
Microsoft Research
nick@microsoft.com

Chung-Kil Hur
University of Cambridge
ckh25@cam.ac.uk

Abstract

We define logical relations between the denotational semantics of a simply typed functional language with recursion and the operational behaviour of low-level programs in a variant SECD machine. The relations, which are defined using biorthogonality and step-indexing, capture what it means for a piece of low-level code to implement a mathematical, domain-theoretic function and are used to prove correctness of a simple compiler. The results have been formalized in the Coq proof assistant.

1. Introduction

Proofs of compiler correctness have been studied for over forty years (McCarthy and Painter 1967; Dave 2003) and have recently been the subject of renewed attention, firstly because of increased interest in security and certification in a networked world and secondly because of advances in verification technology, both theoretical (e.g. separation logic, step-indexed logical relations) and practical (e.g. developments in model checking and improvements in interactive proof assistants).

There are many notions of correctness or safety that one might might wish to establish of a compiler. For applying language-based techniques in operating systems design, as in proof-carrying code, one is primarily interesting in broad properties such as type-safety, memory-safety or resource-boundedness. Although these terms are widely used, they are subject to a range of interpretations. For example, type-safety sometimes refers to a simple syntactic notion (Morrisett et al. 1999) ('is the generated code typable using these rules?') and sometimes to a deeper semantic one ('does the observable behaviour of the code satisfy this desirable property?'). In previous work (Benton 2006; Benton and Zarfaty 2007; Benton and Tabareau 2009), we have looked at establishing type-safety in the latter, more semantic, sense. Our key notion is that a high-level type translates to a low-level specification that should be satisfied by any code compiled from a source language phrase of that type. These specifications are inherently relational, in the usual style of PER semantics, capturing the meaning of a type A as a predicate on low-level heaps, values or code fragments *together* with a notion of A-equality thereon. These relations express what it means for a source-level abstractions (e.g. functions of type $A \to B$) to be respected by low-level code (e.g. 'taking A-equal arguments to B-equal results'). A crucial property of our low-level specifications is that they are defined in terms of the behaviour of low-level programs; making no reference to any intensional details of the code produced by a particular compiler or the grammar of the source language. Of course, the specifications *do* involve low-level details of data representations and calling conventions – these are part of the interface to compiled code – but up to that, code from any source that behaves sufficiently *like* code generated by the compiler should meet the specification, and this should be independently verifiable.

Ideally, one might wish to establish the sense in which a compilation scheme is *fully abstract*, meaning that the compiled versions of two source phrases of some type are in the low-level relation interpreting that type iff the original source phrases are contextually equivalent. If low-level specifications are used for checking linking between the results of compiling open programs and code from elsewhere[1] and full abstraction does not hold, then source level abstractions become 'leaky': reasoning about equivalence or encapsulation at the source level does not generally translate to the target, which can lead to unsound program transformations in optimizing compilers and to security vulnerabilities (Abadi 1998; Kennedy 2006). Ahmed and Blume (2008) also argue that fully abstract translation should be the goal, and prove full abstraction for (source to source) typed closure conversion for a polymorphic lambda calculus with recursive and existential types. Later on we will say something about why we believe that 'full' full abstraction may not, in practice, be quite the right goal, but we certainly *do* want 'sufficiently abstract' compilation, i.e. the preservation of the reasoning principles that we actually use in an optimizing compiler or in proving security properties.

The low-level relations of our previous work are not, however, really even sufficiently abstract, having roughly comparable power to a denotational semantics in continuation-passing style (CPS). This is a very strong and useful constraint on the behaviour of machine-code programs, but does not suffice to prove all the equations we might like between low-level programs: even something as simple as the commutativity of addition does not hold for arbitrary integer computations (just as it doesn't in a lambda calculus with control), even though it does in our pure source language. To understand how to refine our low-level relations further, it is natural to look at logical relations between low-level code and elements of the domains arising in a standard, direct-style, denotational model of our language, which is what we'll do here. Such relations induce low-level PERs (of the sort we were previously defining directly) by taking two low-level programs to be equivalent just when they are related to some common high-level denotational value.

The relations we define will establish a full functional correctness theorem for a simple compiler, not merely a semantic type safety theorem. Just as for type-safety, there are several approaches

[1] This is obviously important for foreign function interfaces and multilanguage interoperability, but can be an issue even for separate compilation using the same compiler. It also covers the simpler and ubiquitous case of handcrafted implementations of standard library routines.

to formulating such correctness theorems in the literature. A common one, used for example by Leroy (2006), is to define an operational semantics for both high- and low-level languages and prove a simulation result between source programs and their compiled versions, allowing one to conclude that if a closed high-level program terminates with a particular observable result, then its compiled version terminates with the same result. One limitation of this kind of one-way simulation result is that it says nothing about diverging programs. Another is that it is not as compositional (modular) or extensional (behavioural) as we would like: we want maximally permissive specifications that capture just what it means for a piece of low-level code to behave like, or *realize*, a particular high-level computation. A slogan here is that 'the ends justify the means': we wish to allow low-level programs that extensionally get the right answers, whilst intensionally using any means necessary.

The main results here will involve logical relations between a cpo-based denotational semantics of a standard CBV lambda calculus with recursion and programs in an extended SECD-like target, chosen to be sufficiently low-level to be interesting, yet simple enough that the important ideas are not lost in detail. The relations will involve both biorthogonality and step-indexing, and in the next section we briefly discuss these constructions in general terms, before turning to the particular use we make of them.

The results in the paper have been formalized and proved in the Coq proof assistant, making use of a formalization of domain theory and denotational semantics that we describe elsewhere (Benton et al. 2009).

2. Orthogonality and Step-Indexing

2.1 Biorthogonality

Biorthogonality is a powerful and rather general idea that has been widely used in diffent kinds of semantics in recent years, beginning with the work of Pitts and Stark (1998) and of Krivine (1994). One way of understanding the basic idea is as a way of 'contextualizing' properties of parts of a system, making them compositional and behavioural. In the unary case, we start with some set of systems $\mathcal{S}$ (e.g. configurations of some machine, lambda terms, denotations of programs, processes) and some predicate $\mathcal{O} \subseteq \mathcal{S}$, which we call an observation, over these systems (e.g. those that execute without error, those that terminate, those that diverge). Then there is some way of combining, or plugging, program fragments $p \in \mathcal{P}$ in whose properties we are interested (bits of machine code, terms, denotations of terms, processes) with complementary contexts $c \in \mathcal{C}$ (frame stacks, terms with holes, continuations, other processes) to yield complete systems. The plugging $- \circ - : \mathcal{P} \times \mathcal{C} \to \mathcal{S}$ might be effected by appending bits of program, substituting terms, applying continuations to arguments or composing processes in parallel. In any such situation, there is covariant map $(\cdot)^\perp : \mathbb{P}(\mathcal{P}) \to \mathbb{P}(\mathcal{C})$ given by

$$P^\perp = \{c \in \mathcal{C} \mid \forall p \in P,\, p \circ c \in \mathcal{O}\}$$

and a homonymous one in the other direction, $(\cdot)^\perp : \mathbb{P}(\mathcal{C}) \to \mathbb{P}(\mathcal{P})$

$$C^\perp = \{p \in \mathcal{P} \mid \forall c \in C,\, p \circ c \in \mathcal{O}\}$$

yielding a contravariant Galois connection, so that, amongst many other things, $(\cdot)^{\perp\perp}$ is a closure operator (inflationary and idempotent) on $\mathbb{P}(\mathcal{P})$, and that any set of the form $C^\perp$ is $(\cdot)^{\perp\perp}$-closed. The binary version of this construction starts with a binary relation (e.g. an equivalence relation) on $\mathcal{S}$ and proceeds in the obvious way.

For compiler correctness, we want the interpretations of source-level types or, indeed, source-level values, to be compositional and extensional: properties of low-level program fragments that we can check independently and that make statements about the observable behavior of the complete configurations that arise when we plug the fragments into 'appropriate' contexts. This set of 'appropriate' contexts can be thought of as a set of tests: a low-level fragment is in the interpretation of a source type, or correctly represents a source value, just when it passes all these tests. Thus these low-level interpretations will naturally be $(\cdot)^{\perp\perp}$-closed sets. For a simply typed source language, we can define these sets by induction on types, either positively, starting with an over-intensional set and then taking its $(\cdot)^{\perp\perp}$-closure, or negatively, by first giving an inductive definition of a set of contexts and then taking its orthogonal. See Vouillon and Melliès (2004) for more on the use of biorthogonality in giving semantics to types.

2.2 Step-Indexing

Logical predicates and relations can be used with many different styles of semantics. When dealing with languages with recursion and working denotationally, however, we often need extra admissibility properties, such as closure under limits of ω-chains. Operational logical relations for languages with recursion generally need to satisfy some analogue of admissibility. One such that has often been used with operational semantics based on lambda terms (Plotkin 1977; Pitts and Stark 1998) considers replacing the recursion construct $\texttt{rec}\, f\, x = M$ (or fixpoint combinator) with a family of finite approximations: $\texttt{rec}_n\, f\, x = M$ for $n \in \mathbb{N}$, unfolds the recursive function n times in M and thereafter diverges. Appel and McAllester (2001) introduced step-indexed logical relations, which have since been refined and succesfully applied by various authors to operational reasoning problems for both high and low level languages, many of which involve challenging language features (Ahmed 2006; Appel et al. 2007; Benton and Tabareau 2009; Ahmed et al. 2009). Step-indexing works with small-step operational semantics $\mathbb{N}$-indexed sets of values, with $(n, v) \in P$ (or $v \in P_n$) meaning 'value v has property P for n steps of reduction'. An interesting feature of step-indexing is that one usually works directly with this family of approximants; the limits that one feels are being approximated (like $\{v \mid \forall n, v \in P_n\}$) do not play much of a rôle.

2.3 On Using Both

Amongst the useful properties of the operational $(\cdot)^{\perp\perp}$-closure operation used by Pitts and Stark (1998) is that it yields admissible sets (in the $\texttt{rec}_n$ sense). The same is true of its natural denotational analogue (Abadi 2000). Our earlier work on low-level interpretations of high-level types (Benton and Tabareau 2009) used both step-indexing and orthogonality, but there was some question as to whether the step-indexing was really necessary. Maybe our closed sets are automatically already appropriately 'admissible', just by construction, and there is no need to add extra explicit indexing? Slightly to our surprise, it turns out that there *are* good reasons to use both $(\cdot)^{\perp\perp}$-closure and step-indexing, for which we now try to give some intuition.

The aim is to carve out interpretations of high-level types and values as 'well-behaved' subsets of low-level, untyped programs. The essence of these interpretations generally only depends upon these well-behaved subsets: we'll (roughly) interpret a function type $A \to B$ as the set of programs that when combined with a good argument of type A and a good continuation expecting something of type B, yield good behaviour. So exactly what range of impure, potentially type-abstraction violating, operations are available in the untyped language (the range of 'any means necessary' above) does not seem to affect the definitions or key results. Programs that use low-level features in improper ways will simply not be in the interpretations if high-level entities, and nothing more needs to be said. For a simply-typed total language *without* recursion, this intuition is quite correct: a Krivine-style realizability interpretation is essentially unaffected by adding extra operations to the untyped tar-

get. Even though orthogonality introduces quantification over bigger sets of contexts, nothing relies explicitly on properties of the untyped language as a whole.

In the presence of recursion, the situation changes. The proof that Pitts and Stark's $(\cdot)^{\perp\perp}$-closed relations are admissible depends on a 'compactness of evaluation' result, sometimes called an 'unwinding theorem', which is a global property of all untyped programs. In the denotational case, attention is already restricted to operations that can be modelled by continuous functions in the chosen domains. But realistic targets often support operations that can violate these global properties. Examples of such egregiously non-functional operations include the 'reflection' features of some high-level languages (such as Java or C♯) or, more interestingly, the ability of machine code programs to switch on code pointers or read executable machine instructions.[2] We have found that the presence of such seriously non-functional operations does not just make the proofs harder, but can actually make the 'natural' theorems false. Appendix A shows how the addition of equality testing on lambda terms to an untyped lambda calculus breaks a 'standard' syntactic interpretation of simple types as sets of untyped terms in the presence of term-level recursion in the source.

Fortunately, as we will see, step-indexing sidesteps this problem. In place of appealing to a global property that holds of all untyped programs, we build a notion of approximation, and the requirement to behave well with respect to it, directly into the definition of our logical relations. In fact, we will also do something very similar on the denotational side, closing up explicitly under limits of chains.

3. Source Language

Our high-level language PCF$_v$ is a conventional simply-typed, call-by-value functional language with types built from integers and booleans by products and function spaces, with type contexts Γ defined in the usual way:

$$\begin{array}{rcl} t & := & \mathtt{Int} \mid \mathtt{Bool} \mid t \to t' \mid t \times t' \\ \Gamma & := & x_1 : t_1 \,, \ldots \,, x_n : t_n \end{array}$$

We separate syntactic values (canonical forms), ranged over by V, from general expressions, ranged over by M and restrict the syntax to ANF, with explicit sequencing of evaluation by `let` and explicit inclusion of values into expressions by $[\cdot]$.

The typing rules for values and for expressions are shown in Figure 1. Note that there are really two forms of judgement, but we refrain from distinguishing them syntactically. The symbol $\star$ stands for an arbitrary integer-valued binary operation on integers, whilst $>$ is a representative boolean-valued one.

PCF$_v$ has the obvious CBV operational semantics, which we elide here, and a conventional, and computationally adequate, denotational semantics in the category of ω-cpos and continuous maps, which we now briefly summarize to fix notation. Types and environments are interpreted as cpos:

$$\begin{array}{rcl} [\![\mathtt{Int}]\!] & \stackrel{\text{def}}{=} & \mathbb{N} \\ [\![\mathtt{Bool}]\!] & \stackrel{\text{def}}{=} & \mathbb{B} \\ [\![t \to t']\!] & \stackrel{\text{def}}{=} & [\![t]\!] \Rightarrow [\![t']\!]_\perp \\ \\ [\![x_1 : t_1, \ldots, x_n : t_n]\!] & \stackrel{\text{def}}{=} & [\![t_1]\!] \times \cdots \times [\![t_n]\!] \end{array}$$

[2] Real implementations, even of functional languages, can make non-trivial use of such features. For example, interpreting machine instructions that would normally be executed in order to advance to a safe-point for interruption, building various runtime maps keyed on return addresses, doing emulation, JIT-compilation or SFI.

where $\Rightarrow$ is the cpo of continuous functions and $\times$ is the Cartesian product cpo. Typing judgements for values and expressions are then interpreted as continuous maps:

$$\begin{array}{rcl} [\![\Gamma \vdash V : t]\!] & : & [\![\Gamma]\!] \to [\![t]\!] \\ [\![\Gamma \vdash M : t]\!] & : & [\![\Gamma]\!] \to [\![t]\!]_\perp \end{array}$$

defined by induction. So, for example

$$\begin{array}{l} [\![\Gamma \vdash \mathrm{Fix}\, f\, x \,=\, M : A \to B]\!]\, \rho \\ = \\ \mu d_f . \lambda d_x \in [\![A]\!] . [\![\Gamma, f : A \to B, x : A \vdash M : B]\!]\, (\rho, d_f, d_x) \end{array}$$

We write $[\cdot] : D \to D_\perp$ for the unit of the lift monad. We elide the full details of the denotational semantics as they are essentially the same as those found in any textbook, such as that of Winskel (1993). The details can also be found, along with discussion of the Coq formalization of the semantics, in Benton et al. (2009).

4. Target Language and Compilation

4.1 An SECD Machine

The low-level target is a variant of an SECD virtual machine (Landin 1964). We have chosen such a machine rather than a lower-level assembly language, such as that of our previous work, so as to keep the formal development less cluttered with detail. But we are emphatically not interested in the SECD machine as something that is inherently "for" executing programs in a language like our source. We have included an equality test instruction, `Eq`, that works on arbitrary values, including closures, so a counterexample to a naive semantics of types like that in Appendix A can be constructed. Furthermore, the logical relations we present have been carefully constructed with applicability to lower-level machines in mind.

The inductive type *Instruction*, ranged over by i, is defined by

$$\begin{array}{rcl} i & := & \mathtt{Swap} \mid \mathtt{Dup} \mid \mathtt{PushV}\, n \mid \mathtt{Op}\star \mid \mathtt{PushC}\, c \mid \mathtt{PushRC}\, c \mid \mathtt{App} \mid \\ & & \mathtt{Ret} \mid \mathtt{Sel}\, (c_1, c_2) \mid \mathtt{Join} \mid \mathtt{MkPair} \mid \mathtt{Fst} \mid \mathtt{Snd} \mid \mathtt{Eq} \end{array}$$

where c ranges over *Code*, the set of lists of instructions, n ranges over integers, and $\star$ over binary operations on integers. The set *Value* of runtime values, ranged over by v is defined by

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

where e ranges over *Env*, defined to be *list Value*. So a *Value* is either an integer literal, a closure containing an environment and some code, a recursive closure, or a pair values. We also define

$$\begin{array}{rcl} \mathit{Stack} & = & \mathit{list}\, \mathit{Value} \\ \mathit{Dump} & = & \mathit{list}\, (\mathit{Code} \times \mathit{Env} \times \mathit{Stack}) \\ \mathit{CESD} & = & \mathit{Code} \times \mathit{Env} \times \mathit{Stack} \times \mathit{Dump} \end{array}$$

CESD is the set of configurations of our virtual machine. A configuration $\langle c,\ e,\ s,\ d \rangle$ 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.

We write $\mathit{cesd} \mapsto^k$ to mean that the configuration cesd takes at least k steps without getting stuck, and $\mathit{cesd} \mapsto^* \not\downarrow$ to mean that cesd eventually terminates:

$$\mathit{cesd} \mapsto^* \not\downarrow \quad \stackrel{\text{def}}{\iff} \quad \neg(\forall k, \mathit{cesd} \mapsto^k)$$

4.2 Compiling PCF$_v$ to SECD

The compiler comprises two mutually-inductive functions mapping (typed) PCF$_v$ values and expressions into *Code*. We overload $(\!|\cdot|\!)$ for both of these functions, whose definitions are shown in Figure 3.

Values:

$$[TVAR]\ \frac{}{\Gamma, x : t \vdash x : t} \qquad [TBOOL]\ \frac{}{\Gamma \vdash b : \mathtt{Bool}}\ (b \in \mathbb{B}) \qquad [TINT]\ \frac{}{\Gamma \vdash n : \mathtt{Int}}\ (n \in \mathbb{N})$$

$$[TFIX]\ \frac{\Gamma, f : t \to t', x : t \vdash M : t'}{\Gamma \vdash \text{Fix}\, f\, x\ =\ M : t \to t'} \qquad [TP]\ \frac{\Gamma \vdash V_i : t_i\ (i = 1, 2)}{\Gamma \vdash \langle V_1, V_2 \rangle : t_1 \times t_2}$$

Expressions:

$$[TVAL]\ \frac{\Gamma \vdash V : t}{\Gamma \vdash [V] : t} \qquad [TLET]\ \frac{\Gamma \vdash M : t \quad \Gamma, x : t \vdash N : t'}{\Gamma \vdash \mathtt{let}\ x = M\ \mathtt{in}\ N : t'}$$

$$[TAPP]\ \frac{\Gamma \vdash V_1 : t \to t' \quad \Gamma \vdash V_2 : t}{\Gamma \vdash V_1\, V_2 : t'} \qquad [TIF]\ \frac{\Gamma \vdash V : \mathtt{Bool} \quad \Gamma \vdash M_1 : t \quad \Gamma \vdash M_2 : t}{\Gamma \vdash \mathtt{if}\ V\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 : t}$$

$$[TOP]\ \frac{\Gamma \vdash V_1 : \mathtt{Int} \quad \Gamma \vdash V_2 : \mathtt{Int}}{\Gamma \vdash V_1 \star V_2 : \mathtt{Int}} \qquad [TGT]\ \frac{\Gamma \vdash V_1 : \mathtt{Int} \quad \Gamma \vdash V_2 : \mathtt{Int}}{\Gamma \vdash V_1 > V_2 : \mathtt{Bool}}$$

$$[TFST, TSND]\ \frac{\Gamma \vdash V : t_1 \times t_2}{\Gamma \vdash \pi_i(V) : t_i\ (i = 1, 2)}$$

Figure 1. Typing rules for PCF_v

$$\begin{array}{rcll}
\langle \mathtt{Swap} :: c,\, e,\, v_1 :: v_2 :: s,\, d\rangle & \mapsto & \langle c,\, e,\, v_2 :: v_1 :: s,\, d\rangle & \\
\langle \mathtt{Dup} :: c,\, e,\, v :: s,\, d\rangle & \mapsto & \langle c,\, e,\, v :: v :: s,\, d\rangle & \\
\langle \mathtt{PushV}\, n :: c,\, [v_1, \ldots, v_k],\, s,\, d\rangle & \mapsto & \langle c,\, [v_1, \ldots, v_k],\, v_n :: s,\, d\rangle & \\
\langle \mathtt{PushN}\, n :: c,\, e,\, s,\, d\rangle & \mapsto & \langle c,\, e,\, \underline{n} :: s,\, d\rangle & \\
\langle \mathtt{PushC}\ bod :: c,\, e,\, s,\, d\rangle & \mapsto & \langle c,\, e,\, \mathtt{CL}\,(e, bod) :: s,\, d\rangle & \\
\langle \mathtt{PushRC}\ bod :: c,\, e,\, s,\, d\rangle & \mapsto & \langle c,\, e,\, \mathtt{RCL}\,(e, bod) :: s,\, d\rangle & \\
\langle \mathtt{App} :: c,\, e,\, v :: \mathtt{CL}\,(e', bod) :: s,\, d\rangle & \mapsto & \langle bod,\, v :: e',\, [],\, (c, e, s) :: d\rangle & \\
\langle \mathtt{App} :: c,\, e,\, v :: \mathtt{RCL}\,(e', bod) :: s,\, d\rangle & \mapsto & \langle bod,\, v :: \mathtt{RCL}\,(e', bod) :: e',\, [],\, (c, e, s) :: d\rangle & \\
\langle \mathtt{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 \mathtt{Ret} :: c,\, e,\, v :: s,\, (c', e', s') :: d\rangle & \mapsto & \langle c',\, e',\, v :: s',\, d\rangle & \\
\langle \mathtt{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 \mathtt{Sel}\,(c_1, c_2) :: c,\, e,\, \underline{0} :: s,\, d\rangle & \mapsto & \langle c_2,\, e,\, s,\, (c, [], []) :: d\rangle & \\
\langle \mathtt{Join} :: c,\, e,\, s,\, (c', e', s') :: d\rangle & \mapsto & \langle c',\, e,\, s,\, d\rangle & \\
\langle \mathtt{MkPair} :: c,\, e,\, v_1 :: v_2 :: s,\, d\rangle & \mapsto & \langle c,\, e,\, \mathtt{PR}\,(v_2, v_1) :: s,\, d\rangle & \\
\langle \mathtt{Fst} :: c,\, e,\, \mathtt{PR}\,(v_1, v_2) :: s,\, d\rangle & \mapsto & \langle c,\, e,\, v_1 :: s,\, d\rangle & \\
\langle \mathtt{Snd} :: c,\, e,\, \mathtt{PR}\,(v_1, v_2) :: s,\, d\rangle & \mapsto & \langle c,\, e,\, v_2 :: s,\, d\rangle & \\
\langle \mathtt{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 \mathtt{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)
\end{array}$$

Figure 2. Operational Semantics of SECD Machine

5. Logical Relations

In this section we define logical relations between components of the low-level SECD machine and elements of semantic domains, with the intention of capturing just when a piece of low-level code realizes a semantic object. In fact, there will be two relations, one defining when a low-level component approximates a domain element, and one saying when a domain element approximates a low-level component. These roughly correspond to the soundness and adequacy theorems that one normally proves to show a correspondence between an operational and denotational semantics, but are rather more complex.

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

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

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

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

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

Values:

$$
\begin{array}{rcl}
(\!|x_1 : t_1, \ldots, x_n : t_n \vdash x_i : t_i|\!) & = & [\mathtt{PushV}\, i] \\
(\!|\Gamma \vdash \mathit{true} : \mathtt{Bool}|\!) & = & [\mathtt{PushN}\, 1] \\
(\!|\Gamma \vdash \mathit{false} : \mathtt{Bool}|\!) & = & [\mathtt{PushN}\, 0] \\
(\!|\Gamma \vdash n : \mathtt{Int}|\!) & = & [\mathtt{PushN}\, n] \\
(\!|\Gamma \vdash \langle V_1, V_2 \rangle : t_1 \times t_2|\!) & = & (\!|\Gamma \vdash V_1 : t_1|\!) +\!\!+ (\!|\Gamma \vdash V_2 : t_2|\!) +\!\!+ [\mathtt{MkPair}] \\
(\!|\Gamma \vdash \mathrm{Fix}\, f\, x\, =\, M : t \to t'|\!) & = & [\mathtt{PushRC}\, ((\!|\Gamma, f : t \to t', x : t \vdash M : t'|\!) +\!\!+ [\mathtt{Ret}])]
\end{array}
$$

Expressions:

$$
\begin{array}{rcl}
(\!|\Gamma \vdash [V] : t|\!) & = & (\!|\Gamma \vdash V : t|\!) \\
(\!|\Gamma \vdash \mathtt{let}\ x = M\ \mathtt{in}\ N : t'|\!) & = & [\mathtt{PushC}\, ((\!|\Gamma, x : t \vdash N : t'|\!) +\!\!+ [\mathtt{Ret}])] +\!\!+ (\!|\Gamma \vdash M : t|\!) +\!\!+ [\mathtt{App}] \\
(\!|\Gamma \vdash V_1\, V_2 : t'|\!) & = & (\!|\Gamma \vdash V_1 : t \to t'|\!) +\!\!+ (\!|\Gamma \vdash V_2 : t|\!) +\!\!+ [\mathtt{App}] \\
(\!|\Gamma \vdash \mathtt{if}\ V\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 : t|\!) & = & (\!|\Gamma \vdash V : \mathtt{Bool}|\!) +\!\!+ [\mathtt{Sel}\, (((\!|\Gamma \vdash M_1 : t|\!) +\!\!+ [\mathtt{Join}]), ((\!|\Gamma \vdash M_2 : t|\!) +\!\!+ [\mathtt{Join}]))] \\
(\!|\Gamma \vdash V_1 \star V_2 : \mathtt{Int}|\!) & = & (\!|\Gamma \vdash V_1 : \mathtt{Int}|\!) +\!\!+ (\!|\Gamma \vdash V_2 : \mathtt{Int}|\!) +\!\!+ [\mathtt{Op}\, \star] \\
(\!|\Gamma \vdash V_1 > V_2 : \mathtt{Bool}|\!) & = & (\!|\Gamma \vdash V_1 : \mathtt{Int}|\!) +\!\!+ (\!|\Gamma \vdash V_2 : \mathtt{Int}|\!) +\!\!+ [\mathtt{Op}\, (\lambda(n_1, n_2).n_1 > n_2 \supset 1 \mid 0)]
\end{array}
$$

Figure 3. Compiler for PCF_v

5.1 Approximating Denotational By Operational

The logical relation expressing what it means for low-level computations to approximate denotational values works with step-indexed entities. We write *iValue* for $\mathbb{N} \times$ *Value*, *iComp* for $\mathbb{N} \times$ *Comp* and *iCESD* for $\mathbb{N} \times$ *CESD* and define an *Env*-parameterized observation $\mathcal{O}\, e$ over pairs of indexed computations from *iComp* and indexed contexts from *iCESD*:

$$\mathcal{O}\, e\, (i, \mathit{comp})\, (j, \mathit{cesd}) \overset{\text{def}}{\Longleftrightarrow} (\mathit{comp} \bowtie (e \smile \mathit{cesd})) \mapsto^{\min(i,j)}$$

So, given an environment e, $\mathcal{O}\, e$ holds of an indexed computation and an indexed context just when the configuration that results from appending the environment e and the code and stack from the computation onto the corresponding components of the context steps for at least the minimum of the indices of the *iComp* and the *iCESD*. We also define an observation on pairs of indexed values and indexed contexts by lifting values to computations:

$$\mathcal{O}\, e\, (i, v)\, (j, \mathit{cesd}) \overset{\text{def}}{\Longleftrightarrow} \mathcal{O}\, e\, (i, \widehat{v})\, (j, \mathit{cesd})$$

Now we follow the general pattern of orthogonality, but with some small twists. We actually have a collection of observations, indexed by environments, made over step-indexed components of configurations. And each observation gives rise to two, closely related, Galois connections: one between predicates on (indexed) values and predicates on (indexed) contexts, and the other between those on (indexed) computations and (indexed) contexts. So there are four contravariant maps associated with each e : *Env*. Our definitions use these two of them:

$$
\begin{array}{rcl}
\downarrow^e (\cdot) & : & \mathbb{P}(\mathit{iValue}) \to \mathbb{P}(\mathit{iCESD}) \\
\downarrow^e (P) & = & \{(j, \mathit{cesd}) \mid \forall (i, v) \in P,\, \mathcal{O}\, e\, (i, v)\, (j, \mathit{cesd})\} \\
\Uparrow^e (\cdot) & : & \mathbb{P}(\mathit{iCESD}) \to \mathbb{P}(\mathit{iComp}) \\
\Uparrow^e (Q) & = & \{(i, \mathit{comp}) \mid \forall (j, \mathit{cesd}) \in Q,\, \mathcal{O}\, e\, (i, \mathit{comp})\, (j, \mathit{cesd})\}
\end{array}
$$

To explain the notation: down arrows translate positive predicates (over values and computations) into negative ones (over contexts), whilst up arrows go the other way. We use single arrows for the operations relating to values and double arrows for those relating to computations.

Now we can start relating the low-level machine to the high-level semantics. If D is a cpo and $RD_i \subseteq \mathit{Value} \times D$ is a $\mathbb{N}$-indexed relation between machine values and elements of D then define the indexed relation $[RD]_n \subseteq \mathit{Value} \times D_\perp$ by

$$[RD]_n = \{(v, dv) \mid \exists d \in D, [d] \sqsubseteq dv \wedge (v, d) \in RD_n\}$$

If, furthermore, S is a cpo and $RS_i \subseteq \mathit{Env} \times S$ an indexed relation between machine environments and elements of S, then we define an indexed relation

$$
\begin{array}{l}
(RS \overset{\lhd}{\to} RD_\perp)_i \ \subseteq\ \mathit{Comp} \times (S \Rightarrow D_\perp) \\
=\ \{(\mathit{comp}, \mathit{df}) \mid \forall k \leq i, \forall (e, \mathit{de}) \in RS_k, \\
\qquad (k, \mathit{comp}) \in \Uparrow^e (\downarrow^e (\{(j, v) \mid (v,\ \mathit{df}\ \mathit{de}) \in [RD]_j\}))\}
\end{array}
$$

which one should see as the relational action of the lift monad, relative to a relation on environments. The definition looks rather complex, but the broad shape of the definition is 'logical': relating machine computations to denotational continuous maps just when RS-related environments yield $[RD]$-related results. Then there is a little extra complication caused by threading the step indices around, but this is also of a standard form: computations are in the relation at i when they take k-related arguments to k-related results for all $k \leq i$. Finally, we use biorthogonality to close up on the right hand side of the arrow; rather than making an intensional direct-style definition that the computation 'yields' a value v that is related to the denotational application $\mathit{df}\ \mathit{de}$, we take the set of all such related results, flip it across to an orthogonal set of contexts with $\downarrow^e (\cdot)$ and then take that back to a set of computations with $\Uparrow^e (\cdot)$.

A special case of indexed relations between machine environments and denotational values is that for the empty environment. We define $I_i \subseteq \mathit{Env} \times 1$, where 1 is the one-point cpo, by $I_i = \{([], *)\}$. We can now define the 'real' indexed logical relation of approximation between machine values and domain elements

$$\lhd^t_i \subseteq \mathit{Value} \times [\![t]\!]$$

where t is a type and i is a natural number index like this:

$$\begin{aligned}
\trianglelefteq_i^{\mathtt{Int}} &= \{(\underline{n}, n) \mid n \in \mathbb{N}\} \\
\trianglelefteq_i^{\mathtt{Bool}} &= \{(\underline{0}, \mathit{false})\} \cup \{(\underline{n+1}, \mathit{true}) \mid n \in \mathbb{N}\} \\
\trianglelefteq_i^{t \times t'} &= \{(\mathtt{PR}\,(v_1, v_2), (dv_1, dv_2)) \mid (v_1, dv_1) \in \trianglelefteq_i^t \wedge \\
&\qquad\qquad (v_2, dv_2) \in \trianglelefteq_i^{t'}\} \\
\trianglelefteq_i^{t \to t'} &= \{(f, df) \mid \forall k \leq i, \forall (v, dv) \in \trianglelefteq_k^t, \\
&\qquad ((\mathtt{[App]}, [v, f]),\ \lambda * : 1.df\ dv) \in (I \xrightarrow{\triangleleft} (\trianglelefteq_v^{t'})_\perp)_k\}
\end{aligned}$$

This says that machine integers approximate the corresponding denotational ones, the machine zero approximates the denotational 'false' value, and all non-zero machine integers approximate the denotational 'true' value, reflecting the way in which the low-level conditional branch instruction works and the way in which we compile source-level booleans. Pair values on the machine approximate denotational pairs pointwise.

As usual, the interesting case is that for functions. The definition says that a machine value f and a semantic function df are related at type $t \to t'$ if whenever v is related to dv at type t, the computation whose code part is a single application instruction and whose stack part is the list $[v, f]$ is related to the constantly $(df\ dv)$ function, of type $1 \Rightarrow [\![t']\!]_\perp$, by the monadic lifting of the approximation relation at type t'.

Having defined the relation for values, we lift it to environments in the usual pointwise fashion. If Γ is $x_1 : t_1, \ldots, x_n : t_n$ then $\trianglelefteq_i^\Gamma \subseteq \mathit{Env} \times [\![\Gamma]\!]$ is given by

$$\trianglelefteq_i^\Gamma = \{([v_1, \ldots, v_n], (d_1, \ldots, d_n)) \mid \forall l, (v_l, d_l) \in \trianglelefteq_i^{t_l}\}$$

For computations in context, we define $\trianglelefteq_i^{\Gamma,t} \subseteq \mathit{Comp} \times ([\![\Gamma]\!] \Rightarrow [\![t]\!]_\perp)$ using the monadic lifting again

$$\trianglelefteq_i^{\Gamma,t} = (\trianglelefteq_e^\Gamma \xrightarrow{\triangleleft} (\trianglelefteq_v^t)_\perp)_i$$

These relations are antimonotonic in the step indices and monotone in the domain-theoretic order:

Lemma 1.

1. *If* $(v, d) \in \trianglelefteq_i^t$, $d \sqsubseteq d'$ *and* $j \leq i$ *then* $(v, d') \in \trianglelefteq_j^t$.
2. *If* $(e, \rho) \in \trianglelefteq_i^\Gamma$, $\rho \sqsubseteq \rho'$ *and* $j \leq i$ *then* $(e, \rho') \in \trianglelefteq_j^\Gamma$.
3. *If* $(\mathit{comp}, f) \in \trianglelefteq_i^{\Gamma,t}$, $f \sqsubseteq f'$ *and* $j \leq i$ *then* $(\mathit{comp}, f') \in \trianglelefteq_j^{\Gamma,t}$.

and the relation on computations extends that on values:

Lemma 2. *If* $(v, d) \in \trianglelefteq_i^t$ *then* $(\widehat{v}, \lambda * : 1.[d]) \in \trianglelefteq_i^{[],t}$.

The non-indexed versions of the approximation relations are then given by universally quantifying over the indices. (We also switch to infix notation for our relations at this point.)

$$\begin{aligned}
v \trianglelefteq^t d &\iff^{\text{def}} \forall i, v \trianglelefteq_i^t d \\
e \trianglelefteq^\Gamma \rho &\iff^{\text{def}} \forall i, e \trianglelefteq_i^\Gamma \rho \\
\mathit{comp} \trianglelefteq^{\Gamma,t} \mathit{df} &\iff^{\text{def}} \forall i, \mathit{comp} \trianglelefteq_i^{\Gamma,t} \mathit{df}
\end{aligned}$$

5.2 Approximating Operational By Denotational

Our second logical relation captures what it means for a denotational value to be 'less than or equal to' a machine computation. This way around we will again use biorthogonality, but this time with respect to the observation of termination. This is intuitively reasonable, as showing that the operational behaviour of a program is at least as defined as some domain element will generally involve showing that reductions terminate. We will not use operational step-indexing to define the relation this way around, but an explicit admissible closure operation will play a similar rôle.

For $e \in \mathit{Env}$, $\mathit{comp} \in \mathit{Comp}$ and $\mathit{cesd} \in \mathit{CESD}$ our termination observation is defined by

$$\mathcal{T}\ e\ \mathit{comp}\ \mathit{cesd} \iff^{\text{def}} (\mathit{comp} \bowtie (e \smile \mathit{cesd})) \mapsto^* \lightning$$

which we again lift to values $v \in \mathit{Value}$:

$$\mathcal{T}\ e\ v\ \mathit{cesd} \iff^{\text{def}} \mathcal{T}\ e\ \widehat{v}\ \mathit{cesd}$$

and again the observations generate two e-parameterized Galois connections, one between predicates on values and predicates on contexts, and the other between predicates on computations and predicates on contexts. Once more we use two of the four maps:

$$\begin{aligned}
\underline{\downarrow}^e(\cdot) &: \mathbb{P}(\mathit{Value}) \to \mathbb{P}(\mathit{CESD}) \\
\underline{\downarrow}^e(P) &= \{\mathit{cesd} \mid \forall v \in P, \mathcal{T}\ e\ v\ \mathit{cesd}\} \\
\overline{\Uparrow}^e(\cdot) &: \mathbb{P}(\mathit{CESD}) \to \mathbb{P}(\mathit{Comp}) \\
\overline{\Uparrow}^e(Q) &= \{\mathit{comp} \mid \forall \mathit{cesd} \in Q, \mathcal{T}\ e\ \mathit{comp}\ \mathit{cesd}\}
\end{aligned}$$

to define a relational action for the lift monad. If S and D are cpos, $RS \subseteq \mathit{Env} \times S$ and $RD \subseteq \mathit{Value} \times D$, then define

$$\begin{aligned}
(RS \xrightarrow{\triangleright} RD_\perp) &\subseteq \mathit{Comp} \times (S \Rightarrow D_\perp) \\
&= \{(\mathit{comp}, \mathit{df}) \mid \forall (e, de) \in RS, \forall d, [d] \sqsubseteq (\mathit{df}\ de) \\
&\qquad \implies \mathit{comp} \in \overline{\Uparrow}^e(\underline{\downarrow}^e(\{v \mid (v, d) \in RD\}))\}
\end{aligned}$$

which follows a similar pattern to our earlier definition, in using biorthogonality on the right hand side of the arrow: starting with all values related to d, flipping that over to the set of contexts that terminate when plugged into any of those values and then coming back to the set of all computations that terminate when plugged into any of those contexts.

Now we can define the second logical relation

$$\trianglerighteq^t \subseteq \mathit{Value} \times [\![t]\!]$$

by induction on the type t:

$$\begin{aligned}
\trianglerighteq^{\mathtt{Int}} &= \{(\underline{n}, n) \mid n \in \mathbb{N}\} \\
\trianglerighteq^{\mathtt{Bool}} &= \{(\underline{0}, \mathit{false})\} \cup \{(\underline{n+1}, \mathit{true}) \mid n \in \mathbb{N}\} \\
\trianglerighteq^{t \times t'} &= \{(\mathtt{PR}\,(v_1, v_2), (dv_1, dv_2)) \mid (v_1, dv_1) \in \trianglerighteq^t \wedge \\
&\qquad\qquad (v_2, dv_2) \in \trianglerighteq^{t'} \\
\trianglerighteq^{t \to t'} &= \{(f, df) \mid \forall (v, dv) \in \trianglerighteq^t, \\
&\qquad ((\mathtt{[App]}, [v, f]),\ \lambda * : 1.df\ dv) \in (I \xrightarrow{\triangleright} (\trianglerighteq^{t'})_\perp)\}
\end{aligned}$$

This relation also lifts pointwise to environments. For $\Gamma = x_1 : t_1, \ldots, x_n : t_n$ define $\trianglerighteq^\Gamma \subseteq \mathit{Env} \times [\![\Gamma]\!]$ by

$$\trianglerighteq^\Gamma = \{([v_1, \ldots, v_n], (dv_1, \ldots, dv_n)) \mid \forall l, (v_l, dv_l) \in \trianglerighteq^{t_l}\}$$

and then for computations in context, $\trianglerighteq^{\Gamma,t} \subseteq ([\![\Gamma]\!] \Rightarrow [\![t]\!]_\perp)$ is given by

$$\trianglerighteq^{\Gamma,t} = (\trianglerighteq^\Gamma \xrightarrow{\triangleright} (\trianglerighteq^t)_\perp)$$

For a fixed v, it turns out that $\{d \mid (v, d) \in \trianglerighteq^t\}$ is not always admissible. So, considering a chain of domain elements as an approximation to its least upper bound, we take the ideal closure:

$$\begin{aligned}
v \succeq^t dv &\iff^{\text{def}} \exists \langle d_i \rangle, dv \sqsubseteq \bigsqcup_i d_i \wedge \forall i, v \trianglerighteq^t d_i \\
e \succeq^\Gamma de &\iff^{\text{def}} \exists \langle de_i \rangle, de \sqsubseteq \bigsqcup_i de_i \wedge \forall i, e \trianglerighteq^\Gamma de_i \\
\mathit{comp} \succeq^{\Gamma,t} \mathit{df} &\iff^{\text{def}} \exists \langle \mathit{df}_i \rangle, \mathit{df} \sqsubseteq \bigsqcup_i \mathit{df}_i \wedge \forall i, \mathit{comp} \trianglerighteq^{\Gamma,t} \mathit{df}_i
\end{aligned}$$

where the notation $\langle d_i \rangle$ means an ω-chain.

It is easy to see that the $\succeq$ relations are all down-closed on the denotational side:

Lemma 3.

1. *If* $v \succeq^t d$ *and* $d' \sqsubseteq d$ *then* $v \succeq^t d'$
2. *If* $e \succeq^\Gamma \rho$ *and* $\rho' \sqsubseteq \rho$ *then* $e \succeq^\Gamma \rho'$
3. *If* $comp \succeq^{\Gamma,t} f$ *and* $f' \sqsubseteq f$ *then* $comp \succeq^{\Gamma,t} f'$

and the relation for computations extends that for values:

Lemma 4. *If* $v \succeq^t d$ *then* $\widehat{v} \succeq^{[],t} (\lambda * : 1.[d])$.

5.3 Realizability and Equivalence

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

$$\begin{array}{rcl} v \models^t d & \stackrel{\text{def}}{\iff} & v \preceq^t d \ \wedge \ v \succeq^t d \\ e \models^\Gamma de & \stackrel{\text{def}}{\iff} & e \preceq^\Gamma de \ \wedge \ e \succeq^\Gamma de \\ comp \models^{\Gamma,t} df & \stackrel{\text{def}}{\iff} & comp \preceq^{\Gamma,t} df \ \wedge \ comp \succeq^{\Gamma,t} df \end{array}$$

and these relations then induce typed notions of equivalence between machine components. We just give the case for computations

$$\begin{array}{rl} & \Gamma \vdash comp_1 \sim comp_2 : t \\ \stackrel{\text{def}}{\iff} & \exists df \in (\llbracket \Gamma \rrbracket \Rightarrow \llbracket t \rrbracket_\perp), comp_1 \models^{\Gamma,t} df \ \wedge comp_2 \models^{\Gamma,t} df \end{array}$$

and we overload this notation to apply to simple pieces of code too:

$$\Gamma \vdash c_1 \sim c_2 : t \stackrel{\text{def}}{\iff} \Gamma \vdash \widehat{c_1} \sim \widehat{c_2} : t$$

So now we have a notion of what it means for a piece of SECD machine code to be in the semantic interpretation of a source language type, and what it means for two pieces of code to be equal when considered at that type. Clearly, we are going to use this to say something about the correctness of our compilation scheme, but note that the details of just what code the compiler produces have not really shown up at all in the definitions of our logical relations: it is only the interfaces to compiled code – the way in which integers, booleans and pairs are encoded and the way in which function values are tested via application – that are mentioned in the logical relation. In fact, equivalence classes of our low-level relations can be seen as defining a perfectly good compositional denotational semantics for the source language in their own right.

A feature of this semantics is that there are no statements about what (non-observable) intermediate configurations should look like. For example, we never say that when a function is entered with a call stack that looks like 'x', then eventually one reaches a configuration that looks like 'y'. At the end of the day, all we ever talk about is termination and divergence of complete configurations, which we need to connect with the intended behaviour of closed programs of ground type, which we do by considering a range of possible external test contexts, playing the role of top-level continuations.

If $c \in Code$ then we say c diverges unconditionally if

$$\forall cesd, (\widehat{c} \bowtie cesd) \mapsto^\omega$$

The following says that if a piece of code realizes the denotational bottom value at any type, then it diverges unconditionally:

Lemma 5 (Adequacy for bottom)**.** *For any* $c \in Code$ *and type* t*, if*

$$c \models^{[],t} (\lambda * : 1.\perp_{\llbracket t \rrbracket})$$

then c *diverges unconditionally.*

For ground type observations, we say a computation $comp$ converges to a particular integer value n if plugging it into an arbitrary context equiterminates with plugging n into that context:

$$\begin{array}{c} \forall cesd, \ \widehat{n} \bowtie cesd \mapsto^* \not\downarrow \implies comp \bowtie cesd \mapsto^* \not\downarrow \\ \wedge \ \widehat{n} \bowtie cesd \mapsto^\omega \implies comp \bowtie cesd \mapsto^\omega . \end{array}$$

And we can then show that if a piece of code realizes a non-bottom element $[n]$ of $\llbracket \texttt{Int} \rrbracket_\perp$ in the empty environment, then it converges to n:

Lemma 6 (Adequacy for ground termination)**.** *For any* $c \in Code$*, if*

$$c \models^{[],t} (\lambda * : 1.[n])$$

then $\widehat{c}$ *converges to* n*.*

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.

We finally show the compositionality of our realizability semantics.

Lemma 7 (Compositionality for application)**.** *For any* $cf, cx \in Code$ *and* $df \in \llbracket \Gamma \rrbracket \Rightarrow (\llbracket t \rrbracket \Rightarrow \llbracket t' \rrbracket_\perp)_\perp, dx \in \llbracket \Gamma \rrbracket \Rightarrow \llbracket t \rrbracket_\perp$*, if*

$$cf \models^{\Gamma, t \to t'} df \ \wedge \ cx \models^{\Gamma,t} dx$$

then

$$cf +\!\!+ cx +\!\!+ [\texttt{App}] \quad \models^{\Gamma,t'} \quad \lambda de : \llbracket \Gamma \rrbracket. \ (df \ de) \circledcirc (dx \ de)$$

where $\circledcirc$ *denotes the lifted application.*

6. Applications

In this section, we illustrate the kind of results one can establish using the logical relations of the previous section.

6.1 Compiler Correctness

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

Theorem 1.

1. *For all* Γ*,*V*,*t*, if* $\Gamma \vdash V : t$ *then*
$$(\!|\Gamma \vdash V : t|\!) \ \preceq^{\Gamma,t} \ [\,\llbracket \Gamma \vdash V : t \rrbracket\,]$$
2. *For all* Γ*,*M*,*t*, if* $\Gamma \vdash M : t$ *then*
$$(\!|\Gamma \vdash M : t|\!) \ \preceq^{\Gamma,t} \ \llbracket \Gamma \vdash M : t \rrbracket$$

The two parts are proved simultaneously by induction on typing derivations, as in most logical relations proofs. In the case for recursive functions, there is a nested induction over the step indices.

Theorem 2.

1. *For all* Γ*,*V*,*t*, if* $\Gamma \vdash V : t$ *then*
$$(\!|\Gamma \vdash V : t|\!) \ \succeq^{\Gamma,t} \ [\,\llbracket \Gamma \vdash V : t \rrbracket\,]$$
2. *For all* Γ*,*M*,*t*, if* $\Gamma \vdash M : t$ *then*
$$(\!|\Gamma \vdash M : t|\!) \ \succeq^{\Gamma,t} \ \llbracket \Gamma \vdash M : t \rrbracket$$

This is another simultaneous induction on typing derivations. This time, the proof for recursive functions involves showing that each of the domain elements in the chain whose limit is the denotation is in the relation and then concluding that the fixpoint is in the relation by admissibility.

Corollary 1.

1. *For all* Γ*,*V*,*t*, if* $\Gamma \vdash V : t$ *then*
$$(\!|\Gamma \vdash V : t|\!) \ \models^{\Gamma,t} \ [\,\llbracket \Gamma \vdash V : t \rrbracket\,]$$
2. *For all* Γ*,*M*,*t*, if* $\Gamma \vdash M : t$ *then*
$$(\!|\Gamma \vdash M : t|\!) \ \models^{\Gamma,t} \ \llbracket \Gamma \vdash M : t \rrbracket$$

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

Corollary 2. *For any M with $[] \vdash M : \mathtt{Int}$,*

- *If $[\![[] \vdash M : \mathtt{Int}]\!] = \bot$ then $(\!|[] \vdash M : \mathtt{Int}|\!)$ diverges unconditionally.*
- *If $[\![[] \vdash M : \mathtt{Int}]\!] = [n]$ then $(\!|[] \vdash M : \mathtt{Int}|\!)$ converges to n.*

which follows by the adequacy lemmas above. And of course, by composing with the result that the denotational semantics is adequate with respect to the operational semantics of the source language, one obtains another corollary, that the operational semantics of complete source programs agrees with that of their compiled versions. 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 Low-level Equational Reasoning

In this section we give some simple examples of typed equivalences one can prove on low-level code. We first define some macros for composing SECD programs. If $c, cf, cx \in \mathit{Code}$ then define

$$\begin{array}{rcl} \mathtt{LAMBDA}(c) & = & [\mathtt{PushC}\,(c +\!\!+ [\mathtt{Ret}])] \\ \mathtt{APP}(cf, cx) & = & cf +\!\!+ cx +\!\!+ [\mathtt{App}] \end{array}$$

6.2.1 Example: Commutativity of addition

Define the following source term

$$\mathit{plussrc} \;=\; (\text{Fix}\; f\, x \;=\; [\text{Fix}\; g\, y \;=\; x + y])$$

and its compiled code

$$\mathit{pluscode}(\Gamma) \;=\; (\!|\Gamma \vdash \mathit{plussrc} : \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int}|\!).$$

Now we can show the following:

Lemma 8. *For any Γ, for any $c_1, c_2 \in \mathit{Code}$, and $dc_1, dc_2 \in [\![\Gamma]\!] \Rightarrow [\![\mathtt{Int}]\!]_\perp$, if*

$$c_1 \models^{\Gamma,\mathtt{Int}} dc_1 \quad \textit{and} \quad c_2 \models^{\Gamma,\mathtt{Int}} dc_2$$

then

$$\begin{array}{l} \Gamma \vdash \mathtt{APP}(\mathtt{APP}(\mathit{pluscode}\,\Gamma,\, c_1),\, c_2) \\ \qquad \sim \mathtt{APP}(\mathtt{APP}(\mathit{pluscode}\,\Gamma,\, c_2),\, c_1) : \mathtt{Int} \end{array}$$

In other words, for any code fragments c_1, c_2 that are in the interpretation of the source language type `Int`, manually composing those fragments with the code produced by the compiler for the curried addition function in either order yields equivalent behaviour of type `Int`.

6.2.2 Example: First projection

Define the source term

$$\mathit{projfstsrc} \;=\; (\text{Fix}\; f\, x \;=\; [\text{Fix}\; g\, y \;=\; x])$$

and the compiled code

$$\mathit{projfstcode}(\Gamma, t, t') \;=\; (\!|\Gamma \vdash \mathit{projfstsrc} : t \to t' \to t|\!)$$

then

Lemma 9. *For any Γ, t,t, for any $c_1, c_2 \in \mathit{Code}$ and $dc_1 \in [\![\Gamma]\!] \Rightarrow [\![t]\!]_\perp$ and $dc_2 \in [\![\Gamma]\!] \Rightarrow [\![t']\!]_\perp$, if*

$$c_1 \models^{\Gamma,t} dc_1 \quad \textit{and} \quad c_2 \models^{\Gamma,t'} dc_2$$

and furthermore

$$\forall de \in [\![\Gamma]\!],\, \exists dv \in [\![t']\!]\,(dc_2\; de) = [dv]$$

which says that the code c_2 realizes some total *denotational computation of type t' in context Γ, then*

$$\Gamma \vdash \mathtt{APP}(\mathtt{APP}(\mathit{projfstcode}(\Gamma, t, t'),\, c_1),\, c_2) \;\sim\; c_1 : t.$$

This says that the compiled version of *projfstsrc* behaves like the first projection, *provided* that the second argument does not diverge.

6.2.3 Example: Optimizing iteration

Our last example is slightly more involved, and makes interesting use of the non-functional equality test in the target language.

We start by compiling the identity function on integers

$$\begin{array}{rcl} \mathit{idsrc} & = & \text{Fix}\; \mathit{id}\, x \;=\; x \\ \mathit{idcode}(\Gamma) & = & (\!|\Gamma \vdash \mathit{idsrc} : \mathtt{Int} \to \mathtt{Int}|\!) \end{array}$$

and then define a higher-order function *appnsrc* that takes a function f from integers to integers, an iteration count n and an integer v, and returns f applied n times to v. We present the definition in ML-like syntax rather than our ANF language to aid readability:

```
fun f => letrec apf n = fun v =>
   if n > 0 then
     f (apf (n-1) v)
   else v
```

and we let

$$\mathit{appncode}(\Gamma) \;=\; (\!|\Gamma \vdash \mathit{appnsrc} : (\mathtt{Int} \to \mathtt{Int}) \to \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int}|\!)$$

Now we define a handcrafted optimized version, *appnopt* in the SECD language, which would, if one could write it, look something like this in an ML-like syntax:

```
fun f => fun n => fun v =>
   if f = idcode
   then v
   else appncode f n v
```

The optimized code checks to see if it has been passed the literal closure corresponding to the identity function, and if so simply returns v without doing any iteration.

We are able to show that for any Γ,

$$\Gamma \vdash \mathit{appnoptcode}(\Gamma) \sim \mathit{appncode}(\Gamma) : (\mathtt{Int} \to \mathtt{Int}) \to \mathtt{Int} \to \mathtt{Int}$$

showing that the optimized version, which could not be written in the source language, is equivalent to the unoptimized original.

7. Discussion

We have given a realizability relation between the domains used to model a simply-typed functional language with recursion and the low level code of an SECD machine with non-functional features. This relation was used to establish a semantic compiler correctness result and to justify typed equational reasoning on handcrafted low-level code. The relations make (we modestly believe…) elegant use of biorthogonality and step-indexing, and the work sheds interesting new light on the interaction of these two useful constructions.

As we said in the introduction, there are many other compiler correctness proofs in the literature, but they tend not to be so compositional or semantic in character as the present one. The classic work on the VLISP verified Scheme compiler by Guttman et al. (1995) is very close in character, being based on relating a denotational semantics to, ultimately, real machine code. The untyped PreScheme language treated in that work is significantly more realistic than the toy language of the present work, though the proofs were not mechanized. The denotational semantics used there was in CPS and the main emphasis was on the behaviour of complete programs. Chlipala (2007) has also used Coq to formalize

a correctness relation between a high-level functional language and low-level code, though in that case the source language is total and so can be given an elementary semantics in Sets.

For ML-like languages (pure or impure), contextual equivalence at higher types is highly complex, depending subtly on exactly what primitives one allows. This is why, as we said in the introduction, we feel that fully abstract compilation might not be quite the right thing to aim for. For example, Longley (1999) shows that there are extensionally pure (and even useful) functionals that are only defineable in the presence of impure features, such as references. Adding such functionals is not conservative – they refine contextual equivalence at order four and above in pure CBV languages – yet it seems clear that they will be implementable in many low-level machines. Complicating one's specifications to rule out these exotic programs in pursuit of full abstraction is not obviously worthwhile: it seems implausible that any compiler transformations or information flow policies will be sensitive to the difference. As another example, the presence of strong reflective facilities at the low-level, such as being able to read machine code instructions, might well make parallel-or defineable; this would obviously break full abstraction with respect to the source language, but we might well wish to allow it.[3]

The case against full abstraction becomes stronger when one considers that one of our aims is to facilitate semantically type safe linking of code produced from *different* programming languages. There is a fair degree of 'wiggle room' in deciding just how strong the semantic assumptions and guarantees should be across these interfaces. They should be strong enough to support sound and useful reasoning from the point of view of one language, but not insist on what one might almost think of as 'accidental' properties of one language that might be hard to interpret or ensure from the point of view of others.

The Coq formalization of these results was pleasantly straightforward. Formalizing the SECD machine, compiler, logical relations and examples, including the compiler correctness theorem, took a little over 4000 lines, not including the library for domain theory and the semantics of the source language. The extra burden of mechanized proof seems fairly reasonable in this case, and the results, both about the general theory and the examples, are sufficiently delicate that our confidence in purely paper proofs would be less than complete.

There are many obvious avenues for future work, including the treatment of richer source languages and type systems, and of lower-level target languages. We intend particularly to look at source languages with references and polymorphism and at a target machine like the idealized assembly language of our previous work. We would also like to give low-level specifications that are more independent of the source language - the current work doesn't mention source language terms, but does still talk about particular cpos. We would like to express essetially the same constraints in a more machine-oriented relational Hoare logic which might be more language neutral and better suited for independent verification. It would also be interesting to look at type systems that are more explicitly aimed at ensuring secure information flow.

Acknowledgments

Thanks to Andrew Kennedy, Neel Krishnaswami, Nicolas Tabareau and Carsten Varming for many useful discussions. Extra thanks to Andrew and Carsten for their work on the Coq formalization of the source language and its denotational semantics.

[3] Discussing the sense in which these two candidate full abstraction-breaking operations are incompatible would take us too far afield, however.

References

M. Abadi. $\top\top$-closed relations and admissibility. *Mathematical Structures in Computer Science*, 10(3), 2000.

M. Abadi. Protection in programming-language translations. In *25th International Colloquium on Automata, Languages and Programming (ICALP)*, 1998.

A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *Proc. 13th ACM International Conference on Functional Programming (ICFP '08)*. ACM, 2008.

A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.

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.

A.W. Appel, P.-A. Melliès, C.D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. *Proc. 34th ACM Symposium on Principles of Programming Languages (POPL '07)*, pages 109–122, 2007.

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

N. Benton and N. Tabareau. Compiling functional types to relational specifications for low level imperative code. In *TLDI*, 2009.

N. Benton and U. Zarfaty. Formalizing and verifying semantic type soundness of a simple compiler. In *Proc. 9th ACM International Conference on Principles and Practice of Declarative Programming (PPDP '07)*, 2007.

N. Benton, A. Kennedy, and C. Varming. Some domain theory and denotational semantics in Coq, 2009. To appear.

A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proc. 2007 ACM Conference on Programming Language Design and Implementation (PLDI '07)*, 2007.

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

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

A. Kennedy. Securing the .NET programming model. *Theoretical Computer Science*, 364(3), 2006.

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

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

X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. 33rd ACM Symposium on Principles of Programming Languages (POPL '06)*, 2006.

J. Longley. When is a functional program not a functional program? In *4th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1999.

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

G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):527–568, 1999.

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

G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

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

G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

A. The Problem With Realizing Recursion

This appendix gives a concrete example of how defining semantics for functional types as sets of untyped programs can run into problems in the case that the source language includes recursion and the untyped target includes non-functional operations. By non-functional operations, we particularly mean operations that examine the actual syntax or code of functional terms. The example we take here is a syntactic equality test: an operation that takes two arguments and returns true if the two (possibly functional) terms are syntactically equal and otherwise returns false.

Definition of VULE. As an example of a target language, we consider the call-by-value untyped lambda calculus with the syntactic equality test (VULE). We give the formal definition and operational semantics of VULE below. We fix a countable set $\mathbf{V}$ of variables. The set of values Val and the set of terms Term with variables in $\mathbf{V}$ are mutually inductively defined by the following rule:

$$\begin{array}{rcl} \mathsf{Val} & := & x \\ & \mid & \lambda x.\, t \\ \mathsf{Term} & := & v \\ & \mid & t\ s \\ & \mid & u \equiv_\alpha v \\ & \mid & \texttt{ERROR} \end{array}$$

where $x \in \mathbf{V}$, $u, v \in \mathsf{Val}$, $t, s \in \mathsf{Term}$. As usual, we assume that the appplication is left-associative. $\mathsf{FVar}(t)$ for any term t denotes the set of free variables in t, and $t\,[x \mapsto s]$ the capture-avoiding substitution of the term s for the variable x in the term t.

We also define some syntactic sugar for boolean operation and recursion.

$$\begin{array}{rcl} \texttt{TRUE} & \triangleq & \lambda x.\, \lambda y.\, x\ y \\ \texttt{FALSE} & \triangleq & \lambda x.\, \lambda y.\, y\ x \\ \texttt{if}\ t\ \texttt{then}\ s_1\ \texttt{else}\ s_2 & \triangleq & t\ (\lambda x.\, s_1)\ (\lambda x.\, s_2) \\ & & \quad \left(x \notin \mathsf{FVar}(s_1) \cup \mathsf{FVar}(s_2)\right) \\ \texttt{rec}\ t & \triangleq & \lambda x.\, (\lambda y.\, t\ (\lambda z.\, y\ y\ z))\ (\lambda y.\, t\ (\lambda z.\, y\ y\ z))\ x \\ & & \quad \left(x, y \notin \mathsf{FVar}(t), z \neq y\right) \end{array}$$

The small-step operational semantics of VULE is given as follows:

$$\begin{array}{rcl} t \rightsquigarrow t' \implies t\ s & \rightsquigarrow & t'\ s \\ s \rightsquigarrow s' \implies v\ s & \rightsquigarrow & v\ s' \\ (\lambda x.\, t)\ v & \rightsquigarrow & t\,[x \mapsto v] \\ u \approx_\alpha v \implies u \equiv_\alpha v & \rightsquigarrow & \texttt{TRUE} \\ u \not\approx_\alpha v \implies u \equiv_\alpha v & \rightsquigarrow & \texttt{FALSE} \\ \texttt{ERROR}\ v & \rightsquigarrow & \texttt{ERROR} \end{array}$$

where $x \in \mathbf{V}$, $u, v \in \mathsf{Val}$, $t, t', s, s' \in \mathsf{Term}$ and where $u \approx_\alpha v$ denotes that u is alpha-equivalent to v. For convenience, we define the multi-step relation $\overset{+}{\rightsquigarrow}$ by setting $t \overset{+}{\rightsquigarrow} t'$ iff the term t reaches t' in one or more steps.

We observe some properties of the syntactic sugar. The following hold for any $v \in \mathsf{Val}$ and $t, s_1, s_2 \in \mathsf{Term}$:

(Prop 1) TRUE, FALSE and rec t are values;

(Prop 2) if TRUE then s_1 else $s_2 \overset{+}{\rightsquigarrow} s_1$;

(Prop 3) if FALSE then s_1 else $s_2 \overset{+}{\rightsquigarrow} s_2$;

(Prop 4) $(\texttt{rec}\ t)\ v \overset{+}{\rightsquigarrow} t\ (\texttt{rec}\ t)\ v$.

Definition of the problem. We now define the problem. Let $\mathbf{Type}$ be the set of simple types defined by the rule

$$T \in \mathbf{Type} := \texttt{Bool} \mid T \to T\ .$$

As usual, the type constructor $\to$ is right-associative. We consider the following seemingly natural, and very minimal, conditions on a semantics $[\![-]\!] : \mathbf{Type} \to \mathscr{P}(\mathsf{Term})$.

(Asm 1) TRUE, FALSE $\in [\![\texttt{Bool}]\!]$.

(Asm 2) For a value u and $n \geq 1$, if $u\ \ v_1\ \ \ldots\ \ v_n \overset{+}{\rightsquigarrow}$ ERROR for some values $v_1 \in [\![A_1]\!]\ \ldots\ v_n \in [\![A_n]\!]$, then $u \notin [\![A_1 \to \ldots \to A_n \to B]\!]$.

(Asm 3) For a value u, if $u\ v \overset{+}{\rightsquigarrow} v$ for all values $v \in [\![A]\!]$, then $u \in [\![A \to A]\!]$.

For any semantics $[\![-]\!]$ satisfying the above conditions, we show that the Y-combinator Y defined by

$$Y \triangleq \lambda f.\, \texttt{rec}\ f$$

is not in the set $[\![(\texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool}]\!]$ as follows. For the value F defined by

$$F \triangleq \lambda g.\, \lambda f.\, \texttt{if}\ f \equiv_\alpha (\texttt{rec}\ g)\ \texttt{then ERROR else}\ f\ ,$$

one can easily make the following observations, for an arbitrary value $v \in \mathsf{Val}$:

(Obs 1):

$$(\texttt{rec}\ F)\ v \overset{+}{\rightsquigarrow} F\ (\texttt{rec}\ F)\ v \overset{+}{\rightsquigarrow} \begin{cases} \texttt{ERROR} & \text{if } v \approx_\alpha \texttt{rec rec}\ F \\ v & \text{otherwise} \end{cases}$$

(Obs 2): $(\texttt{rec rec}\ F)\ \texttt{TRUE} \overset{+}{\rightsquigarrow} (\texttt{rec}\ F)\ (\texttt{rec rec}\ F)\ \texttt{TRUE} \overset{+}{\rightsquigarrow} \texttt{ERROR TRUE} \rightsquigarrow \texttt{ERROR}$

(Obs 3): $Y\ (\texttt{rec}\ F)\ \texttt{TRUE} \rightsquigarrow (\texttt{rec rec}\ F)\ \texttt{TRUE} \overset{+}{\rightsquigarrow} \texttt{ERROR}$

From these observations, we can conclude that

(Con 1): $\texttt{rec rec}\ F \notin [\![\texttt{Bool} \to \texttt{Bool}]\!]$ by **(Asm 1)**, **(Asm 2)** and **(Obs 2)**;

(Con 2): $\texttt{rec}\ F \in [\![(\texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool}]\!]$ by **(Asm 3)**, **(Obs 1)** and **(Con 1)**;

(Con 3): $Y \notin [\![(\texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool}]\!]$ by **(Asm 1)**, **(Asm 2)**, **(Obs 3)** and **(Con 2)**.

If the source language includes non-terminating programs, the Y-combinator should be of type $((\texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool}$ because a sensible compiler may compile the Y-combinator of the source language into the Y-combinator of VULE. This shows that the above assumptions are too strong when recursion is considered.

Discussion. As the assumptions **(Asm 1)** and **(Asm 2)** cannot be too strong, the problematic assumption is the assumption **(Asm 3)**. The above example tells us that when one decides whether a value u has a type $A \to B$, it is not enough to test u on only valid arguments $v \in [\![A]\!]$. One also has to take values outside $[\![A]\!]$ into account. This is just what step-indexing will do for us.