Step-Indexing: The Good, the Bad and the Ugly

Nick Benton[1] and Chung-Kil Hur[2]⋆

[1] Microsoft Research Cambridge, UK
nick@microsoft.com
[2] PPS, Université Paris Diderot
Chung-Kil.Hur@pps.jussieu.fr

Abstract. Over the last decade, step-indices have been widely used for the construction of operationally-based logical relations in the presence of various kinds of recursion. We first give an argument that step-indices, or something like them, seem to be required for defining realizability relations between high-level source languages and low-level targets, in the case that the low-level allows egregiously intensional operations such as reflection or comparison of code pointers. We then show how, much to our annoyance, step-indices also seem to prevent us from exploiting such operations as aggressively as we would like in proving program transformations.

Introduction

Since their introduction by Appel and McAllester [4], step-indexed logical relations (both binary and unary) have been widely used for operational reasoning about about many kinds of 'difficult' language features, including recursive types [1], first-class references [2] and objects [10]. In all these cases, simple induction on types is inapplicable and one naturally finds oneself trying to construct relations as solutions to some contra- or mixed-variance recursive equation that, interpreted naively, would for simple cardinality reasons have only trivial, or even no, solutions. Denotational techniques (using minimal invariance properties of solutions to recursive domain equations [11] or metric constructions [9]) are an alternative approach, but are often perceived as involving more sophisticated mathematics and seem harder to apply to, for example, concurrency or unstructured low-level languages. Step-indexed relations are stratified by the number of execution steps available for testing relatedness: the more steps available, the finer the relation. Well-founded recursive definitions can then be made if the definition of the relation at level k only ever depends on itself at levels $j < k$, which is typically natural when examining a value (e.g. following a pointer or calling a function) takes at least one step.

Many authors have noticed the 'ugly' side of step-indices, which is that definitions and proofs have a tendency to become cluttered with extra indices and even arithmetic, which are really playing the role of 'construction lines'. This clutter has been largely alleviated by more recent progress in replacing the slightly vulgar concrete natural numbers with more abstract modal operators [5].

⋆ Research supported by Digiteo/Ile-de-France project COLLODI (2009-28HD).

We have employed step-indexing in some recent work on compiler correctness [8,7], as a way of capturing low-level analogues of denotational admissibility (closure under limits of chains) or operational unwinding (behaving well with respect to syntactic approximations to a fixpoint combinator) properties of high-level relations. Such admissibility properties play a well-understood role in high-level relational reasoning even when just term-level recursion or looping is present, and step-indexing seems a natural way to transfer those properties to low-level programs [6,12]. The first part of this talk derives from [7], and explains the 'good' side of step-indices for working with relations over low-level code. We show how the presence of non-extensional operations in an untyped model of computation can break a standard realizability interpretation of types when the source language includes term-level recursion, and how the extra tests required by a step-indexed interpretation fix the problem. The second part shows the 'bad': the step indexed interpretation requires certain bad computations not to go wrong 'too quickly', which ends up ruling out some clever low-level transformations that we believe *should* be valid, but which seem to optimize too well.

Defining the meaning of simple types via a logical predicate over terms in an untyped lambda-calculus based language is entirely standard. We'll consider what happens when we try to do the same thing over an untyped language in which terms can also be tested intensionally for α-equivalence, an operation which we regard as a proxy for the ability of machine code programs to compare code pointers for equality or to read executable instructions, Java programs to perform dynamic type tests or use reflection, or programs in popular dynamic languages to do all sorts of things that shouldn't be spoken of in polite company.

One's natural intuition is that the nature of logical relations is that one always says 'a thing is good if whenever it's put into a good context, the resulting behaviour is good', and that this means that the range of possible bad stuff in the untyped model of computation doesn't really affect the form of such a definition. And indeed, for interpreting the pure, total, simply typed calculus in the spirit of the Curry-Howard isomorphism, that intuition is quite correct. The issue arises when we try to apply the same ideas to interpret a typed language with recursion, in which (amongst other things) every type is inhabited by divergence, and we expect a fixpoint combinator to inhabit its natural types. It is well-known that everything works out straightforwardly, with only minor differences from the normalizing case, when the untyped model is the usual untyped lambda calculus. Let's see what happens when we instead use our lambda calculus with equality testing.

Abstract low-level language $\boldsymbol{\lambda_{ve}}$

As an abstract setting in which to investigate the issues raised by the presence of very intensional operations in low-level languages, we will work with an untyped lambda calculus, λ_{ve}, including reflective construct that tests arbitrary values for syntactic equality. 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 defined mutually inductively as follows:

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

where $x \in \mathbf{V}$. As usual, application 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 (implemented via a CBV fixed point combinator).[3]

$$\begin{aligned}\texttt{TRUE} &\triangleq \lambda x.\ \lambda y.\ x\ y \\ \texttt{FALSE} &\triangleq \lambda x.\ \lambda y.\ y\ x \\ \texttt{if}\ t\ \texttt{then}\ s_1\ \texttt{else}\ s_2 &\triangleq t\ (\lambda x.\ s_1)\ (\lambda x.\ s_2) \\ &\qquad\quad \big[x \notin \mathsf{FVar}(s_1) \cup \mathsf{FVar}(s_2)\big] \\ \texttt{rec}\ t &\triangleq \lambda x.\ (\lambda y.\ t\ (\lambda z.\ y\ y\ z))\ (\lambda y.\ t\ (\lambda z.\ y\ y\ z))\ x \\ &\qquad\quad \big[x, y \notin \mathsf{FVar}(t), z \neq y\big]\end{aligned}$$

The small-step call by value operational semantics is given as follows:

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

where $x \in \mathbf{V}$, $u, v \in \mathsf{Val}$, $t, t', s, s' \in \mathsf{Term}$ and where $u \approx_\alpha v$ means that u is alpha-equivalent to v. For convenience, we define the multi-step relations $\overset{k}{\rightsquigarrow}$ and $\overset{*}{\rightsquigarrow}$ by setting $t \overset{k}{\rightsquigarrow} t'$ iff the term t reaches t' in k steps; and $t \overset{*}{\rightsquigarrow} t'$ iff t reaches t' in zero or more steps. Note that `ERROR`, and terms in which it appears in an evaluation position, are stuck. We also define the proper (i.e. stuck-free) termination and divergence relations: $t \downarrow$ iff $\exists v.\ t \overset{*}{\rightsquigarrow} v$; and $t \uparrow$ iff $\forall k.\ \exists t'.\ t \overset{k}{\rightsquigarrow} t'$.

Standard logical relation

Due to the syntactic equality test, the obvious *untyped* notion of contextual equivalence for λ_{ve} collapses to the syntactic alpha equality [13]. For instance,

[3] One could add language constructs for booleans and recursion, rather than using these encodings, without affecting the arguments at all. This choice just keeps the basic calculus smaller.

the terms $\lambda x.\ x$ and $\lambda x.\ (\lambda y.\ y)\ x$ can be distinguished by the context `if` $[-] \equiv_\alpha \lambda x.\ x$ `then TRUE else` *div* for some divergent term *div*. Our aim, however, is to interpret types as relations that relate equivalent *well-behaved* untyped terms, where the requirement to be 'well-behaved' only involves behaviour when placed in appropriately well-behaved contexts. We expect to rule out contexts such as the one we just gave, and hence, for example, to relate the untyped identity function and its eta-expansion at the interpretation of any type of the form $T \to T$.

One may think that a completely standard logical relation will give such a notion of typed equivalence. Let us try it here. For simplicity, we only consider boolean and arrow types:

$$\mathbf{Type} \ni T \ := \ \texttt{Bool} \mid T \to T\ ;$$

and relate closed values and terms, which are those that have no free variables in them: $\approx^T \subseteq \ \mathsf{ClosedVal} \times \mathsf{ClosedVal}$ and $[\approx]^T \ \subseteq\ \mathsf{ClosedTerm} \times \mathsf{ClosedTerm}$ for $T \in \mathbf{Type}$ defined by a mutual induction on T:

$$\begin{aligned}
\approx^{\texttt{Bool}} &= \{\, (\texttt{TRUE}, \texttt{TRUE}), (\texttt{FALSE}, \texttt{FALSE}) \,\} \\
\approx^{T_1 \to T_2} &= \{\, (u, u') \mid \forall v \approx^{T_1} v'.\ u\ v\, [\approx]^{T_2}\, u'\ v' \,\} \\
[\approx]^T &= \{\, (t, t') \mid (t \uparrow\ \wedge t' \uparrow) \vee \\
&\qquad\qquad (\exists v, v'.\ t \ \overset{*}{\rightsquigarrow}\ v \ \wedge\ t' \ \overset{*}{\rightsquigarrow}\ v' \ \wedge\ v \approx^T v') \,\}
\end{aligned}$$

$[\approx]^T$ is symmetric by definition and its transitivity can be shown by proving the following proposition by induction over T:

$$\begin{gathered}
(\forall v \approx^T v'.\ v \approx^T v) \ \wedge\ (\forall t\, [\approx]^T\, t'.\ t\, [\approx]^T\, t) \wedge \\
\approx^T \ \text{ is transitive } \ \wedge\ [\approx]^T \ \text{ is transitive.}
\end{gathered}$$

Thus, $[\approx]^T$ forms a partial equivalence relation on $\mathsf{ClosedTerm}$; that is, an equivalence relation on

$$[\![T]\!] = \{\, t \mid t\, [\approx]^T\, t \,\} \subseteq \mathsf{ClosedTerm}\ .$$

One can think of $[\![T]\!]$ as the set of all good terms of type T and $[\approx]^T$ as the semantic equivalence on $[\![T]\!]$. This typed equivalence $[\approx]^T$ gives a compositional and adequate notion of equivalence in the following sense:

Theorem 1 (Compositionality).

1. $\forall t, s.\ t \in [\![T_1 \to T_2]\!] \ \wedge\ s \in [\![T_1]\!] \implies t\ s \in [\![T_2]\!]$.
2. $\forall t, t', s, s'.\ t\, [\approx]^{T_1 \to T_2}\, t' \ \wedge\ s\, [\approx]^{T_1}\, s' \implies t\ s\, [\approx]^{T_2}\, t'\ s'$.

Theorem 2 (Adequacy).

1. $\forall t \in [\![T]\!].\ t \downarrow\ \vee\ t \uparrow$.
2. $\forall t\, [\approx]^T\, t'.\ (t \downarrow\ \wedge t' \downarrow) \ \vee\ (t \uparrow\ \wedge t' \uparrow)$.

Note that it is the compositionality that guarantees that linking of good terms is safe (*i.e.*, it yields a good term).

Problem with standard logical relation

Though the standard logical relation gives an adequate notion of equivalence, it is not big enough to admit all sensibly good terms. As an example, we show that the fixed point combinator Y defined by

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

is *not* in the set

$$[\![((\texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool}]\!],$$

which means that the innocent Y-combinator is treated as a bad term and thus ruled out.

We first note that the semantics of types $[\![T]\!]$ defined via the logical relation satisfies the following properties.

(Prop 1) $\texttt{TRUE}, \texttt{FALSE} \in [\![\texttt{Bool}]\!]$.
(Prop 2) Given a value $u \in [\![A_1 \to \ldots \to A_n \to \texttt{Bool}]\!]$ and values

$$v_1 \in [\![A_1]\!],\ \ldots,\ v_n \in [\![A_n]\!],$$

the application of u to the v_is does not go wrong: $u\ v_1\ \ldots\ v_n \overset{*}{\not\rightsquigarrow} \texttt{ERROR}$.
(Prop 3) For a value u, if $u\ v \overset{*}{\rightsquigarrow} v$ for all values $v \in [\![T]\!]$, then $u \in [\![T \to T]\!]$.

Our argument will actually depend only on these (rather minimal) properties of the logical relation, not on any more specific details of its definition. Let the value F be defined by

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

and observe the following facts about the behaviour of F:

(Obs 1) For any value $v \in \mathsf{Val}$,

$$\begin{aligned}(\texttt{rec}\ F)\ v &\overset{2}{\rightsquigarrow} F\ (\texttt{rec}\ F)\ v \\ &\overset{5}{\rightsquigarrow} \begin{cases} \lambda x.\, \texttt{ERROR} & \text{if } v \approx_\alpha \texttt{rec}\ \texttt{rec}\ F \\ v & \text{otherwise} \end{cases}\end{aligned}$$

(Obs 2) $(\texttt{rec}\ \texttt{rec}\ F)\ v \overset{2}{\rightsquigarrow} (\texttt{rec}\ F)\ (\texttt{rec}\ \texttt{rec}\ F)\ v \overset{7}{\rightsquigarrow} (\lambda x.\, \texttt{ERROR})\ v \overset{1}{\rightsquigarrow} \texttt{ERROR}$, for $v \in \{\, \texttt{TRUE}, \texttt{FALSE} \,\}$
(Obs 3) $Y\ (\texttt{rec}\ F)\ \texttt{TRUE} \rightsquigarrow (\texttt{rec}\ \texttt{rec}\ F)\ \texttt{TRUE} \overset{*}{\rightsquigarrow} \texttt{ERROR}$

From these observations, we can conclude that

(Con 1) $\texttt{rec}\ \texttt{rec}\ F \notin [\![\texttt{Bool} \to \texttt{Bool}]\!]$ because $\texttt{TRUE} \in [\![\texttt{Bool}]\!]$ by **(Prop 1)**, $(\texttt{rec}\ \texttt{rec}\ F)\ \texttt{TRUE} \overset{*}{\rightsquigarrow} \texttt{ERROR}$ by **(Obs 2)**, and well-typed applications don't go wrong by **(Prop 2)**.

(Con 2) $\texttt{rec}\ F \in [\![(\texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool}]\!]$: because $\texttt{rec}\ F$ behaves as the identity on all values except $\texttt{rec rec}\ F$ by **(Obs 1)** and $\texttt{rec rec}\ F$ is *not* in $[\![\texttt{Bool} \to \texttt{Bool}]\!]$ by **(Con 1)**, $\texttt{rec}\ F$ must behave as the identity on all values that *are* in $[\![\texttt{Bool} \to \texttt{Bool}]\!]$ and thus we get the conclusion by **(Prop 3)**.

(Con 3) Now we see that

$$Y \notin [\![((\texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool}]\!]$$

because

$$\begin{array}{lr} Y\ (\texttt{rec}\ F)\ \texttt{TRUE} \overset{*}{\leadsto} \texttt{ERROR} & \textbf{(Obs 3)} \\ \texttt{rec}\ F \in [\![(\texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool}]\!] & \textbf{(Con 2)} \\ \texttt{TRUE} \in [\![\texttt{Bool}]\!] & \textbf{(Prop 1)} \end{array}$$

and well-typed applications don't go wrong by **(Prop 2)**.

This is not at all what we wanted! One would expect **(Con 2)** to be false, since F is clearly highly suspicious, and then the blameless Y could have the expected type. Our analysis of the problem is that **(Prop 3)** is the only one of the properties that could be modified in order to get the expected result. In short, just testing with 'good' arguments is actually insufficient grounds for concluding that a function is good: we need some extra tests on 'partially good' values, which is just what step-indexing will supply.

Step-indexed logical relation

We modify the standard logical relation using the idea of step-indexing and see how this fixes the above problem. The basic idea is that we consider terms that behaves well up to k number of steps as partially good terms up to k steps, and say that good functions are those that behaves well up to k steps for any arguments that are partially good up to k steps for $k \in \mathbb{N}$.

The step-indexed families of relations $\approx_k^T \subseteq \mathsf{ClosedVal}^2$ and $[\approx]_k^T \subseteq \mathsf{ClosedTerm}^2$ for $T \in \textbf{Type}$ and $k \in \mathbb{N}$ are defined by a mutual induction on T:

$$\begin{aligned} \approx_k^{\texttt{Bool}} &= \{ (\texttt{TRUE}, \texttt{TRUE}), (\texttt{FALSE}, \texttt{FALSE}) \} \\ \approx_k^{T_1 \to T_2} &= \{ (u, u') \mid \forall j \le k,\ v \approx_j^{T_1} v'.\ u\ v\ [\approx]_j^{T_2}\ u'\ v' \} \\ [\approx]_k^T &= \{ (t, t') \mid (\exists s, s'.\ t \overset{k}{\leadsto} s \wedge t' \overset{k}{\leadsto} s') \vee \\ &\qquad (\exists j, v, j', v'.\ t \overset{j}{\leadsto} v \wedge t' \overset{j'}{\leadsto} v' \wedge v \approx_{k-\min(j,j')}^T v') \} \end{aligned}$$

We take good terms as those that are partially good up to any number of steps:

$$[\approx]^T = \{ (t, t') \mid \forall k \in \mathbb{N}.\ t\ [\approx]_k^T\ t' \}\ .$$

As there seems no general reason to expect that the relations $[\approx]^T$ are transitive, we take a transitive closure, say $[\approx]^{T+}$, to obtain a notion of equivalence.

$$[\![T]\!] = \{ t \mid t\ [\approx]^T\ t \} \subseteq \mathsf{ClosedTerm}$$

$$[=]^{T} = \{ (t,t') \in [\![T]\!] \times [\![T]\!] \mid t\,[\approx]^{T+}\,t' \}$$

This typed equivalence is also compositional and adequate:

Theorem 3 (Compositionality).

1. $\forall t, s.\ t \in [\![T_1 \to T_2]\!] \ \wedge\ s \in [\![T_1]\!] \implies t\ s \in [\![T_2]\!]$.
2. $\forall t, t', s, s'.\ t\,[=]^{T_1 \to T_2}\,t' \ \wedge\ s\,[=]^{T_1}\,s' \implies t\ s\,[=]^{T_2}\,t'\ s'$.

Theorem 4 (Adequacy).

1. $\forall t \in [\![T]\!].\ t\downarrow \vee\ t\uparrow$.
2. $\forall t\,[=]^{T}\,t'.\ (t\downarrow \wedge\ t'\downarrow) \ \vee\ (t\uparrow \wedge\ t'\uparrow)$.

Furthermore, by simple calculation, one can show that

$$Y \in [\![((\mathtt{Bool} \to \mathtt{Bool}) \to \mathtt{Bool} \to \mathtt{Bool}) \to \mathtt{Bool} \to \mathtt{Bool}]\!]\ .$$

To see the reason intuitively, let us see how the suspicious term $\mathtt{rec}\ F$ is ruled out. Though $\mathtt{rec}\ \mathtt{rec}\ F$ is not in $[\![\mathtt{Bool} \to \mathtt{Bool}]\!]$, it is good up to 9 steps, *i.e.*, $\mathtt{rec}\ \mathtt{rec}\ F\,[\approx]_9^{\mathtt{Bool}\to\mathtt{Bool}}\,\mathtt{rec}\ \mathtt{rec}\ F$, because $(\mathtt{rec}\ \mathtt{rec}\ F)\ v$ for $v \in \{\mathtt{TRUE}, \mathtt{FALSE}\}$ takes at least 9 steps by **(Obs 2)**. Thus, if we assume that $\mathtt{rec}\ F$ is in $[\![(\mathtt{Bool} \to \mathtt{Bool}) \to \mathtt{Bool} \to \mathtt{Bool}]\!]$, then $(\mathtt{rec}\ F)\ (\mathtt{rec}\,\mathtt{rec}\ F)$ should be good up to 9 steps. As the application reduces to the value $\lambda x.\ \mathtt{ERROR}$ at 7 steps by **(Obs 1)**, the resulting value should be good up to 2 steps. However this is not the case because its application to the values TRUE and FALSE reduces to ERROR at one step. Thus we can conclude that the term $\mathtt{rec}\ F$ is not good, so the argument in the previous section does not apply to the step-indexed logical relation.

Problem with step-indexing

The step-indexed logical relation gives a well-behaved notion of equivalence, admitting all the 'obviously' well-typed terms (those which don't use equality testing) and validating most of the equivalences on those that one would expect. That's the 'good'. The 'bad' is that it seems to rule out more than we would really like it to. One would hope that not all uses of of low-level intensional operations would be forbidden, but only those that violate the contract on observable behaviour associated with our types. There are many practically useful transformations that safely exploit the ability to compare code pointers or read instructions, without changing observable behaviour. But we'll now show how a natural example of such a transformation, using our syntactic equality test to perform a 'clever' optimization, is ruled out by the step-indexed logical relation, so showing that step-indexing is still rather too intensional.

Consider the three terms I, Ω and A defined as follows:

$$\begin{aligned} I &:= \lambda x.\ (\lambda y.\ y)\ \ldots\ (\lambda y.\ y)\ x \qquad \text{(100 times)} \\ \Omega &:= \mathtt{rec}\ (\lambda f.\ \lambda x.\ f\ x) \\ A &:= \lambda f.\ \lambda x.\ \mathtt{if}\ f \equiv_\alpha I\ \mathtt{then}\ x\ \mathtt{else}\ f\ x \end{aligned}$$

I is an identity function but takes 100 steps when applied to values, Ω is a function that always diverges when applied to any value, and A takes two values f and v and essentially applies f to v, except that if f is syntactically equal to I, so A detects that it has been given an expensive version of the identity function, then A optimizes by just returning the argument v directly.

The term A seems to perform an entirely correct and useful optimization, but is unfortunately rejected by our step-indexed logical relation. To see why, let us assume that $A \in [\![\mathtt{Bool}^4]\!]$ for $\mathtt{Bool}^4 = (\mathtt{Bool} \rightarrow \mathtt{Bool}) \rightarrow \mathtt{Bool} \rightarrow \mathtt{Bool}$. Since both I and Ω take at least 50 steps (say) whenever applied to any value, we have that $I \approx_{50}^{\mathtt{Bool}\rightarrow\mathtt{Bool}} \Omega$. As A is a value and $A\,[\approx]^{\mathtt{Bool}^4}\,A$, we must have $A \approx_k^{\mathtt{Bool}^4} A$ for any k. In particular, $A \approx_{50}^{\mathtt{Bool}^4} A$, so $A\ I\,[\approx]_{50}^{\mathtt{Bool}\rightarrow\mathtt{Bool}}\,A\ \Omega$. Since

$$A\ I \overset{1}{\rightsquigarrow} (\lambda x.\ \mathtt{if}\ I \equiv_\alpha I\ \mathtt{then}\ x\ \mathtt{else}\ I\ x)$$

and

$$A\ \Omega \overset{1}{\rightsquigarrow} (\lambda x.\ \mathtt{if}\ \Omega \equiv_\alpha I\ \mathtt{then}\ x\ \mathtt{else}\ \Omega\ x),$$

that means

$$(\lambda x.\ \mathtt{if}\ I \equiv_\alpha I\ \mathtt{then}\ x\ \mathtt{else}\ I\ x) \approx_{49}^{\mathtt{Bool}\rightarrow\mathtt{Bool}} (\lambda x.\ \mathtt{if}\ \Omega \equiv_\alpha I\ \mathtt{then}\ x\ \mathtt{else}\ \Omega\ x).$$

Trivially, $\mathtt{TRUE} \approx_{49}^{\mathtt{Bool}} \mathtt{TRUE}$, and since $(\lambda x.\ \mathtt{if}\ I \equiv_\alpha I\ \mathtt{then}\ x\ \mathtt{else}\ I\ x)\ \mathtt{TRUE} \overset{3}{\rightsquigarrow} \mathtt{TRUE}$, we can easily deduce that $(\lambda x.\ \mathtt{if}\ \Omega \equiv_\alpha I\ \mathtt{then}\ x\ \mathtt{else}\ \Omega\ x)\ \mathtt{TRUE}$ must terminate, which clearly does not hold. Hence our original assumption that $A \in [\![\mathtt{Bool}^4]\!]$ must have been false.

The problem here is attributable to the extra intensional requirements imposed by step indexing (and which we argued were helpful in the previous section). It does not suffice to yield good behaviour in good contexts, but one must also yield approximately good (e.g. taking a certain number of steps without error) behaviour in approximately good contexts (e.g. ones supplying function arguments that take take some steps). The difficulty with A from the point of view of the step indexed relation is that it optimizes “too well” to respect the extra intensional requirements.

Conclusion

In the presence of both source-level recursion and very intensional low-level operations of the sort that are both present and practically exploited in real low-level machines, a naive attempt to transfer the kind of realizability interpretation of types that works for more structured untyped computational models goes wrong. Step-indexing fixes that problem, but still fails to allow realizers (like those Appel describes in [3]) that seriously exploit the intensional operations. Unfortunately, we do not yet have a better alternative.

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. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009.
3. A. Appel. Intensional equality ;-) for continuations. *ACM SIGPLAN Notices*, 31(2), 1996.
4. A. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5), 2001.
5. A.W. Appel, P.-A. Melliès, C.D. Richards, and J. Vouillon. A Very Modal Model of a Modern, Major, General Type System. *34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.
6. N. Benton. A typed, compositional logic for a stack-based abstract machine. In *Proc. 3rd Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, 2005.
7. N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *Proceedings of ICFP'09*, 2009.
8. 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.
9. L. Birkedal, K. Stovring, and J. Thamsborg. Realizability semantics of paramteric polymorphism, general references and recursive types. In *Proceedings of FOSSACS'09*, 2009.
10. C. Hritcu and J. Schwinghammer. A step-indexed semantics of imperative objects. *Logical Methods in Computer Science*, 5(4), 2009.
11. A. M. Pitts. Relational properties of domains. *Inf. Comput.*, 127, 1996.
12. G. Tan and A. Appel. A compositional logic for control flow. In *Proceedings of VMCAI'06*, 2006.
13. M. Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10, 1998.