# The Journey of Biorthogonal Logical Relations to the Realm of Assembly Code

Guilhem Jaber[1,2] and Nicolas Tabareau[3]

[1] ENS Cachan
[2] École des Mines de Nantes
[3] INRIA

**Abstract.** Logical relations appeared to be very fruitful for the development of modular proofs of compiler correctness. In this field, logical relations are parametrized by a high-level type system, and are even sometimes directly relating low level pieces of code to high-level programs. All those works rely crucially on biorthogonality to get extensionality and compositionality properties. But the use of biorthogonality in the definitions also complicates matters when it comes to operational correctness. Most of the time, such correctness results amount to show an unfolding lemma that makes reduction more explicit than in a biorthogonal definition. Unfortunately, unfolding lemmas are not easy to derive for rich languages and in particular for assembly code. In this paper, we focus on three different situations that enable to reach step-by-step the assembly code universe: the use of Curry-style polymorphism, the presence of syntactical equality in the language and finally an ideal assembly code with a notion of code pointer.

Logical relations—and more particularly biorthogonal logical relations—have shown to be very fruitful for the development of modular proofs of compiler correctness. Such biorthogonal logical relations are indexed by a high-level type system. When focusing on a particular high level language for compiler correctness, logical relations relate low level pieces of code to high-level programs [4,6]. But to give an intrinsic semantics to assembly code, they can also relate directly low level pieces of codes together [5]. In this paper, we will focus on this second line of work. All those works rely crucially on biorthogonality definitions to get extentionality and compositionality properties. Indeed, using it we can define good terms in a logical relation as those which interact well with good contexts, rather than defining them explicitly. More precisely, orthogonality is used to lift relations on values to relations on terms. Indeed, logical relations are defined in three layers. A relation on values $\mathcal{V}[\![\tau]\!]$ is defined for each type $\tau$ of the high-level type system, then relations on context

$$\mathcal{K}[\![\tau]\!] = \{(K_1, K_2) \mid \forall (v_1, v_2) \in \mathcal{V}[\![\tau]\!] . \mathcal{O}(K_1[v_1], K_2[v_2])\}$$

are obtained by considering contexts that interacts well with values of $\mathcal{V}[\![\tau]\!]$, which is called the orthogonal of $\mathcal{V}[\![\tau]\!]$. Orthogonality is defined with respect to

an abstract notion of observation $\mathcal{O}$ that will be equitermination in this paper. Finally, relations on terms

$$\mathcal{E}[\![\tau]\!] = \{(t_1, t_2) \mid \forall (K_1, K_2) \in \mathcal{K}[\![\tau]\!] . \mathcal{O}(K_1[t_1], K_2[t_2])\}$$

are obtained by taking the orthogonal of $\mathcal{K}[\![\tau]\!]$. But biorthogonality also complicates matters when it comes to operational correctness. Indeed, such results rely on the fact $\mathcal{V}[\![\tau]\!]$ and $\mathcal{E}[\![\tau]\!]$ match on values. Such a result is proved using the intuition that good contexts of type $\tau$ has to be seen as tests for type $\tau$. Then, one has to investigate the discriminating power of such contexts. For instance, when contexts are very weak, good terms can be any terms, and to the opposite, when they are very powerful, good terms are only good values.

In this article, we will focus on three different situations related to assembly code. First, we will work in the context of a Curry-style polymorphic $\lambda$-calculus, wondering if contexts should be allowed to test terms more than one time with different types. This is an important issue in the assembly code framework, because Church-style polymorphism is hardly expressible at this level. Then we will deal with contexts that can use intentional equality on terms during the test, wondering if in this case contexts are too powerful and discriminate too many terms. Indeed, in assembly code, it is easy to express such contexts using pointer arithmetic. Finally, we will use those two case studies to investigate how to define contexts for an ideal assembly code, with a notion of code pointer. In this case, we have to give up the usual representation of a context as term with an hole. As in [6], we will rather define contexts has return addresses. Then, we will see that our two previous results still fit well in this low-level setting.

## 1 Biorthogonal Logical Relations for a polymorphic Curry-style $\lambda$-calculus

In this part, we will focus on Curry-style $\lambda$-calculus, which is useful in our enterprise to deal with low-level code. Indeed, the main difference between Curry-style and Church-style polymorphism is that there is no explicit values $\Lambda\alpha.\tau$ in Curry-style, which is the case for many assembly languages.

### 1.1 The Language

We consider a standard call-by-value $\lambda$-calculus with universal polymorphism, noted $\mathcal{L}$. The syntax, typing rules and operational semantics of this language are defined in Figure 1. In each typing rule, $\Delta$ is the context for free type variables, and $\Gamma$ for free term variables.

Figure 2 introduces standard type indexed logical relations for $\mathcal{L}$. The relations $\mathcal{K}[\![\tau]\!]$ and $\mathcal{E}[\![\tau]\!]$ are defined in terms of an abstract notion of observation $\mathcal{O}$, which will be instantiated by the equidivergence relation. Notice that stuck terms which are not values are not in $\mathcal{O}$. Because of universal quantification, we have to deal with open types. This is the purpose of the environment $\eta$ which associates relations on terms to free type variables. In the definition of $\mathcal{V}[\![\forall\alpha.\tau]\!]_\eta$, $\mathrm{Rel}_{\sigma_1,\sigma_2}$ is the set of relations on values of types $\sigma_1$ and $\sigma_2$.

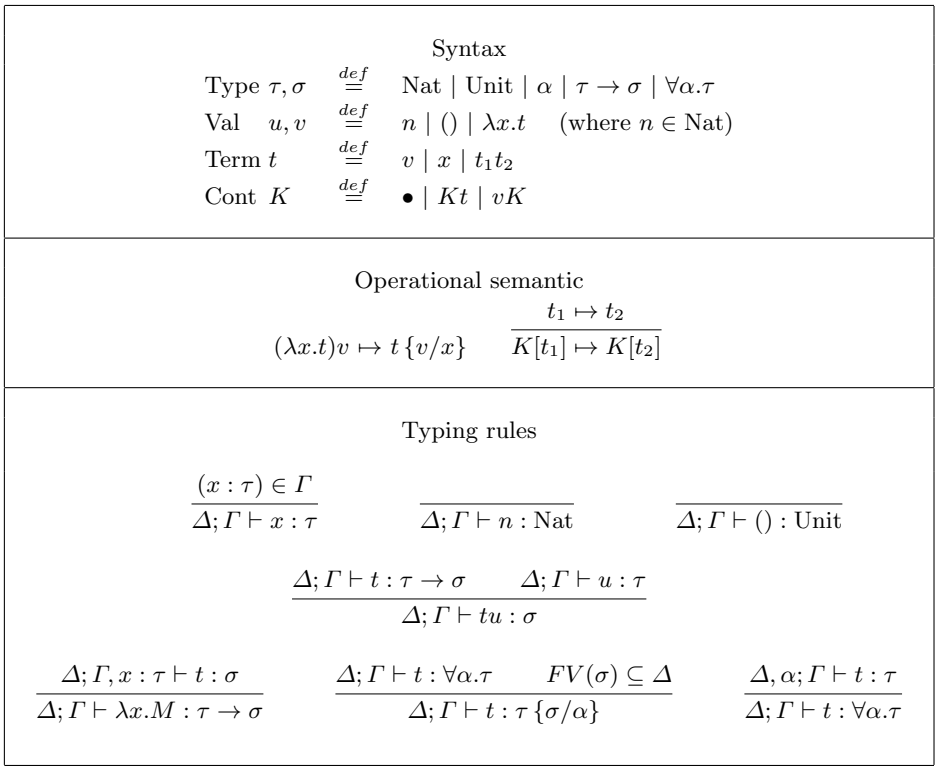


**Fig. 1.** The Language $\mathcal{L}$.

### 1.2 Parametricity

As it is well known, terms in $\mathcal{V}[\![\forall\alpha.\tau]\!]$ and $\mathcal{E}[\![\forall\alpha.\tau]\!]$ have to be *parametric*, i.e. uniform with respect to the instantiation of $\alpha$. This is due to the different instantiation of $\alpha$ in the definition of $\mathcal{V}[\![\forall\alpha.\tau]\!]_\eta$, with $\sigma_1$ on the left and $\sigma_2$ on the right. This enforces good terms not to adapt their behaviour to the instantiated type. For example, consider the instruction `isBool` whose semantics is :

$$\texttt{isBool}\, u \mapsto \begin{cases} \text{true} & \text{if } u \text{ is a boolean (true or false)} \\ \text{false} & \text{otherwise} \end{cases}$$

`isBool` has type $\forall\alpha.(\alpha \to \text{Bool})$ but has a very different behaviour when used with a boolean or with a term of another type. This means that `isBool` is not parametric. Indeed, we can prove that $(\texttt{isBool}, \texttt{isBool}) \notin \mathcal{V}[\![\forall\alpha.\alpha \to \text{Bool}]\!]$. Consider the relation $r \in \text{Rel}_{\text{Bool,Nat}}$ defined by $r = \{(\text{true}, 0)\}$, then we should have $(\texttt{isBool}\,\text{true}, \texttt{isBool}\,0) \in \mathcal{E}[\![\text{Bool}]\!]$ which is false since $\texttt{isBool}\,\text{true} \mapsto \text{true}$ and $\texttt{isBool}\,0 \mapsto \text{false}$.

$$
\begin{array}{lcl}
\mathcal{V}\left[\!\left[\alpha\right]\!\right]_\eta & \stackrel{def}{=} & \{\eta(\alpha)\}^{\perp_v\perp_k} \\
\mathcal{V}\left[\!\left[\mathrm{Unit}\right]\!\right]_\eta & = & \{((),())\} \\
\mathcal{V}\left[\!\left[\mathrm{Nat}\right]\!\right]_\eta & \stackrel{def}{=} & \{(n,n) \mid n \in \mathrm{Nat}\} \\
\mathcal{V}\left[\!\left[\tau \to \sigma\right]\!\right]_\eta & = & \{(u_1,u_2) \mid \forall (v_1,v_2) \in \mathcal{V}\left[\!\left[\tau\right]\!\right]_\eta . (u_1v_1, u_2v_2) \in \mathcal{E}\left[\!\left[\sigma\right]\!\right]_\eta\} \\
\mathcal{V}\left[\!\left[\forall\alpha.\tau\right]\!\right]_\eta & = & \{(v_1,v_2) \mid \forall \sigma_1,\sigma_2 : \mathrm{Type}, \forall r \in \mathrm{Rel}_{\sigma_1,\sigma_2} . (v_1,v_2) \in \mathcal{V}\left[\!\left[\tau\right]\!\right]_{\eta,\alpha\mapsto r}\} \\
 & & \\
\mathcal{K}\left[\!\left[\tau\right]\!\right]_\eta & \stackrel{def}{=} & \mathcal{V}\left[\!\left[\tau\right]\!\right]_\eta^{\perp_k} \\
\mathcal{E}\left[\!\left[\tau\right]\!\right]_\eta & \stackrel{def}{=} & \mathcal{K}\left[\!\left[\tau\right]\!\right]_\eta^{\perp_t} \\
 & & \\
X^{\perp_k} & \stackrel{def}{=} & \{(K_1,K_2) \mid \forall (v_1,v_2) \in X, \mathcal{O}(K_1[v_1], K_2[v_2])\} \\
X^{\perp_v} & \stackrel{def}{=} & \{(v_1,v_2) \mid \forall (K_1,K_2) \in X, \mathcal{O}(K_1[v_1], K_2[v_2])\} \\
X^{\perp_t} & \stackrel{def}{=} & \{(t_1,t_2) \mid \forall (K_1,K_2) \in X, \mathcal{O}(K_1[t_1], K_2[t_2])\}
\end{array}
$$

**Fig. 2.** Logical relations for $\mathcal{L}$

### 1.3 Fundamental property of the logical relation

Let us now prove that terms of types $\tau$ are in the diagonal of $\mathcal{E}\left[\!\left[\tau\right]\!\right]$. To do so, we have to prove the so-called *compatibility lemmas*—one for each typing rule—which make the induction on typing derivations possible. But the one about the introduction rule of the of the universal quantifier is problematic in Curry-style, due to the absence of proper values of type $\forall\alpha.\tau$. To be able to prove it, we have to unfold biorthogonality to get a direct definition of $\mathcal{E}\left[\!\left[\tau\right]\!\right]$ in terms of $\mathcal{V}\left[\!\left[\tau\right]\!\right]$.

**Lemma 1.** *For every type $\tau, \sigma$ and every relation $r \in \mathrm{Rel}_\sigma$ and every environment $\eta$,*

- $\mathcal{V}\left[\!\left[\forall\alpha.\tau\right]\!\right]_\eta \subseteq \mathcal{V}\left[\!\left[\tau\right]\!\right]_{\eta\cdot[\alpha\mapsto r]}$
- $\mathcal{K}\left[\!\left[\tau\right]\!\right]_{\eta\cdot[\alpha\mapsto r]} \subseteq \mathcal{K}\left[\!\left[\forall\alpha.\tau\right]\!\right]_\eta$
- $\mathcal{E}\left[\!\left[\forall\alpha.\tau\right]\!\right]_\eta \subseteq \mathcal{E}\left[\!\left[\tau\right]\!\right]_{\eta\cdot[\alpha\mapsto r]}$

*Proof.* The first inclusion is direct by definition, the other ones come from the basic properties of orthogonality.

**Lemma 2 (Adequacy of $\mathcal{E}$ and $\mathcal{V}$ on values).** *For every type $\tau$, every map $\eta$ s.t. $\mathrm{dom}(\eta) = FV(\tau)$ and every values $v_1, v_2$, $(v_1,v_2) \in \mathcal{E}\left[\!\left[\tau\right]\!\right]_\eta$ iff $(v_1,v_2) \in \mathcal{V}\left[\!\left[\tau\right]\!\right]_\eta$*

*Proof.* The proof is done by induction on $\tau$ :

- If $\tau$ is a type variable $\alpha$, then $\mathcal{E}\left[\!\left[\alpha\right]\!\right]_\eta = \eta(\alpha)^{\perp_t\perp_k\perp_v\perp_k} \subseteq \eta(\alpha)^{\perp_v\perp_k\perp_v\perp_k} = \eta(\alpha)^{\perp_v\perp_k} = \mathcal{V}\left[\!\left[\alpha\right]\!\right]_\eta$
- If $\tau$ is $\forall\alpha.\tau$, then Let $(v_1,v_2) \in \mathcal{E}\left[\!\left[\forall\alpha.\tau\right]\!\right]_\eta$, we will prove that $(v_1,v_2) \in \mathcal{V}\left[\!\left[\forall\alpha.\tau\right]\!\right]_\eta$, i.e. for every couple of types $(\sigma_1,\sigma_2)$ and every relation $r \in \mathrm{Rel}_{\sigma_1,\sigma_2}, (v_1,v_2) \in \mathcal{V}\left[\!\left[\tau\right]\!\right]_{\eta\cdot[\alpha\mapsto r]}$. By induction hypothesis we can simply show

that $(v_1, v_2) \in \mathcal{E}[\![\tau]\!]_{\eta\cdot[\alpha\mapsto r]}$, i.e. for every $(K_1, K_2) \in \mathcal{K}[\![\tau]\!]_{\eta\cdot[\alpha\mapsto r]}, \mathcal{O}(K_1[v_1], K[v_2])$. But from Lemma 1 we know that such $(K_1, K_2)$ are in fact in $\mathcal{K}[\![\forall\alpha.\tau]\!]_\eta$ so we can conclude since $(v_1, v_2) \in \mathcal{E}[\![\forall\alpha.\tau]\!]_\eta$.
- The proof of other cases are classic, see [**?**].

**Lemma 3 (Unfolding of biorthogonality).** *Let $t_1, t_2$ be two terms, then $(t_1, t_2) \in \mathcal{E}[\![\tau]\!]_\eta$ iff $t_1, t_2$ both diverge or there exists two values $(v_1, v_2) \in \mathcal{V}[\![\tau]\!]_\eta$ s.t. $t_i \mapsto^* v_i$.*

*Proof.*
- Direct side : If $(t_1, t_2) \in \mathcal{E}[\![\tau]\!]_\eta$ do not both diverge, then they both normalize since the empty context is in the diagonal of $\mathcal{K}[\![\tau]\!]_\eta$. So from Lemma 2, there exists values $(v_1, v_2) \in \mathcal{V}[\![\tau]\!]_\eta$ s.t. $t_i \mapsto^* v_i$.
- Reverse side : Let $(K_1, K_2) \in \mathcal{K}[\![\tau]\!]_\eta$, then from $(K_1[v_1], K_2[v_2]) \in \mathcal{O}$ we get $(K_1[t_1], K_2[t_2]) \in \mathcal{O}$, i.e. $(t_1, t_2) \in \mathcal{E}[\![\tau]\!]_\eta$.

**Lemma 4 (Compatibility lemma for the polymorphic introduction rule).** *Let $t_1, t_2$ be two terms such that for every types $\sigma_1, \sigma_2$ and every relation $r \in \mathrm{Rel}_{\sigma_1,\sigma_2}, (t_1, t_2) \in \mathcal{E}[\![\tau]\!]_{\eta\cdot\alpha\mapsto r}$ then $(t_1, t_2) \in \mathcal{E}[\![\forall\alpha.\tau]\!]_\eta$*

*Proof.* By Lemma 3, there are two cases. Either $t_1$ and $t_2$ both diverge then they are trivially in $\mathcal{E}[\![\forall\alpha.\tau]\!]_\eta$. Otherwise there exists $(v_1, v_2)$ s.t. for every types $\sigma_1, \sigma_2$ and every relation $r \in \mathrm{Rel}_{\sigma_1,\sigma_2}$, $(v_1, v_2) \in \mathcal{V}[\![\tau]\!]_{\eta\cdot\alpha\mapsto r}$ and $t_i \mapsto^* v_i$, i.e. $(v_1, v_2) \in \mathcal{V}[\![\forall\alpha.\tau]\!]_\eta$, and we conclude using the reverse side of lemma 3.

Thanks to this lemma, we can prove the desired theorem.

**Theorem 1 (Fundamental property).** *Let $t$ be a term such that $\Delta, \Gamma \vdash t : \tau$, then for every environment $\gamma$ and $\eta$ compatible respectively with $\Gamma$ and $\Delta$, we get $(\gamma t, \gamma t) \in \mathcal{E}[\![\tau]\!]_\eta$*

*Proof.* The proof is done by induction on the typing proof, using compatibility lemmas.

Finally, here is an example of what kind of operational correctness results we can get.

**Theorem 2 (Operational correctness for the type** Nat**).** *If $(t_1, t_2) \in \mathcal{E}[\![\mathrm{Nat}]\!]$ then $t_1$ and $t_2$ both diverges or there exists $n \in \mathrm{Nat}$ such that $t_1$ and $t_2$ both normalize to $n$.*

*Proof.* Straightforward using lemma 3 and the definition of $\mathcal{V}[\![\mathrm{Nat}]\!]$.

### 1.4 Addition of mutable states

To deal with the problem of program equivalence for a language with mutable states, the so-called *Kripke logical relations* have been developed [10]. The idea is to use a notion of *world* to represent relations on heaps. Building such worlds in presence of higher-order references is difficult, since there is circularity problems in the definition. To overcome this problem, it is now common to use step-index

Syntax

$$
\begin{array}{lcl}
\text{Type } \tau,\sigma & \stackrel{def}{=} & \text{Nat} \mid \text{Unit} \mid \alpha \mid \tau \to \sigma \mid \forall\alpha.\tau \mid \text{ref}\,\tau \\
\text{Val} \quad u,v & \stackrel{def}{=} & n \mid () \mid \lambda x.t \mid l \quad (\text{where } n \in \text{Nat and } l \in \text{Loc}) \\
\text{Term } t & \stackrel{def}{=} & v \mid x \mid t_1 t_2 \mid \text{ref}_\tau t \mid t_1 := t_2 \mid !t \\
\text{Cont } K & \stackrel{def}{=} & \bullet \mid Kt \mid vK \mid \text{ref}_\tau K \mid K := t \mid v := K \mid !K
\end{array}
$$

Operational semantic

$$
\begin{array}{l}
((\lambda x.t)v, h) \mapsto (t\{v/x\}, h) \\
(\text{ref}\, v, h) \quad \mapsto (l, h \bullet [l \mapsto v]) \\
(l := v, h) \quad \mapsto ((), h[l \mapsto v]) \\
(!l, h) \qquad \mapsto (h(l), h)
\end{array}
\qquad
\frac{(t_1, h_1) \mapsto (t_2, h_2)}{(K[t_1], h_1) \mapsto (K[t_2], h_2)}
$$

New typing rules

$$
\frac{\Upsilon(l) = T}{\Delta;\Gamma;\Upsilon \vdash l : \text{ref}\, T}
\qquad
\frac{\Delta;\Gamma;\Upsilon \vdash t : \tau}{\Delta;\Gamma;\Upsilon \vdash \text{ref}\, t : \text{ref}\, \tau}
$$

$$
\frac{\Delta;\Gamma;\Upsilon \vdash t_1 : \text{ref}\, \tau \qquad \Delta;\Gamma;\Upsilon \vdash t_2 : \tau}{\Delta;\Gamma;\Upsilon \vdash t_1 := t_2 : \text{Unit}}
\qquad
\frac{\Delta;\Gamma;\Upsilon \vdash t : \text{ref}\, \tau}{\Delta;\Gamma \vdash !t : \tau}
$$

**Fig. 3.** The Extention $\mathcal{L}_{ref}$.

method [2,1] which induces a stratification on worlds. However, it is possible to hide completely such technicalities in the definition, as it is shown in [8].

In fact here we only want to prove the fundamental lemma, i.e. to prove $(t,t) \in \mathcal{E}[\![\tau]\!]$, so we can work with really simple world, which are predicates on heaps, specifying the free locations $l$ of $t$. More precisely, from a typing environment $\Upsilon$ of free location one associate the world

$$w \stackrel{def}{=} \{(l, v, w') \mid l \in \text{dom}(\Upsilon), v \text{ is in the diagonal of } \mathcal{V}[\![\Upsilon(l)]\!]_{w',\eta}\}$$

We will call such worlds "simple predicative". In the following, we will note $h \in w$ as a shortcut for $\text{dom}(h) = \text{dom}(w) \wedge \forall l \in \text{dom}(h), (l, h(l), w) \in w$. Such worlds can be useful only if the type system ensure weak updates of heap (i.e. type preserving one).

**Lemma 5 (Weak updates).** *If $\Delta;\Gamma;\Upsilon \vdash t : \tau$ and $h$ is a heap satisfying $\Upsilon$ and $(t,h) \mapsto^* (t',h')$, then $h'$ satisfies $\Upsilon$.*

*Proof.* This property is ensure by the restriction of monomorphic types for references.

Then we will have to consider only typable terms in our logical relations to get this property.

**Lemma 6 (One-Step Unfolding of biorthogonality).** *Let $t$ be a typable term which is not a value and $w$ a simple predicative world, then $(t,t) \in \mathcal{E}[\![\tau]\!]_{w,\eta}$ iff for all heaps $h \in w$, there exists a simple predicative world $w' \sqsupseteq w$, a term $t'$ s.t. $(t',t') \in \mathcal{E}[\![\tau]\!]_{w',\eta}$ and a heap $h' \in w'$ s.t. $(t,h) \mapsto (t',h')$.*

*Proof.* – Direct side : For every heap $h \in w$ there exists a unique term $t'$ and heap $h'$ such that $(t,h) \mapsto (t',h')$. We are going to build the world $w' \sqsupseteq w$ by case analysis on this reduction step :
  - If $h' = h$ then we can simply take $w' = w$.
  - If $t$ can be written as $K[\mathrm{ref}_\tau v]$ and $t'$ as $K[l]$ then then one can take $w' = w \cup \{((l,w'',v) \mid (v,v) \in \mathcal{V}[\![\tau]\!]_{w'',\eta}]\}$
  - If If $t$ can be written as $K[l := v]$, then from lemma 5, $h'$ will be in $w$ so we take $w' = w$.

– Reverse side : Let $(K_1,K_2) \in \mathcal{K}[\![\tau]\!]_{w,\eta}$ then we have to show

$$((K_1[t],h),(K_2[t],h)) \in \mathcal{O}.$$

To do so we will just prove that $((K_1[t'],h'),(K_2[t'],h')) \in \mathcal{O}$, which comes from the fact that $(t',t') \in \mathcal{E}[\![\tau]\!]_{w',\eta}$ and by monotonicity of $\mathcal{K}[\![\tau]\!]$ with respect to worlds, $(K_1,K_2) \in \mathcal{K}[\![\tau]\!]_{w',\eta}$.

**Lemma 7 (Adequacy of $\mathcal{E}$ and $\mathcal{V}$ on values).** *For every type $\tau$, every map $\eta$ s.t. $\mathrm{dom}(\eta) = FV(\tau)$, every values $v_1, v_2$ and every world $w$, $(v_1,v_2) \in \mathcal{E}[\![\tau]\!]_{w,\eta}$ iff $(v_1,v_2) \in \mathcal{V}[\![\tau]\!]_{w,\eta}$*

*Proof.* Same as for lemma 2.

**Lemma 8 (Unfolding of biorthogonality).** *Let $t$ be a typable term and $w$ a simple predicative world, then $(t,t) \in \mathcal{E}[\![\tau]\!]_{w,\eta}$ iff for all heaps $h \in w$, $(t,h)$ diverges or there exists a simple predicative world $w' \sqsupseteq w$, a value $v$ in the diagonal of $\mathcal{V}[\![\tau]\!]_{w,\eta}$ and a heap $h' \in w'$ s.t. $(t,h) \mapsto^* (v,h')$.*

*Proof.* – Direct side : The proof is done combining lemmas 6 and 7.

– Reverse side : Let $(K_1,K_2) \in \mathcal{K}[\![\tau]\!]_{w,\eta}$ then we have to show

$$((K_1[t_1],h_1),(K_2[t_2],h_2)) \in \mathcal{O}.$$

To do so we will just prove that $((K_1[v_1],h_1'),(K_2[v_2],h_2')) \in \mathcal{O}$, which comes from the fact that $(v_1,v_2) \in \mathcal{V}[\![\tau]\!]_{w',\eta}$ and by monotonicity of $\mathcal{K}[\![\tau]\!]$ with respect to worlds, $(K_1,K_2) \in \mathcal{K}[\![\tau]\!]_{w',\eta}$.

Notice that this lemma is too weak to be useful to get operational correctness from $(t_1,t_2) \in \mathcal{E}[\![\tau]\!]$, since it talks only about the diagonal of $\mathcal{E}[\![\tau]\!]$. It seems hard to extend this result to the whole relation, since we do not know how to build $w' \sqsupseteq w$ when worlds are relations on heaps and terms $t_1$ and $t_2$ do not modify their heap in the same way. This seems to be a weakness of Kripke logical relations, since worlds are only preconditions and there is no postconditions which assert the way the heap is modified by the terms. We plan to further

investigate this point using the language of separation logic, more precisely Hoare Type Theory [9], to extend logical relations with postconditions. Nevertheless, it is enough to prove the compatibility lemma for the introduction rule of the universal quantifier.

**Lemma 9 (Compatibility lemma for the polymorphic introduction rule).** *Let $t_1, t_2$ be two typable terms such that for every type $\sigma_1, \sigma_2$ and every relation $r \in \mathrm{Rel}_{\sigma_1,\sigma_2}, (t_1, t_2) \in \mathcal{E}[\![\tau]\!]_{w,\eta\cdot\alpha\mapsto r}$ then $(t_1, t_2) \in \mathcal{E}[\![\forall\alpha.\tau]\!]_{w,\eta}$*

*Proof.* Same as for lemma 4.

### 1.5 Intersection types

For the time, we have only work with a parametric quantification, which implies a uniform behaviour of programs. We will now add intersection types $\tau \cap \sigma$ with a subtyping rule `<:` defined by the following rule :

$$\frac{\Gamma \vdash \sigma' <: \sigma \qquad \Gamma \vdash \tau <: \tau'}{\Gamma \vdash \sigma \to \tau <: \sigma' \to \tau'} \qquad \frac{}{\Gamma \vdash \sigma \cap \tau <: \sigma} \qquad \frac{}{\Gamma \vdash \sigma \cap \tau <: \tau}$$

and new typing rules :

$$\frac{\Gamma \vdash t : \sigma \qquad \Gamma \vdash \sigma <: \tau}{\Gamma \vdash t : \tau} \qquad \frac{\Gamma \vdash t : \sigma \qquad \Gamma \vdash t : \tau}{\Gamma \vdash t : \sigma \cap \tau}$$

Then we define $\mathcal{V}[\![\sigma \cap \tau]\!] = \mathcal{V}[\![\sigma]\!] \cap \mathcal{V}[\![\tau]\!]$ while $\mathcal{K}[\![\sigma \cap \tau]\!]$ and $\mathcal{E}[\![\sigma \cap \tau]\!]$ are defined as usual by orthogonality. We can easily check that the previous proof of the various lemmas are still working.

**Lemma 10.** *For every types $\sigma, \tau$, $\sigma <: \tau$ implies $\mathcal{E}[\![\sigma]\!]_\eta \subseteq \mathcal{E}[\![\tau]\!]_\eta$.*

*Proof.* By induction on the proof of $\sigma <: \tau$ :

- If $\sigma = \alpha \to \beta$ and $\tau = \alpha' \to \beta'$ with $\alpha' <: \alpha$ and $\vdash \beta <: \beta'$, then we have to prove that for all $(t_1, t_2) \in \mathcal{E}[\![\alpha \to \beta]\!]$ and $(v_1, v_2) \in \mathcal{V}[\![\alpha']\!], (t_1 v_1, t_2 v_2) \in \mathcal{E}[\![\beta]\!]$.
  By induction hypothesis, $(v_1, v_2) \in \mathcal{E}[\![\alpha]\!]$, and using lemma 2 we get that $(v_1, v_2) \in \mathcal{V}[\![\alpha]\!]$, so we can easily conclude.
- If $\sigma = \alpha \cap \tau$ and $\tau = \alpha$ then one has to unfold biorthogonality using lemma 3, following the same proof than lemma 4.

However, compared to values in $\mathcal{V}[\![\forall\alpha.\tau]\!]$, those in $\mathcal{V}[\![\sigma \cap \tau]\!]$ does not have to have the same behaviour for $\sigma$ and $\tau$. For example the function $\lambda n.\texttt{if } n > 0 \texttt{ then } n \texttt{ else } \Omega$ is in the diagonal of $\mathcal{V}[\![(\mathrm{Nat}_0 \to \mathrm{Nat}_0) \cap (\mathrm{Pos} \to \mathrm{Pos})]\!]$. In comparison, terms in $\mathcal{V}[\![\forall\alpha.\tau]\!]$ equidiverge regardless of the instantiation of $\alpha$. In fact one can even prove that `isBool` is in the diagonal of $\mathcal{V}[\![(\mathrm{Nat} \to \mathrm{Bool}) \cap (\mathrm{Bool} \to \mathrm{Bool})]\!]$.

Notice that all these results are still true when we add an implicit depend types $\forall n : \mathrm{Nat}.\tau_n$ as in [5], with the following subtyping rule

$$\frac{}{\Gamma \vdash \forall n : \mathrm{Nat}.\tau_n <: \tau_n}$$

Then $\mathcal{V}[\![\forall n : \mathrm{Nat}.\tau_n]\!]_\eta$ is defined as

$$\bigcap_{n \in \mathbb{N}} \mathcal{V}[\![\tau_n]\!]$$

## 2 Intentional equality

Like in [3], we consider a language with an intentional equality, and we are going to investigate the discriminating power of contexts in this setting. More precisely, consider the language $\mathcal{L}_\alpha$ where we add to $\mathcal{L}$ the relation $\simeq_\alpha$ encoding $\alpha$-equivalence:

$$t_1 \simeq_\alpha t_2 \text{ iff } t_1 \text{ and } t_2 \text{ are } \alpha\text{-equivalent.}$$

We will note $\mathrm{Val}_\alpha$ and $\mathrm{Term}_\alpha$ the corresponding syntactical classes of $\mathcal{L}_\alpha$.

Adding such an intentional equality, contextual equivalence collapses to syntactical equality, so one should not seek correctness of logical relations anymore. But if we consider our previous logical relation $\mathcal{E}$, this relation collapse to $\mathcal{V}$ on basic types. Indeed let $K = \texttt{if } \bullet \simeq_\alpha n \texttt{ then } \Omega \texttt{ else ()}$, then we have $(K, K) \in \mathcal{K}[\![\mathrm{Nat}]\!]$ and $(K, K)$ will be able to discriminate between a term $t$ which reduces to $n$ and the value $n$, so $\mathcal{E}[\![\mathrm{Nat}]\!] = \mathcal{V}[\![\mathrm{Nat}]\!]$.

To avoid this problem we have to restrict contexts to the form

$$\mathrm{Cont}_\alpha = \{\texttt{let } x = \bullet \texttt{ in } M \mid M \in \mathrm{Term}_\alpha\}$$

Then we force the evaluation of the term to happen first, and only the value we get as the result can be tested. As we will see later, this is in fact the shape of low-level contexts. The addition of $\simeq_\alpha$ breaks the rule

$$\frac{t_1 \mapsto t_2}{K[t_1] \mapsto K[t_2]}.$$

which provides an up-to context reasoning for operational semantics. But if we know that $K_1, K_2$ are in this restricted shape, this rule is true again, so does the unfolding of biorthogonality.

Consider

$$\mathcal{K}_\alpha[\![\tau]\!] \stackrel{def}{=} \{(K_1, K_2) \in \mathrm{Cont}_\alpha . \forall (v_1, v_2) \in \mathcal{V}[\![\tau]\!], (K_1[v_1], K_2[v_2]) \in \mathcal{O}\}$$

and

$$\mathcal{E}_\alpha[\![\tau]\!] = \{(t_1, t_2) \in \mathrm{Term} . \forall (K_1, K_2) \in \mathcal{K}_\alpha[\![\tau]\!], (K_1[t_1], K_2[t_2]) \in \mathcal{O}\}.$$

Here $\mathcal{V}[\![\tau]\!]$ corresponds to the previous relation on Val, and not on $\text{Val}_\alpha$, while $\mathcal{E}[\![\tau]\!]$ is restricted to elements of Term and not of $\text{Term}_\alpha$. Indeed, we are only interested in the logical relations on elements of $\mathcal{L}$, and we just want to see if extending the power of contexts changes something, which is not the case :

**Theorem 3.** *For every type $\tau$ and every environment $\eta$, $\mathcal{E}_\alpha[\![\tau]\!]_\eta = \mathcal{E}[\![\tau]\!]_\eta$.*

*Proof.* First we prove $\mathcal{E}[\![\tau]\!] \subseteq \mathcal{E}_\alpha[\![\tau]\!]$. Let $(e_1, e_2) \in \mathcal{E}[\![\tau]\!]$, then unfolding biorthogonality we know that $(e_1, e_2)$ both diverge or there exists $(v_1, v_2) \in \mathcal{V}[\![\tau]\!]$ such that $e_i \mapsto^* v_i$. But since $\mathcal{K}_\alpha[\![\tau]\!] = \mathcal{V}[\![\tau]\!]^\perp$, taking $(K_1, K_2) \in \mathcal{K}_\alpha[\![\tau]\!]$ we get $(v_1, v_2) \perp (K_1, K_2)$, so $(e_1, e_2) \perp (K_1, K_2)$, i.e. $(e_1, e_2) \in \mathcal{E}_\alpha[\![\tau]\!]$.

Finally, $\mathcal{K}[\![\tau]\!] \subseteq \mathcal{K}_\alpha[\![\tau]\!]$, because they are orthogonal of the same set of values - not with the same notion of orthogonality, but $\text{Cont} \subseteq \text{Cont}_\alpha$ - so $\mathcal{E}_\alpha[\![\tau]\!] \subseteq \mathcal{E}[\![\tau]\!]$.

Looking carefully at the proof, we see that it does not depend on the new added instruction—here $\simeq_\alpha$—but relies only on the possibility to unfold biorthogonality. Of course, it is not always the case, even with restricted shapes of contexts, as one can see by considering the `call/cc` instruction.

## 3 Logical Relations for Assembly code

Now we will investigate previous results in the setting of low-level code.

### 3.1 A variant of the SECD

We will work with an ideal assembly code, represented by a variant of the SECD machine. Compared to previous articles [7],[4], we use a machine with an explicit code pointer. Indeed, while with $\lambda$-calculus or usual abstract machines we can define contexts as code with an hole (and possibly an environment and a stack), this is no more possible when we have an explicit code pointer, since code has to be constant. The machine is defined in Figure 1.

Notice that the instruction `Stop` has no reduction rule. Indeed, it is here to indicate final configurations. Then, the whole compiled program associate to a term $t$ will be $\textbf{compile}\,(t, 0)$`++`[`Stop`], where **compile** is defined in Figure 2. But to prove operational correctness for $\textbf{compile}\,(t, 0)$, this is in fact $\textbf{compile}\,(t, 0)$`++`[`Return`] which will be used, to force context in the dump to be evaluated.

We keep $\mathcal{O}$ as the equidivergence, but one has to define what it means to be a good final configuration, to avoid stuck configurations. $(l, e, s, d)$ is a good final configuration with respect to $c$ if :

- $l = \texttt{size}(c) + 1$
- $s = [v]$
- $d = []$

Syntax

| | | | |
|---|---|---|---|
| Loc | $l$ | $\stackrel{def}{=}$ | Nat |
| `Inst` | $i$ | $\stackrel{def}{=}$ | $\texttt{App} \mid \texttt{Access}\, n \mid \texttt{Return} \mid \texttt{Jump}\, l \mid \texttt{PushN}\, v \mid \texttt{PushC}\, v \mid \texttt{Sel}\, l_1 l_2 \mid \texttt{Op} \circ v_1 v_2 \mid \texttt{Stop}$ |
| `Code` | $c$ | $\stackrel{def}{=}$ | List `Inst` |
| `Env` | $e$ | $\stackrel{def}{=}$ | List Val |
| `Stack` | $s$ | $\stackrel{def}{=}$ | List Val |
| Val | $v$ | $\stackrel{def}{=}$ | $\texttt{V}(n) \mid \texttt{C}(l, e)$ |
| Cont | $k$ | $\stackrel{def}{=}$ | Loc $\times$ `Env` $\times$ `Stack` |
| `Dump` | $d$ | $\stackrel{def}{=}$ | List Cont |
| Conf | $\Phi$ | $\stackrel{def}{=}$ | Loc $\times$ `Env` $\times$ `Stack` $\times$ `Dump` |

Transition rules

| | | |
|---|---|---|
| $c(l) = \texttt{Return}$ | $: (l, e, v :: s, (l', e', s') :: d)$ | $\succ_c (l', e', v :: s', d)$ |
| $c(l) = \texttt{Jump}\, l'$ | $: (l, e, v :: s, d)$ | $\succ_c (l', e', v :: s', d)$ |
| $c(l) = \texttt{Access}\, n$ | $: (l, e, s, d)$ | $\succ_c (l+1, e, e(n) :: s, d)$ |
| $c(l) = \texttt{PushN}\, n$ | $: (l, e, s, d)$ | $\succ_c (l+1, e, \texttt{V}(n) :: s, d)$ |
| $c(l) = \texttt{PushC}\, l'$ | $: (l, e, s, d)$ | $\succ_c (l+1, e, \texttt{C}(l', e) :: s, d)$ |
| $c(l) = \texttt{PushRC}\, l'$ | $: (l, e, s, d)$ | $\succ_c (l+1, e, \texttt{RC}(l', e) :: s, d)$ |
| $c(l) = \texttt{App}$ | $: (l, e, \texttt{C}(l', e') :: v :: s, d)$ | $\succ_c (l', v :: e', s, (l, e, s) :: d)$ |
| $c(l) = \texttt{Op} \odot$ | $: (l, e, \texttt{V}(n_1) :: \texttt{V}(n_2) :: s, d)$ | $\succ_c (l+1, e, \texttt{V}(n_2 \odot n_1) :: s, d)$ |
| $c(l) = \texttt{Sel}\, l_1 l_2$ | $: (l, e, v :: s, d)$ | $\succ_c \begin{cases} (l_1, e, s, d) & \text{if } v = \texttt{V}(0) \\ (l_2, e, s, d) & \text{otherwise.} \end{cases}$ |

**Table 1.** A Variant of the SECD with code pointer

$$
\begin{aligned}
&\textbf{compile}\,(n, l) && = \texttt{PushN}\, n \\
&\textbf{compile}\,(\text{false}, l) && = \texttt{PushN}\, 0 \\
&\textbf{compile}\,(\text{true}, l) && = \texttt{PushN}\, 1 \\
&\textbf{compile}\,(\text{Var}_i, l) && = \texttt{Access}\, i \\
&\textbf{compile}\,(\lambda.t, l) && = [\texttt{PushC}\,(l+2), \texttt{Jump}\,(l+2+k+1)] \texttt{++} \textbf{compile}\,(t, l+2) \texttt{++} [\texttt{Return}] \\
& && \quad \text{where } k = \texttt{size}(\textbf{compile}\,(t, l+2)) \\
&\textbf{compile}\,(t\, u, l) && = \textbf{compile}\,(u, l) \texttt{++} \textbf{compile}\,(t, l+k) \texttt{++} [\texttt{App}] \\
& && \quad \text{where } k = \texttt{size}(\textbf{compile}\,(t, l)) \\
&\textbf{compile}\,(t_1 \odot t_2, l) && = \textbf{compile}\,(t_1, l) \texttt{++} \textbf{compile}\,(t_2, l+k) \texttt{++} [\texttt{Op} \odot] \\
& && \quad \text{where } k = \texttt{size}(\textbf{compile}\,(t_1, l)) \\
&\textbf{compile}\,(\text{If}\, t_c\, t_1\, t_2, l) && = \textbf{compile}\,(t_c, l) \texttt{++} [\texttt{Sel}\,(l+k+1)(l+k+1+i)] \texttt{++} \\
& && \quad \textbf{compile}\,(t_1, l+k+1) \texttt{++} \textbf{compile}\,(t_2, l+k+1+i) \\
& && \quad \text{where } k = \texttt{size}(\textbf{compile}\,(t_c, l)) \\
& && \quad \text{and} \quad i = \texttt{size}(\textbf{compile}\,(t_1, l+k+1))
\end{aligned}
$$

**Table 2.** The compiler

and two configurations are in $\mathcal{O}_{c_1,c_2}$ if they both diverges or they both reduces to good final configuration with respect to $c_1$ and $c_2$.

Then one define an orthogonality relation $\perp$, indexed by codes $c_1, c_2$, between values in Val and contexts in Cont :

$$(v_1, v_2) \perp_{c_1,c_2} ((l_1, e_1, s_1), (l_2, e_2, s_2)) \stackrel{def}{=} ((l_1, e_1, v_1 :: s_1, []), (l_2, e_2, v_2 :: s_2, [])) \in \mathcal{O}_{c_1,c_2}$$

One also define the orthogonality between configuration inf Conf and contexts in Cont :

$$((l_1, e_1, s_1, d_1), (l_2, e_2, s_2, d_2)) \perp_{c_1,c_2} ((l_1', e_1', s_1'), (l_2', e_2', s_2'))$$
$$\stackrel{def}{=}$$
$$((l_1, e_1, s_1, d_1 \texttt{++}[(l_1', e_1', s_1')]), (l_2, e_2, s_2, d_2 \texttt{++}[(l_2', e_2', s_2')])) \in \mathcal{O}_{c_1,c_2}$$

Contexts have to be seen as return addresses, as in [6]. Just like the restricted contexts in part 2, such contexts can only test values and not full configurations, since they are evaluated by an instruction `Return`.

The fact that the code $c$ is constant induces some conceptual changes. Indeed, the syntactic duality between terms and values blurs to a duality on the control flow between a result and its computation. But since we still want to consider contexts as tests, we have to allow to extend $c$ to add tests of results in it. We could also imagine, as in [6], to use Kripke logical relations with worlds describing code $c_1, c_2$. This is in fact closer to the spirit to low-level language since code is in fact stocked in the memory, and can modify itself.

Then, we defined our logical relation :

$$\begin{aligned}
\mathcal{V}[\![\alpha]\!]_{(c_1,c_2),\eta} &= \eta(\alpha) \\
\mathcal{V}[\![\mathrm{Nat}]\!]_{(c_1,c_2),\eta} &= \{(n,n) \mid n \in \mathrm{Nat}\} \\
\mathcal{V}[\![\tau \to \sigma]\!]_{(c_1,c_2),\eta} &= \{(u_1,u_2) \mid \forall (v_1,v_2) \in \mathcal{V}[\![\tau]\!]_{(c_1,c_2),\eta}.(u_1 v_1, u_2 v_2) \in \mathcal{E}[\![\sigma]\!]_{(c_1,c_2),\eta}\} \\
\mathcal{V}[\![\forall \alpha.\tau]\!]_{(c_1,c_2),\eta} &= \{(v_1,v_2) \mid \forall \sigma_1, \sigma_2 : \mathrm{Type}, \forall r \in \mathrm{Rel}_{\sigma_1,\sigma_2}.(v_1,v_2) \in \mathcal{V}[\![\tau]\!]_{(c_1,c_2),\eta,\alpha \mapsto r}\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{K}[\![\tau]\!]_{(c_1,c_2),\eta} &= \{(k_1,k_2) \mid \forall (c_1',c_2') \sqsupseteq (c_1\texttt{++}[\texttt{Return}], c_2\texttt{++}[\texttt{Return}]), \\
&\quad \forall (v_1,v_2) \in \mathcal{V}[\![\tau]\!]_{(c_1',c_2'),\eta}.(v_1,v_2) \perp_{c_1',c_2'} (k_1,k_2)\} \\
\mathcal{E}[\![\tau]\!]_{(c_1,c_2),\eta} &= \{(\Phi_1,\Phi_2) \mid \forall (c_1',c_2') \sqsupseteq (c_1\texttt{++}[\texttt{Return}], c_2\texttt{++}[\texttt{Return}]), \\
&\quad \forall (k_1,k_2) \in \mathcal{K}[\![\tau]\!]_{(c_1',c_2'),\eta}.(\Phi_1,\Phi_2) \perp_{c_1',c_2'} (k_1,k_2)\}
\end{aligned}$$

Notice the quantification $\forall (c_1', c_2') \sqsupseteq (c_1\texttt{++}[\texttt{Return}], c_2\texttt{++}[\texttt{Return}])$ in the definition of $\mathcal{K}[\![\tau]\!]_{(c_1,c_2),\eta}$ and $\mathcal{E}[\![\tau]\!]_{(c_1,c_2),\eta}$, to allow contexts $K_1, K_2$ to be extended so that they can test configurations.

### 3.2 Unfolding biorthogonality

As we have seen before, writing $\mathcal{E}[\![\tau]\!]$ directly in term of $\mathcal{V}[\![\tau]\!]$ is crucial to prove fact about $\mathcal{E}[\![\forall \alpha.\tau]\!]$, just like to get operational results from $(t_1, t_2) \in \mathcal{E}[\![\tau]\!]$.

However, to adapt this result to our low-level machine is not straightforward, because code $c$ is fixed so one cannot replace a value $v$ by a term $t$ in a context. What we rather want to express is that a configuration is going to reduce to a value $v$, and that the context behave the same way with respect to this value and to this configuration.

To prove this result, we have to adapt the usual rule $\dfrac{t_1 \mapsto t_2}{K[t_1] \mapsto K[t_2]}$ in our setting, which is done in the following lemma.

**Lemma 11.** $(l, e, s, d) \succ_c^* (\texttt{size}(c) + 1, e', [v], [])$ *implies that for all code* $c' \sqsupseteq c$ *and dump* $d'$, $((l, e, s, d \texttt{++} d') \succ_{c'}^* (\texttt{size}(c) + 1, e', [v], d')$

*Proof.* By case analysis on the instructions and their transitions.

**Lemma 12.** $\begin{cases} \Phi_1 \succ_c^* (\texttt{size}(c) + 1, e_1', [v_1], []) \\ \Phi_2 \succ_c^* (\texttt{size}(c) + 1, e_2', [v_2], []) \end{cases}$ *implies that for all* $(c_1', c_2') \sqsupseteq (c_1 \texttt{++}[\texttt{Return}], c_2 \texttt{++}[\texttt{Return}])$, $(\Phi_1, \Phi_2) \bot_{c_1', c_2'} (k_1, k_2)$ *iff* $(v_1, v_2) \bot_{c_1', c_2'} (k_1, k_2)$

*Proof.* Using previous lemma, for $i \in \{1, 2\}$, if $k_i = (l_i, e_i, s_i)$ then $\Phi_i \mapsto_{c_i'}^* (\texttt{size}(c_i) + 1, e_i', [v_i], [(l_i, e_i, s_i)])$ and $c_i'(\texttt{size}(c_i) + 1) = \texttt{Return}$ so we get the configuration $(l_i, e_i, v_i :: s_i, [])$.

**Theorem 4.** *If* $(\Phi_1, \Phi_2) \in \mathcal{E}[\![\tau]\!]_{(c_1,c_2),\eta}$ *then* $\Phi_1, \Phi_2$ *both diverges or there exists* $(v_1, v_2) \in \mathcal{V}[\![\tau]\!]_{(c_1,c_2),\eta}$ *such that* $\Phi_i \mapsto (l_i, e_i, [v_i], [])$.

*Proof.*
- Direct side : If $(\Phi_1, \Phi_2) \in \mathcal{E}[\![\tau]\!]_{(c_1,c_2),\eta}$ do not both diverge, then they both normalize to good final configurations since the empty context, which corresponds here to the code pointer to the instruction `Stop`, is in the diagonal of $\mathcal{K}[\![\tau]\!]_{(c_1',c_2'),\eta}$. So from the previous theorem there exists values $(v_1, v_2) \in \mathcal{V}[\![\tau]\!]_{(c_1,c_2),\eta}$ s.t. $\Phi_i \mapsto^* (l_i, e_i, [v_i], [])$.
- Reverse side : Let $(k_1, k_2) \in \mathcal{K}[\![\tau]\!]_{(c_1',c_2'),\eta}$, then from $(v_1, v_2) \bot_{c_1', c_2'} (k_1, k_2)$ we get $(\Phi_1, \Phi_2) \bot_{c_1', c_2'} (k_1, k_2)$, i.e. $(\Phi_1, \Phi_2) \in \mathcal{E}[\![\tau]\!]_{(c_1,c_2),\eta}$.

Then, using this result on can again prove our fundamental lemma for our language with Curry-style polymorphism and intersection types :

**Theorem 5 (Fundamental property).** *Let* $t$ *be a term such that* $\Delta, \Gamma \vdash t : \tau$, *then for every environment* $\gamma$ *and* $\eta$ *compatible respectively with* $\Gamma$ *and* $\Delta$, *and* $c = \mathbf{compile}(\gamma t_1, 0)$ *then* $(0, [], [], [])$ *is in the diagonal of* $\mathcal{E}[\![\tau]\!]_{(c,c),\eta}$

As said in part 1.5, this result also extend to implicit dependent types, and so it solves one of the open problems of [5].

### 3.3 Code pointer equality

Finally, we are going to adapt results of Section 2 to our low-level setting. We are not going to try to compile our $\alpha$-equivalence, since we are working directly with De Bruijn indices. We will rather add a new instruction `EqPoint` which is able to test code pointer equality in closures.

$$c(l) = \texttt{EqPoint} : (l, e, \texttt{C}(l_1, e_1) :: \texttt{C}(l_2, e_2) :: s, d) \succ_c (l+1, e, \texttt{V}(l_1 = l_2) :: s, d)$$

Then just like in Section 2 one has to make the distinction between the two set of instructions, to forbid values to use `EqPoint`. However we cannot do that syntactically on the code $c$, otherwise we would also forbid contexts to use it, which is definitely not what we want to do (or the result would be trivial). So we have to define it semantically : $\texttt{C}(l, e) \in \mathrm{Val}_{good,c}$ iff every reduction made from the configuration $(l, e, s, d)$ with code $c$ does not use the instruction `EqPoint`. Then logical relations $\mathcal{V}[\![\tau]\!]_{c_1,c_2}$ are defined restricted to $\mathrm{Val}_{good,c_1} \times \mathrm{Val}_{good,c_2}$.

In the same way, one define what is a good context $(l, e, s)$ in $\mathrm{Cont}_{good,c}$ and a good configuration $(l, e, s, d) \in \mathrm{Conf}_{good,c}$. Then one can easily show the equality of Theorem 3, defining $\mathcal{K}[\![\tau]\!]_{(c_1,c_2),\eta}$ restricted to $\mathrm{Cont}_{good,c_1} \times \mathrm{Cont}_{good,c_2}$ , $\mathcal{E}[\![\tau]\!]_{(c_1,c_2),\eta}$ as

$$\{(\Phi_1, \Phi_2) \mid \Phi_1 \in \mathrm{Conf}_{good,c_1}, \Phi_2 \in \mathrm{Conf}_{good,c_2} \text{ and } \forall (c_1', c_2') \sqsupseteq (c_1 \texttt{++}[\texttt{Return}], c_2 \texttt{++}[\texttt{Return}]), \forall (k_1, k_2) \in \mathcal{K}[\![\tau]\!]_{(c_1',c_2'),\eta}.(\Phi_1, \Phi_2) \bot_{c_1',c_2'} (k_1, k_2)\}$$

and $\mathcal{E}_\alpha[\![\tau]\!]_{(c_1,c_2),\eta}$ as

$$\{(\Phi_1, \Phi_2) \mid t_1 \in \mathrm{Conf}_{good,c_1}, t_2 \in \mathrm{Conf}_{good,c_2} \text{ and } \forall (c_1', c_2') \sqsupseteq (c_1 \texttt{++}[\texttt{Return}], c_2 \texttt{++}[\texttt{Return}]), \forall (k_1, k_2) \in \mathcal{K}_\alpha[\![\tau]\!]_{(c_1',c_2'),\eta}.(\Phi_1, \Phi_2) \bot_{c_1',c_2'} (k_1, k_2)\}$$

**Theorem 6.** *For every type $\tau$ and every environment $\eta$, $\mathcal{E}_\alpha[\![\tau]\!]_\eta = \mathcal{E}[\![\tau]\!]_\eta$.*

*Proof.* The proof of $\mathcal{E}[\![\tau]\!]_\eta \subseteq \mathcal{E}_\alpha[\![\tau]\!]_\eta$ is the same than in Theorem 3. The other side is proved using the fact that $\mathrm{Cont}_{good,c} \subseteq \mathrm{Cont}$.

## References


1. A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. *Programming Languages and Systems*, pages 69–83, 2006.
2. A.W. 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):657–683, 2001.
3. N. Benton and C.-K. Hur. Step-indexing: the good, the bad and the ugly. In *Proceedings of Dagstuhl Seminar 10351: Modelling, Controlling and Reasoning About State*.
4. N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 97–108, New York, NY, USA, 2009. ACM.

5. N. Benton and N. Tabareau. Compiling functional types to relational specifications for low level imperative code. In *TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation*, pages 3–14, New York, NY, USA, 2009. ACM.
6. C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. *POPL '11: Proceedings of the 38th ACM Symposium on Principles of Programming Languages*, 46(1):133–146, 2011.
7. G. Jaber and N. Tabareau. Krivine realizability for compiler correctness. *LOLA workshop*, 2010.
8. G. Jaber and N. Tabareau. Decomposing Logical Relations with Forcing. *Submitted*, 2011.
9. A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and Separation in Hoare Type Theory. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming*, page 73. ACM, 2006.
10. A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*. CUP, 1998.