THEORETICAL PEARLS

Correctness of Compiling Polymorphism to Dynamic Typing

Kuen-Bang Hou (Favonia)

Carnegie Mellon University
(e-mail: favonia@cs.cmu.edu)

Nick Benton

Microsoft Research
(e-mail: nick@microsoft.com)

Robert Harper

Carnegie Mellon University
(e-mail: rwh@cs.cmu.edu)

Abstract

The connection between polymorphic and dynamic typing was originally considered by Curry, *et al.* in the form of “polymorphic type assignment” for untyped λ -terms. Types are assigned after the fact to what is, in modern terminology, a dynamic language. Interest in type assignment was revitalized by the proposals of Bracha, *et al.* and Bank, *et al.* to enrich Java with polymorphism (generics), which in turn sparked the development of other languages, such as Scala, with similar combinations of features. In that setting it is desirable to compile polymorphism to dynamic typing in such a way that as much static typing as possible is preserved, relying on dynamics only insofar as genericity is actually required.

The basic approach is to compile polymorphism using embeddings from each type into a universal ‘top’ type, dyn , and partial projections that go in the other direction. This scheme is intuitively reasonable, and, indeed, has been used in practice many times. Proving its correctness, however, is non-trivial. This paper studies the compilation of System F to an extension of Moggi’s computational metalanguage with a dynamic type and shows how the compilation may be proved correct using a logical relation.

1 Introduction

The close connection between polymorphic and dynamic typing was first explored by Curry *et al.* (1972) under the name “polymorphic type assignment.” Types are assigned to untyped λ -terms using rules such as these:

$$\frac{M : A}{M : \forall X. A} \qquad \frac{M : \forall X. A}{M : A[B/X]}$$

According to this view, which Reynolds (2003) terms *extrinsic*, a term, M , already has a definite meaning (say, as an element of a reflexive domain); types are assigned after the

fact to express some aspects of their behavior. For example, $\lambda x.x$ has the type $\forall X.X \rightarrow X$, essentially because it may be seen as having all types of the form $A \rightarrow A$ for any choice of A .

This approach to typing inspired much work on its adaptation to programming languages, including the original formulation of the highly influential ML type system by Milner (1978). However, a more refined analysis reveals that the type assignment viewpoint is not the only possible way to understand polymorphism in programming languages. For example, the Girard-Reynolds polymorphic typed λ -calculus, known as System F (Girard, 1972; Reynolds, 1974), expresses a subtly different concept characterized by these rules:

$$\frac{M : A}{\Lambda X.M : \forall X.A} \qquad \frac{M : \forall X.A}{MB : A[B/X]}$$

Here type abstraction and application are an explicit part of the construction of the term. More generally, according to this *intrinsic* view, types are seen as defining what terms exist, rather than describing the behavior of pre-existing terms.

A key difference between the two approaches lies in their execution behavior when endowed with an operational semantics. Under a type assignment regime the programs are given independently of any type information. Such programs may be seen as dynamically typed, which, in the presence of multiple classes of data, implies run-time overhead to express and enforce proper classification of values. Types are purely descriptive, and play no role in determining the run-time behavior of a program. In contrast, under a type checking regime the formation of programs and their execution behavior are influenced by, and in some cases determined by, the types involved. This leads to better execution behavior, because it avoids the overhead of dynamic class checks from the outset, and hence is more amenable to modular compilation and composition.

Interest in polymorphic type assignment was revitalized by the proposals of Bracha *et al.* (1998) and Bank *et al.* (1997) to enrich Java with polymorphism (“generics”), which then inspired similar treatment of generics in languages such as C[‡] and Scala. Such languages are statically typed, but feature a universal type (`Object` in Java, but herein called `dyn`) of dynamically typed values. The question arose as to how to compile these extensions, given that little or no change could be made to the language’s established monomorphic run-time structure. Abstracting from the language-specific details, the question may be re-phrased as “how can we compile System F to a simply-typed language, which will here be called λ_{dyn} , with a type of dynamic values?”

From the point of view of classical type assignment, nothing needs to be done other than to “erase” the polymorphic types. But, as previously remarked, this is not quite adequate, both because of the overhead of dynamic typing, and because these languages admit static types, such as integers and arrays, that are handled specially. A more refined question, then, is how we may compile polymorphism in such a way that static type information is preserved as much as possible? So, for example, a function of type $\text{nat} \rightarrow \text{nat}$ ought to be compiled as such, and not incur the overhead of class checking inherent in dynamic languages. This is clearly a laudable goal, but it is not obvious that it can always be achieved, because the $\text{nat} \rightarrow \text{nat}$ function in question might be (or involve) an instance of a polymorphic term. The problem is that a polymorphic function such as $\Lambda X.\lambda x:X.x$ is naturally compiled as $\lambda x:\text{dyn}.x$ of type $\text{dyn} \rightarrow \text{dyn}$, reflecting the genericity of the argu-

ment x in the body of the function. Corresponding to the instantiation of the polymorphic function at the type nat are a pre- and post-composition of the dynamically typed identity with untagging and tagging operations, respectively. On the other hand we can at least ensure that, for example, the obvious doubling function compiles to the very same doubling function, because it does not involve polymorphism.

So, roughly speaking, the overall plan is to translate a Sytem F type A into a λ_{dyn} type $A^\dagger$ for some structure-preserving transformation $(\cdot)^\dagger$ in which occurrences of type variables, X , map to dyn . Rather than replace A *in toto* by the single type dyn , as is done in type assignment, we instead preserve as much of the static type information as possible, retreating to dynamic values only where polymorphism is truly required. This immediately raises the question of how to relate $(A[B/X])^\dagger$ to $A^\dagger$, which is to say how to manage polymorphic instantiation. Indeed, this is the heart of the translation given by the aforementioned authors.

The main idea is to rely on the existence of an embedding, i , of each λ_{dyn} type into the type dyn , equipped with a corresponding projection, j , which recovers the embedded object, which is to say that j is post-inverse to i , up to observational equivalence, $j \circ i \cong \text{id}$. Notice that j is not also pre-inverse to i , because there is no reason to expect that a given value of type dyn lies in the image of the embedding. Put in other terms, every λ_{dyn} type is a retract of dyn , with retraction given by the composition $i \circ j$, an idempotent endomorphism of dyn .

The embedding of every λ_{dyn} type into dyn lifts functorially into an embedding, I , from $(A[B/X])^\dagger$ to $A^\dagger$, accompanied by a corresponding projection, J , going in the other direction. These lifted embeddings and projections are used to mediate polymorphic instantiation, so that an instantiation of the polymorphic identity (given above) at the type nat is translated to the function of type $\text{nat} \rightarrow \text{nat}$

$$\lambda x:\text{nat}. j_{\text{nat}}((\lambda x:\text{dyn}. x) i_{\text{nat}}(x)),$$

where i_{nat} and j_{nat} are the embedding and projection pair for nat . The pre- and post-compositions with the embeddings and projections arise from the functorial action of the type constructor $X \mapsto X$, thought of as a function of X . To projection to type $\text{nat} \rightarrow \text{nat}$ requires that we embed the argument into dyn , execute the translation of the polymorphic identity, and project the result back to dyn . This function is observationally equivalent to the identity on nat , because the context may only provide natural numbers as arguments, and expect natural numbers as results.

This general idea leads to a relatively straightforward, and generally well-understood, translation of System F into λ_{dyn} . Our contribution is not in the method of compilation, but rather in the means by which it may be proved correct. The overall goal is to show that an expression and its compilation are observably equivalent, which means that they engender the same behavior in all (appropriately equivalent) program contexts of observable type (say, the booleans). A simple induction on types or terms just using the properties of embeddings and projections does not work; nor did our initial attempt at using a logical relations argument inspired by that used by Meyer & Wand (1985) to show correctness of CPS translation for the simply-typed lambda calculus. That attempt used a relation that explicitly mentioned the projection functions. Here we present a more carefully formulated parametric logical relation that directly relates terms with their translations, together with

$$\begin{array}{lcl}
A, B & ::= & X \mid \mathbf{nat} \mid A \rightarrow B \mid \forall X. A \\
M, N & ::= & x \mid \mathbf{z} \mid \mathbf{suc}(M) \mid \mathbf{ifz}(M; N_0; x. N_1) \mid \lambda x: A. M \mid MN \mid \Lambda X. M \mid MA \\
\Delta & ::= & \cdot \mid \Delta, X \\
\Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}$$

$$\begin{array}{c}
\boxed{\Delta \vdash A} \\
\\
\Delta, X \vdash X \qquad \Delta \vdash \mathbf{nat} \qquad \frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \rightarrow B} \qquad \frac{\Delta, X \vdash A}{\Delta \vdash \forall X. A} \\
\\
\boxed{\Delta; \Gamma \vdash M : A} \\
\\
\Delta; \Gamma, x : A \vdash x : A \qquad \Delta; \Gamma \vdash \mathbf{z} : \mathbf{nat} \qquad \frac{\Delta; \Gamma \vdash M : \mathbf{nat}}{\Delta; \Gamma \vdash \mathbf{suc}(M) : \mathbf{nat}} \\
\\
\frac{\Delta; \Gamma \vdash M : \mathbf{nat} \quad \Delta; \Gamma \vdash N_0 : A \quad \Delta; \Gamma, x : \mathbf{nat} \vdash N_1 : A}{\Delta; \Gamma \vdash \mathbf{ifz}(M; N_0; x. N_1) : A} \qquad \frac{\Delta; \Gamma \vdash M : A \rightarrow B \quad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash MN : B} \\
\\
\frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda x: A. M : A \rightarrow B} \qquad \frac{\Delta \vdash \Gamma \quad \Delta, X \vdash A \quad \Delta, X; \Gamma \vdash M : A}{\Delta; \Gamma \vdash \Lambda X. M : \forall X. A} \qquad \frac{\Delta; \Gamma \vdash M : \forall X. A}{\Delta; \Gamma \vdash MB : A[B/X]}
\end{array}$$

Fig. 1. Syntax and type rules for System F

a separate key lemma that captures the way in which the relation respects the embeddings and projections. This allows us to give what is, to our knowledge, the first correctness proof of this method of compiling polymorphism (generics) to dynamic typing. A secondary contribution is the proof method itself, which may be useful in other settings.

2 Source Language

Our source language is System F, the Girard-Reynolds polymorphic lambda calculus, extended with natural numbers and conditionals. The syntax and typing rules are shown in Figure 1. We say a type A is *closed* if $\cdot \vdash A$.

The source language is equipped with the usual transition relation on open terms, which is the precongruence closure of the following rules:

$$\begin{array}{lcl}
(\Lambda X. M) B & \rightsquigarrow & M[B/X] \\
(\lambda x : A. M) N & \rightsquigarrow & M[N/x] \\
\mathbf{ifz}(\mathbf{z}; N_0; x. N_1) & \rightsquigarrow & N_0 \\
\mathbf{ifz}(\mathbf{suc}(M); N_0; x. N_1) & \rightsquigarrow & N_1[M/x]
\end{array}$$

3 Target Language

We translate source terms into expressions of λ_{dyn} , an extension of Moggi's (1991) computational metalanguage, λML_T . The computational metalanguage is a simply-typed lambda calculus with the usual equational theory of CCCs (in particular, β and η laws), extended

with a type constructor, $\mathcal{T}$, with associated introduction and elimination forms, and equations corresponding to a strong monad. This language may be modelled in, for example, the category of ω -cpo (predomains).¹

For our translation of polymorphism, the monad serves two roles. Firstly, the equational axioms required of the universal type (making it a model of an untyped lambda calculus) imply the existence of fixpoint operators implementing general recursion, and hence divergent computations. The monad accounts for this possibility of divergence. Secondly, when a value is projected out of the universal type to be used at a more specific type, there is the possibility of a runtime error. For example, injecting a function value into the universal type and then projecting (casting) it out as an natural number should cause a dynamic error. The monad also accounts for such errors.

The translation is, however, parametric in exactly what the monad is, provided that it *is* a monad, and also has the extra structure required to interpret dynamic errors. We simply require that $\mathcal{T}$ comes equipped with a polymorphic constant $\text{err} : \mathcal{T}D$ for any D . For example

1. Take $\mathcal{T}D = D_\perp$, the lifting monad, and $\text{err} = \perp$, so dynamic errors are just modelled by divergence.
2. Take $\mathcal{T}D = (1 + D)_\perp$, the lifted error (maybe, option) monad, and $\text{err} = [\text{inl}(*)]$, so dynamic errors are modelled by a terminating, failing computation.

We abuse notation slightly by writing $[\cdot]$ for the unit of the monad in all cases. We also abbreviate the usual monadic bind construct $\text{let } x \leftarrow e \text{ in } f$ by just $x \leftarrow e; f$.

λ_{dyn} has a base type $\mathbb{N}$ for the natural numbers (thought of as a discrete cpo). If $n : \mathbb{N}$ is a natural, we write $\bar{n}$ for the corresponding System F normal form $\text{suc}(\dots(\text{suc}(z)\dots))$.

λ_{dyn} also has a universal type dyn equipped with operations

$$\begin{array}{ll} \text{num} : \mathbb{N} \rightarrow \text{dyn} & \text{num?} : \text{dyn} \rightarrow \mathcal{T}\mathbb{N} \\ \text{fun} : (\text{dyn} \rightarrow \mathcal{T}\text{dyn}) \rightarrow \text{dyn} & \text{fun?} : \text{dyn} \rightarrow \mathcal{T}(\text{dyn} \rightarrow \mathcal{T}\text{dyn}) \end{array}$$

subject to the equations

$$\begin{array}{ll} \text{num?}(\text{num } n) &= [n] \\ \text{num?}(x) &= \text{err otherwise} \\ \text{fun?}(\text{fun } f) &= [f] \\ \text{fun?}(x) &= \text{err otherwise} \end{array}$$

The canonical way to satisfy these requirements is to take dyn to be the least solution to the recursive domain equation

$$\text{dyn} \cong \mathcal{T}(\text{nat} + (\text{dyn} \rightarrow \mathcal{T}\text{dyn}))$$

¹ Note that although the source language is presented operationally, via a transition relation, we take an equational view of the target. We could, of course, have presented the target using an operational semantics as well, and derived a notion of contextual equivalence. But it is entirely standard that the equational theory of λML_T is adequate for contextual equivalence for an operational semantics of the metalanguage itself, or for a non-monadically-typed CBV language translated into the metalanguage.

$$\begin{array}{rcl}
& \boxed{A^\dagger} & \\
\text{nat}^\dagger & = & \mathbb{N} \\
X^\dagger & = & \text{dyn} \\
(A \rightarrow B)^\dagger & = & A^\dagger \rightarrow \mathcal{T}B^\dagger \\
(\forall X.A)^\dagger & = & A^\dagger
\end{array}$$

Fig. 2. Translation of types

with

```

num(x) = roll([inl(x)])
num?(d) = x ← unroll(d); case x of inl(n) ⇒ [n] | inr(f) ⇒ err
fun f = roll([inr(f)])
fun?(d) = x ← unroll(d); case x of inl(n) ⇒ err | inr(f) ⇒ [f]

```

where `roll()` and `unroll()` are the components of the isomorphism in the solution of the equation for `dyn`. However, nothing that follows relies on other properties of any specific domain: we just need the equations.

It is interesting to observe that the correctness of the translation does not actually *require* any interesting properties of errors. In particular, we do not need to specify that `err` is natural, that the monad is strict in errors (i.e. that $x \leftarrow \text{err}; f = \text{err}$), or even that errors are disjoint from values ($\forall d, \text{err} \neq [d]$), though these properties do hold for our examples of concrete monads. Indeed, one could remove errors entirely, replacing them with arbitrary default values, without materially affecting what follows. The reason for this is that the correctness theorem only talks about ‘good’ behaviour – if everything in the context is good then the translated term is good – so the precise nature of ‘bad’ is not very important. But in any realistic setting, in which one needs to link translated code with target code obtained by other means, one would want to use a well-structured error mechanism.

4 Translation

The translation/interpretation of types follows Moggi’s (1991) call-by-value translation, with type variables interpreted as the universal type, `dyn`. This is shown in Figure 2

4.1 Embeddings and Projections

Before we can define the translation of terms, we need some auxiliary definitions on the target side. First we have an embedding, i , and a projection, j , mapping between $A^\dagger$ and `dyn` for each source type A . The definitions are shown in Figure 3. Note that the embedding is total (any value of type $A^\dagger$ can be mapped into the universal domain), although the projection is partial, which is why the monad appears in the return type. Only well-behaved elements of `dyn` may be mapped back to $A^\dagger$; projecting an ill-behaved value may fail immediately or, in the case of function types, when the projected value is later actually applied.

$$\begin{array}{lcl}
\boxed{i_A : A^\dagger \rightarrow \mathbf{dyn}} & \text{and} & \boxed{j_A : \mathbf{dyn} \rightarrow \mathcal{T}A^\dagger} \\
i_{\mathbf{nat}}(x) & = & \mathbf{num}\,x \\
j_{\mathbf{nat}}(x) & = & \mathbf{num?}\,x \\
i_X(x) & = & x \\
j_X(x) & = & [x] \\
i_{A \rightarrow B}(f) & = & \mathbf{fun}\,\lambda d:\mathbf{dyn}.v \leftarrow j_A(d); r \leftarrow f\,v; [i_B(r)] \\
j_{A \rightarrow B}(f) & = & f' \leftarrow \mathbf{fun?}\,f; [\lambda a.r \leftarrow f'(i_A(a)); j_B(r)] \\
i_{\forall X.A}(x) & = & i_A(x) \\
j_{\forall X.A}(x) & = & j_A(x)
\end{array}$$

Fig. 3. Embeddings and projections

An embedding followed by the corresponding projection is always morally the identity (actually, the unit of the monad):

Lemma 1

For any A and $x : A^\dagger$,

$$j_A(i_A(x)) = [x]$$

Proof

Induction on the structure of A .

- $\mathbf{nat}$

$$j_{\mathbf{nat}}(i_{\mathbf{nat}}(x)) = [x]$$

- X

$$j_{\mathbf{nat}}(i_{\mathbf{nat}}(x)) = [x]$$

- $A \rightarrow B$

$$\begin{aligned}
& j_{A \rightarrow B}(i_{A \rightarrow B}(f)) \\
&= f' \leftarrow \mathbf{fun?}(\mathbf{fun}\,\lambda d:\mathbf{dyn}.v \leftarrow j_A(d); r \leftarrow f\,v; [i_B(r)]); [\lambda a.r \leftarrow f'(i_A(a)); j_B(r)] \\
&= f' \leftarrow [\lambda d:\mathbf{dyn}.v \leftarrow j_A(d); r \leftarrow f\,v; [i_B(r)]]; [\lambda a.r \leftarrow f'(i_A(a)); j_B(r)] \\
&= [\lambda a.r \leftarrow (v \leftarrow j_A(i_A(a)); r \leftarrow f\,v; [i_B(r)]); j_B(r)] \\
&= [\lambda a.r \leftarrow (r \leftarrow f\,a; [i_B(r)]); j_B(r)] \\
&= [\lambda a.r \leftarrow f\,a; j_B(i_B(r))] \\
&= [\lambda a.r \leftarrow f\,a; [r]] \\
&= [\lambda a.f\,a] \\
&= [f]
\end{aligned}$$

- $\forall X.A$

$$j_{\forall X.A}(i_{\forall X.A}(x)) = j_A(i_A(x)) = [x]$$

□

$$\boxed{I_{X.A}^B : (A[B/X])^\dagger \rightarrow A^\dagger} \text{ and } \boxed{J_{X.A}^B : A^\dagger \rightarrow \mathcal{T}(A[B/X]^\dagger)}$$

$$\begin{array}{ll}
I_{X.X}^B(x) &= i_B(x) \\
J_{X.X}^B(x) &= j_B(x) \\
I_{X.Y}^B(x) &= x \\
J_{X.Y}^B(x) &= [x] \\
I_{X.\text{nat}}^B(x) &= x \\
J_{X.\text{nat}}^B(x) &= [x] \\
I_{X.A_1 \rightarrow A_2}^B(f) &= \lambda a. a' \leftarrow J_{X.A_1}^B(a); r \leftarrow f a'; [I_{X.A_2}^B(r)] \\
J_{X.A_1 \rightarrow A_2}^B(f) &= [\lambda a. r \leftarrow f(I_{X.A_1}^B(a)); J_{X.A_2}^B(r)] \\
I_{X.\forall Y.A}^B(x) &= I_{X.A}^B(x) \\
J_{X.\forall Y.A}^B(x) &= J_{X.A}^B(x)
\end{array}$$

Fig. 4. Lifted embeddings and projections

In the concrete situation where $\mathcal{T}$ is lifting on cpos, there is an inequality that provides a partial converse to Lemma 1, viz. $x \leftarrow j_A(d); [i_A(x)] \sqsubseteq [d]$, but one cannot say much in the more general case.

4.2 Lifted Embeddings and Projections

The translation $A^\dagger$ of a type A with a free type variable X has dyn in positions corresponding to the occurrences of X in A . Type application in the source involves substitution of a type B for those occurrences of X ; translating the application requires the use of functions $J_{X.A}^B$ from $A^\dagger$ to $\mathcal{T}(A[B/X]^\dagger)$, the monad applied to the translation of the substituted type. The result is wrapped in the monad because dyn -values produced by the argument in places corresponding to positive occurrences of X in A are not necessarily well-behaved. Just as with the embeddings and projections of the previous section, the definition of $J_{X.A}^B$ is not only inductive on A , but mutually inductive with that of a function going in the other direction, $I_{X.A}^B$ from $A[B/X]^\dagger$ to $A^\dagger$. The definitions are shown in Figure 4.

The following is a lifted version of Lemma 1.

Lemma 2

$$\Gamma \vdash J_{X.A}^B(I_{X.A}^B(x)) = [x]$$

Proof

Induction on A .

- $X.X$

$$J_{X.X}^B(I_{X.X}^B(x)) = j_B(i_B(x)) = [x]$$

- $X.Y$

$$J_{X.Y}^B(I_{X.Y}^B(x)) = [x]$$

- $X.\text{nat}$

$$J_{X.\text{nat}}^B(I_{X.\text{nat}}^B(x)) = [x]$$

	$(\Delta; \Gamma \vdash M : A)^*$	
$(\Delta; \Gamma, x:A \vdash x : A)^*$	$=$	$[x]$
$(\Delta; \Gamma \vdash z : \mathbf{nat})^*$	$=$	$[0]$
$(\Delta; \Gamma \vdash \mathbf{succ}(M) : \mathbf{nat})^*$	$=$	$d \leftarrow (\Delta; \Gamma \vdash M : \mathbf{nat})^*; [d+1]$
$(\Delta; \Gamma \vdash \mathbf{ifz}(M; N_0; x.N_1) : A)^*$	$=$	$d \leftarrow (\Delta; \Gamma \vdash M : \mathbf{nat})^*;$ $\mathbf{ifz}(d; (\Delta; \Gamma \vdash N_0 : A)^*; x.(\Delta; \Gamma, x : \mathbf{nat} \vdash N_1 : A)^*)$
$(\Delta; \Gamma \vdash \lambda x:A.M : A \rightarrow B)^*$	$=$	$[\lambda x:A^\dagger. (\Delta; \Gamma, x:A \vdash M : B)^*]$
$(\Delta; \Gamma \vdash MN : B)^*$	$=$	$e \leftarrow (\Delta; \Gamma \vdash M : A \rightarrow B)^*; d \leftarrow (\Delta; \Gamma \vdash N : A)^*; ed$
$(\Delta; \Gamma \vdash \Lambda X.M : \forall X.A)^*$	$=$	$(\Delta, X; \Gamma \vdash M : A)^*$
$(\Delta; \Gamma \vdash MB : A[B/X])^*$	$=$	$d \leftarrow (\Delta; \Gamma \vdash M : \forall X.A)^*; J_{X.A}^B(d)$

Fig. 5. Term translation

- $X.A_1 \rightarrow A_2$

$$\begin{aligned}
& J_{X.A_1 \rightarrow A_2}^B(I_{X.A_1 \rightarrow A_2}^B(x)) \\
&= [\lambda a.r \leftarrow (\lambda a.d' \leftarrow J_{X.A_1}^B(a); r \leftarrow f a'; [I_{X.A_2}^B(r)])(I_{X.A_1}^B(a)); J_{X.A_2}^B(r)] \\
&= [\lambda a.r \leftarrow (a' \leftarrow J_{X.A_1}^B(I_{X.A_1}^B(a)); r \leftarrow f a'; [I_{X.A_2}^B(r)]); J_{X.A_2}^B(r)] \\
&= [\lambda a.r \leftarrow (r \leftarrow f a; [I_{X.A_2}^B(r)]); J_{X.A_2}^B(r)] \\
&= [\lambda a.r \leftarrow f a; J_{X.A_2}^B(I_{X.A_2}^B(r))] \\
&= [\lambda a.r \leftarrow f a; [r]] \\
&= [\lambda a.f a] \\
&= [f]
\end{aligned}$$

- $\forall X.A$

$$j_{\forall X.A}(i_{\forall X.A}(x)) = j_A(i_A(x)) = [x]$$

□

4.3 Term Translation

Just as was the case for types, the translation $(\cdot)^*$ of terms in context is Moggi's usual CBV translation, extended to use $J_{X.A}^B$ to translate type application. The formal definition is shown in Figure 5.

Note that uniqueness of typing in the source language ensures that the type A appearing on the right hand side of the application case is uniquely determined, so this is indeed a good definition. It is also appropriately typed:

Lemma 3

If $\Delta; \Gamma \vdash M : A$ then $\Gamma^\dagger \vdash (\Delta; \Gamma \vdash M : A)^* : \mathcal{S}A^\dagger$. □

$$\boxed{\mathcal{R}_A^w \subseteq \text{CT}(A[B_j/X_j]) \times A^\dagger}$$

$$\begin{aligned}
\mathcal{R}_{X_i}^w &= \mathcal{R}_i \\
\mathcal{R}_{\text{nat}}^w &= \{(M, n) \mid M \rightsquigarrow^* \bar{n}\} \\
\mathcal{R}_{A_1 \rightarrow A_2}^w &= \{(M, d) \mid M \rightsquigarrow^* \lambda x : A_1[B_j/X_j].M' \wedge \\
&\quad \forall (M_2, d_2) \in \mathcal{R}_{A_1}^w, (M'[M_2/x], d d_2) \in \mathcal{R}_{A_2}^w\} \\
\mathcal{R}_{\forall X.A}^w &= \{(M, d) \mid M \rightsquigarrow^* \Lambda X.M' \wedge \forall (B, \mathcal{R}) \in \mathcal{F}, (M'[B/X], d) \in \mathcal{R}_A^{w, X \mapsto (B, \mathcal{R})}\} \\
&\quad \text{where } \mathcal{R}_A^w = \{(M, [d]) \mid (M, d) \in \mathcal{R}_A^w\}.
\end{aligned}$$

Fig. 6. Logical relation

5 Logical Relation

If B is a closed source type, write $\text{CT}(B) = \{M \mid \cdot; \cdot \vdash M : B\}$ for the set of closed terms of type B . Given $\mathcal{R} \subseteq \text{CT}(B) \times \text{dyn}$, a relation between closed source terms of type B and elements of dyn , then we say $\mathcal{R}$ is *admissible* if it closed under antireduction; that is

$$(M', d) \in \mathcal{R} \wedge M \rightsquigarrow M' \implies (M, d) \in \mathcal{R}.$$

We write $\Delta \vdash w$ to mean that the *world* w is a map from the finite set of type variables Δ to pairs comprising a closed type and an admissible relation on that type. Formally

$$w : \Delta \rightarrow \mathcal{F}, \text{ where } \mathcal{F} = \sum_{\{B \mid \vdash B\}} \{\mathcal{R} \subseteq \text{CT}(B) \times \text{dyn} \mid \mathcal{R} \text{ is admissible}\}.$$

If $\Delta = X_1, \dots, X_n$ and $w(X_j) = (B_j, \mathcal{R}_j)$ for each $1 \leq j \leq n$, then we define the relations $\mathcal{R}_A^w \subseteq \text{CT}(A[B_j/X_j]) \times A^\dagger$ and $\mathcal{R}_A^w \subseteq \text{CT}(A[B_j/X_j]) \times A^\dagger$ and for each A such that $\Delta \vdash A$ by induction on A as shown in Figure 6. To reduce the boilerplates in writing, we also define the relation $\mathcal{R}_A^w = \{(M, [d]) \mid (M, d) \in \mathcal{R}_A^w\} \subseteq \text{CT}(A[B_j/X_j]) \times \mathcal{T}(A^\dagger)$.

Observe that, as in other work on relationally parametric models of polymorphism (Reynolds, 1983), the clause for polymorphic types involves quantification over *all* relations from a pre-defined set. This enforces parametricity and also avoids a potential circularity due to impredicativity, which would arise were one to consider instantiating just with $\mathcal{R}_B^w$ for each B .

Lemma 4 (Admissibility)

For all w and A , $\mathcal{R}_A^w$ and $\mathcal{R}_A^w$ are admissible. $\square$

Lemma 5 (Weakening)

If $\Delta \vdash A$ and $\Delta \vdash w$ then for any B and $\mathcal{R}$,

$$\mathcal{R}_A^{w, X \mapsto (B, \mathcal{R})} = \mathcal{R}_A^w.$$

$\square$

The crucial lemma is the following, which connects the logical relation at a substituted type, $A[B/X]$, with the relation at the type A in an extended world, mediated by the lifted embeddings and projections. The statement involves instantiating a type variable with a particular, well chosen, relation.

Lemma 6 (Type substitution)

Let $\Delta \vdash B$, $\Delta \vdash w$, and $w(X_j) = (B_j, \mathcal{R}_j)$ for each j . Define the admissible relation $\widehat{\mathcal{R}}_B^w \subseteq \text{CT}(B[B_j/X_j] \times \text{dyn})$ by $\widehat{\mathcal{R}}_B^w = \{(M, d) \mid (M, j_B(d)) \in \mathcal{IR}_B^w\}$ and the extended world $w' = w, X \mapsto (B[B_j/X_j], \widehat{\mathcal{R}}_B^w)$. Then for any A and M , with $\Delta, X \vdash A$, $\cdot \vdash M : (A[B/X])[B_j/X_j]$,

1. For any d , $(M, d) \in \mathcal{R}_{A[B/X]}^w$ implies $(M, I_{X.A}^B(d)) \in \mathcal{R}_A^{w'}$.
2. For any d , $(M, d) \in \mathcal{R}_A^{w'}$ implies $(M, J_{X.A}^B(d)) \in \mathcal{IR}_{A[B/X]}^w$.

Proof

The two parts are proved by simultaneous induction on A .

- X .
 1. $(M, j_B(i_B(d))) = (M, [d]) \in \mathcal{IR}_B^w$ and thus $(M, i_B(d)) \in \widehat{\mathcal{R}}_B^w = \mathcal{R}_X^{w'}$.
 2. By definition $\mathcal{R}_X^{w'} = \widehat{\mathcal{R}}_B^w$. Thus $(M, j_B(d)) \in \mathcal{IR}_B^w = \mathcal{IR}_{X[B/X]}^w$.
- Y .
 1. $(M, I_{X.A}^B(d)) = (M, d) \in \mathcal{R}_Y^w = \mathcal{R}_Y^{w'}$.
 2. By weakening $\mathcal{R}_Y^{w'} = \mathcal{R}_Y^w$. Therefore $(M, [d]) \in \mathcal{IR}_Y^w = \mathcal{IR}_{Y[B/X]}^w$.
- nat .
 1. $(M, I_{X.A}^B(d)) = (M, d) \in \mathcal{R}_{\text{nat}}^w = \mathcal{R}_{\text{nat}}^{w'}$.
 2. By weakening $\mathcal{R}_{\text{nat}}^{w'} = \mathcal{R}_{\text{nat}}^w$. Therefore $(M, [d]) \in \mathcal{IR}_{\text{nat}}^w = \mathcal{IR}_{\text{nat}[B/X]}^w$.
- $A_1 \rightarrow A_2$.
In either part $M \rightsquigarrow^* \lambda x. M'$ for some M' .

1. Expanding the definition, we know it is sufficient to show

$$(M'[M_2/x], a' \leftarrow J_{X.A_1}^B(d_2); r \leftarrow f a'; [I_{X.A_2}^B(r)]) \in \mathcal{IR}_{A_2}^{w'}$$

for any $(M_2, d_2) \in \mathcal{R}_{A_1}^{w'}$. Pick any such (M_2, d_2) . By inductive hypothesis $(M_2, J_{X.A_1}^B(d_2)) \in \mathcal{IR}_{A_1[B/X]}^w$, which means there is d'_2 such that $J_{X.A_1}^B(d_2) = [d'_2]$ and $(M_2, d'_2) \in \mathcal{R}_{A_1[B/X]}^w$.

By definition $(M'[M_2/x], f d'_2) \in \mathcal{IR}_{A_2[B/X]}^w$, which is to say there exists r' such that $f d'_2 = [r']$ and $(M'[M_2/x], r') \in \mathcal{R}_{A_2[B/X]}^w$. We can then simplify the goal as

$$(a' \leftarrow J_{X.A_1}^B(d_2); r \leftarrow f a'; [I_{X.A_2}^B(r)]) = (r \leftarrow f d'_2; [I_{X.A_2}^B(r)]) = [I_{X.A_2}^B(r')]$$

and thus it suffices to show

$$(M'[M_2/x], I_{X.A_2}^B(r')) \in \mathcal{R}_{A_2}^{w'},$$

which directly follows the inductive hypothesis.

2. Expanding the definition, we know it is sufficient to show

$$(M'[M_2/x], r \leftarrow d(I_{X.A_1}^B(d_2)); J_{X.A_2}^B(r)) \in \mathcal{IR}_{A_2[B/X]}^w$$

for any $(M_2, d_2) \in \mathcal{R}_{A_1}^{w'}$. Pick any such (M_2, d_2) . By inductive hypothesis $(M_2, I_{X.A_1}^B(d_2)) \in \mathcal{R}_{A_1}^{w'}$. Therefore

$$(M'[M_2/x], d(I_{X.A_1}^B(d_2))) \in \mathcal{IR}_{A_2}^{w'},$$

12

Favonia, N. Benton, and R. Harper

which means there exists r' such that $d(I_{X.A_1}^B(d_2)) = [r']$ and $(M'[M_2/x], r') \in \mathcal{R}_{A_2}^{w'}$. This implies

$$(r \leftarrow d(I_{X.A_1}^B(d_2)); J_{X.A_2}^B(r)) = J_{X.A_2}^B(l'),$$

and thus it suffices to show

$$(M'[M_2/x], J_{X.A_2}^B(r')) \in \mathcal{R}_{A_2[B/X]}^{w'}$$

which directly follows the inductive hypothesis.

- $\forall Y.A'$.

In either part $M \rightsquigarrow^* \Lambda Y.M'$ for some M' .

1. Expanding the definition, we know it is sufficient to show

$$(M'[C/Y], I_{X.A'}^B(d)) \in \mathcal{R}_{A'}^{w', Y \mapsto (C, \mathcal{R}_C)}$$

for any $(C, \mathcal{R}_C) \in \mathcal{F}$. Fix a pair $(C, \mathcal{R}_C)$. By definition

$$(M'[C/Y], d) \in \mathcal{R}_{A'[B/X]}^{w, Y \mapsto (C, \mathcal{R}_C)}$$

and by induction

$$(M'[C/Y], I_{X.A'}^B(d)) \in \mathcal{R}_{A'}^{w, Y \mapsto (C, \mathcal{R}_C), X \mapsto (B[B_j/X_j], \widehat{\mathcal{R}}_B^{w, Y \mapsto (C, \mathcal{R}_C)})}.$$

By weakening (and exchange, implicit in the treatment of worlds as maps), we have the goal

$$(M'[C/Y], I_{X.A'}^B(d)) \in \mathcal{R}_{A'}^{w', Y \mapsto (C, \mathcal{R}_C)}.$$

2. Expanding the definition, we know it is sufficient to show

$$(M'[C/Y], J_{X.A'}^B(d)) \in \mathcal{R}_{A'[B/X]}^{w, Y \mapsto (C, \mathcal{R}_C)}$$

for any $(C, \mathcal{R}_C) \in \mathcal{F}$. Fix a pair $(C, \mathcal{R}_C)$. By definition

$$(M'[C/Y], d) \in \mathcal{R}_{A'}^{w', Y \mapsto (C, \mathcal{R}_C)}$$

and, by weakening,

$$(M'[C/Y], d) \in \mathcal{R}_{A'}^{w, Y \mapsto (C, \mathcal{R}_C), X \mapsto (B[B_j/X_j], \widehat{\mathcal{R}}_B^{w, Y \mapsto (C, \mathcal{R}_C)})}.$$

By inductive hypothesis we have the desired statement

$$(M'[C/Y], J_{X.A'}^B(d)) \in \mathcal{R}_{A'[B/X]}^{w, Y \mapsto (C, \mathcal{R}_C)}.$$

□

Armed with Lemma 6, we are now in a position to show the ‘Fundamental lemma’: that each (open) source term is logically related to its translation. The relation is defined on closed terms, so the statement of the lemma involves substituting arbitrary types and relations for free type variables, and arbitrary – but related – closed source and target terms for free term variables.

Lemma 7 (Fundamental lemma)

Suppose $\Delta; \Gamma \vdash M : A$, where $\Delta = X_1, \dots, X_m$ and $\Gamma = x_1 : A_1, \dots, x_n : A_n$. Let w be such that $\Delta \vdash w$ and $w(X_j) = (B_j, \mathcal{R}_j)$ for each $1 \leq j \leq m$. Then for any list of source terms $V_i : A_i[B_j/X_j]$ and target terms $t_i : A_i^\dagger$, $1 \leq i \leq n$, such that $(V_i, t_i) \in \mathcal{R}_{A_i}^w$ for each i , we have

$$(M[B_j/X_j][V_i/x_i], (\Delta; \Gamma \vdash M : A)^*[t_i/x_i]) \in \mathcal{TR}_A^w.$$

Proof

Induction on the derivation of $\Delta; \Gamma \vdash M : A$. We first define some abbreviations, writing $\widehat{w}$ for the type substitution $[B_j/X_j]$, $\widehat{V}$ for the source term substitution $[V_i/x_i]$, and $\widehat{t}$ for the target term substitution $[t_i/x_i]$.

- x . Directly from the assumption.
- z . Directly from the definition.
- $\text{suc}(M)$.

By inductive hypothesis

$$(\widehat{V}(\widehat{w}(M)), \widehat{t}((\Delta; \Gamma \vdash M : \text{nat})^*)) \in \mathcal{TR}_{\text{nat}}^w.$$

By definition of the relation, there exists n such that $\widehat{t}((\Delta; \Gamma \vdash M : \text{nat})^*) = [n]$ and

$$(\widehat{V}(\widehat{w}(M)), n) \in \mathcal{R}_{\text{nat}}^w.$$

So $\widehat{V}(\widehat{w}(M)) \rightsquigarrow^* \bar{n}$ and, by the dynamics, $\widehat{V}(\widehat{w}(\text{suc}(M))) \rightsquigarrow^* \text{suc}(\bar{n})$. Hence

$$\begin{aligned} & (\widehat{V}(\widehat{w}(\text{suc}(M))), n+1) \in \mathcal{R}_{\text{nat}}^w && \text{by definition} \\ \implies & (\widehat{V}(\widehat{w}(\text{suc}(M))), [n+1]) \in \mathcal{TR}_{\text{nat}}^w && \text{definition} \\ \implies & (\widehat{V}(\widehat{w}(\text{suc}(M))), d \leftarrow [n]; [d+1]) \in \mathcal{TR}_{\text{nat}}^w && \text{monad} \\ \implies & (\widehat{V}(\widehat{w}(\text{suc}(M))), d \leftarrow \widehat{t}((\Delta; \Gamma \vdash M : \text{nat})^*); [d+1]) \in \mathcal{TR}_{\text{nat}}^w && \\ \implies & (\widehat{V}(\widehat{w}(\text{suc}(M))), \widehat{t}(d \leftarrow (\Delta; \Gamma \vdash M : \text{nat})^*; [d+1])) \in \mathcal{TR}_{\text{nat}}^w && \\ \implies & (\widehat{V}(\widehat{w}(\text{suc}(M))), \widehat{t}((\Delta; \Gamma \vdash \text{suc}(M) : \text{nat})^*)) \in \mathcal{TR}_{\text{nat}}^w && \text{translation} \end{aligned}$$

- $\text{ifz}(M; N_0; x.N_1)$.

By induction,

$$(\widehat{V}(\widehat{w}(M)), \widehat{t}((\Delta; \Gamma \vdash M : \text{nat})^*)) \in \mathcal{TR}_{\text{nat}}^w.$$

Hence there exists n such that $\widehat{t}((\Delta; \Gamma \vdash M : \text{nat})^*) = [n]$ and $(\widehat{V}(\widehat{w}(M)), n) \in \mathcal{R}_{\text{nat}}^w$, so $\widehat{V}(\widehat{w}(M)) \rightsquigarrow^* \bar{n}$. Then

$$\begin{aligned} & \widehat{t}((\Delta; \Gamma \vdash \text{ifz}(M; N_0; x.N_1) : A)^*) \\ = & \widehat{t}(d \leftarrow [n]; \text{ifz}(d; (\Delta; \Gamma \vdash N_0 : A)^*; x.(\Delta; \Gamma, x : \text{nat} \vdash N_1 : A)^*)) \\ = & \text{ifz}(n; \widehat{t}((\Delta; \Gamma \vdash N_0 : A)^*); x.\widehat{t}((\Delta; \Gamma, x : \text{nat} \vdash N_1 : A)^*)) \\ = & \begin{cases} \widehat{t}((\Delta; \Gamma \vdash N_0 : A)^*) & \text{if } n = 0 \\ \widehat{t}((\Delta; \Gamma, x : \text{nat} \vdash N_1 : A)^*)[n'/x] & \text{if } n = n' + 1 \end{cases} \end{aligned}$$

— In the case that $n = 0$, $\widehat{V}(\widehat{w}(\text{ifz}(M; N_0; x.N_1))) \rightsquigarrow^* \widehat{V}(\widehat{w}(N_0))$ and since, by induction,

$$(\widehat{V}(\widehat{w}(N_0)), \widehat{t}((\Delta; \Gamma \vdash N_0 : A)^*)) \in \mathcal{TR}_A^w,$$

we are done by Lemma 4 (admissibility).

14

Favonia, N. Benton, and R. Harper

- If $n = n' + 1$ then $\widehat{V}(\widehat{w}(\text{ifz}(M; N_0; x.N_1))) \rightsquigarrow^* \widehat{V}(\widehat{w}(N_1))[\overline{n'}/x]$. Since $(\overline{n'}, n') \in \mathcal{R}_{\text{nat}}^w$, induction gives

$$(\widehat{V}(\widehat{w}(N_1))[\overline{n'}/x], \hat{i}((\Delta; \Gamma, x : \text{nat} \vdash N_1 : A)^*)[n'/x]) \in \mathcal{TR}_A^w$$

we are again done by Lemma 4.

- $\lambda x : A.M$.

Since $\hat{i}((\Delta; \Gamma \vdash \lambda x : A.M : A \rightarrow B)^*) = [\lambda x : A^\dagger. \hat{i}((\Delta; \Gamma, x : A \vdash M : B)^*)]$ it suffices to show

$$(\widehat{V}(\widehat{w}(\lambda x : A.M)), \lambda x : A^\dagger. \hat{i}((\Delta; \Gamma, x : A \vdash M : B)^*)) \in \mathcal{R}_{A \rightarrow B}^w$$

Suppose $(M_2, d_2) \in \mathcal{R}_A^w$, then we need to show

$$(\widehat{V}(\widehat{w}(M))[\overline{M_2}/x], \hat{i}((\Delta; \Gamma, x : A \vdash M : B)^*)[d_2/x]) \in \mathcal{TR}_B^w$$

which follows by induction.

- MN .

By the inductive hypotheses and the monadic relation, there are target values d and e such that

$$\hat{i}((\Delta; \Gamma \vdash M : A \rightarrow B)^*) = [d] \quad \text{and} \quad (\widehat{V}(\widehat{w}(M)), d) \in \mathcal{R}_{A \rightarrow B}^w$$

and

$$\hat{i}((\Delta; \Gamma \vdash N : A)^*) = [e] \quad \text{and} \quad (\widehat{V}(\widehat{w}(N)), e) \in \mathcal{R}_A^w.$$

Thus we know $\widehat{V}(\widehat{w}(M)) \rightsquigarrow^* \lambda x : A.M'$ for some M' such that

$$(M'[\widehat{V}(\widehat{w}(N))/x], de) \in \mathcal{TR}_B^w.$$

Since $\widehat{V}(\widehat{w}(MN)) \rightsquigarrow^* M'[\widehat{V}(\widehat{w}(N))/x]$, Lemma 4 gives

$$(\widehat{V}(\widehat{w}(MN)), de) \in \mathcal{TR}_B^w.$$

And since

$$\begin{aligned} \hat{i}((\Delta; \Gamma \vdash MN : B)^*) &= d' \leftarrow \hat{i}((\Delta; \Gamma \vdash M : A \rightarrow B)^*); e' \leftarrow \hat{i}((\Delta; \Gamma \vdash N : A)^*); d' e' \\ &= d' \leftarrow [d]; e' \leftarrow [e]; d' e' \\ &= de \end{aligned}$$

we are done.

- $\Lambda X.M$.

By induction we know that for any $(B, \mathcal{R}) \in \mathcal{F}$,

$$(\widehat{V}(\widehat{w}(M)[B/X]), \hat{i}((\Delta, X; \Gamma \vdash M : A)^*)) \in \mathcal{TR}_A^{w, X \mapsto (B, \mathcal{R})}$$

and then as $(\Delta; \Gamma \vdash \Lambda X.M : \forall X.A)^* = (\Delta, X; \Gamma \vdash M : A)^*$ we are done by the definition of $\mathcal{R}_{\forall X.A}^w$.

- MB .

By induction, we know

$$(\widehat{V}(\widehat{w}(M)), \hat{i}((\Delta; \Gamma \vdash M : \forall X.A)^*)) \in \mathcal{TR}_{\forall X.A}^w.$$

So there is a d such that

$$\hat{t}((\Delta; \Gamma \vdash M : \forall X.A)^*) = [d] \quad \text{and} \quad (\hat{V}(\hat{w}(M)), d) \in \mathcal{R}_{\forall X.A}^w.$$

Unfolding the logical relation for quantified types and instantiating with $(\hat{w}(B), \hat{\mathcal{R}}_B^w) \in \mathcal{F}$ yields that $\hat{V}(\hat{w}(M)) \rightsquigarrow^* \Lambda X.M'$ for some M' with

$$(M'[\hat{w}(B)/X], d) \in \mathcal{R}_A^{w, X \mapsto (\hat{w}(B), \hat{\mathcal{R}}_B^w)}.$$

By the second part of Lemma 6, the key type substitution property, this implies

$$(M'[\hat{w}(B)/X], J_{X.A}^B(d)) \in \mathcal{TR}_{A[B/X]}^w.$$

Since $\hat{V}(\hat{w}(MB)) = \hat{V}(\hat{w}(M)) \hat{w}(B) \rightsquigarrow^* (\Lambda X.M') \hat{w}(B) \rightsquigarrow M'[(\hat{w}(B)/X)]$, Lemma 4 gives

$$(\hat{V}(\hat{w}(MB)), J_{X.A}^B(d)) \in \mathcal{TR}_{A[B/X]}^w.$$

By definition of the translation,

$$\begin{aligned} \hat{t}((\Delta; \Gamma \vdash MB : A[B/X])^*) &= d' \leftarrow \hat{t}((\Delta; \Gamma \vdash M : \forall X.A)^*); J_{X.A}^B(d') \\ &= d' \leftarrow [d]; J_{X.A}^B(d') \\ &= J_{X.A}^B(d) \end{aligned}$$

So

$$(\hat{V}(\hat{w}(MB)), \hat{t}((\Delta; \Gamma \vdash MB : A[B/X])^*)) \in \mathcal{TR}_{A[B/X]}^w.$$

as required.

□

An immediate consequence of Lemma 7 is that the behaviour of a program (closed term of ground type) and its translation agree:

Corollary 1

If $\cdot; \cdot \vdash M : \text{nat}$ then there exists an n such that $M \rightsquigarrow^* \bar{n}$ and $(\cdot; \cdot M : \text{nat})^* = [n]$. □

Observe that the above also incorporates the fact that the source language is total, in the sense that every program reduces to a canonical value.

6 Discussion

Using logical relations it is possible to prove the correctness of the compilation of polymorphic types to dynamic types in such a way that overhead is imposed only insofar as polymorphism is actually used. This compilation method lies at the heart of the implementation of generic extensions to Java, and of polymorphic languages such as Scala, on the Java Virtual Machine, with the type `Object` playing the role of our `dyn`. As far as we are aware this is the first correctness proof of this compilation strategy, and is novel insofar as it only relies on an embedding into `dyn`, rather than a stronger condition such as isomorphism. In this respect the proof may be useful in other situations where the correctness of a compilation method is required.

Semantically, the underlying idea of interpreting types as retracts of a universal domain is an old one, going back to work of Scott (1976) and McCracken (1979). It has been

adapted and used for various purposes in programming, including by Benton (2005) and Ramsey (2011) for interfacing typed languages with untyped ones, and by many authors studying run-time enforcement of contracts (Findler & Felleisen, 2002) in dynamic languages, and the correct assignment of blame should violations occur (Ahmed *et al.*, 2011).

The broad shape of the proof presented here is that of *adequacy*: showing agreement between an operational and a denotational (translational) semantics via a logical relation (Plotkin, 1977; Amadio, 1993). Similar logical relations have also been used for the closely-related task of establishing the correctness of compilers (Minamide *et al.*, 1996; Benton & Hur, 2010; Hur & Dreyer, 2011).

One possible extension to this work is to consider the extension of System F with general recursion at the expression level, or, more generally, with recursive types. It appears that handling general recursion is straightforward, following directly the strategy outlined in Chapter 48 of the third author's text (Harper, 2012), which requires that admissible relations be closed under limits of suitable chains, and which employs fixed point induction in establishing the main theorem. The extension to product and sum types is entirely straightforward. Recursive types require more sophisticated techniques pioneered by Pitts (Pitts, 1996), and adapted to the operational setting by Crary & Harper (2007). Step-indexed methods, such as those introduced by Dreyer *et al.* (2009) may also be useful in this respect.

Another possible extension is to consider System F with higher-order polymorphism, namely System F_ω , which enables programmers to abstract over even type constructors, such as lists or trees which themselves are polymorphic in their element type. Such higher-order polymorphism has been materialized in dynamic typing, for example in Scala, by Moors *et al.* (2008), and it is conceivable, for studying the correctness, to migrate the method to System F_ω as we did to System F. Together with the work by Rossberg *et al.* (2010) which compiles ML modules to System F_ω , an alternative account for the dynamics of ML modules, in terms of dynamic typing, can possibly be made.

Acknowledgements

This paper originated from a homework exercise in an undergraduate programming language theory class that was developed by the third author with Roger Wolff, to whom we are grateful for many helpful discussions and suggestions. Thanks also to Yitzhak Mandelbaum for his numerous suggestions. Thanks to Carlo Angiuli, Shayak Sen and the PoP group at Carnegie Mellon who gave valuable comments on earlier drafts of this work.

Bibliography

- Ahmed, Amal, Findler, Robert Bruce, Siek, Jeremy G., & Wadler, Philip. (2011). Blame for all. *Pages 201–214 of: Proceedings of the 38th annual acm sigplan-sigact symposium on principles of programming languages*. POPL '11. New York, NY, USA: ACM.
- Amadio, Roberto. (1993). On the adequacy of PER models. *18th international symposium on mathematical foundations of computer science (mfcs)*. LNCS, vol. 711. Springer.
- Bank, Joseph A., Myers, Andrew C., & Liskov, Barbara. (1997). Parameterized types for Java. *Pages 132–145 of: Proceedings of the 24th acm sigplan-sigact symposium on principles of programming languages*. POPL '97. New York, NY, USA: ACM.
- Benton, Nick. (2005). Embedded interpreters. *Journal of functional programming*, **15**(4).
- Benton, Nick, & Hur, Chung-Kil. (2010). *Realizability and compositional compiler correctness for a polymorphic language*. Tech. rept. MSR-TR-2010-62. Microsoft Research.
- Bracha, Gilad, Odersky, Martin, Stoutamire, David, & Wadler, Philip. (1998). Making the future safe for the past: adding genericity to the Java programming language. *Acm sigplan notices*, **33**(10), 183–200.
- Crary, Karl, & Harper, Robert. (2007). Syntactic logical relations for polymorphic and recursive types. *Electronic notes in theoretical computer science*, **172**(0), 259 – 299. Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.
- Curry, H. B., Hindley, J. R., & Seldin, J. P. (1972). *Combinatory logic, volume ii*. North-Holland.
- Dreyer, D., Ahmed, A., & Birkedal, L. (2009). Logical step-indexed logical relations. *Pages 71–80 of: Logic in computer science, 2009. lics '09. 24th annual ieee symposium on*.
- Findler, Robbie, & Felleisen, Matthias. (2002). Contracts for higher-order functions. *Proceedings of the international conference on functional programming (icfp)*.
- Girard, Jean-Yves. (1972). *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. thesis, Université Paris 7.
- Harper, Robert. (2012). *Practical foundations for programming languages*. Cambridge University Press.
- Hur, Chung-Kil, & Dreyer, Derek. (2011). A Kripke logical relation between ML and assembly. *Proceedings of the 38th acm sigplan-sigact symposium on principles of programming languages (popl)*. ACM.
- McCracken, Nancy. 1979 (June). *An investigation of a programming language with a polymorphic type structure*. Ph.D. thesis, Syracuse University.
- Meyer, Albert R., & Wand, Mitchell. (1985). Continuation semantics in typed lambda-calculi. *Pages 219–224 of: Parikh, Rohit (ed), Logics of programs*. Lecture Notes in Computer Science, vol. 193. Springer Berlin Heidelberg.
- Milner, Robin. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, **17**(3), 348–375.
- Minamide, Yasuhiko, Morrisett, Greg, & Harper, Robert. (1996). Typed closure conversion. *Proceedings of the 23rd acm sigplan-sigact symposium on principles of programming languages (popl)*. ACM.
- Moggi, Eugenio. (1991). Notions of computation and monads. *Information and computation*, **93**, 55–92.

- Moors, Adriaan, Piessens, Frank, & Odersky, Martin. (2008). Generics of a higher kind. *Sigplan not.*, **43**(10), 423–438.
- Pitts, Andrew M. (1996). Relational properties of domains. *Information and computation*, **127**(2), 66 – 90.
- Plotkin, Gordon D. (1977). LCF considered as a programming language. *Theoretical computer science*, **5**, 223–255.
- Ramsey, Norman. (2011). Embedding an interpreted language using higher-order functions and types. *Journal of functional programming*, **21**(6).
- Reynolds, John. (1983). Types, abstraction and parametric polymorphism. *Information processing 83*. Elsevier Science Publishers B.V. (North-Holland).
- Reynolds, John C. (1974). Towards a theory of type structure. *Pages 408–423 of: Programming symposium, proceedings colloque sur la programmation*. Springer-Verlag.
- Reynolds, John C. (2003). What do types mean? from intrinsic to extrinsic semantics. *Pages 309–327 of: McIver, Annabelle, & Morgan, Carroll (eds), Programming methodology*. Monographs in Computer Science. Springer New York.
- Rosberg, Andreas, Russo, Claudio V., & Dreyer, Derek. (2010). F-ing modules. *Pages 89–102 of: Proceedings of the 5th acm sigplan workshop on types in language design and implementation*. TLDI '10. New York, NY, USA: ACM.
- Scott, Dana. (1976). Data types as lattices. *SIAM journal of computing*, **4**.