Type Refinements for Compiler Correctness

Abstract

Type refinements, introduced by Freeman and Pfenning and explored by Davies and Dunfield, unify the ontological and epistemic views of typing. Types tell us what programming language constructs exist, whereas refinements express properties of the values of a type. Here we show that refinements are very useful in compiler correctness proofs, wherein it often arises that two expressions that are inequivalent in general are equivalent in the particular context in which they occur. Refinements serve to restrict the contexts sufficiently so that the desired equivalence holds. For example, an expression might be replaced by a more efficient one, even though it is not generally equivalent to the original, but is interchangeable in any context satisfying a specified refinement of the type of those expressions.

We study here in detail a particular problem of compiler correctness, namely the correctness of compiling polymorphism (generics) to dynamic types by treating values of variable type as values of a universal dynamic type. Although this technique is widely used (for example, to compile Java generics), no proof of its correctness in System F has been given to date. Surprisingly, standard arguments based on logical relations do not suffice, precisely because it is necessary to record deeper invariants about the compiled code than is expressible in their types alone. We show that refinements provide an elegant solution to this problem by capturing the required invariants so that a critical invertibility property that is false is general can be proved to hold in the contexts that arise in the translated code. This proof not only establishes the correctness of this compilation method, but also exemplifies the importance of refinements for compiler correctness proofs more generally.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Invariants; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

Keywords Refinement, Logical Relation, Parametricity, Polymorphism, Dynamic Typing

1. Introduction

Traditionally, type systems are divided into two general classifications, the *ontological*, or *prescriptive*, form and the *epistemic*, or *descriptive* form. The ontological form, usually associated with Church, treats types as determining the language itself; programs do not exist except insofar as they are well-typed, and their dynamics arises from Gentzen's inversion principle. The epistemic form, usually associated with Curry, treats types as descriptions of the behavior of untyped (more accurately, uni-typed) terms equipped with a given dynamics. From the point of view of constructive logic, the ontological view corresponds to the syntactic propositions-as-types principle [16] as exemplified by Coq [6] and Agda [21], whereas the epistemic view corresponds to the semantic propositions-as-types principle (also known as realizability) [5] and exemplified by NuPrl.

Type refinements, introduced by Freeman and Pfenning [12] and further explored by Davies [9] and Dunfield [11], unify and generalize these two views of typing by separating the structural concept of a type from the logical concept of a (type) refinement. Types are conceived ontologically, refinements epistemically as predicates over the values of a type. The judgment $\rho \sqsubseteq \sigma$ specifies that the refinement ρ is a predicate describing the behavior of expressions of type σ. The judgment $d \in_\sigma \rho$ means, for an expression d of type σ, that it satisfies the property expressed by ρ. The purely ontological view may be expressed by the judgment $d \in_\sigma \top_\sigma$, where $\top_\sigma$ is the always-true refinement of type σ. The usual epistemic view may be expressed by the judgment $e \in_{\mathtt{dyn}} \rho$, where ρ now plays the role of an (epistemic) type describing the behavior of a term of the universal type, `dyn`.

Thought of as a logic of programs, refinements necessarily conform to the crucial *compositionality*, or *modularity*, principle, which may be stated in the following form:[1]

$$\frac{d_1 \in_{\sigma_1} \rho_1 \qquad x \in_{\sigma_1} \rho_1 \vdash d \in_\sigma \rho}{[d_1/x]d \in_\sigma \rho}$$

The first premise states that d_1 satisfies the invariant expressed by ρ_1, and the second states that d_1 satisfies the invariant expressed by ρ_2, under the assumption that the unknown, x, satisfies the invariant ρ_1. Crucially, the refinement mediates the interaction between the two components d_1 and d_2; the only information propagated to d_2 is that which is expressed by ρ_1. Put in other terms, the refinement ρ_1 limits the "experiments" that d_2 can perform on d_1 to those that are validated by the refinement ρ_1.

The restrictions imposed by refinements and modularity is central to their use in compiler correctness proofs. Briefly, by restricting the contexts in which an expression arises, a refinement allows us to consider two expressions to be equivalent *in that context* even though they are inequivalent *as members of their common type*. This pattern of reasoning arises frequently in compiler correctness proofs, in particular when a typical compiler optimization replaces on piece of code, d_1, by another, d_2, that may not be equivalent in general, but is equivalent at the point of replacement. For example, an optimizing compiler may choose to omit overflow checking when such possibility is ruled out by the context, or to skip computing the absolute value when the argument is known to be positive. More generally, the two pieces of code may be considered equiva-

[1] Conceptually, this is what distinguishes a program logic from a program analysis.

lent in any context satisfying a stated assumption, which we express here by a refinement. The ubiquity of such reasoning is central to the overall enterprise of compiler verification. The purpose of this paper is to show that refinements provide an elegant means of handling a class of such problems that, we anticipate, may be scaled to many similar situations.

To justify our claim we consider a well-known compilation technique for polymorphism introduced by Bracha *et al.* [4] and Bank *et al.* [2] and used in the compilation of Java generics and *Scala* [22]. Rather than treat polymorphism as a primitive notion in its own right, programs are compiled down to a simpler language with a universal type, here called `dyn`, of which all other types are retracts. Values of unknown type are compiled as values of type `dyn`, the universal type; instantiation is compiled by projection from the universal type to the instantiated type.

The question considered here is why is this compilation strategy correct? To our knowledge no proof has been given. Moreover, to our surprise, the argument is more complex than one might at first imagine. We expected that one could give a fairly simple type-based translation to a typed intermediate language (in the style of, for example, [18]), and show that it is correct using a relatively straightforward logical relations argument similar to that given by Meyer and Wand [17] in their treatment of typed CPS conversion. However, the proof requires a much more subtle argument for which refinements provide an elegant solution. Briefly, the proof of correctness requires a proof of equivalence of two expressions that is valid *only at the point where it arises*, but is *not valid in general*. More generally, we are concerned with situations in which an equivalence is valid only under additional assumptions that refine the type of the involved expressions, but that fails without these assumptions. By formulating the required condition as a refinement we constrain the possible contexts in which the expressions may occur, and admit the replacement of one by the other in all such contexts (but not others, in general).

The reasoning required here seems fairly typical of a pattern that occurs often in compiler correctness proofs. By giving a precise example of their use in a moderately complex argument we hope to exemplify what we believe to be a broadly applicable tool for compiler verification. Just as refinements may be used by programmers express and enforce critical invariants in code [9, 11, 12], so refinements may be used by compiler writers to state and verify the correctness of program transformations. Thus, we view the main contribution of this paper to be the use of refinements to facilitate compiler correctness proofs, with a secondary contribution being the first proof of correctness of a widely used compilation strategy given using this method.

2. Overview

To make these ideas more precise we give an overview of the technical development in the remainder of the paper. We consider the compilation of Girard and Reynolds' System F [14, 24] to an extension of Plotkin's PCF [23], called DPCF, with a single recursive type, `dyn`, into which all other types may be embedded. The type `dyn` may be considered a universal type in that it embeds the full uni-typed λ-calculus, and may readily be extended to encompass the constructs of any dynamically typed language [15].[2]

The most obvious compilation of System F to DPCF is by *type erasure*: simply disregard types entirely and treat System F terms as dynamically λ-terms. Clearly this approach incurs excessive overhead, since the static type discipline of System F is completely disregarded, so that even elementary arithmetic involves run-time tagging and untagging [15]. A better approach [2, 4] is to preserve as much static type information as possible, resorting to dynamic typing only insofar as polymorphism is used in a program. In such a case simple numeric codes are compiled to run with no additional overhead; the only cost is associated with the use of polymorphism. More precisely, variable types, t, are replaced by the universal type, `dyn`, during compilation. For example, the polymorphic identity function $\Lambda t.\lambda x{:}t.x$ is compiled as $\lambda x{:}\mathtt{dyn}.x$ of type $\mathtt{dyn} \rightharpoonup \mathtt{dyn}$ (suppressing some irrelevant details for the time being).

So, roughly speaking, the overall plan is to translate a System F type τ involving a type variable t into a DPCF type $[\mathtt{dyn}/t]\tau^*$ for some structure-preserving transformation τ^* in which occurrences of t are replaced by `dyn`. In particular, polymorphic types $\forall t.\tau$ are translated to function types $\mathtt{unit} \rightharpoonup [\mathtt{dyn}/t]\tau^*$, degenerating the type abstraction to abstraction over the `unit` type so as to preserve values. This raises the question of how to relate the instance type, $[\sigma/t]\tau$, to $[\mathtt{dyn}/t]\tau^*$, the result of applying the closure to a trivial argument to activate its body.

The key is to define an embedding, i, of each DPCF type σ into the type `dyn`, simultaneously with a corresponding projection, j, from `dyn` back to σ such that $j \circ i \cong \mathsf{id}$, which is to say that $j(i(x)) \cong x$ for each $x : \sigma$, but one cannot expect that $i \circ j \cong \mathsf{id}$ in general, because the argument may not lie within the image of the embedding. Thus, j recovers an embedded value of type σ, but is otherwise undefined.[3] The embeddings and projections may be lifted to mediate between $[\sigma/t]\tau^*$ and $[\mathtt{dyn}/t]\tau^*$ by the functorial action of τ^* with argument t. The lifted projection is again left inverse to the lifted embedding, but one cannot expect it to be right inverse in general, for the same reasons.

With this in hand, polymorphic instantiation may be compiled by using the lifted projection from $[\mathtt{dyn}/t]\tau^*$ to $[\sigma/t]\tau^*$. In particular the instantiation of the polymorphic identity at the type `nat` is compiled by projecting from $\mathtt{dyn} \rightharpoonup \mathtt{dyn}$ to $\mathtt{nat} \rightharpoonup \mathtt{nat}$ to obtain

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

which has the desired type $\mathtt{nat} \rightharpoonup \mathtt{nat}$. This function is *observationally equivalent* to the identity on `nat`, since the context is restricted to apply it to a natural number, and to use the result as a natural number, which is to say

$$\lambda x{:}\mathtt{nat}.j_{\mathtt{nat}}((\lambda x{:}\mathtt{dyn}.x)\, i_{\mathtt{nat}}(x)) \cong_{\mathtt{nat}\to\mathtt{nat}} \lambda x{:}\mathtt{nat}.x. \quad (1)$$

Formally, the compilation can be written as an extension to the standard System F typing judgment, $\Delta;\Gamma \vdash e : \tau$, where Δ is the type context, Γ is the expression context and e is the source term of type τ, to the following form.

$$\Delta;\Gamma \vdash e : \tau \Rightarrow d,$$

where d is the translation result in DPCF. The translation of polymorphic instantiation, written $e[\tau]$, meaning that a polymorphic term e is instantiated with the type τ, is given as the following critical rule

$$\frac{\Delta \vdash \tau_2\ \mathsf{type} \qquad \Delta;\Gamma \vdash e : \forall t.\tau_1 \Rightarrow d}{\Delta;\Gamma \vdash e[\tau_2] : [\tau_2/t]\tau_1 \Rightarrow J(d\,\langle\rangle)}$$

where the premise $\Delta \vdash \tau_2\ \mathsf{type}$ guarantees the well-formedness of τ_2 and J in the conclusion is some fitting projection determined by τ_1 and τ_2, which will be defined carefully below. (The dummy argument $\langle\rangle$ here is to force the thunk we create for each polymorphic term so that values are preserved by compilation.)

We wish to show that an expression and its compilation are observationally equivalent, which means that they give rise to the same observable behavior in all program contexts. As might be expected, the idea of the proof is to define a simulation relation

[2] The type `dyn` is analogous to the type `Object` in Java, the type of all objects in the language.

[3] In practice j would of course raise an exception when applied to a value outside the image of i.

between the source and target languages that is the identity at observable type. The main tool for establishing such a relation for a higher-order language is the method of logical relations [25], which has been widely used for proving compiler correctness (for example, [10, 17].) Since our strategy involves embeddings and projections into a universal type, our approach is somewhat similar to that of Meyer and Wand [17], who considered the correctness of the CPS translation for the simply typed λ-calculus.

The general idea of such a proof is to set up a simulation relation, $\mathcal{R}$, indexed by source language types, and to arrange that every well-typed expression is related, at its type, to its translation. The main theorem will imply that if $e : \tau \Rightarrow d$ is derivable then the relation $\mathcal{R}(e, d)$ holds. By ensuring that the relation coincides with Kleene equivalence at observable type, we obtain the desired observational correctness property of the compilation.

However, in attempting to follow this strategy, we encountered a seemingly insuperable difficulty arising from the contravariance of the function type. At some critical points we need to show J commutes with application in the sense that $J(e_1^*)\,J(e_2^*) \cong J(e_1^*\,e_2^*)$. The projection associated to a function type involves the embedding of its argument at the domain type and the projection of its result at the codomain type, which means the left-hand side is observationally equivalent to $J(e_1^*\,I(J(e_2^*)))$. Comparing this to the right-hand side $J(e_1^*\,e_2^*)$, one might wish to argue that $I \circ J \cong \mathsf{id}$ to finish the proof, but this identity does not hold in general. For example, $J(\mathsf{id}_{\mathtt{dyn}})$ is observationally equivalent to $\mathsf{id}_{\mathtt{nat}}$, but $I(J(\mathsf{id}_{\mathtt{dyn}}))$ is not observationally equivalent to the $\mathsf{id}_{\mathtt{dyn}}$, as would be required. The reason is that the former term amounts to $\lambda x{:}\mathtt{dyn}.i_{\mathtt{nat}}(j_{\mathtt{nat}}(x))$, which contains class checks that restrict the domain of the function to those values with numeric class, rather than admitting all dynamic values.

The central problem here is that the embedding and projection define a retraction that is not an isomorphism. Nonetheless, the required isomorphism does hold provided that the context satisfies an invariant sufficient to ensure that no run-time class check failures can occur (*i.e.*, so that no projection ever fails to be defined). The required condition can be captured using refinements. Intuitively, we must exploit the static typing of the source language so as to ensure that any use of polymorphic instantiation cannot lead to a run-time error. Typing alone is not sufficient, because the type system of DPCF is not sufficiently strong to express the required invariants (and is in any case viewed ontologically in our work). Rather what we require is a system of refinements, which we call *polymorphic constructor refinements*, or *pcr*'s, which carry invariants arising during compilation. Pcr's are designed to express the following key properties:

1. *Error-free termination* of function evaluation. This is required to ensure that no use of projection in compiled code is undefined, which allows us to replace $i \circ j$ by the identity.
2. The *class* (*constructor*) of a dynamic value. In the present setting the class can be either `nil`, classifying an embedded nullary tuple $\langle\rangle$, `num`, classifying an embedded number, or `fun`, classifying an embedded function, but there is no inherent limit on what classes of data may be supported.
3. *Polymorphic refinements* to state that the same invariant holds in multiple positions in a value. In keeping with the epistemic character of refinements, polymorphic refinements incur no run-time overhead; they are purely a matter of reasoning about the correctness of the compilation.

In particular, we have a pcr $\mathtt{num}(\mathbb{N})$ of type `dyn`, specifying successful termination with a dynamic value of the numeral class `num`, and $\rho_1 \to \rho_2$ of the function type, asserting totality of the function whose outcome always satisfies the pcr ρ_2 whenever the argument satisfies the pcr ρ_1.

Notably, no notion of "subtyping" or "entailment" for refinements is necessary (or sufficient) for the present problem, nor is there any notion of intersection (meet) or truth (top), as in other systems of refinements [9, 12].

To see how this system works, let us reconsider the functions $\mathsf{id}_{\mathtt{dyn}}$ and $I(\mathsf{id}_{\mathtt{nat}})$ in the example given above: they cannot be distinguished in a context that provides an argument of class `num` and expects a result of class `num`. This requirement is expressed by the refinement $\mathtt{num}(\mathbb{N}) \to \mathtt{num}(\mathbb{N})$, and we may say that $\mathsf{id}_{\mathtt{dyn}}$ and $I(\mathsf{id}_{\mathtt{nat}})$ are equivalent *with respect to this refinement*, written

$$I(\mathsf{id}_{\mathtt{nat}}) \mathrel{\dot{\cong}}_{\mathtt{num}(\mathbb{N})\to\mathtt{num}(\mathbb{N})} \mathsf{id}_{\mathtt{dyn}},$$

or, after expanding the definition of I and id,

$$\lambda x{:}\mathtt{dyn}.i_{\mathtt{nat}}((\lambda x{:}\mathtt{nat}.x)\,j_{\mathtt{nat}}(x)) \mathrel{\dot{\cong}}_{\mathtt{num}(\mathbb{N})\to\mathtt{num}(\mathbb{N})} \lambda x{:}\mathtt{dyn}.x$$

that closely mimics Equation 1. Similar reasoning lies at the heart of the correctness proof given below for the compilation of polymorphism to dynamic typing. In particular we will use refinements to ensure that, in situations arising in compiled code, the projection, j, is right-inverse to its corresponding embedding, i, *up to the restrictions imposed by a suitable refinement*. The refinement allows us to consider a coarser equivalence than contextual equivalence, precisely because it limits the contexts in which the replacement may be made. Crucially, the compilation algorithm ensures that the limitation is always met, and we may then push the correctness proof outlined above through to completion.

3. Translation of Polymorphism

3.1 The Languages and Notation

We use the following syntax for System F

$$\begin{array}{llll} \mathsf{Typ} & \tau & ::= & t \mid \mathtt{nat} \mid \tau_1 \to \tau_2 \mid \forall t.\tau \\ \mathsf{Exp} & e & ::= & x \mid \mathtt{z} \mid \mathtt{suc}(e) \mid \mathtt{ifz}(e; e_0; x.e_1) \mid \\ & & & \lambda x{:}\tau.e \mid e_1\,e_2 \mid \Lambda t.e \mid e[\tau] \end{array}$$

and the following for DPCF.

$$\begin{array}{llll} \mathsf{Typ} & \sigma & ::= & \mathtt{unit} \mid \mathtt{nat} \mid \mathtt{dyn} \mid \sigma_1 \rightharpoonup \sigma_2 \\ \mathsf{Exp} & d & ::= & x \mid \langle\rangle \mid \mathtt{z} \mid \mathtt{suc}(d) \mid \mathtt{ifz}(d; d_0; x.d_1) \mid \\ & & & \lambda x{:}\sigma.d \mid d_1\,d_2 \mid \mathtt{fix}[\sigma](x.d) \mid c\,!\,d \mid d\,?\,c \\ \mathsf{Cls} & c & ::= & \mathtt{nil} \mid \mathtt{num} \mid \mathtt{fun} \end{array}$$

The expression "$d\ ?\ c$" casts d of type `dyn` to another type indicated by the class c, and "$c!d$" tags d with some class c, forming a new expression of type `dyn`. Figure 1 lists all casting and tagging rules in DPCF. The important invariant is that casting is always left inverse to tagging, which says if $(c\,!\,d)\ ?\ c$ is well-typed, it should be observationally equivalent to d.

We choose `nat` to be the type of observation, deciding that two expressions are observationally equivalent if they produce the same number under any program context of `nat`. Observational equivalence is written as $\Gamma \vdash d_1 \cong d_2 : \sigma$, or simply $d_1 \cong d_2$ if Γ and σ are clear from the context.[4] As usual, we adopt call-by-name for DPCF to simplify the reasoning, where $c\,!\,d$ is always regarded as a value.

Substitutions are mappings from variables to expressions, which are made concrete as lists of matching pairs. For example, $\xi = (x \hookrightarrow \mathtt{z}) \otimes (y \hookrightarrow \langle\rangle)$ is a substitution which sends x to $\mathtt{z}$ and y to

[4] See Chapters 48 and 49 of [15] for the definition and characterization of observational equivalence for PCF and F. The extension to DPCF is given by Crary and Harper [7] in the more general setting of recursive types. The main property we rely on here is that observational equivalence is the coarsest consistent congruence containing all computation steps. Section 4 also develops the formal definition in a similar way.

$$\frac{\Gamma \vdash d : \mathtt{unit}}{\Gamma \vdash \mathtt{nil}\,!\,d : \mathtt{dyn}} \qquad \frac{\Gamma \vdash d : \mathtt{dyn}}{\Gamma \vdash d\,?\,\mathtt{nil} : \mathtt{unit}}$$

$$\frac{\Gamma \vdash d : \mathtt{nat}}{\Gamma \vdash \mathtt{num}\,!\,d : \mathtt{dyn}} \qquad \frac{\Gamma \vdash d : \mathtt{dyn}}{\Gamma \vdash d\,?\,\mathtt{num} : \mathtt{nat}}$$

$$\frac{\Gamma \vdash d : \mathtt{dyn} \rightharpoonup \mathtt{dyn}}{\Gamma \vdash \mathtt{fun}\,!\,d : \mathtt{dyn}} \qquad \frac{\Gamma \vdash d : \mathtt{dyn}}{\Gamma \vdash d\,?\,\mathtt{fun} : \mathtt{dyn} \rightharpoonup \mathtt{dyn}}$$

Figure 1. Rules for primitive embedding and projection.

$\langle\rangle$; the tensor product ($\otimes$) here extends an existing mapping with a new pair. $\hat{\xi}(\cdot)$ is the induced substitution function $[\mathtt{z}, \langle\rangle/x, y](\cdot)$ that works for all expressions, not only variables. Type substitutions are defined in a similar way, and the symbol $\emptyset$ is reserved for empty mappings.

One remark is that the construct `fix` is never used in the translation. However, the introduction of `dyn` already makes possible embedding the whole untyped λ-calculus, including the fixed-point combinator.

For clarity, the meta variables e and τ will be dedicated to System F and σ, d and c to DPCF for the rest of the paper.

3.2 Embedding and Projection Pairs

While there are only three primitive embedding and projection pairs for `unit`, `nat`, and $\mathtt{dyn} \rightharpoonup \mathtt{dyn}$, we can implement embedding and projection pairs, written i_σ and j_σ, for all types σ's in DPCF. Our criterion of good embedding and projection pairs is

- $i_\sigma : \sigma \rightharpoonup \mathtt{dyn}$.
- $j_\sigma : \mathtt{dyn} \rightharpoonup \sigma$.
- $j_\sigma \circ i_\sigma(d) \cong \mathsf{id} : \sigma \rightharpoonup \sigma$, which is to say that j_σ is a left inverse of i_σ.

which can be fulfilled with the following inductive definition (written in applied forms for clarity):

$$\begin{array}{lcl}
i_{\mathtt{unit}}(x) & = & \mathtt{nil}\,!\,x \\
j_{\mathtt{unit}}(x) & = & x\,?\,\mathtt{nil} \\
i_{\mathtt{nat}}(x) & = & \mathtt{num}\,!\,x \\
j_{\mathtt{nat}}(x) & = & x\,?\,\mathtt{num} \\
i_{\mathtt{dyn}}(x) & = & x \\
j_{\mathtt{dyn}}(x) & = & x \\
i_{\sigma_1 \rightharpoonup \sigma_2}(f) & = & \mathtt{fun}\,!\,\lambda x{:}\mathtt{dyn}.i_{\sigma_2}(f(j_{\sigma_1}(x))) \\
j_{\sigma_1 \rightharpoonup \sigma_2}(f) & = & \lambda x{:}\sigma_1.j_{\sigma_2}((f\,?\,\mathtt{fun})(i_{\sigma_1}(x)))
\end{array}$$

To specialize the generic identity function of type $\mathtt{dyn} \rightharpoonup \mathtt{dyn}$ to an identity function for `nat` of type $\mathtt{nat} \rightharpoonup \mathtt{nat}$, we need to consider lifting $i_{\mathtt{nat}}$ and $j_{\mathtt{nat}}$ for arbitrary type abstractions with one variable, namely *type operators*, functorially. The relevent type operator in this example is $\square \rightharpoonup \square$ where $\square$ is the distinguished variable marking the changing parts at which embedding or projecting is happening. We may summarize the syntax as follows.[5]

$$\mathsf{TypOp} \quad \omega \quad ::= \quad \square \mid \mathtt{unit} \mid \mathtt{nat} \mid \mathtt{dyn} \mid \omega_1 \rightharpoonup \omega_2$$

We write lifted functions as I^σ_ω and J^σ_ω where ω is the type operator and σ indicates the pair i_σ and j_σ upon which the lifting is based. Continuing the discussion of identity functions, $J^{\mathtt{nat}}_{\square \rightharpoonup \square}$ is the desired lifted projection that converts an expression of type $\mathtt{dyn} \rightharpoonup \mathtt{dyn}$ to $\mathtt{nat} \rightharpoonup \mathtt{nat}$. It turns out that we can build lifted functions I^σ_ω and J^σ_ω for all type operators from basic building blocks i_σ and j_σ which satisfy these important conditions:

- $I^\sigma_\omega : [\sigma/\square]\omega \rightharpoonup [\mathtt{dyn}/\square]\omega$.
- $J^\sigma_\omega : [\mathtt{dyn}/\square]\omega \rightharpoonup [\sigma/\square]\omega$.
- $J^\sigma_\omega \circ I^\sigma_\omega \cong \mathsf{id} : [\sigma/\square]\omega \rightharpoonup [\sigma/\square]\omega$. That is, J^σ_ω is a left inverse of I^σ_ω.

The following is our implementation:

$$\begin{array}{lcl}
I^\sigma_\square(x) & = & i_\sigma(x) \\
J^\sigma_\square(x) & = & j_\sigma(x) \\
I^\sigma_{\mathtt{unit}}(x) & = & x \\
J^\sigma_{\mathtt{unit}}(x) & = & x \\
I^\sigma_{\mathtt{nat}}(x) & = & x \\
J^\sigma_{\mathtt{nat}}(x) & = & x \\
I^\sigma_{\mathtt{dyn}}(x) & = & x \\
J^\sigma_{\mathtt{dyn}}(x) & = & x \\
I^\sigma_{\omega_1 \rightharpoonup \omega_2}(f) & = & \lambda(x{:}[\mathtt{dyn}/\square]\omega_1).I^\sigma_{\omega_2}(f(J^\sigma_{\omega_1}(x))) \\
J^\sigma_{\omega_1 \rightharpoonup \omega_2}(f) & = & \lambda(x{:}[\sigma/\square]\omega_1).J^\sigma_{\omega_2}(f(I^\sigma_{\omega_1}(x)))
\end{array}$$

Now we are ready to present the formal translation rules.

3.3 Translation

We have defined two meta operations and one judgment form for the translation, all shown in Figure 2. The first meta operation is the syntax-directed type translation, written $\tau^\dagger = \sigma$. The second one, written $[\tau]_t = \omega$, is similar to the first one except that one particular type variable t is turned into a marker ($\square$) instead of `dyn`. This generates the type operators in lifted embedding and projection pairs. These two meta operations should interact compositionally as follows:

- For any τ and t, $[\mathtt{dyn}/\square]([\tau]_t) = \tau^\dagger$.
- For any τ and τ_1 and t, $[\tau_1^\dagger/\square]([\tau]_t) = ([\tau_1/t]\tau)^\dagger$.

For example,

$$\begin{array}{rl}
[\mathtt{nat}/\square]([t \to t]_t) = [\mathtt{nat}/\square](\square \rightharpoonup \square) & = \mathtt{nat} \rightharpoonup \mathtt{nat} \\
& = (\mathtt{nat} \to \mathtt{nat})^\dagger = ([\mathtt{nat}/t](t \to t))^\dagger
\end{array}$$

The judgment form extends the standard typing judgment form in System F, written $\Delta; \Gamma \vdash e : \tau \Rightarrow d$, meaning that the expression e of type τ in System F is translated into a DPCF expression d. Notice that the translation of $\Lambda t.e$ is always wrapped as a thunk, which guarantees that all values may be translated to values and makes the translation operationally faithful.

As a sanity check, we have the following theorem for statics.

Theorem 1 (statics). *If $\Delta; \Gamma \vdash e : \tau \Rightarrow d$, then $\Gamma^\dagger \vdash d : \tau^\dagger$ where $\Gamma^\dagger$ is defined as applying the type translation to each type in the context.*

Proof. Induction on the derivation of $\Delta; \Gamma \vdash e : \tau \Rightarrow d$. □

4. Equational Reasoning with Refinements

4.1 A Refinement System

In this section we will design the pcr system, on top of the existing DPCF, that facilitates the correctness proof which aims at showing that the image of the compilation will always terminate without run-time errors due to mismatched classes. Intuitively, knowing the underlying class of an expressions of type `dyn` should be sufficient for safety and thus pcr's should be capable of recording such information. To state that the same invariant holds in multiple positions

[5] Instead of variables and substitutions, an alternative formulation could use contexts and replacements. We think substitution is more accurate because substitution avoids variable name collision but replacement does not. That said, the $\square$ is the only variable here and the distinction might not be clear.

$\boxed{\tau^\dagger = \sigma}$

$$\begin{array}{lcl} t^\dagger & = & \mathtt{dyn} \\ \mathtt{nat}^\dagger & = & \mathtt{nat} \\ (\tau_1 \to \tau_2)^\dagger & = & \tau_1^\dagger \rightharpoonup \tau_2^\dagger \\ (\forall t.\tau)^\dagger & = & \mathtt{unit} \rightharpoonup \tau^\dagger \end{array}$$

$\boxed{[\tau]_t = \omega}$

$$\begin{array}{lcll} [t]_t & = & \Box & \\ [u]_t & = & \mathtt{dyn} & \text{for } u \neq t \\ [\mathtt{nat}]_t & = & \mathtt{nat} & \\ [\tau_1 \to \tau_2]_t & = & [\tau_1]_t \rightharpoonup [\tau_2]_t & \\ [\forall u.\tau_1]_t & = & \mathtt{unit} \rightharpoonup [\tau_1]_t & \text{(assuming } u \neq t) \end{array}$$

$\boxed{\Delta; \Gamma \vdash e : \tau \Rightarrow d}$

$$\frac{}{\Delta; \Gamma, x{:}\tau \vdash x : \tau \Rightarrow x}\ (\mathtt{var}) \qquad \frac{}{\Delta; \Gamma \vdash \mathtt{z} : \mathtt{nat} \Rightarrow \mathtt{z}}\ (\mathtt{z})$$

$$\frac{\Delta; \Gamma \vdash e : \mathtt{nat} \Rightarrow d}{\Delta; \Gamma \vdash \mathtt{suc}(e) : \mathtt{nat} \Rightarrow \mathtt{suc}(d)}\ (\mathtt{suc})$$

$$\frac{\Delta; \Gamma \vdash e : \mathtt{nat} \Rightarrow d \qquad \Delta; \Gamma \vdash e_0 : \tau \Rightarrow d_0 \qquad \Delta; \Gamma, x{:}\mathtt{nat} \vdash e_1 : \tau \Rightarrow d_1}{\Delta; \Gamma \vdash \mathtt{ifz}(e; e_0; x.e_1) : \tau \Rightarrow \mathtt{ifz}(d; d_0; x.d_1)}\ (\mathtt{ifz})$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \qquad \Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2 \Rightarrow d}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2 \Rightarrow \lambda x{:}\tau_1^\dagger.d}\ (\mathtt{lam})$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \Rightarrow d_1 \qquad \Delta; \Gamma \vdash e_2 : \tau_1 \Rightarrow d_2}{\Delta; \Gamma \vdash e_1\, e_2 : \tau_2 \Rightarrow d_1\, d_2}\ (\mathtt{ap})$$

$$\frac{\Delta, t\ \mathsf{type}; \Gamma \vdash e : \tau \Rightarrow d}{\Delta; \Gamma \vdash \Lambda t.e : \forall t.\tau \Rightarrow \lambda\_{:}\mathtt{unit}.d}\ (\mathtt{Lam})$$

$$\frac{\Delta \vdash \tau_2\ \mathsf{type} \qquad \Delta; \Gamma \vdash e : \forall t.\tau_1 \Rightarrow d}{\Delta; \Gamma \vdash e[\tau_2] : [\tau_2/t]\tau_1 \Rightarrow J^{\tau_2^\dagger}_{[\tau_1]_t}(d\,\langle\rangle)}\ (\mathtt{Ap})$$

Figure 2. Rules for translation.

in a value, pcr's should also mirror the polymorphic typing structures in System F. These considerations gives rise to the refinement syntax

$$\mathsf{Ref} \quad \rho \quad ::= \quad t \mid \mathbb{1} \mid \mathbb{N} \mid \rho_1 \to \rho_2 \mid c(\rho) \mid \forall t.\rho.$$

The trivial refinements $\mathbb{1}$ and $\mathbb{N}$ are always true of values of their type. The constructor refinements $c(\rho)$ govern the type $\mathtt{dyn}$, and specify that the class of a dynamic value is c and that its associated value satisfies ρ. Polymorphic refinements $\forall t.\rho$ are used to express correlations within a value of a type. Each refinement guarantees successful termination of expressions satisfying it. For example, if d satisfies $c(\rho)$, then d terminates with a value of class c; it cannot be undefined. Similarly, the refinement $\rho_1 \to \rho_2$ ensures the successful termination of any function that satisfies it, provided that it is applied to an argument satisfying ρ_1, because the result will satisfy ρ_2.

One might notice that all refinement variables in our formulation must refine $\mathtt{dyn}$. The reason is that it is undesirable to make function refinements parametric, that is, free from induction principles. Whenever we are decomposing a function type $\sigma_1 \rightharpoonup \sigma_2$ into

$\boxed{\Theta \vdash \rho \sqsubseteq \sigma}$

$$\frac{}{\Theta, t{\sqsubseteq}\mathtt{dyn} \vdash t \sqsubseteq \mathtt{dyn}} \qquad \frac{}{\Theta \vdash \mathbb{1} \sqsubseteq \mathtt{unit}} \qquad \frac{}{\Theta \vdash \mathbb{N} \sqsubseteq \mathtt{nat}}$$

$$\frac{\Theta \vdash \rho_1 \sqsubseteq \sigma_1 \qquad \Theta \vdash \rho_2 \sqsubseteq \sigma_2}{\Theta \vdash \rho_1 \to \rho_2 \sqsubseteq \sigma_1 \rightharpoonup \sigma_2} \qquad \frac{\Theta \vdash \rho \sqsubseteq \mathtt{unit}}{\Theta \vdash \mathtt{nil}(\rho) \sqsubseteq \mathtt{dyn}}$$

$$\frac{\Theta \vdash \rho \sqsubseteq \mathtt{nat}}{\Theta \vdash \mathtt{num}(\rho) \sqsubseteq \mathtt{dyn}} \qquad \frac{\Theta \vdash \rho \sqsubseteq \mathtt{dyn} \rightharpoonup \mathtt{dyn}}{\Theta \vdash \mathtt{fun}(\rho) \sqsubseteq \mathtt{dyn}}$$

$$\frac{\Theta, t{\sqsubseteq}\mathtt{dyn} \vdash \rho \sqsubseteq \sigma}{\Theta \vdash \forall t.\rho \sqsubseteq \sigma}$$

Figure 3. Rules for the formation of pcr's.

its domain σ_1 and codomain σ_2, we wish to decompose the corresponding refinement ρ in the same manner, under the assumption that it must be of the form $\rho_1 \to \rho_2$; such assumption, however, would be false if the refinement could be a variable. For example, $t{\sqsubseteq}(\mathtt{nat} \to \mathtt{unit}) \vdash t \sqsubseteq (\mathtt{nat} \to \mathtt{unit})$ would be provable but t, unlike $\rho_1 \to \rho_2$, can never be decomposed into two parts that match the domain type $\mathtt{nat}$ and the codomain type $\mathtt{unit}$ respectively, which is often required in our correctness proof. In principle, refinements should only contain more structural information than the types they refine. It seems that we could allow variables refining base types other than $\mathtt{dyn}$, but for simplicity we did not pursue this direction.

Our pcr system that assigns refinements to expressions is given by the rules in Figure 4. One interesting bit is that the final two rules are exactly the polymorphic type assignments which have been considered by Curry, *et al.* [8]. The compatibility of the pcr system with the underlying type system is justified by the following lemma which asserts that well-refined expressions are also well-typed.

Lemma 1. *If* $\Theta; \Gamma \vdash d \in_\sigma \rho$ *then* $\Gamma \vdash d : \sigma$.

4.2 Equational Reasoning

This section is meant to establish standard tools for equational reasoning with the presence of both types and refinements. It is easy to derive the concept of observational equivalence from a syntactical system and a chosen observable refinement $\mathbb{N} \sqsubseteq \mathtt{nat}$, by considering all suitable expression contexts where the outcome may be observed. We write out the refinement of a context $\mathcal{C}$ as

$$\mathcal{C} \in (\Theta; \Gamma \triangleright_\sigma \rho) \rightsquigarrow (\Theta'; \Gamma' \triangleright_{\sigma'} \rho')$$

to mean that if $\Theta; \Gamma \vdash d \in_\sigma \rho$ then $\Theta'; \Gamma' \vdash \mathcal{C}\{d\} \in_{\sigma'} \rho'$. The observational equivalence is then defined by choosing contexts that make expressions complete programs:

Definition 1 (Kleene equivalence)**.** Two closed expressions of type $\mathtt{nat}$ are *Kleene equivalent* if both reduce to the same numeral $\overline{n}$.

Definition 2 (refined observational equivalence)**.** Suppose $\Theta; \Gamma \vdash d_1 \in_\sigma \rho$ and $\Theta; \Gamma \vdash d_2 \in_\sigma \rho$. We say d_1 and d_2 are *observational equivalent*, written $\Theta; \Gamma \vdash d_1 \cong d_2 \in_\sigma \rho$, iff $\mathcal{C}\{d_1\}$ and $\mathcal{C}\{d_2\}$ are Kleene equivalent under any context $\mathcal{C} \in (\Theta; \Gamma \triangleright_\sigma \rho) \rightsquigarrow (\cdot; \cdot \triangleright_{\mathtt{nat}} \mathbb{N})$.

Since refinements are stricter than the types alone, they further limit the possible contexts for consideration and make the equivalence coarser. The following lemma says that, for well-refined terms, this refined equivalence is indeed coarser than the standard observational equivalence.

$\boxed{\Theta;\Gamma \vdash d \in_\sigma \rho}$

$$\frac{}{\Theta;\Gamma, x{\in_\sigma}\rho \vdash x \in_\sigma \rho} \qquad \frac{}{\Theta;\Gamma \vdash \langle\rangle \in_{\mathtt{unit}} \mathbb{1}}$$

$$\frac{}{\Theta;\Gamma \vdash \mathtt{z} \in_{\mathtt{nat}} \mathbb{N}} \qquad \frac{\Theta;\Gamma \vdash d \in_{\mathtt{nat}} \mathbb{N}}{\Theta;\Gamma \vdash \mathtt{suc}(d) \in_{\mathtt{nat}} \mathbb{N}}$$

$$\frac{\Theta;\Gamma \vdash d \in_{\mathtt{nat}} \mathbb{N} \quad \Theta;\Gamma \vdash d_0 \in_\sigma \rho \qquad \Theta;\Gamma, x{\in_{\mathtt{nat}}}\mathbb{N} \vdash d_1 \in_\sigma \rho}{\Theta;\Gamma \vdash \mathtt{ifz}(d; d_0; x.d_1) \in_\sigma \rho}$$

$$\frac{\Theta;\Gamma, x{\in_{\sigma_1}}\rho_1 \vdash d \in_{\sigma_2} \rho_2}{\Theta;\Gamma \vdash \lambda x{:}\sigma_1.d \in_{\sigma_1\to\sigma_2} \rho_1 \to \rho_2}$$

$$\frac{\Theta;\Gamma \vdash d_1 \in_{\sigma_1\to\sigma_2} \rho_1 \to \rho_2 \qquad \Theta;\Gamma \vdash d_2 \in_{\sigma_1} \rho_1}{\Theta;\Gamma \vdash d_1\, d_2 \in_{\sigma_2} \rho_2}$$

$$\frac{\Theta;\Gamma \vdash d \in_{\mathtt{unit}} \rho}{\Theta;\Gamma \vdash \mathtt{nil}\,!\,d \in_{\mathtt{dyn}} \mathtt{nil}(\rho)} \qquad \frac{\Theta;\Gamma \vdash d \in_{\mathtt{dyn}} \mathtt{nil}(\rho)}{\Theta;\Gamma \vdash d\,?\,\mathtt{nil} \in_{\mathtt{unit}} \rho}$$

$$\frac{\Theta;\Gamma \vdash d \in_{\mathtt{nat}} \rho}{\Theta;\Gamma \vdash \mathtt{num}\,!\,d \in_{\mathtt{dyn}} \mathtt{num}(\rho)} \qquad \frac{\Theta;\Gamma \vdash d \in_{\mathtt{dyn}} \mathtt{num}(\rho)}{\Theta;\Gamma \vdash d\,?\,\mathtt{num} \in_{\mathtt{nat}} \rho}$$

$$\frac{\Theta;\Gamma \vdash d \in_{\mathtt{dyn}\to\mathtt{dyn}} \rho}{\Theta;\Gamma \vdash \mathtt{fun}\,!\,d \in_{\mathtt{dyn}} \mathtt{fun}(\rho)} \qquad \frac{\Theta;\Gamma \vdash d \in_{\mathtt{dyn}} \mathtt{fun}(\rho)}{\Theta;\Gamma \vdash d\,?\,\mathtt{fun} \in_{\mathtt{dyn}\to\mathtt{dyn}} \rho}$$

$$\frac{\Theta, t{\sqsubseteq}\mathtt{dyn};\Gamma \vdash d \in_\sigma \rho}{\Theta;\Gamma \vdash d \in_\sigma \forall t.\rho} \qquad \frac{\Theta \vdash \rho_1 \sqsubseteq \mathtt{dyn} \qquad \Theta;\Gamma \vdash d \in_\sigma \forall t.\rho}{\Theta;\Gamma \vdash d \in_\sigma [\rho_1/t]\rho}$$

Figure 4. Rules for the pcr system.

Lemma 2. *Suppose* $\Theta;\Gamma \vdash d_1 \in_\sigma \rho$ *and* $\Theta;\Gamma \vdash d_2 \in_\sigma \rho$. $\Gamma \vdash d_1 \cong d_2 : \sigma$ *implies* $\Theta;\Gamma \vdash d_1 \mathrel{\dot\cong} d_2 \in_\sigma \rho$.

Every observational equivalence, including the standard one and the refined one here, has the nice property that it is the coarsest consistent congruent equivalence whose consistency is defined as being finer than Kleene equivalence at the observable type, for example `nat` here.[6] However, it could be difficult to establish such relation since it considers all suitable contexts. The standard method to simplify the reasoning is a logical equivalence that exploits typing and refinement structures. Refinement variables borrowed from System F, however, bring in an extra complexity of impredicativity and force us to consider all logical relation *candidates*, not only the relation we are interested in. Here the criterion of being a candidate is to respect observational equivalence and converse evaluation, and is called *admissibility*:

Definition 3 (admissible refined relation)**.** A relation $\mathcal{R}$ between two expressions of refinements $\rho_1, \rho_2 \sqsubseteq \mathtt{dyn}$ (written $\mathcal{R} \in \rho_1 \leftrightarrows \rho_2$) is *admissible* if it is closed under refined observational equivalence and converse evaluation.

We write the logical equivalence as $d_1 \mathrel{\dot\sim}_\rho d_2\ [\eta \in \theta_1 \leftrightarrows \theta_2]$ where d_1 is of type $\hat\theta_1(\rho)$, d_2 is of type $\hat\theta_2(\rho)$, and $\eta(t)$ is some admissible relation candidate between $\delta_1(t)$ and $\delta_2(t)$ (formally $\eta(t) \in \delta(t) \leftrightarrows \rho(t)$) for each t.

[6] Note that the meaning of congruence is germane to the system in consideration, and so the refined observational equivalence is the coarsest with respect to the pcr system, while the standard one the underlying type system.

Definition 4 (refined logical equivalence)**.** Suppose $\Theta;\cdot \vdash d_1 \in_\sigma \rho$ and $\Theta;\cdot \vdash d_2 \in_\sigma \rho$ and $\theta_1, \theta_2 : \Theta$. The relation $d_1 \mathrel{\dot\sim}_\rho d_2\ [\eta \in \theta_1 \leftrightarrows \theta_2]$ is defined inductively on the structure of ρ as follows:

- $d_1 \mathrel{\dot\sim}_t d_2\ [\eta \in \theta_1 \leftrightarrows \theta_2]$ iff $\eta(t)(d_1, d_2)$.
- $d_1 \mathrel{\dot\sim}_{\mathbb{1}} d_2\ [\eta \in \theta_1 \leftrightarrows \theta_2]$ always holds.
- $d_1 \mathrel{\dot\sim}_{\mathbb{N}} d_2\ [\eta \in \theta_1 \leftrightarrows \theta_2]$ iff Kleene equivalent.
- $d_1 \mathrel{\dot\sim}_{\rho_1\to\rho_2} d_2\ [\eta \in \theta_1 \leftrightarrows \theta_2]$ iff
  for any d_1' and d_2' such that $d_1' \mathrel{\dot\sim}_{\rho_1} d_2'\ [\eta \in \theta_1 \leftrightarrows \theta_2]$,
  $d_1\, d_1' \mathrel{\dot\sim}_{\rho_2} d_2\, d_2'\ [\eta \in \theta_1 \leftrightarrows \theta_2]$.
- $d_1 \mathrel{\dot\sim}_{c(\rho)} d_2\ [\eta \in \theta_1 \leftrightarrows \theta_2]$ iff
  $(d_1\ ?\ c) \mathrel{\dot\sim}_\rho (d_2\ ?\ c)\ [\eta \in \theta_1 \leftrightarrows \theta_2]$.
- $d_1 \mathrel{\dot\sim}_{\forall t.\rho} d_2\ [\eta \in \theta_1 \leftrightarrows \theta_2]$ iff
  for every $\rho_1, \rho_2 \sqsubseteq \mathtt{dyn}$ and $\mathcal{R} \in \rho_1 \leftrightarrows \rho_2$,
  $d_1 \mathrel{\dot\sim}_\rho d_2\ [\eta \otimes (t \hookrightarrow \mathcal{R}) \in \theta_1 \otimes (t \hookrightarrow \rho_1) \leftrightarrows \theta_2 \otimes (t \hookrightarrow \rho_2)]$.

Definition 5 (refined open logical equivalence)**.** We write $\Theta;\Gamma \vdash d_1 \mathrel{\dot\sim} d_2 \in_\sigma \rho$ iff $\hat\gamma_1(d_1) \mathrel{\dot\sim}_\rho \hat\gamma_2(d_2)\ [\eta \mathrel{\dot\sim} \delta_1 \leftrightarrows \delta_2]$ whenever $\gamma_1 \mathrel{\dot\sim}_\Gamma \gamma_2\ [\eta \in \delta_1 \leftrightarrows \delta_2]$.

What is important is that logical and observational equivalences coincide, which is to say that both are the coarsest consistent congruence.

Lemma 3. $\Theta;\Gamma \vdash d_1 \mathrel{\dot\sim} d_2 \in_\sigma \rho$ *iff* $\Theta;\Gamma \vdash d_1 \mathrel{\dot\cong} d_2 \in_\sigma \rho$.

5. Refined Analysis on Translation

In light of pcr's, we are able to carry out the argument about correctness easily. In particular, pcr's capture the sufficient condition that makes I and J inverse to each other. The overall plan here is to show that the image of the translation is well-refined, that all (lifted and unlifted) embeddings are invertible under certain contexts, and finally a simulation relation between the expressions and their translations.

5.1 Statics

First, we show that our construction of embedding and projection gives well-refined expressions i_σ and j_σ that can handle *any* refinement ρ of σ. We write ρ^* as the refinement of the result of the embedding, called *embedded refinement*, which must be a refinement of `dyn`.

Definition 6 (refinement embedding)**.**

$$\begin{array}{lcl} t^* & = & t \\ \mathbb{1}^* & = & \mathtt{nil}(\mathbb{1}) \\ \mathbb{N}^* & = & \mathtt{num}(\mathbb{N}) \\ (\rho_1 \to \rho_2)^* & = & \mathtt{fun}(\rho_1^* \to \rho_2^*) \\ c(\rho)^* & = & c(\rho) \\ (\forall t.\rho)^* & = & \forall t.\rho^* \end{array}$$

Lemma 4. *If* $\Theta \vdash \rho \sqsubseteq \sigma$ *then* $\Theta \vdash \rho^* \sqsubseteq \mathtt{dyn}$.

Lemma 5. *Suppose* $\Theta \vdash \rho \sqsubseteq \sigma$.

- $\Theta;\cdot \vdash i_\sigma \in_{\sigma\to\mathtt{dyn}} \rho \to \rho^*$.
- $\Theta;\cdot \vdash j_\sigma \in_{\mathtt{dyn}\to\sigma} \rho^* \to \rho$.

Lemma 5 justifies the notations i_ρ and j_ρ since σ is uniquely determined by ρ. It also makes clear the intention of embedding and projection from the viewpoint of refinements.

We would like to show that similar properties hold for lifted embedding and projection as well, which requires the introduction of the concept *refinements for type operators*, or *refinement operators* for short, as the counterpart of type operators. The intuition is that, in the framework with refinements, the distinguished variable $\square$ actually ranges over a type and an associated refinement. Here we reuse the symbol $\square$ in refinement operators to account for that

refinement.

$$\mathsf{RefOp} \quad \varepsilon \quad ::= \quad \Box \mid t \mid \mathbb{1} \mid \mathbb{N} \mid \varepsilon_1 \to \varepsilon_2 \mid c(\rho) \mid \forall t.\varepsilon$$

The formation judgment can be extended to support refinement operators. We say $\varepsilon \sqsubseteq \omega$ if $\rho \sqsubseteq \sigma$ implies $[\rho/\Box]\varepsilon \sqsubseteq [\rho/\Box]\omega$, which can also be symbolized as the rule

$$\frac{}{\Theta \vdash \Box \sqsubseteq \Box}$$

In any case, the lifted embedding and projection pairs work for any refinement operator ε and refinement ρ, which in turn justifies the notations I^ρ_ε and J^ρ_ε.

Lemma 6. *Suppose* $\Theta \vdash \rho \sqsubseteq \sigma$ *and* $\Theta \vdash \varepsilon \sqsubseteq \omega$.

- $\Theta; \cdot \vdash I^\sigma_\omega \in_{[\sigma/\Box]\omega \to [\mathtt{dyn}/\Box]\omega} [\rho/\Box]\varepsilon \to [\rho^*/\Box]\varepsilon$.
- $\Theta; \cdot \vdash J^\sigma_\omega \in_{[\mathtt{dyn}/\Box]\omega \to [\sigma/\Box]\omega} [\rho^*/\Box]\varepsilon \to [\rho/\Box]\varepsilon$.

Now we are ready to relate the type of an expression in System F and the refinement of the translation. The main difference from type translation is that the pcr system, which is equipped with System F typing structures, can capture parametric polymorphism that is lost in DPCF types. For each source type τ we write $\tau^\ddagger$ to be a particular refinement that suffices to complete the correctness proof (*i.e.*, the simulation relation). The following theorem improves Theorem 1 with refinement information.

Definition 7 (refinement translation)**.**

$$\begin{array}{lcl} t^\ddagger & = & t \\ \mathtt{nat}^\ddagger & = & \mathbb{N} \\ (\tau_1 \to \tau_2)^\ddagger & = & \tau_1^\ddagger \to \tau_2^\ddagger \\ (\forall t.\tau)^\ddagger & = & \mathbb{1} \to \forall t.\tau^\ddagger \end{array}$$

Lemma 7. *If* $\Delta \vdash \tau$ type *then* $\Delta^\ddagger \vdash \tau^\ddagger \sqsubseteq \tau^\dagger$ *where* $\Delta^\ddagger$ *is defined as turning each "*t type*" to "*$t \sqsubseteq \mathtt{dyn}$*".*

Theorem 2 (statics with pcr's)**.** *If* $\Delta; \Gamma \vdash e : \tau \Rightarrow d$*, then* $\Delta^\ddagger; \Gamma^\ddagger \vdash d \in_{\tau^\dagger} \tau^\ddagger$ *where* $\Gamma^\ddagger$ *is defined as applying the refinement translation to each type in the context.*

Proof. Induction on the translation derivation. □

5.2 Invertibility

The next step is to show that embedding and projection are inverse to each other under appropriate contexts, which is already hinted by their total function refinements in Lemmas 5 and 6.

Lemma 8 (invertibility of i and j)**.**

- $\Theta; \cdot \vdash j_\rho \circ i_\rho \mathrel{\dot\sim} \mathsf{id} \in \rho \to \rho$.
- $\Theta; \cdot \vdash i_\rho \circ j_\rho \mathrel{\dot\sim} \mathsf{id} \in \rho^* \to \rho^*$.

Lemma 9 (invertibility of I and J)**.**

- $\Theta; \cdot \vdash J^\rho_\varepsilon \circ I^\rho_\varepsilon \mathrel{\dot\sim} \mathsf{id} \in [\rho/\Box]\varepsilon \to [\rho/\Box]\varepsilon$.
- $\Theta; \cdot \vdash I^\rho_\varepsilon \circ J^\rho_\varepsilon \mathrel{\dot\sim} \mathsf{id} \in [\rho^*/\Box]\varepsilon \to [\rho^*/\Box]\varepsilon$.

5.3 Simulation Relation

Finally, taking the advantages of pcr's, we will establish a simulation relation between expressions in System F and the compiled expressions. To deal with impredicativity in System F, we will follow the standard approach consisting of self-admissibility and compositionality as we did for refined logical equivalence. We begin with the concept of *admissibility*.

Definition 8 (admissible simulation relation)**.** A binary relationship $\mathcal{R}$ between expressions e of System F and d of DPCF such that $e : \tau$ and $d \in_\sigma \rho$ is *admissible* (written $\mathcal{R} : \tau \leftrightarrow \rho$) if it respects *observational equivalence* for e and *refined observational equivalence* for d.

One might expect that ρ to be always $\tau^\ddagger$ (the refinement translation of τ) since e and d should be related if d is a translation of e, and $\tau^\ddagger$ is the translation of τ as stated in Theorem 2. This does not hold in general, however, because substitution does not commute with translation. For example, suppose $e = \lambda x{:}t.x$ with type $t \to t$ and we are interested the substitution $t \hookrightarrow \mathtt{nat}$. On the one hand, if we translate e before the substitution, the result will be $d = \lambda x{:}\mathtt{dyn}.x$ with refinement $\mathtt{num}(\mathbb{N}) \to \mathtt{num}(\mathbb{N})$. On the other hand, if we substitute $\mathtt{nat}$ for t first, then the result will be $d = \lambda x{:}\mathtt{nat}.x$ with refinement $\mathbb{N} \to \mathbb{N}$. The intuition is that once type variables are erased to $\mathtt{dyn}$, the type substitution has no effects on expressions and thus we need projections J's to bridge the discrepancy. This phenomenon will become apparent in Lemma 12.

We write the simulation relation as

$$e \sim_\tau d \; [\eta : \delta \leftrightarrow \theta]$$

where e is a System F expression of type $\hat\delta(\tau)$, d is a DPCF expression of refinement $\widehat{\theta^*}(\tau^\ddagger)$ and $\eta(t)$ is an admissible relation for $\delta(t)$ and $\theta(t)$ (formally $\eta(t) : \delta(t) \leftrightarrow \theta(t)$) for each t. For example,

$$\lambda x{:}\mathtt{nat}.x \sim_{t \to t} \lambda x{:}\mathtt{dyn}.x \; [(t \hookrightarrow \mathcal{R}_{\mathtt{nat}}) : (t \hookrightarrow \mathtt{nat}) \leftrightarrow (t \hookrightarrow \mathbb{N})],$$

where $\mathcal{R}_{\mathtt{nat}}$ is the Kleene equivalence, which asserts that $\lambda x{:}\mathtt{dyn}.x$ serves as an identity function for $\mathtt{nat}$, under the circumstances that t will be substituted with $\mathtt{nat}$.

The reason that the DPCF term d is of refinement $\widehat{\theta^*}(\tau^\ddagger)$ with refinement embedding $(^*)$, rather than simply $\hat\theta(\tau^\ddagger)$, is worth careful discussion. Each type variable t in F is translated to $\mathtt{dyn}$ and so we need some refinement of type $\mathtt{dyn}$ to match up. Moreover, the pcr system only accepts variable refinement of type $\mathtt{dyn}$. Since the refinement $\theta(t)$ that the candidate $\eta(t)$ accepts can be of any type, we need to embed $\theta(t)$ as $\theta(t)^*$ before refinement substitution, and perform a projection $j_{\theta(t)}$ for $\eta(t)$ when the variable case is reached.

The question is then, why do we allow candidates of arbitrary refinements in the first place? That is, why can not we avoid embedding by requesting all candidates to be admissible on a refinement of $\mathtt{dyn}$? The key is that we want the simulation relation itself to be admissible, and since the translation preserves all non-polymorphic typing information, the refinement will, in general, refine a type different from $\mathtt{dyn}$. In sum, the embedding is necessary.

Continuing the above example, despite that the refinement mapping is sending t to $\mathbb{N}$ since the candidate $\mathcal{R}_{\mathtt{nat}}$ is admissible between $\mathtt{nat}$ and $\mathbb{N}$, the DPCF term $\lambda x{:}\mathtt{dyn}.x$ has the refinement $\mathtt{num}(\mathbb{N}) \to \mathtt{num}(\mathbb{N})$, not $\mathbb{N} \to \mathbb{N}$, as the involved refinement embedding turns $\mathbb{N}$ into $\mathtt{num}(\mathbb{N})$ before the substitution. Here is the formal definition of our simulation relation:

Definition 9 (simulation relation)**.** Suppose $e : \hat\delta(\tau)$ and $d \in_{\tau^\dagger} \widehat{\theta^*}(\tau^\ddagger)$. The relation $e \sim_\tau d \; [\eta : \delta \leftrightarrow \theta]$ is defined inductively on the structure of τ as follows:

- $e \sim_t d \; [\eta : \delta \leftrightarrow \theta]$ iff $\eta(t)(e, j_{\theta(t)}(d))$.
- $e \sim_{\mathtt{nat}} d \; [\eta : \delta \leftrightarrow \theta]$ iff Kleene equivalent.
- $e \sim_{\tau_1 \to \tau_2} d \; [\eta : \delta \leftrightarrow \theta]$ iff
  $e_2 \sim_{\tau_1} d_2 \; [\eta : \delta \leftrightarrow \theta]$ implies $e\, e_2 \sim_{\tau_2} d\, d_2 \; [\eta : \delta \leftrightarrow \theta]$.
- $e \sim_{\forall t.\tau_1} d \; [\eta : \delta \leftrightarrow \theta]$ iff
  for every τ_2 and ρ_2 such that $\mathcal{R}_2 : \tau_2 \leftrightarrow \rho_2$,
  $e[\tau_2] \sim_{\tau_1} d\, \langle\rangle \; [\eta \otimes (t \hookrightarrow \mathcal{R}_2) : \delta \otimes (t \hookrightarrow \tau_2) \leftrightarrow \theta \otimes (t \hookrightarrow \rho_2)]$.

We can show that this particular relation is a good candidate, which says it is self-admissible.

Lemma 10 ($\sim$ is admissible)**.** $\cdot \sim_\tau \cdot \; [\eta : \delta \leftrightarrow \theta]$ *is admissible between* $\hat\delta(\tau)$ *and* $\widehat{\theta^*}(\tau^\ddagger)$.

We can also show the weakening and strengthening.

Lemma 11 (weakening and strengthening)**.** *We have*

$$e \sim_\tau d\ [\eta : \delta \leftrightarrow \theta]$$

if and only if

$$e \sim_\tau d\ [\eta' : \delta' \leftrightarrow \theta']$$

when δ', θ' and η' extend δ, θ and η, respectively.

However, the standard way to phrase compositionality, that is, that if the candidate for some type variable t is our simulation relation then we can substitute $\delta(t)$ for t in τ, fails, precisely because substitution and refinement translation do not commute. Formally speaking, this principle asserts that the two relations

$$e \sim_{[\tau_2/t]\tau_1} d\ [\eta : \delta \leftrightarrow \theta]$$

and

$$e \sim_{\tau_1} d\ [\eta \otimes (t \hookrightarrow (\cdot \sim_{\tau_2} \cdot\ [\eta : \delta \leftrightarrow \theta])) : \delta \otimes (t \hookrightarrow \hat{\delta}(\tau_2)) \leftrightarrow \rho \otimes (t \hookrightarrow \widehat{\theta^*}(\tau_2^\ddagger))]$$

should be interprovable, since the relation for the type variable t is the same as the main relation, and they should compose. This is false because changes in the System F type are reflected in the refinement, not the type, of the DPCF expression d, and thus we need projections J's to migrate the changes to the expression and its type. For example, we can rewrite the last example as

$$\lambda x{:}\mathtt{nat}.x \sim_{t\to t} \lambda x{:}\mathtt{dyn}.x\ [(t \hookrightarrow (\cdot \sim_{\mathtt{nat}} \cdot\ [\emptyset : \emptyset \leftrightarrow \emptyset])) : (t \hookrightarrow \mathtt{nat}) \leftrightarrow (t \hookrightarrow \mathbb{N})]$$

because we employ the Kleene equivalence for `nat`. The compositionality principle (phrased in the usual way) would assert

$$\lambda x{:}\mathtt{nat}.x \sim_{\mathtt{nat}\to\mathtt{nat}} \lambda x{:}\mathtt{dyn}.x\ [\emptyset : \emptyset \leftrightarrow \emptyset],$$

where t is substituted by `nat`, but this would be wrong as the translation of $\lambda x{:}\mathtt{nat}.x$ should be (equivalent to) $\lambda x{:}\mathtt{nat}.x$, not $\lambda x{:}\mathtt{dyn}.x$. Instead, the correct version comes with J as

$$\lambda x{:}\mathtt{nat}.x \sim_{\mathtt{nat}\to\mathtt{nat}} J^{\mathbb{N}}_{\square\to\square}(\lambda x{:}\mathtt{dyn}.x)\ [\emptyset : \emptyset \leftrightarrow \emptyset].$$

This correct formulation can be stated as follows.

Lemma 12 (quasi-compositionality)**.** *Let*

$$\mathcal{R}_\tau(e,d) = e \sim_\tau d\ [\eta : \delta \leftrightarrow \theta]$$
$$\mathcal{R}'_\tau(e,d) = e \sim_\tau d\ [\eta \otimes (t \hookrightarrow \mathcal{R}_{\tau_2}) : \delta \otimes (t \hookrightarrow \hat{\delta}(\tau_2)) \leftrightarrow \theta \otimes (t \hookrightarrow \widehat{\theta^*}(\tau_2^\ddagger))]$$

1. $\mathcal{R}'_{\tau_1}(e,d)$ *iff* $\mathcal{R}_{[\tau_2/t]\tau_1}(e, J^{\tau_2^\dagger}_{[\tau_1]_t}(d))$.
2. $\mathcal{R}_{[\tau_2/t]\tau_1}(e,d)$ *iff* $\mathcal{R}'_{\tau_1}(e, I^{\tau_2^\dagger}_{[\tau_1]_t}(d))$.

Proof. Induction on the structure of τ_1. It is sufficient to prove the first part because the second part directly follows from the first part with $d = I(d)$. (We only show the proof for function types.[7])

- $\tau_1 = \tau_3 \to \tau_4$. $[\tau_2/t](\tau_3 \to \tau_4) = [\tau_2/t]\tau_3 \to [\tau_2/t]\tau_4$.
  - ($\Rightarrow$) Suppose there are e_2 and d_2 such that

    $$\mathcal{R}_{[\tau_2/t]\tau_3}(e_2, d_2).$$

    By inductive hypothesis

    $$\mathcal{R}'_{\tau_3}(e_2, I^{\tau_2^\dagger}_{[\tau_3]_t}(d_2)).$$

    By definition of $\sim$ for function types

    $$\mathcal{R}'_{\tau_4}(e\,e_2, d\,(I^{\tau_2^\dagger}_{[\tau_3]_t}(d_2))).$$

    By inductive hypothesis

    $$\mathcal{R}_{[\tau_2/t]\tau_4}(e\,e_2, J^{\tau_2^\dagger}_{[\tau_4]_t}(d\,(I^{\tau_2^\dagger}_{[\tau_3]_t}(d_2)))).$$

    Since

    $$J^{\tau_2^\dagger}_{[\tau_4]_t}(d\,(I^{\tau_2^\dagger}_{[\tau_3]_t}(d_2))) \mathrel{\dot\cong} J^{\tau_2^\dagger}_{[\tau_3\to\tau_4]_t}(d)\,d_2,$$

    we have

    $$\mathcal{R}_{[\tau_2/t]\tau_4}(e\,e_2, J^{\tau_2^\dagger}_{[\tau_3\to\tau_4]_t}(d)\,d_2).$$

  - ($\Leftarrow$) Suppose there are e_2 and d_2 such that

    $$\mathcal{R}'_{\tau_3}(e_2, d_2).$$

    By inductive hypothesis

    $$\mathcal{R}_{[\tau_2/t]\tau_3}(e_2, J^{\tau_2^\dagger}_{[\tau_3]_t}(d_2)).$$

    By definition of $\sim$ for function types

    $$\mathcal{R}_{[\tau_2/t]\tau_4}(e\,e_2, J^{\tau_2^\dagger}_{[\tau_3\to\tau_4]_t}(d)\,(J^{\tau_2^\dagger}_{[\tau_3]_t}(d_2))).$$

    Since $I \circ J \mathrel{\dot\cong} \mathsf{id}$ we know

    $$J^{\tau_2^\dagger}_{[\tau_3\to\tau_4]_t}(d)\,(J^{\tau_2^\dagger}_{[\tau_3]_t}(d_2)) \mathrel{\dot\cong} J^{\tau_2^\dagger}_{[\tau_4]_t}(d\,(I^{\tau_2^\dagger}_{[\tau_3]_t}(J^{\tau_2^\dagger}_{[\tau_3]_t}(d_2)))) \mathrel{\dot\cong} J^{\tau_2^\dagger}_{[\tau_4]_t}(d\,d_2).$$

    By inductive hypothesis

    $$\mathcal{R}'_{\tau_4}(e\,e_2, d\,d_2).$$

□

We are now in a position to relate a System F expression and its translation. The following lemma works for all translation judgments, even the ones with non-empty contexts, saying that if each variable is substituted with some related expressions on both sides, then the result is also related.

Lemma 13 (fundamental lemma)**.** *Suppose $\Delta;\Gamma \vdash e : \tau \Rightarrow d$. For every candidates η, type substitution δ, refinement mapping θ, expression substitutions γ and ξ such that $\eta : \delta \leftrightarrow \theta$ and $\gamma \sim_\Gamma \xi\ [\eta : \delta \leftrightarrow \theta]$ we have*

$$\hat{\gamma}(\hat{\delta}(e)) \sim_\tau \hat{\xi}(d)\ [\eta : \delta \leftrightarrow \theta]$$

where $\gamma \sim_\Gamma \xi\ [\eta : \delta \leftrightarrow \theta]$ means $\gamma(x) \sim_{\Gamma(x)} \xi(x)\ [\eta : \delta \leftrightarrow \theta]$ for every variable x in the context Γ.

Proof. Induction on the derivation of $\Delta;\Gamma \vdash e : \tau \Rightarrow d$. (We only show the expression application and type application cases here.)

- (`ap`).

  $$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \Rightarrow d_1 \qquad \Delta;\Gamma \vdash e_2 : \tau_1 \Rightarrow d_2}{\Delta;\Gamma \vdash e_1\,e_2 : \tau_2 \Rightarrow d_1\,d_2}\ (\mathtt{ap})$$

  By inductive hypotheses $\hat{\gamma}(\hat{\delta}(e_1)) \sim_{\tau_1\to\tau_2} \hat{\xi}(d_1)\ [\eta : \delta \leftrightarrow \theta]$ and $\hat{\gamma}(\hat{\delta}(e_2)) \sim_{\tau_1} \hat{\xi}(d_2)\ [\eta : \delta \leftrightarrow \theta]$. By definition of $\sim$ we have $\hat{\gamma}(\hat{\delta}(e_1\,e_2)) \sim_{\tau_2} \hat{\xi}(d_1\,d_2)\ [\eta : \delta \leftrightarrow \theta]$.
- (`Ap`).

  $$\frac{\Delta \vdash \tau_2\ \mathsf{type} \qquad \Delta;\Gamma \vdash e_1 : \forall t.\tau_1 \Rightarrow d_1}{\Delta;\Gamma \vdash e_1[\tau_2] : [\tau_2/t]\tau_1 \Rightarrow J^{\tau_2^\dagger}_{[\tau_1]_t}(d_1\,\langle\rangle)}\ (\mathtt{Ap})$$

[7] For full details of the omitted proofs in this paper, please see the supplementary document.

By inductive hypothesis,

$$\hat{\gamma}(\hat{\delta}(e_1)) \sim_{\forall t.\tau_1} \hat{\xi}(d_1)\ [\eta : \delta \leftrightarrow \theta].$$

Let $\mathcal{R}_2(e,d) = e \sim_{\tau_2} d\ [\eta : \delta \leftrightarrow \theta]$. By Lemma 10 we have $\mathcal{R}_2 : \hat{\delta}(\tau_2) \leftrightarrow \widehat{\theta^*}(\tau_2^{\ddagger})$, which is to say that $\mathcal{R}_2$ is admissible. Therefore

$$\hat{\gamma}(\hat{\delta}(e_1))[\hat{\delta}(\tau_2)] \sim_{\tau_1} \hat{\xi}(d_1)\ \langle\rangle\ [\eta \otimes (t \hookrightarrow \mathcal{R}_2) : \delta \otimes (t \hookrightarrow \hat{\delta}(\tau_2)) \leftrightarrow \theta \otimes (t \hookrightarrow \widehat{\theta^*}(\tau_2^{\ddagger})],$$

which is

$$\hat{\gamma}(\hat{\delta}(e_1[\tau_2])) \sim_{\tau_1} \hat{\xi}(d_1)\ \langle\rangle\ [\eta \otimes (t \hookrightarrow \mathcal{R}_2) : \delta \otimes (t \hookrightarrow \hat{\delta}(\tau_2)) \leftrightarrow \theta \otimes (t \hookrightarrow \widehat{\theta^*}(\tau_2^{\ddagger})].$$

By Lemma 12 this implies

$$\hat{\gamma}(\hat{\delta}(e_1[\tau_2])) \sim_{[\tau_2/t]\tau_1} J^{\tau_2^{\dagger}}_{[\tau_1]_t}(\hat{\xi}(d_1)\ \langle\rangle)\ [\eta : \delta \leftrightarrow \theta],$$

which is exactly

$$\hat{\gamma}(\hat{\delta}(e_1[\tau_2])) \sim_{[\tau_2/t]\tau_1} \hat{\xi}(J^{\tau_2^{\dagger}}_{[\tau_1]_t}(d_1\ \langle\rangle))\ [\eta : \delta \leftrightarrow \theta],$$

as desired.

□

The main theorem below directly follows the lemma above as a special case, which guarantees that the semantics of every closed expression of type `nat` is preserved faithfully by the translation.

Theorem 3 (dynamics)**.** *If* e : `nat` $\Rightarrow$ d *then* e *and* d *are Kleene equivalent.*

6. Related Research

The idea of using runtime checking to implement parametric polymorphism has a long history of development [2, 4, 8]. Nonetheless, it seems that no arguments, stated in terms of logical relations, have been made for the correctness of this method. Here, we gave a positive answer regarding compiling a minimum language with parametric polymorphism (System F).

There is another line of research led by Ahmed, *et al.* [1], who have developed *polymorphic blame calculus*, which is capable of integrating polymorphism *with* dynamic typing. This is challenging because the runtime class checking might accidentally reveal extra information about the underlying type substituting for a type variable, which in turn breaks parametricity. The problem we study here has a different goal, which is to compile a language with polymorphism *to* another language with dynamic typing. To the best of our understanding, these two seemingly related problems require different formulations and techniques.

The use of embedding and projection pairs in this paper is inspired by the seminal work of Meyer and Wand. [17] on the correctness of CPS transformation. In their terminology we have demonstrated that every type in DPCF is a retraction of the type `dyn` of dynamic values. They also established a logical relation for the correctness of the transformation, but it requires the commutativity between J and expression application to be one of the invariants, which is surprisingly difficult to prove in our setting until the introduction of the refinements.

7. Conclusions and Further Work

Using refinements it is possible to prove the correctness of the compilation that relies on the epistemic invariants which can be awkward or impossible to express in the target language type. Moreover, we demonstrated how a new equivalence, possibly coarser than the standard observational equivalence (for a subset of expressions), can be systematically developed from refinements customized for a particular problem. This seems impossible because the standard observational equivalence is already the coarsest consistent congruence with respect to the type system. We argue that such apparent limitation is merely an illusion; even with the same language, different observational equivalences can be created, on demand, for different correctness criteria, by choosing different type refinements.

One possible direction is to extend the current account to imperative programs by introducing refinements for monadic types. Such refinements could be as simple as the effects considered by Gifford and Lucassen, [13] or as sophisticated as the Hoare triples considered by Nanevski, *et al.* [19, 20]. Birkedal, *et al.* [3] make critical use of such refinements in their correctness proofs, and we expect that their methods can be recast in terms of effect refinements for monads.

References

[1] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 201–214. ACM, 2011. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926409. URL http://doi.acm.org/10.1145/1926385.1926409.

[2] J. A. Bank, A. C. Myers, and B. Liskov. Parameterized types for Java. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 132–145. ACM, 1997. ISBN 0-89791-853-3. doi: 10.1145/263699.263714. URL http://doi.acm.org/10.1145/263699.263714.

[3] L. Birkedal, F. Sieczkowski, and J. Thamsborg. A concurrent logical relation. In *CSL*, pages 107–121, 2012. URL http://d-nb.info/1025751426/34#page=121.

[4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the Java programming language. *ACM SIGPLAN Notices*, 33(10):183–200, Oct. 1998. ISSN 0362-1340. doi: 10.1145/286942.286957. URL http://doi.acm.org/10.1145/286942.286957.

[5] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall, 1986. ISBN 978-0134518329. URL http://www.cs.cornell.edu/info/projects/nuprl/book/doc.html.

[6] Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA-Rocquencourt, 2012. URL http://coq.inria.fr/refman/.

[7] K. Crary and R. Harper. Syntactic logical relations for polymorphic and recursive types. *Electronic Notes in Theoretical Computer Science*, 172(0):259 – 299, 2007. ISSN 1571-0661. doi: 10.1016/j.entcs.2007.02.010. URL http://www.sciencedirect.com/science/article/pii/S1571066107000825. Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.

[8] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic, Volume II*. North-Holland, 1972. ISBN 0-7204-2208-6.

[9] R. Davies. *Practical Refinement-type Checking*. PhD thesis, Carnegie Mellon University, 2005.

[10] D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. In *Logic In Computer Science, 2009. LICS '09. 24th Annual IEEE Symposium on*, pages 71–80, 2009. doi: 10.1109/LICS.2009.34. URL http://dx.doi.org/10.1109/LICS.2009.34.

[11] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007.

[12] T. Freeman and F. Pfenning. Refinement types for ML. *SIGPLAN Not.*, 26(6):268–277, May 1991. ISSN 0362-1340. doi: 10.1145/113446.113468. URL http://doi.acm.org/10.1145/113446.113468.

[13] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 28–38. ACM, 1986. ISBN 0-89791-200-4. doi: 10.1145/319838.319848. URL http://doi.acm.org/10.1145/319838.319848.

[14] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de larithmétique dordre supérieur*. These d'état, Universit de Paris 7, 1972.

[15] R. Harper. *Practical foundations for programming languages*. Cambridge University Press, 2012. ISBN 978-1-107-02957-6. URL http://www.cs.cmu.edu/~rwh/plbook/book.pdf.

[16] W. A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. Reprint of 1969 article.

[17] A. Meyer and M. Wand. Continuation semantics in typed lambda-calculi. In R. Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 219–224. Springer Berlin Heidelberg, 1985. ISBN 978-3-540-15648-2. doi: 10.1007/3-540-15648-8_17. URL http://dx.doi.org/10.1007/3-540-15648-8_17.

[18] G. Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1995. URL http://www.eecs.harvard.edu/~greg/papers/thesis_ps.zip.

[19] A. Nanevski and G. Morrisett. Dependent type theory of stateful higher-order functions. Technical report, Technical Report TR-24-05, Harvard University, 2005.

[20] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. *SIGPLAN Not.*, 43(9):229–240, Sept. 2008. ISSN 0362-1340. doi: 10.1145/1411203.1411237. URL http://doi.acm.org/10.1145/1411203.1411237.

[21] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers, Göteborg University, 2007.

[22] M. Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004. URL http://scala-lang.org/docu/files/ScalaOverview.pdf.

[23] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977. ISSN 03043975. doi: 10.1016/0304-3975(77)90044-5. URL http://dx.doi.org/10.1016/0304-3975(77)90044-5.

[24] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer Berlin Heidelberg, 1974. ISBN 978-3-540-06859-4. doi: 10.1007/3-540-06859-7_148. URL http://dx.doi.org/10.1007/3-540-06859-7_148.

[25] R. Statman. Logical relations and the typed lambda-calculus. *Information and Control*, 65(2):85–97, 1985.