Breaking through the Normalization Barrier: A Self-interpreter for F-omega

Abstract

According to conventional wisdom, a self-interpreter for a strongly normalizing λ-calculus is impossible. We call this the normalization barrier. The normalization barrier stems from a theorem in computability theory that says that a total universal function for the total computable functions is impossible. In this paper we break through the normalization barrier and define a self-interpreter for System F_ω, a strongly normalizing λ-calculus. After a careful analysis of the classical theorem, we show that static type checking in F_ω excludes the proof's diagonalization gadget and leaves open the possibility for a self-interpreter. Along with the self-interpreter, we program four other operations in F_ω, including a continuation-passing style transformation. Our operations rely on a new approach to program representation that may be useful in theorem provers and compilers.

1. Introduction

Barendregt's notion of a *self-interpreter* is a program that recovers a program from its representation and is implemented in the language itself [1]. Specifically for λ-calculus, the challenge is to devise a quoter that maps each term e to a representation $\overline{\mathsf{e}}$, and a self-interpreter u (for *unquote*) such that for every λ-term e we have $(\mathsf{u}\ \overline{\mathsf{e}}) \equiv_\beta \mathsf{e}$. The quoter is an injective function from λ-terms to representations, which are λ-terms in normal form. For untyped λ-calculus, in 1936 Kleene [14] presented the first self-interpreter, and in 1994 Mogensen presented the first *strong* self-interpreter u that satisfies the property $(\mathsf{u}\ \overline{\mathsf{e}}) \longrightarrow^*_\beta \mathsf{e}$. In 2009, Rendel, Ostermann, and Hofer [20] presented the first self-interpreter for a *typed* λ-calculus (F^*_ω), and in 2015 Brown and Palsberg [4] presented the first self-interpreter for a typed λ-calculus with *decidable* type checking (Girard's System U). Those results are all for non-normalizing λ-calculi and they go about as far as one can go before reaching what we call the *normalization barrier*.

The normalization barrier: According to conventional wisdom, a self-interpreter for a strongly normalizing λ-calculus is impossible.

The normalization barrier stems from a theorem in computability theory that says that a total universal function for the total computable functions is impossible. Several books, papers, and web pages have concluded that the theorem about total universal functions carries over to self-interpreters for strongly normalizing languages. For example, Turner states that "For any language in which all programs terminate, there are always terminating programs which cannot be written in it - among these are the interpreter for the language itself" [26, pg. 766]. Similarly, Stuart writes that "Total programming languages are still very powerful and capable of expressing many useful computations, but one thing they can't do is interpret themselves" [23, pg. 264]. Additionally, the Wikipedia page on the *Normalization Property* (accessed in May 2015) explains that a self-interpreter for a strongly normalizing λ-calculus is impossible. That Wikipedia page cites three typed λ-calculi, namely simply typed λ-calculus, System F, and the Calculus of Constructions, each of which is a member of Barendregt's cube of typed λ-calculi [2]. We can easily add examples to that list, particularly the other five corners of Barendregt's λ-cube, including F_ω. The normalization barrier implies that a self-interpreter is impossible for every language in the list. In a seminal paper in 1991 Pfenning and Lee [18] considered whether one can define a self-interpreter for System F or F_ω and found that the answer seemed to be "no". They went on to represent F in F_ω, and F_ω in F^+_ω.

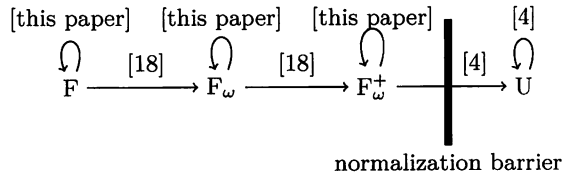

Figure 1: Four typed λ-calculi: $\rightarrow$ denotes "represented in."

In this paper we take up the challenge presented by the normalization barrier.

The challenge: Can we define a self-interpreter for a strongly normalizing λ-calculus?

Our result: Yes, we present a strong self-interpreter for the strongly normalizing λ-calculus F_ω; the program representation is *deep* and supports a variety of other operations. We also present a much simpler self-interpreter that works for each of System F, F_ω, and F^+_ω; the program representation is *shallow* and supports no other operations.

Figure 1 illustrates how our result relates to other representations of typed λ-calculi with decidable type checking. The normalization barrier separates the three strongly-normalizing languages on the left from System U on the right, which is not strongly-normalizing. Our result contributes the self-loops on F, F_ω, and F^+_ω, depicting the first self-representations for strongly-normalizing languages.

Our result breaks through the normalization barrier and relies on three insights. First, we observe that F_ω excludes some problematic λ-terms. Specifically, the proof of the classical theorem in computability theory uses a diagonalization gadget that fails to type check in F_ω, so the proof doesn't carry over to F_ω. Second, for our deep representation we use

a novel *extensional* approach to representing polymorphic terms. We use instantiation functions that describe the relationship between a quantified type and one of its instance types. Each instantiation function takes as input a term of a quantified type, and instantiates it with a particular parameter type. Third, for our deep representation we use a novel representation of types, which helps us type check a continuation-passing-style transformation.

We present five operations on our deep representation, namely a strong self-interpreter, a continuation-passing-style transformation, an intensional predicate for testing whether a closed term is an abstraction or an application, a size measure, and a normal-form checker. Our list of operations extends the lists presented by Rendel et al. [20] and by Brown and Palsberg [4].

Our deep self-representation of F_ω could be useful for type-checking self-applicable metaprograms, with potential for applications in typed macro systems, partial evaluators, compilers, and theorem provers. In particular, F_ω is a subset of the proof language of the Coq proof assistant, and Morrisett has called F_ω *the workhorse of modern compilers* [16].

Our deep representation is the most powerful self-representation of F_ω that we have identified: it supports all the five operations listed above. One can define several other representations for F_ω by using fewer of our insights. Ultimately, one can define a *shallow* representation that supports only a self-interpreter and nothing else. As a stepping stone towards explaining our main result, we will show a shallow representation and a self-interpreter in Section 3.3. That representation and self-interpreter have the distinction of working for System F, F_ω and F_ω^+. Thus, we have solved the two challenges left open by Pfenning and Lee [18].

Rest of the paper. In Section 2 we describe F_ω, in Section 3 we analyze the normalization barrier, in Section 4 we describe instantiation functions, in Section 5 we show how to represent types, in Section 6 we show how to represent terms, in Section 7 we present our operations on program representations, in Section 8 we discuss our implementation and experiments, and in Section 9 we compare with related work. Proofs of theorems stated throughout the paper are provided in an appendix that we have submitted as supplementary material to this POPL submission.

2. System F_ω

System F_ω is a typed λ-calculus within the λ-cube [2]. It combines two axes of the cube: polymorphism and higher-order types (type-level functions). In this section we summarize the key properties of System F_ω used in this paper. We refer readers interested in a complete tutorial to other sources [2, 19]. We give a definition of F_ω in Figure 2. It includes a grammar, rules for type formation and equivalence, and rules for term formation and reduction. The grammar defines the kinds, types, terms, and environments. As usual, types classify terms, kinds classify types, and environments classify free term and type variables. Every syntactically well-formed kind and environment is legal, so we do not include separate formation rules for them. The type formation rules determine the legal types in a given environment, and assigns a kind to each legal type. Similarly, the term formation rules determine the legal terms in a given environment, and assigns a type to each legal term. Our definition is similar to Pierce's [19], with two differences: we use a slightly different syntax, and our semantics is arbitrary β-reduction instead of call-by-value.

It is well known that System F_ω is strongly normalizing, that type checking is decidable, and that types in F_ω are unique up to β-equivalence. Strong normalization means that every sequence of β-reductions eventually terminates with a β-normal form.

We require that representations of terms be *data*, which for λ-calculus usually means a term in β-normal form. In particular, a term e is β-normal if there is no e' such that $\mathsf{e} \longrightarrow \mathsf{e}'$.

3. The Normalization Barrier

In this section, we explore the similarity of a universal computable function in computability theory and a self-interpreter for a programming language. As we shall see, the exploration has a scary beginning and a happy ending. At first, a classical theorem in computability theory seems to imply that a self-interpreter for F_ω is impossible. Fortunately, further analysis reveals that the proof relies on an assumption that a diagonalization gadget can always be defined for a language with a self-interpreter. We show this assumption to be false: by using a *typed representation*, it is possible to define a self-interpreter such that the diagonalization gadget cannot possibly type check. We conclude the section by demonstrating a simple typed self-representation and a self-interpreter for F_ω.

3.1 Functions from numbers to numbers

We recall a classical theorem in computability theory (Theorem 3.2). The proof of the theorem is a diagonalization argument, which we divide into two steps: first we prove a key property (Theorem 3.1) and then we proceed with the proof of Theorem 3.2.

Let $\mathbb{N}$ denote the set of natural numbers $\{0, 1, 2, \ldots\}$. Let $\bar{\cdot}$ be an injective function that maps each total, computable function in $\mathbb{N} \to \mathbb{N}$ to an element of $\mathbb{N}$.

We say that $\mathsf{u} \in (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ is a universal function for the total, computable functions in $\mathbb{N} \to \mathbb{N}$, if for every total, computable function f in $\mathbb{N} \to \mathbb{N}$, we have $\forall \mathsf{v} \in \mathbb{N} : \mathsf{u}(\overline{\mathsf{f}}, \mathsf{v}) = \mathsf{f}(\mathsf{v})$. We let $\mathsf{Univ}(\mathbb{N} \to \mathbb{N})$ denote the set of universal functions for the total, computable functions in $\mathbb{N} \to \mathbb{N}$.

Given a function u in $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$, we define the function $\mathsf{p_u}$ in $\mathbb{N} \to \mathbb{N}$, where $\mathsf{p_u}(\mathsf{x}) = \mathsf{u}(\mathsf{x}, \mathsf{x}) + 1$.

Theorem 3.1. *If* $\mathsf{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$, *then* $\mathsf{p_u}$ *isn't total.*

Proof. Suppose $\mathsf{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$ and $\mathsf{p_u}$ is total. Notice that $\mathsf{p_u}$ is a total, computable function in $\mathbb{N} \to \mathbb{N}$ so $\overline{\mathsf{p_u}}$ is defined. We calculate:

$$\mathsf{p_u}(\overline{\mathsf{p_u}}) = \mathsf{u}(\overline{\mathsf{p_u}}, \overline{\mathsf{p_u}}) + 1 = \mathsf{p_u}(\overline{\mathsf{p_u}}) + 1$$

Given that $\mathsf{p_u}$ is total, we have that $\mathsf{p_u}(\overline{\mathsf{p_u}})$ is defined; let us call the result v. From $\mathsf{p_u}(\overline{\mathsf{p_u}}) = \mathsf{p_u}(\overline{\mathsf{p_u}}) + 1$, we get $\mathsf{v} = \mathsf{v} + 1$, which is impossible. So we have reached a contradiction, hence our assumption (that $\mathsf{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$ and $\mathsf{p_u}$ is total) is wrong. We conclude that if $\mathsf{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$, then $\mathsf{p_u}$ isn't total. □

Theorem 3.2. *If* $\mathsf{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$, *then* u *isn't total.*

Proof. Suppose $\mathsf{u} \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$ and u is total. For every $\mathsf{x} \in \mathbb{N}$, we have that $\mathsf{p_u}(\mathsf{x}) = \mathsf{u}(\mathsf{x}, \mathsf{x}) + 1$. Since u is total, $\mathsf{u}(\mathsf{x}, \mathsf{x}) + 1$ is defined, and therefore $\mathsf{p_u}(\mathsf{x})$ is also defined. Since $\mathsf{p_u}(\mathsf{x})$ is defined for every $\mathsf{x} \in \mathbb{N}$, $\mathsf{p_u}$ is total. However, Theorem 3.1 states that $\mathsf{p_u}$ is not total. Thus we have reached a contradiction, so our assumption (that $\mathsf{u} \in$

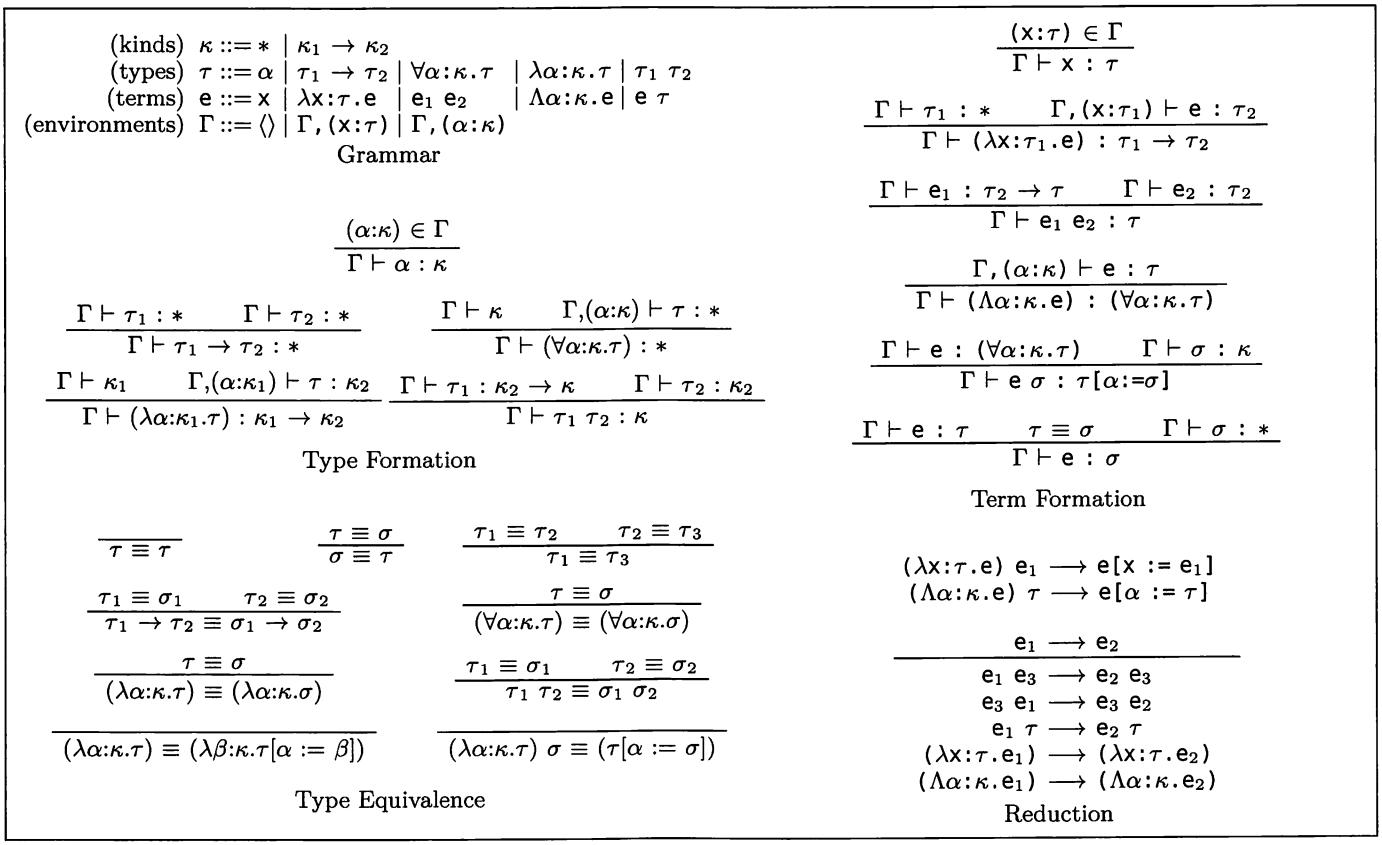

Figure 2: Definition of System F_ω

$\mathsf{Univ}(\mathbb{N} \to \mathbb{N})$ and u is total) is wrong. We conclude that if $u \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$, then u isn't total. □

Intuitively, Theorem 3.2 says that if we write an interpreter for the total, computable functions in $\mathbb{N} \to \mathbb{N}$, then that interpreter must go into an infinite loop on some inputs.

3.2 Typed λ-calculus: F_ω

Does Theorem 3.2 imply that a self-interpreter for F_ω is impossible? Recall that every term in F_ω is strongly normalizing. So, if we have a self-interpreter u for F_ω and we have $(u\ e) \in F_\omega$, then $(u\ e)$ is strongly normalizing, which is intuitively expresses that u is a total function. Thus, Theorem 3.2 seems to imply that a self-interpreter for F_ω is impossible. This is the normalization barrier. Let us examine this intuition via a "translation" of Section 3.1 to F_ω.

Let us recall the definition of a self-interpreter from Section 1, here for F_ω. A quoter is an injective function from terms in F_ω to their representations, which are β-normal terms in F_ω. We write $\overline{e}$ to denote the representation of a term e. We say that $u \in F_\omega$ is a self-interpreter for F_ω, if $\forall e \in F_\omega$: $(u\ \overline{e}) \equiv_\beta e$. We allow $(u\ \overline{e})$ to include type abstractions or applications as necessary, and leave them implicit. We use $\mathsf{SelfInt}(F_\omega)$ to denote the set of self-interpreters for F_ω.

Notice a subtle difference between the definition of a universal function in Section 3.1 and the definition of a self-interpreter. The difference is that a universal function takes both its arguments at the same time, while, intuitively, a self-interpreter is *curried* and takes its arguments one by one. This difference plays no role in our further analysis.

The proof of Theorem 3.1 relies on the diagonalization gadget $(p_u\ \overline{p_u})$, where p_u is a cleverly defined function. The idea of the proof is to achieve the equality $(p_u\ \overline{p_u}) = (p_u\ \overline{p_u}) + 1$. For the F_ω version of Theorem 3.1, our idea is to achieve the equality $(p_u\ \overline{p_u}) \equiv_\beta \lambda y.(p_u\ \overline{p_u})$, where y is fresh. Here, λy plays the role of "+1". Given $u \in F_\omega$, we define $p_u = \lambda x.\ \lambda y.\ ((u\ x)\ x)$, where x,y are fresh, and where we omit suitable type annotations for x,y. We can now state an F_ω version of Theorem 3.1.

Theorem 3.3. *If* $u \in \mathsf{SelfInt}(F_\omega)$, *then* $(p_u\ \overline{p_u}) \notin F_\omega$.

Proof. Suppose $u \in \mathsf{SelfInt}(F_\omega)$ and $(p_u\ \overline{p_u}) \in F_\omega$. We calculate:

$$\begin{aligned} & p_u\ \overline{p_u} \\ \equiv_\beta\ & \lambda y.\ ((u\ \overline{p_u})\ \overline{p_u}) \\ \equiv_\beta\ & \lambda y.\ (p_u\ \overline{p_u}) \end{aligned}$$

From $(p_u\ \overline{p_u}) \in F_\omega$ we have that $(p_u\ \overline{p_u})$ is strongly normalizing. From the Church-Rosser property of F_ω, we have that $(p_u\ \overline{p_u})$ has a unique normal form; let us call it v. From $(p_u\ \overline{p_u}) \equiv_\beta \lambda y.(p_u\ \overline{p_u})$ we get $v \equiv_\beta \lambda y.v$. Notice that v and $\lambda y.v$ are *distinct* yet β-equivalent normal forms. Now the Church-Rosser property implies that β-equivalent terms must have the *same* normal form. Thus $v \equiv_\beta \lambda y.v$ implies $v \equiv_\alpha \lambda y.v$, which is false. So we have reached a contradiction, hence our assumption (that $u \in \mathsf{SelfInt}(F_\omega)$ and $(p_u\ \overline{p_u}) \in F_\omega$) is wrong. We conclude that if $u \in \mathsf{SelfInt}(F_\omega)$, then $(p_u\ \overline{p_u}) \notin F_\omega$. □

What is an F_ω version of Theorem 3.2? Given that every term in F_ω is "total" in the sense described earlier, Theorem 3.2 suggests that we should expect $\mathsf{SelfInt}(F_\omega) = \emptyset$. However this turns out to be wrong and indeed in this paper we will define a self-representation and self-interpreter for F_ω. So, $\mathsf{SelfInt}(F_\omega) \neq \emptyset$.

We saw earlier that Theorem 3.1 helped prove Theorem 3.2. Why does Theorem 3.3 fail to lead the conclusion $\mathsf{SelfInt}(F_\omega) = \emptyset$? Observe that in the proof of Theorem 3.2, the key step was to notice that if u is total, also $\mathsf{p_u}$ is total, which contradicts Theorem 3.1. In contrast, the assumption $\mathsf{u} \in \mathsf{SelfInt}(F_\omega)$ produces no useful conclusion like $(\mathsf{p_u}\ \overline{\mathsf{p_u}}) \in F_\omega$ that would contradict Theorem 3.3. In particular, it is possible for u and $\mathsf{p_u}$ to be typeable in F_ω, and yet for $(\mathsf{p_u}\ \overline{\mathsf{p_u}})$ to be untypeable. So, the door is open for a self-interpreter for F_ω.

3.3 A self-interpreter for F_ω

Inspired by the optimism that emerged in Section 3.2, let us now define a quoter and a self-interpreter for F_ω. The quoter will support *only* the self-interpreter and nothing else. The idea of the quoter is to use a designated variable `id` to block the reduction of every application. The self-interpreter unblocks reduction by substituting the polymorphic identity function for `id`. Below we define the representation $\overline{\mathsf{e}}$ of a closed term e.

$$\Gamma \vdash \mathsf{x} : \tau \rhd \mathsf{x}$$

$$\frac{\Gamma, (\mathsf{x}{:}\tau_1) \vdash \mathsf{e} : \tau_2 \rhd \mathsf{q}}{\Gamma \vdash (\lambda \mathsf{x}{:}\tau_1.\mathsf{e}) : \tau_1 \to \tau_2 \rhd (\lambda \mathsf{x}{:}\tau_1.\mathsf{q})}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \tau_2 \to \tau \rhd \mathsf{q}_1 \qquad \Gamma \vdash \mathsf{e}_2 : \tau_2 \rhd \mathsf{q}_2}{\Gamma \vdash \mathsf{e}_1\ \mathsf{e}_2 : \tau \rhd \mathsf{id}\ (\tau_2 \to \tau)\ \mathsf{q}_1\ \mathsf{q}_2}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash \mathsf{e} : \tau \rhd \mathsf{q}}{\Gamma \vdash (\Lambda \alpha{:}\kappa.\mathsf{e}) : (\forall \alpha{:}\kappa.\tau) \rhd (\Lambda \alpha{:}\kappa.\mathsf{q})}$$

$$\frac{\Gamma \vdash \mathsf{e} : (\forall \alpha{:}\kappa.\tau_1) \rhd \mathsf{q} \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash \mathsf{e}\ \tau_2 : (\tau_1[\alpha := \tau_2]) \rhd \mathsf{id}\ (\forall \alpha{:}\kappa.\tau_1)\ \mathsf{q}\ \tau_2}$$

$$\frac{\Gamma \vdash \mathsf{e} : \tau \rhd \mathsf{q} \qquad \tau \equiv \sigma \qquad \Gamma \vdash \sigma : *}{\Gamma \vdash \mathsf{e} : \sigma \rhd \mathsf{q}}$$

$$\frac{\langle\rangle \vdash \mathsf{e} : \tau \rhd \mathsf{q}}{\overline{\mathsf{e}} = \lambda \mathsf{id}{:}(\forall \alpha{:}*.\ \alpha \to \alpha).\ \mathsf{q}}$$

Our representation is defined in two steps. First, the rules of the form $\Gamma \vdash \mathsf{e} : \tau \rhd \mathsf{q}$ build a pre-representation q from the typing judgment of a term e. The types are needed to instantiate each occurrence of the designated variable `id`. The representation $\overline{\mathsf{e}}$ is defined by abstracting over `id` in the pre-representation. Our self-interpreter takes a representation as input and applies it to the polymorphic identity function:

$$\begin{aligned}\mathsf{unquote} &: \forall \alpha{:}*.\ ((\forall \beta{:}*.\beta \to \beta) \to \alpha) \to \alpha \\ &= \Lambda \alpha{:}*.\ \lambda \mathsf{q}{:}(\forall \beta{:}*.\beta \to \beta) \to \alpha. \\ &\quad\ \mathsf{q}\ (\Lambda \beta{:}*.\ \lambda \mathsf{x}{:}\beta.\ \mathsf{x})\end{aligned}$$

Theorem 3.4.
If $\langle\rangle \vdash \mathsf{e} : \tau$, *then* $\overline{\mathsf{e}}$ *is normal and* $\langle\rangle \vdash \overline{\mathsf{e}} : (\forall \alpha{:}*.\ \alpha \to \alpha) \to \tau$.

Theorem 3.5.
If $\langle\rangle \vdash \mathsf{e} : \tau$, *then* $\mathsf{unquote}\ \tau\ \overline{\mathsf{q}} \longrightarrow^* \mathsf{e}$.

This self-interpreter demonstrates that it is possible to break through the normalization barrier. In fact, we can define a similar self-representation and self-interpreter for System F and for System F^+_ω. However, the representation supports no other operations than `unquote`: parametricity implies that the polymorphic identity function is the *only* possible argument to a representation $\overline{\mathsf{e}}$ [28]. The situation is similar to the one faced by Pfenning and Lee who observed that "evaluation is just about the only useful function definable" for their representation of F_ω in F^+_ω. We call a representation *shallow* if it supports only one operation, and we call representation *deep* if it supports a variety of operations. While our representation above is shallow, we have found it to be a good starting point for designing deep representations.

In Figure 3 we define a *deep* self-representation of F_ω that supports operations as varied as `unquote`, a CPS-transformation, and a normal-form checker. The keys to why this works are two novel techniques along with typed Higher-Order Abstract Syntax (HOAS), all of which we will explain in the following sections. First, in Section 4 we present an extensional approach to representing polymorphism in F_ω. Second, in Section 5 we present a simple representation of types that is sufficient to support our CPS transformation. Third, in Section 6 we present a typed HOAS representation based on church encoding, which supports operations that fold over the term. Finally, in Section 7 we define five operations for our representation.

4. Representing Polymorphism

In this section, we discuss our extensional approach to representing polymorphic terms in our type Higher-Order Abstract Syntax representation. Our approach allows us to define our HOAS representation of F_ω in F_ω itself. Before presenting our extensional approach, we will review the intensional approach used by previous work. As a running example, we consider how to program an important piece of a self-interpreter for a HOAS representation.

Our HOAS representation, like those of Pfenning and Lee [18], Rendel et al. [20], and Brown and Palsberg [4], is based on Church encoding. Operations are defined by cases functions, one for each of λ-abstraction, application, type abstraction, and type application. Our representation differs from the previous work in how we type check the case functions for type abstraction and type applications. Our running example will focus just on the case function for type applications. To simplify further, we consider only the case function for type applications in a self-interpreter.

4.1 The Intensional Approach

The approach of previous work [4, 18, 20] used a *polymorphic type-application function* to encode type applications. A polymorphic type application function can apply any polymorphic term to any type in its domain. The function tapp^+ defined below is a polymorphic type application function for System F^+_ω. System F^+_ω extends F_ω with kind abstractions and applications in terms (written $\Lambda\kappa.\mathsf{e}$ and $\mathsf{e}\ \kappa$ respectively), and kind-polymorphic types (written $\forall^+\kappa.\tau$):

$$\begin{aligned}\mathsf{tapp}^+ &: (\forall^+\kappa.\forall \beta{:}\kappa{\to}*.(\forall \alpha{:}\kappa.\ \beta\ \alpha) \to \forall \gamma.\kappa.\beta\ \gamma) \\ \mathsf{tapp}^+ &= \Lambda^+\kappa.\Lambda \beta{:}\kappa{\to}*.\lambda \mathsf{e}{:}(\forall \alpha{:}\kappa.\beta\ \alpha).\ \Lambda \gamma{:}\kappa.\mathsf{e}\ \gamma\end{aligned}$$

The first two parameters are the domain and codomain of an arbitrary quantified type. The domain of $(\forall \alpha{:}\kappa.\tau)$ is the kind κ, and the codomain is the type function $(\lambda \alpha{:}\kappa.\tau)$ since

τ depends on a type parameter α of kind κ. Instantiating the quantified type with a type parameter σ results in the type $(\lambda\alpha{:}\kappa.\tau)\ \sigma \equiv_\beta (\tau[\alpha{:=}\sigma])$. Since the body of a quantified type must have kind $*$, the codomain function $(\lambda\alpha{:}\kappa.\tau)$ must have kind $(\kappa \to *)$. A quantified type can be expressed in terms of its domain and codomain: $(\forall\alpha{:}\kappa.\tau) \equiv (\forall\alpha{:}\kappa.\ (\lambda\alpha{:}\kappa.\tau)\ \alpha)$. The type of e in tapp^+ is expressed in terms of an arbitrary domain κ and an arbitrary codomain β. For any quantified type σ, it is possible to instantiate κ and β so that the type of e is equivalent to σ.

We call this encoding *intensional* because it abstracts over the parts of a quantified type (its domain κ and codomain β). This ensures that e can only have a quantified type, and that γ ranges over exactly the types to which e can be applied. In other words, γ can be instantiated with σ if and only if e σ is well-typed.

Consider a type application e σ with the derivation:

$$\frac{\Gamma \vdash \mathsf{e} : (\forall\alpha{:}\kappa.\tau) \qquad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash \mathsf{e}\ \sigma : \tau[\alpha{:=}\sigma]}$$

We can encode e σ in F^+_ω as $\mathsf{tapp}^+\kappa\ (\lambda\alpha{:}\kappa.\tau)$ e σ. Since F_ω does not support kind polymorphism, we can't use this technique encode F_ω type applications in itself. To represent F_ω in itself, we need a new approach.

4.2 An Extensional Approach

The approach we use in this paper is *extensional:* we focus on the relationship between a quantified type and its instances. We encode the relationship "$(\tau[\alpha{:=}\sigma])$ is an instance of $(\forall\alpha{:}\kappa.\tau)$" as an *instantiation function* of type $(\forall\alpha{:}\kappa.\tau) \to (\tau[\alpha{:=}\sigma])$. The instantiation function $\mathsf{inst}_{\tau,\sigma} = \lambda\mathsf{x}{:}\tau.\ \mathsf{x}\ \sigma$ instantiates an input term of type τ with the type σ. It is well-typed only when τ is quantified type and σ is in the domain of τ.

The advantage of using instantiation functions is that all quantified types and all instantiations of quantified types are types of kind $*$. Thus, we can encode the rule for type applications in F_ω by abstracting over the quantified type, the instance type, and the instantiation function for them:

```
tapp : (∀α:*. α → β → (α → β) → β)
tapp = Λα:*. λe:α. Λβ:*. λinst:α → β. inst e
```

Using tapp we can encode the type application e σ above as $(\mathsf{tapp}\ (\forall\alpha{:}\kappa.\tau)\ (\tau[\alpha{:=}\sigma])\ \mathsf{e}\ \mathsf{inst}_{((\forall\alpha:\kappa.\tau),\sigma)})$.

Unlike the intensional approach, the extensional approach provides no guarantee that e will always have a quantified type. Furthermore, even if e does have a quantified type, inst is not guaranteed to actually be an instantiation function. In short, the intensional approach provides two Free Theorems [28] that we don't get with our extensional approach. However, the extensional approach has the key advantage of enabling a representation of F_ω in itself.

5. Representing Types

We use type representations to typecheck term representations and operations on term representations. Our type representation is shown as part of the representation of F_ω in Figure 3. The $[\![\tau]\!]$ syntax denotes the pre-representation of the type τ, while $\overline{\tau}$ denotes the representation. A pre-representation is defined using a designated variable F, and a representation abstracts over F.

Our type representation is novel and designed to support three important properties: first, it can represent all types (not just types of kind $*$); second, representation preserves equivalence between types; third, it is expressive enough to typecheck all our benchmark operations. The first and second properties were not supported by the type representations used in previous work [4], and play an important part in our representation of polymorphic terms.

Type representations supports operations that iterate a type function R over the first-order types – arrows and universal quantifiers. Each operation on representations produces results of the form R ($[\![\tau]\!]$[F := R]), which we call the "interpretation of τ under R". For example, the interpretation of $(\forall\alpha{:}*.\ \alpha \to \alpha)$ under R is R ($[\![\forall\alpha{:}*.\ \alpha \to \alpha]\!]$[F := R]) = R ($\forall\alpha{:}*.$ R (R $\alpha \to$ R α)).

As stated previously, type representations are used to typecheck representations of terms and their operations. In particular, a term of type τ is represented by a term of type Exp $\overline{\tau}$, and each operation on term representation produces results with types that are interpretations under some R.

Let's consider the outputs produced by unquote, size, and cps, when applied to a representation of the polymorphic identity function, which has the type $(\forall\alpha{:}*.\ \alpha \to \alpha)$. For unquote, the type function R is the identity function Id = $(\lambda\alpha{:}*.\alpha)$. Therefore, unquote applied to the representation of the polymorphic identity function will produce an output with the type Id $(\forall\alpha{:}*.$ Id (Id $\alpha \to$ Id $\alpha)) \equiv (\forall\alpha{:}*.\alpha \to \alpha)$. For size, R is the constant function KNat = $(\lambda\alpha{:}*.$Nat). Therefore, size applied to the representation of the polymorphic identity function will produce an output with the type KNat $(\forall\alpha{:}*.$ KNat (KNat $\alpha \to$ KNat $\alpha)) \equiv$ Nat. For cps, R is the function Ct = $(\lambda\alpha{:}*.\ \forall\beta{:}*.\ (\alpha \to \beta) \to \beta)$, such that Ct α is the type of a continuation for values of type α. Therefore, cps applied to the representation of the polymorphic identity function will produce an output with the type Ct $(\forall\alpha{:}*.$ Ct (Ct $\alpha \to$ Ct $\alpha))$. This type suggests that every sub-term has been transformed into continuation-passing style.

We also represent higher-order types, since arrows and quantifiers can occur within them. Type variables, abstractions, and applications are represented meta-circularly. Intuitively, the pre-representation of an abstraction is an abstraction over pre-representations. Since pre-representations of κ-types (i.e. types of kind κ) are themselves κ-types, an abstraction over κ-types can also abstract over pre-representations of κ-types. In other words, abstractions are represented as themselves. The story is the same for type variables and applications.

Examples. The representation of $(\forall\alpha{:}*.\ \alpha \to \alpha)$ is:

$$\begin{aligned}&\overline{\forall\alpha{:}*.\ \alpha \to \alpha}\\ &= \lambda\mathsf{F}{:}*\to*.\ [\![\forall\alpha{:}*.\ \alpha \to \alpha]\!]\\ &= \lambda\mathsf{F}{:}*\to*.\ \forall\alpha{:}*.\ \mathsf{F}\ (\mathsf{F}\ \alpha \to \mathsf{F}\ \alpha)\end{aligned}$$

Our representation is defined so that the representations of two β-equivalent types are also β-equivalent. In other words, representation of types preserves β-equivalence. In particular, we can normalize a type before or after representation, with the same result. For example,

$$\begin{aligned}&\overline{\forall\alpha{:}*.(\lambda\gamma{:}*.\gamma \to \gamma)\ \alpha}\\ &= \lambda\mathsf{F}{:}*\to*.\ [\![\forall\alpha{:}*.(\lambda\gamma{:}*.\gamma \to \gamma)\ \alpha]\!]\\ &= \lambda\mathsf{F}{:}*\to*.\ \forall\alpha{:}*.\ \mathsf{F}\ ((\lambda\gamma{:}*.\mathsf{F}\ \gamma \to \mathsf{F}\ \gamma)\ \alpha)\\ &\equiv_\beta \lambda\mathsf{F}{:}*\to*.\ \forall\alpha{:}*.\ \mathsf{F}\ (\mathsf{F}\ \alpha \to \mathsf{F}\ \alpha)\\ &= \overline{\forall\alpha{:}*.\ \alpha \to \alpha}\end{aligned}$$

Properties. We now discuss some properties of our type representation that are important for representing terms. First, we can pre-represent legal types of any kind and in any environment. Since a representation abstracts over the designated type variable F in a pre-representation, the

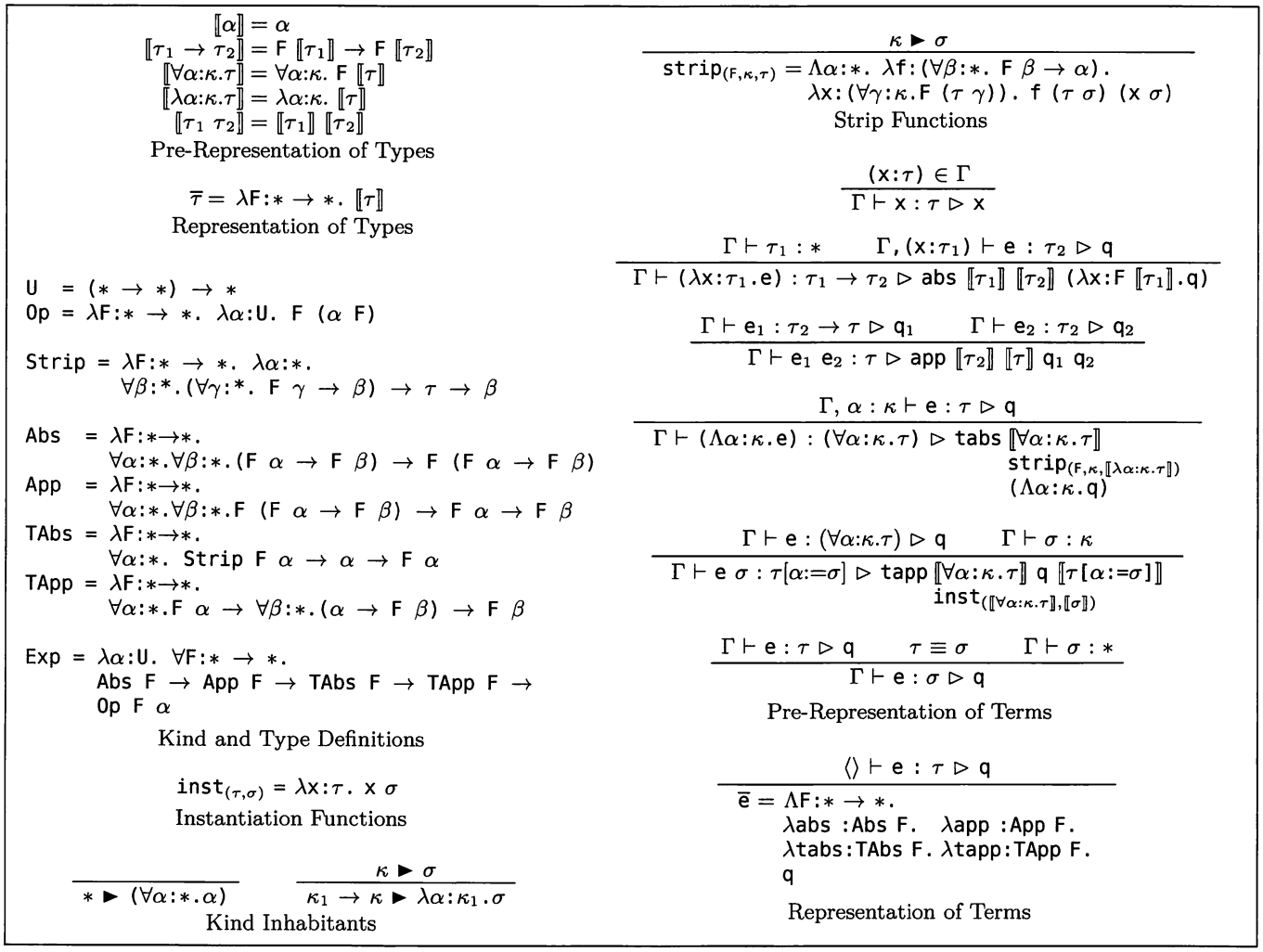

Figure 3: Self-Representation of F_ω

representation of a κ-type is a type of kind $(* \to *) \to \kappa$. In particular, base types (i.e. types of kind $*$) are represented by a type of kind $(* \to *) \to *$. This kind will be important for representing terms, so in Figure 3 we define $\mathsf{U} = (* \to *) \to *$.

Theorem 5.1. *If* $\Gamma \vdash \tau : \kappa$, *then* $\Gamma \vdash \overline{\tau} : (* \to *) \to \kappa$.

Equivalence preservation relies on the following substitution theorem, which will also be important for our representation of terms.

Theorem 5.2. *For any types* τ *and* σ, *and any type variable* α, $[\![\tau]\!][\alpha := [\![\sigma]\!]] = [\![\tau[\alpha := \sigma]]\!]$.

We now formally state the equivalence preservation property of type pre-representation and representation.

Theorem 5.3. $\tau \equiv \sigma$ *if and only if* $\overline{\tau} \equiv \overline{\sigma}$.

6. Representing Terms

In this section we describe our representation of F_ω terms. Our representations are typed to ensure that only well-typed terms can be represented. We typecheck representations of terms using type representations. In particular, a term of type τ is represented by a term of type $\mathsf{Exp}\ \overline{\tau}$.

Our representation is similar to those of Rendel et al. [20], and Brown and Palsberg [4]. We use Parametric Higher-Order Abstract Syntax (PHOAS) [9, 29]. As usual in Higher-Order Abstract Syntax (HOAS), we represent variables and abstractions meta-circularly, that is, as variables and abstractions. This avoids the need to implement capture-avoiding substitution on our operations – we inherit it from the host language implementation. In PHOAS representations, the types of variables are parametric. In our case, they are parametric in the type function F that defines an interpretation of types.

Our representation of F_ω terms is shown in Figure 3. We define our representation in two steps, as we did for types. The pre-representation of a term is defined using the designated variables F, abs, app, tabs, and tapp. The representation abstracts over these variables in the pre-representation.

While the pre-representation of types can be defined by the type alone, the pre-representation of a term depends on its typing judgment. We call the function that maps typing judgments to pre-representations the *pre-quoter*. We write

$\Gamma \vdash \mathsf{e} : \tau \triangleright \mathsf{q}$ to denote "given an input judgment $\Gamma \vdash \mathsf{e} : \tau$ the pre-quoter outputs a pre-representation q". The pre-representation of a term is defined by a type function F that defines pre-representations of types, and by four *case functions* that together define a fold over the structure of a term. The types of each case function depends on the type function F. The case functions are named abs, app, tabs, and tapp, and respectively represent λ-abstraction, function application, type-abstraction, and type application.

The representation $\overline{\mathsf{e}}$ of a closed term e is obtained by abstracting over the variables F, abs, app, tabs, and tapp in the pre-representation of e. If e has type τ, its pre-representation has type $\mathsf{F}\ [\![\tau]\!]$, and its representation has type $\mathsf{Exp}\ \overline{\tau}$. The choice of τ can be arbitrary because typings are unique up to β-equivalence and type representation preserves β-equivalence.

Stripping redundant quantifiers. In addition to the inst functions discussed in Section 4, our quoter embeds a specialized variant of instantiation functions into representations. These functions can strip redundant quantifiers, which would otherwise limit the expressiveness of our HOAS representation. For example, our size operation will use them to remove the redundant quantifier from intermediate values with types of the form $(\forall\alpha{:}\kappa.\mathsf{Nat})$. The type Nat is closed, so in particular α does not occur free in Nat. This is why the quantifier is said to be redundant. This problem of redundant quantifiers is well known, and applies to other HOAS representations than ours [20].

We can strip a redundant quantifier with a type application: if e has type $(\forall\alpha{:}\kappa.\mathsf{Nat})$ and σ is a type of kind κ, then $\mathsf{e}\ \sigma$ has the type Nat. We can also use the instantiation function $\mathsf{inst}_{(\forall\alpha:\kappa.\mathsf{Nat}),\sigma}$, which has type $(\forall\alpha{:}\kappa.\mathsf{Nat}) \to \mathsf{Nat}$. The choice of σ is arbitrary – it can be any type of kind κ. It happens that in F_ω all kinds are inhabited, which means we can always find an appropriate σ to strip a redundant quantifier.

Our quoter generates a single strip function for each type abstraction in a term and embeds it into the representation. At the time of quotation most quantifiers are not redundant – redundant quantifiers are introduced by certain operations like size. Whether a quantifier will become redundant depends on the result type function F for an operation. In our operations, redundant quantifiers are introduced when F is a constant function. The operation size has results typed using the constant Nat function $\mathsf{KNat} = (\lambda\alpha{:}*.\mathsf{Nat})$. Each strip function is general enough to work for multiple operations that introduce redundant quantifiers, and to still allow operations like unquote that need the quantifier.

To provide this generality, the strip functions take some additional inputs that help establish that a quantifier is redundant before stripping it. Each strip function will have a type of the form $\mathsf{Strip}\ \mathsf{F}\ [\![\forall\alpha{:}\kappa.\tau]\!] \equiv (\forall\beta{:}*.\ (\forall\gamma{:}*.\ \mathsf{F}\ \gamma \to \beta) \to [\![\forall\alpha{:}\kappa.\tau]\!] \to \beta)$. The type F is the result type function of an operation. The type $[\![\forall\alpha{:}\kappa.\tau]\!]$ is the quantified type with the redundant quantifier being stripped. Recall that $[\![\forall\alpha{:}\kappa.\tau]\!] = (\forall\alpha{:}\kappa.\mathsf{F}\ [\![\tau]\!])$. The type term of type $(\forall\gamma{:}*.\ \mathsf{F}\ \gamma \to \beta)$ shows that F is a constant function that always returns β. The strip function uses it to turn the type $(\forall\alpha{:}\kappa.\mathsf{F}\ [\![\tau]\!])$ into the type $(\forall\alpha{:}\kappa.\beta)$ where α has become redundant. For size, recall that $\mathsf{F} = \mathsf{KNat} = (\lambda\alpha{:}*.\mathsf{Nat})$. We show that KNat is the constant Nat function with an identity function $(\Lambda\gamma{:}*.\lambda\mathsf{x}{:}\mathsf{KNat}\ \gamma.\ \mathsf{x})$. The type of this function is $(\forall\gamma{:}*.\mathsf{KNat}\ \gamma \to \mathsf{KNat}\ \gamma)$, which is equivalent to $(\forall\gamma{:}*.\mathsf{KNat}\ \gamma \to \mathsf{Nat})$.

Types of case functions. The types of the four case functions that define an interpretation (Abs, App, TAbs, and TApp) are shown in Figure 3. The types of each function rely on invariants about pre-representations of types. For example, the type App F uses the fact that the pre-representation of an arrow type $[\![\tau_1 \to \tau_2]\!]$ is equal to $\mathsf{F}\ [\![\tau_1]\!] \to \mathsf{F}\ [\![\tau_2]\!]$. In other words, App F abstracts over the types $[\![\tau_1]\!]$ and $[\![\tau_2]\!]$ that can change, and makes explicit the structure $\mathsf{F}\ \alpha \to \mathsf{F}\ \beta$ that is invariant. These types allow the implementation of each case function to use this structure – it is part of the "interface" of representations, and plays an important role in the implementation of each operation.

Building representations. The first rule of pre-representation handles variables. As in our type representation, variables are represented meta-circularly, that is, by other variables. We will re-use the variable name, but change its type: a variable of type τ is represented by a variable of type $\mathsf{F}\ [\![\tau]\!]$. This type is the same as the type of a pre-representation. In other words, variables in a pre-representation abstract over pre-representations.

The second rule of pre-representation handles λ-abstractions. We recursively pre-quote the body, in which a variable x can occur free. Since variables are represented meta-circularly, x can occur free in the pre-representation q of the body. Therefore, we bind x in the pre-representation. This is standard for Higher-Order Abstract Syntax representations. Note that we change of the type of x from τ_1 to $\mathsf{F}\ [\![\tau_1]\!]$ in the representation. This is consistent with our earlier discussion about pre-representations of variables. It may be helpful to think of q as the "open pre-representation of e", in the sense that x can occur free, and to think of $(\lambda\mathsf{x}{:}\mathsf{F}\ [\![\tau_1]\!].\ \mathsf{q})$ as the "closed pre-representation of e". The open pre-representation of e has type $\mathsf{F}\ [\![\tau_2]\!]$ in an environment that assigns x the type $\mathsf{F}\ [\![\tau_1]\!]$. The closed pre-representation of e has type $\mathsf{F}\ [\![\tau_1]\!] \to \mathsf{F}\ [\![\tau_2]\!]$. The pre-representation of $(\lambda\mathsf{x}{:}\tau_1.\ \mathsf{e})$ is built by applying the case function abs to the types $[\![\tau_1]\!]$ and $[\![\tau_2]\!]$ and the closed pre-representation of e.

The third rule of pre-representation handles applications. We build the representation of an application $\mathsf{e}_1\ \mathsf{e}_2$ by applying the case function app to the types $[\![\tau_2]\!]$ and τ and the pre-representations of e_1 and e_2.

The fourth rule of pre-representation handles type abstractions. As for λ-abstractions, we call q the open pre-representation of e, and abstract over α to get the closed pre-representation of e. Unlike for λ-abstractions, we do not pass the domain and codomain of the type to the case function tabs, since that would require kind-polymorphism as discussed in Section 4. Instead, we pass to tabs the pre-representation of the quantified type directly. We also pass to tabs a quantifier stripping function that enables tabs to remove the quantifier from $[\![\forall\alpha{:}\kappa.\ \mathsf{F}\ \tau]\!]$ in case F is a constant function. Note that the strip function is always defined, since $[\![\forall\alpha{:}\kappa.\ \mathsf{F}\ \tau]\!] = \forall\alpha{:}\kappa.\mathsf{F}\ [\![\tau]\!]$.

The fifth rule of pre-quotation handles type applications. As for type abstractions, we don't decompose the quantified type $[\![\forall\alpha{:}\kappa.\tau]\!]$, but pass it to the case function tapp whole. We pre-represent the type argument σ, and construct an instantiation function $\mathsf{inst}_{([\![\forall\alpha:\kappa.\tau]\!],[\![\sigma]\!])}$, which can apply any term of type $[\![\forall\alpha{:}\kappa.\tau]\!]$ to the type $[\![\sigma]\!]$. Since $[\![\forall\alpha{:}\kappa.\tau]\!] = (\forall\alpha{:}\kappa.\mathsf{F}\ [\![\tau]\!])$, the instantiation function has type $[\![\forall\alpha{:}\kappa.\tau]\!] \to \mathsf{F}\ [\![\tau[\alpha{:=}\sigma]]\!]$.

The last rule of pre-quotation handles the type-conversion rule. Unsurprisingly, the pre-representation of e is the same when e has type σ as when it has type τ. When e has type τ, its pre-representation will have type $\mathsf{F}\ [\![\tau]\!]$. When e has type

σ, its pre-representation will have type F $[\![\sigma]\!]$. By Theorem 5.3, these two types are equivalent, so q can be given either type.

Examples. We now give two example representations. Our first example is the representation of the polymorphic identity function $\Lambda\alpha{:}*.\lambda x{:}\alpha.x$:

```
ΛF:* → *.
λabs:Abs F. λapp:App F.
λtabs:TAbs F. λtapp:TApp F.
tabs [[∀α:*. α → α]] strip_{F,[[∀α:*.α→α]]}
  (Λα:*. abs α α (λx:F α. x))
```

We begin by abstracting over the type function F that defines an interpretation of types, and the four case functions that define an interpretation of terms. Then we build the pre-representation of $\Lambda\alpha{:}*.\lambda x{:}\alpha.x$. We represent the type abstraction using tabs, the term abstraction using abs, and the variable x as another variable also named x.

Our second example is representation of $(\lambda x{:}(\forall\alpha{:}*.\ \alpha \to \alpha).\ x\ (\forall\alpha{:}*.\ \alpha \to \alpha)\ x)$, which applies an input term to itself.

```
ΛF:* → *.
λabs:Abs F. λapp:App F.
λtabs:TAbs F. λtapp:TApp F.
abs [[∀α:*. α → α]] [[∀α:*. α → α]]
 (λx: F [[∀α:*. α → α]].
  app [[∀α:*. α → α]] [[∀α:*. α → α]]
    (tapp [[∀α:*. α → α]] x
          [[(∀α:*. α → α) → (∀α:*. α → α)]]
          inst_{[[∀α:*.α→α]],[[∀α:*.α→α]]}))
```

The overall structure is similar to above: we begin with the five abstractions that define interpretations of types and terms. We then use the case functions to build the pre-representation of the term. The instantiation function $\mathtt{inst}_{[\![\forall\alpha:*.\alpha\to\alpha]\!],[\![\forall\alpha:*.\alpha\to\alpha]\!]}$ has the type $[\![\forall\alpha{:}*.\ \alpha \to \alpha]\!] \to [\![\forall\alpha{:}*.\ \alpha \to \alpha]\!] \to [\![\forall\alpha{:}*.\ \alpha \to \alpha]\!]$. Here, the quantified type being instantiated is $[\![\forall\alpha{:}*.\ \alpha \to \alpha]\!] = \forall\alpha{:}*.\ \mathtt{F}\ [\![\alpha \to \alpha]\!]$, the instantiation parameter is also $[\![\forall\alpha{:}*.\ \alpha \to \alpha]\!]$, and the instantiation type is $\mathtt{F}\ [\![(\forall\alpha{:}*.\ \alpha \to \alpha) \to (\forall\alpha{:}*.\ \alpha \to \alpha)]\!]$. By lemma 5.2, we have that $[\![\alpha \to \alpha]\!][\alpha := [\![\forall\alpha{:}*.\ \alpha \to \alpha]\!]] = [\![(\alpha \to \alpha)[\alpha := \forall\alpha{:}*.\ \alpha \to \alpha]]\!] = [\![(\forall\alpha{:}*.\ \alpha \to \alpha) \to (\forall\alpha{:}*.\ \alpha \to \alpha)]\!]$.

Properties. We typecheck pre-quotations under a modified environment that changes the types of term variables and binds the variables F, abs, app, tabs, and tapp. The bindings of type variables are unchanged.

The environment for pre-quotations of closed terms only contains bindings for F, abs, app, tabs, and tapp. The representation of a closed term abstracts over these variables, and so can be typed under an empty environment.

Theorem 6.1. *If* $\langle\rangle \vdash \mathtt{e} : \tau$, *then* $\langle\rangle \vdash \overline{\mathtt{e}} : \mathtt{Exp}\ \overline{\tau}$.

Our representations are *data*, which for F_ω means a β-normal form.

Theorem 6.2. *If* $\langle\rangle \vdash \mathtt{e} : \tau$, *then* $\overline{\mathtt{e}}$ *is* β*-normal.*

Our quoter preserves equality of terms up to equivalence of types. That is, if two terms are equal up to equivalence of types, then their representations are equal up to equivalence of types as well. Our quoter is also injective up to equivalence of types, so the converse is also true: if the representations of two terms are equal up to equivalence of types, then the terms are themselves equal up to equivalence of types.

Definition 6.1 (Equality up to equivalence of types). *We write* $\mathtt{e}_1 \sim \mathtt{e}_2$ *to denote that terms* $\mathtt{e}_1$ *and* $\mathtt{e}_2$ *are equal up to equivalence of types.*

$$\mathtt{x} \sim \mathtt{x}$$

$$\frac{\tau \equiv_\beta \tau' \qquad \mathtt{e} \sim \mathtt{e}'}{(\lambda x{:}\tau.\mathtt{e}) \sim (\lambda x{:}\tau'.\mathtt{e}')} \qquad \frac{\mathtt{e}_1 \sim \mathtt{e}_1' \qquad \mathtt{e}_2 \sim \mathtt{e}_2'}{(\mathtt{e}_1\ \mathtt{e}_2) \sim (\mathtt{e}_2'\ \mathtt{e}_2')}$$

$$\frac{\mathtt{e} \sim \mathtt{e}'}{(\Lambda\alpha{:}\kappa.\mathtt{e}) \sim (\Lambda\alpha{:}\kappa.\mathtt{e}')} \qquad \frac{\mathtt{e} \sim \mathtt{e}' \qquad \tau \equiv_\beta \tau'}{(\mathtt{e}\ \tau) \sim (\mathtt{e}'\ \tau')}$$

Now we can formally state that our quoter is injective and preserves equivalence.

Theorem 6.3. *If* $\langle\rangle \vdash \mathtt{e}_1 : \tau_1$, *and* $\langle\rangle \vdash \mathtt{e}_2 : \tau_2$, *then* $\mathtt{e}_1 \sim \mathtt{e}_2$, *if and only if* $\overline{\mathtt{e}_1} \sim \overline{\mathtt{e}_2}$.

7. Operations

Our suite of operations is given in Figure 4. It consists of a self-interpreter unquote, a continuation-passing-style transformation cps, a simple intensional predicate isAbs, a size measure size, and a normal-form checker nf. Our suite extends those of each previous work on typed self-representation[4, 20]. Rendel et al. define a self-interpreter and a size measure, while Brown and Palsberg define a self-interpreter, a CPS transformation, and the intensional predicate isAbs. Our normal-form checker is the first for a self-representation.

Each operation is defined using a function foldExp for programming folds. We also define encodings of booleans, pairs of booleans, and natural numbers that we use in our operations. We use a declaration syntax for types and terms. For example, the term declaration x : τ = e asserts that e has the type τ (i.e. $\langle\rangle \vdash \mathtt{e} : \tau$ is derivable), and substitutes e for x (essentially inlining x) in the subsequent declarations. We have machine checked the type of each declaration.

We give formal semantic correctness proofs for four of our operations: unquote, isAbs, size, and nf. The proofs demonstrate qualitatively that our representation is not only expressive but also easy to reason with. In the remainder of this section we briefly discuss the correctness theorems.

Each operation has a type of the form $\forall\alpha{:}\mathtt{U}.\ \mathtt{Exp}\ \alpha \to \mathtt{Op}\ \mathtt{R}\ \alpha$ for some type function R. When α is instantiated with a type representation $\overline{\tau}$, the result type Op R $\overline{\tau}$ is an interpretation under R:

Theorem 7.1. Op R $\overline{\tau} \equiv$ R ($[\![\tau]\!]$[F := R]).

Each operation is defined using the function foldExp that constructs a fold over term representations. An interpretation of a term is obtained by substituting the designated variables F, abs, app, tabs, and tapp with the case functions that define an operation. The following theorem states that a fold constructed by foldExp maps representations to interpretations:

Theorem 7.2. *If* f = foldExp R abs′ app′ tabs′ tapp′, *and* $\Gamma \vdash \mathtt{e} : \tau \rhd \mathtt{q}$, *then* f $\overline{\tau}$ $\overline{\mathtt{e}}$. $\longrightarrow^*$ (q[F:=R, abs:=abs′, app:=app′, tabs:=tabs′, tapp:=tapp′]).

unquote. Our first operation on term representations is our self-interpreter unquote, which recovers a term from its representation. Its results have types of the form Op Id

```
Bool  : *    = ∀α:*. α → α → α
true  : Bool = Λα:*. λt:α. λf:α. t
false : Bool = Λα:*. λt:α. λf:α. f
and   : Bool → Bool → Bool =
  λb1:Bool. λb2:Bool. Λα:*. λt:α. λf:α.
  b1 α (b2 α t f) f

Bools : * = ∀α:*. (Bool → Bool → α) → α
bools : Bool → Bool → Bools =
  λb1:Bool. λb2:Bool.
  Λα:*. λf:Bool → Bool → α. f b1 b2
fst : Bools → Bool =
  λbs:Bools. bs Bool (λb1:Bool. λb2:Bool. b1)
snd : Bools → Bool =
  λbs:Bools. bs Bool (λb1:Bool. λb2:Bool. b2)

Nat  : *   = ∀α:*. α → (α → α) → α
zero : Nat = Λα:*. λz:α. λs:α → α. z
succ : Nat → Nat =
  λn:Nat. Λα:*. λz:α. λs:α → α. s (n α z s)
plus : Nat → Nat → Nat =
  λm:Nat. λn:Nat. m Nat n succ
```

Definitions and operations of Bool, Bools, and Nat.

```
foldExp : (∀F:* → *.
           Abs F → App F → TAbs F → TApp F →
           ∀α:U. Exp α → Op F α) =
  ΛF:* → *.
  λabs  : Abs F.  λapp  : App F.
  λtabs : TAbs F. λtapp : TApp F.
  Λα:U. λe:Exp α. e F abs app tabs tapp
```

Implementation of foldExp

```
Id  : * → * = λα:*.α

unAbs : Abs Id = Λα:*.Λβ:*.λf:α→β.f
unApp : App Id = Λα:*.Λβ:*.λf:α→β.λx:α.f x
unTAbs : TAbs Id = Λα:*.λs:Strip Id α.λf:α.f
unTApp : TApp Id = Λα:*.λf:α.Λβ:*.λg:α→β.g f

unquote : (∀α:U. Exp α → Op Id α) =
  foldExp Id unAbs unApp unTAbs unTApp
```

Implementation of unquote

```
KBool : * → * = λα:*. Bool

isAbsAbs : Abs KBool =
  Λα:*. Λβ:*. λf:Bool → Bool. true
isAbsApp : App KBool =
  Λα:*. Λβ:*. λf:Bool. λx:Bool. false
isAbsTAbs : TAbs KBool =
  Λα:*. λstrip:Strip KBool α. λf:α. true
isAbsTApp : TApp KBool =
  Λα:*. λf:Bool. Λβ:*. λinst:α → Bool. false

isAbs : (∀α:U. Exp α → Bool) =
  foldExp KBool isAbsAbs isAbsApp
           isAbsTAbs isAbsTApp
```

Implementation of isAbs.

```
Ct  : * → * = λα:*. ∀β:*. (α → β) → β
CPS : U → * = Op Ct

cpsAbs : Abs Ct =
  Λα:*. Λβ:*. λf:(Ct α → Ct β).
  ΛV:*. λk : (Ct α → Ct β) → V.
  k f
cpsApp : App Ct =
  Λα:*. Λβ:*. λf:Ct (Ct α → Ct β). λx:Ct α.
  ΛV:*. λk:β → V.
  f V (λg:Ct α → Ct β. g x V k)
cpsTAbs : TAbs Ct =
  Λα:*. λstrip:Strip Ct α. λf: α.
  ΛV:*. λk:α → V.
  k f
cpsTApp : TApp Ct =
  Λα:*. λf: Ct α.
  Λβ:*. λinst:α → Ct β.
  ΛV:*. λk:β → V.
  f V (λe:α. inst e V k)

cps : (∀α:U. Exp α → CPS α) =
  foldExp Ct cpsAbs cpsApp cpsTAbs cpsTApp
```

Implementation of cps.

```
KNat : * → * = λα:*. Nat

sizeAbs : Abs KNat =
  Λα:*. Λβ:*. λf:Nat → Nat. succ (f (succ zero))
sizeApp : App KNat =
  Λα:*. Λβ:*. λf:Nat. λx:Nat. succ (plus f x)
sizeTAbs : TAbs KNat =
  Λα:*. λstrip:Strip KNat α. λf:α.
  succ (strip Nat (Λα:*. λx:Nat. x) f)
sizeTApp : TApp KNat =
  Λα:*. λf : Nat. Λβ:*. λinst:α → Nat. succ f

size : (∀α:U. Exp α → Nat) =
  foldExp KNat sizeAbs sizeApp sizeTAbs sizeTApp
```

Implementation of size.

```
KBools : * → * = λα:*. Bools

nfAbs : Abs KBools =
  Λα:*. Λβ:*. λf:Bools → Bools.
  bools (fst (f (bools true true))) false
nfApp : App KBools =
  Λα:*. Λβ:*. λf:Bools. λx:Bools.
  bools (and (snd f) (fst x)) (and (snd f) (fst x))
nfTAbs : TAbs KBools =
  Λα:*. λstrip:Strip KBools α. λf:α.
  bools (fst (strip Bools (Λα:*.λx:Bools.x) f))
        false
nfTApp : TApp KBools =
  Λα:*. λf:Bools. Λβ:*. λinst:(α → Bools).
  bools (snd f) (snd f)

nf : (∀α:U. Exp α → Bool) =
  Λα:U. λe:Exp α.
  fst (foldExp KBools nfAbs nfApp nfTAbs nfTApp e)
```

Implementation of nf.

Figure 4: Five operations on representations of F_ω terms.

$\overline{\tau}$. The type function `Id` is the identity function, and the operation `Op Id` recovers a type from its representation.

Theorem 7.3. *If* $\Gamma \vdash \tau : *$, *then* `Op Id` $\overline{\tau} \equiv \tau$.

Theorem 7.4. *If* $\langle\rangle \vdash$ `e` $: \tau$, *then* `unquote` $\overline{\tau}\ \overline{\texttt{e}} \longrightarrow^*$ `e`.

isAbs. Our second operation `isAbs` is a simple intensional predicate that checks whether its input represents an abstraction or an application. It returns a boolean on all inputs. Its result types are interpretations under `KBool`, the constant `Bool` function. The interpretation of any type under `KBool` is equivalent to `Bool`:

Theorem 7.5. *If* $\Gamma \vdash \tau : *$, *then* `Op KBool` $\overline{\tau} \equiv$ `Bool`.

Theorem 7.6. *Suppose* $\langle\rangle \vdash$ `e` $: \tau$. *If* `e` *is an abstraction then* `isAbs` $\overline{\tau}\ \overline{\texttt{e}} \longrightarrow^*$`true`. *Otherwise* `e` *is an application and* `isAbs` $\overline{\tau}\ \overline{\texttt{e}} \longrightarrow^*$`false`.

cps. Our third operation `cps` is a call-by-name continuation-passing-style transformation. Its result types are interpretations under `Ct`. We have also implemented a call-by-value CPS transformation, though we omit the details because it is rather similar to our call-by-name CPS. We do not formally prove the correctness of our CPS transformation. However, being defined in F_ω it is guaranteed to terminate for all inputs, and the types of the case functions provide some confidence in its correctness.

size. Our fourth operation `size` measures the size of its input representation. Its result types are interpretations under `KNat`, the constant `Nat` function. The interpretation of any type under `KNat` is equivalent to `Nat`:

Theorem 7.7. *If* $\Gamma \vdash \tau : *$, *then* `Op KNat` $\overline{\tau} \equiv$ `Nat`.

The size of a term excludes the types. We formally define the size of a term in order to state the correctness of `size`.

Definition 7.1. *The size of a term* `e`, *denoted* `|e|`, *is defined as:*

$$\begin{array}{ll} |\texttt{x}| & = 1 \\ |\lambda \texttt{x}{:}\tau.\texttt{e}| & = 1 + |\texttt{e}| \\ |\texttt{e}_1\ \texttt{e}_2| & = 1 + |\texttt{e}_1| + |\texttt{e}_2| \\ |\Lambda\alpha{:}\kappa.\texttt{e}| & = 1 + |\texttt{e}| \\ |\texttt{e}\ \tau| & = 1 + |\texttt{e}| \end{array}$$

The results of `size` are Church encodings of natural numbers. We define a type `Nat` and a `zero` element and a successor function `succ`. We use the notation $\texttt{church}_n$ to denote the Church-encoding of the natural number n. For example, $\texttt{church}_0$ `= zero`, $\texttt{church}_1$ `= succ zero`, $\texttt{church}_2$ `= succ (succ zero)`, and so on.

Theorem 7.8. *If* $\langle\rangle \vdash$ `e` $: \tau$ *and* `|e|`$=n$, *then* `size` $\overline{\tau}\ \overline{\texttt{e}} \longrightarrow^* \texttt{church}_n$

nf. Our fifth operation checks whether its input term is in β-normal form. Its results have types that are interpretations under `KBools`, the constant `Bools` function, where `Bools` is the type of pairs of boolean values.

Theorem 7.9. *If* $\Gamma \vdash \tau : *$, *then* `Op KBools` $\overline{\tau} \equiv$ `Bools`.

We program `nf` in two steps: first, we compute a pair of booleans by folding over the input term. Then we return the first component of the pair. The first boolean encodes whether a term is β-normal. The second encodes whether a term is normal and *neutral*. Intuitively, a neutral term is one that can be used in function position of an application without introducing a redex. We provide a formal definition of normal and neutral in the Appendix.

Theorem 7.10. *Suppose* $\langle\rangle \vdash$ `e` $: \tau$.

1. *If* `e` *is* β*-normal, then* `nf` $\overline{\tau}\ \overline{\texttt{e}} \longrightarrow^*$`true`.
2. *If* `e` *is not* β*-normal, then* `nf` $\overline{\tau}\ \overline{\texttt{e}} \longrightarrow^*$`false`.

8. Experiments

We have validated our techniques using an implementation of F_ω in Haskell, consisting of a parser, type checker, evaluator, β-equivalence checker, and our quoter. Each operation has been programmed, type checked, and tested. We have also confirmed that the representation of each operation type checks with the expected type.

Each of our operations are self-applicable, meaning it can be applied to a representation of itself. We have checked that the self-application of each operation type checks with the expected type. Further, we have checked that the self-application of `unquote` is β-equivalent to itself:

$$\begin{array}{ll} & \texttt{unquote}\ \overline{(\forall\alpha{:}\texttt{U}.\ \texttt{Exp}\ \alpha \to \texttt{Op Id}\ \alpha)}\ \overline{\texttt{unquote}} \\ \equiv_\beta & \texttt{unquote} \end{array}$$

We plan to submit our implementation for artifact evaluation.

9. Related Work

Typed Self-Interpretation. Pfenning and Lee [18] studied self-interpretation of Systems F and F_ω. They concluded that it seemed to be impossible for each language, and defined representations and self-interpreters of System F in F_ω and F_ω in F_ω^+. They used the intensional approach discussed in Section 4, and did not consider our extensional approach.

Rendel, et al. [20] presented the first typed self-representation and self-interpreter. Their language System F_ω^* extends F_ω with a `Type:Type` rule that unifies the levels of types and kinds. As a result, F_ω^* is not strongly-normalizing, and type checking is undecidable. They used the intensional approach to representing polymorphism. They implemented `unquote` and `size` operations. Their implementation of `size` relied on a special $\bot$ type to strip redundant quantifiers. The type $\bot$ inhabits every kind, but is not used to type check terms. We strip redundant quantifiers using special instantiation functions that are generated by the quoter.

Jay and Palsberg [13] presented a typed self-representation and self-interpreter for a combinator calculus, with a λ-calculus surface syntax. Their calculus had undecidable type checking and was not strongly normalizing.

Brown and Palsberg [4] presented a typed self-representation for System U, which is not strongly normalizing but does have decidable type checking. This was the first self-representation for a language with decidable type checking. They implemented `unquote`, `isAbs`, and `cps` operations. They also represented types, though they only represented types of kind $*$ and did not have a substitution theorem like our Theorem 5.2. They instead used a kind of coercion to change the type of a representation after a type application, which ensured that type was properly represented. Our type representation is designed to avoid the need for such coercions, which simplifies our representation and the proofs of our theorems.

Typed Meta-Programming. Typed self-interpretation is a particular instance of typed meta-programming, which involves a typed representation of one language in a possibly

different language, and operations on that representation. Typed meta-programming has been studied extensively, and continues to be an active research area. Chen and Xi [7, 8] demonstrated that types can make meta-programming less error-prone.

Carette et al. [5] introduced tagless representations, which are more efficient than other techniques and use simpler types. Our representation is also tagless, though we use ordinary λ-abstractions to abstract over the case functions of an operation, while they use Haskell type classes or OCaml modules. The object languages they represented did not include polymorphism. Our extensional technique could be used to program tagless representations of polymorphic languages in Haskell or OCaml.

MetaML [24] supports *generative* typed meta-programming for multi-stage programming. It includes a built-in unquoter, while we program `unquote` as a typed F_ω term.

Trifonov et al. [25] define a language with fully reflexive intensional type analysis, which supports type-safe runtime type introspection. Instead of building representations of types, their language includes special operators to support iterating over types. They programmed generic programs like marshalling values for transmission over a network. Generic programming and meta-programming are different techniques: generic programs operate on programs or program values, and meta-programs operate on representations of programs. These differences mean that each technique is better suited to some problems than other.

Dependently-Typed Representation. Some typed representations use dependent types to ensure that only well-typed terms can be represented. For example, Harper and Licata [12] represented simply-typed λ-calculus in LF, and Schürmann et al. [21] represented F_ω in LF. Chapman [6] presented a meta-circular representation of a dependent type theory in Agda. These representations are quite useful for *mechanized metatheory* – machine-checked proofs of the metatheorems for the represented language. The demands of mechanized metatheory appear to be rather different from those of self-interpretation. It is an open question whether a dependently-typed self-representation can support a self-interpreter.

Untyped Representation. The literature contains many examples of untyped representations for typed languages, including for Coq [3] and Haskell [17]. Untyped representations generally use a single type like `Exp` to type check all representations, and permit ill-typed terms to be represented. Template Haskell [22] uses an untyped representation and supports user-defined operations on representations. Since representations are not guaranteed to be well-typed by construction, generated code needs to be type checked.

Coercions. Our instantiation functions are similar to coercions or *retyping functions*: they change the type of a term without affecting its behavior. Cretin and Rémy [10] studied erasable coercions for System F_η [15], including coercions that perform instantiations. We conjecture that our self-representation technique would work for an extension of F_ω with erasable coercions for instantiations, and that erasable coercions could replace instantiation functions in our extensional approach to representing polymorphism.

10. Conclusion

Our self-interpreter for System F_ω opens to door to self-representations and self-interpreters for other strongly normalizing languages. For example, it might be possible to use kind-instantiation functions to define a deep self-representation of F^+_ω. Similarly, universe-instantiation functions might enable self-representation for languages like Coq that include an infinite hierarchy of universes and universe polymorphism.

We have solved two open problems posed by Pfenning and Lee: First, we define a shallow self-representation technique that supports self-interpretation for each of System F and System F_ω. Second, we define a deep self-representation for System F_ω that supports a variety of operations including a self-interpreter. It is still an open question whether System F can support a deep self-representation.

Our techniques create new opportunities for type-checking self-applicable metaprograms, with potential applications in typed macro systems, partial evaluators, compilers, and theorem provers.

References

[1] Henk Barendregt. Self-interpretations in lambda calculus. *J. Funct. Program*, 1(2):229–233, 1991.

[2] HP Barendregt. *Handbook of Logic in Computer Science (vol. 2): Background: Computational Structures: Abramski, S. (ed)*, chapter Lambda Calculi with Types. Oxford University Press, Inc., New York, NY, 1993.

[3] Bruno Barras and Benjamin Werner. Coq in coq. Technical report, 1997.

[4] Matt Brown and Jens Palsberg. Self-Representation in Girard's System U. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 471–484, New York, NY, USA, 2015. ACM.

[5] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

[6] James Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009.

[7] Chiyan Chen and Hongwei Xi. Meta-Programming through Typeful Code Representation. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 275–286, Uppsala, Sweden, August 2003.

[8] Chiyan Chen and Hongwei Xi. Meta-Programming through Typeful Code Representation. *Journal of Functional Programming*, 15(6):797–835, 2005.

[9] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. ACM.

[10] Julien Cretin and Didier Rémy. On the power of coercion abstraction. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 361–372, New York, NY, USA, 2012. ACM.

[11] N. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

[12] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, July 2007.

[13] Barry Jay and Jens Palsberg. Typed self-interpretation by pattern matching. In *Proceedings of ICFP'11, ACM SIGPLAN International Conference on Functional Programming*, pages 247–258, Tokyo, September 2011.

[14] Stephen C. Kleene. λ-definability and recursiveness. *Duke Math. J.*, pages 340–353, 1936.

[15] John C. Mitchell. Polymorphic type inference and containment. *Inf. Comput.*, 76(2-3):211–249, February 1988.

[16] Greg Morrisett. F-omega – the workhorse of modern compilers. http://www.eecs.harvard.edu/ greg/cs256sp2005/lec16.txt, 2005.

[17] Matthew Naylor. Evaluating Haskell in Haskell. *The Monad.Reader*, 10:25–33, 2008.

[18] Frank Pfenning and Peter Lee. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science*, 89(1):137–159, 1991.

[19] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[20] Tillmann Rendel, Klaus Ostermann, and Christian Hofer. Typed self-representation. In *Proceedings of PLDI'09, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–303, June 2009.

[21] Carsten Schürmann, Dachuan Yu, and Zhaozhong Ni. A representation of f_ω in lf. *Electronic Notes in Theoretical Computer Science*, 58(1):79 - 96, 2001.

[22] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.

[23] T. Stuart. *Understanding Computation: Impossible Code and the Meaning of Programs.* Understanding Computation. O'Reilly Media, Incorporated, 2013.

[24] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. In *Theoretical Computer Science*, pages 203–217. ACM Press, 1999.

[25] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. *SIGPLAN Not.*, 35(9):82–93, September 2000.

[26] David Turner. Total functional programming. *Journal of Universal Computer Science*, 10:187–209, 2004.

[27] N.K. Vereshchagin and A. Shen. *Computable Functions.* Student mathematical library. American Mathematical Society, 2003.

[28] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.

[29] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 249–262, New York, NY, USA, 2003. ACM.