Typed Self-Evaluation via Intensional Type Functions

Matt Brown Jens Palsberg

University of California at Los Angeles, USA

msb@cs.ucla.edu palsberg@ucla.edu

Abstract

Many popular languages have a self-interpreter, that is, an interpreter for the language written in itself. So far, work on polymorphically-typed self-interpreters has concentrated on self-recognizers that merely recover a program from its representation. A larger and until now unsolved challenge is to implement a polymorphically-typed self-evaluator that evaluates the represented program and produces a representation of the result. We present F_{ω}^{μ} , the first λ -calculus that supports a polymorphically-typed self-evaluator. Our calculus extends F_{ω} with recursive types and intensional type functions and has decidable type checking. Our key innovation is a novel implementation of type equality proofs that enables us to define a versatile representation of programs. Our results establish a new category of languages that can support polymorphically-typed self-evaluators.

Categories and Subject Descriptors D.3.4 [Processors]: Interpreters; F.3.3 [Studies of Program Constructs]: Type structure

General Terms Languages; Theory

Keywords Lambda Calculus; Self Representation; Self Interpretation; Self Evaluation; Meta Programming; Type Equality

1. Introduction

Many popular languages have a self-interpreter, that is, an interpreter for the language written in *itself*; examples include Haskell [26], JavaScript [17], Python [32], Ruby [44], Scheme [3], and Standard ML [33]. The use of *itself* as implementation language is cool, demonstrates expressiveness, and has key advantages. In particular, a self-interpreter enables the language designer to easily modify, extend, and grow the language [31], and do other forms of meta-programming [6].

What is the type of an interpreter that can interpret a representation of itself? The classical answer to such questions is to work with a single type for all program representations. For example, the single type could be `String` or it could be `Syntax Tree`. The single-type approach enables an interpreter to have type, say, $(\text{String} \rightarrow \text{String})$, where the input string represents a program and where the output string represents the result. However, this approach ignores that the source program type checks, and gives no guarantee that the interpreter preserves the type of its input.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

POPL'17, January 15–21, 2017, Paris, France
ACM. 978-1-4503-4660-3/17/01...\$15.00
http://dx.doi.org/10.1145/3009837.3009853

Figure 1: Self-recognizers and self-evaluators.

How can we do better type checking of self-interpreters? First, suppose we have a better representation scheme $quote(\cdot)$ and a type function Exp such that if $e : T$, then $quote(e) : Exp\ T$. This enables us to consider two polymorphic types of self-interpreters:

(self-recognizer) $unquote : \forall T. Exp\ T \rightarrow T$ (1)

(self-evaluator) $eval : \forall T. Exp\ T \rightarrow Exp\ T$ (2)

The functionality of a self-recognizer $unquote$ is to recover a program from its representation, while the functionality of a self-evaluator $eval$ is to evaluate the represented program and produce a representation of the result. The relationship between a self-recognizer and a self-evaluator is illustrated in Figure 1. The meta-level function $quote$ maps a term e to its representation $\bar{e}$. A meta-level *evaluation* function maps e to a value v . A self-recognizer $unquote$ inverts $quote$, while a self-evaluator $eval$ implements *evaluation* on representations. There can be multiple evaluation functions and self-evaluators for a particular language, implementing different evaluation strategies. The thinner arrows indicate mappings up to equivalence: the application of $unquote$ to $\bar{e}$ is *equivalent* to e , but is not *identical* to e .

There are several examples of self-recognizers with type (1) in the literature. Specifically, Rendel, Ostermann, and Hofer [31] presented the *first* self-recognizer with type (1) for the typed λ -calculus F_{ω}^* . In previous work we presented self-recognizers with type (1) for System U [7], a typed λ -calculus with *decidable* type checking, and for F_{ω} [8], a *strongly normalizing* language.

Implementing a self-evaluator with type (2) has remained an open problem until now. Our goal is to identify a core calculus for which we can solve the problem.

The challenge: Can we define a self-evaluator with type (2) for a typed λ -calculus?

Our result: Yes, we present three self-evaluators for a typed λ -calculus with decidable type checking. Our calculus, F_{ω}^{μ} , extends F_{ω} with recursive types and intensional type functions.

Our starting point is an evaluator for simply-typed λ -calculus (STLC) written in Haskell. The evaluator has type (2) and operates on a representation of STLC based on generalized algebraic data types (GADTs). The gap between the meta-language (Haskell)

and the object-language (STLC) is large. To reduce this gap, we apply a series of translations to reduce our GADT-based evaluator of STLC to lower-level constructs: higher-order polymorphism, recursive types, and a theory of type equality. We close the gap in $F_{\omega}^{\mu i}$, which is designed to support these constructs.

The key challenge of self-representation – “tying the knot” – is to balance the competing needs for a single language to be simultaneously the object language and the meta-language. A more powerful language can represent more, but also has more that needs to be represented. Previous work on self-representation has focused on tying the knot as it pertains to polymorphism [7, 8, 31]. A similar challenge arises for type equality, and this is our main focus in this paper.

To tie the knot for a language with type equality, we need to consider two questions. First, how expressive must a theory of type equality be in order to implement a typed evaluator for a particular object language? Second, what meta-language features are needed to represent and evaluate a particular theory of type equality? In Section 2 we show that to evaluate STLC, type equality between arrow types should be decomposable. In particular, if we know $(A \rightarrow B) = (S \rightarrow T)$, then we also know $A = S$ and $B = T$. What then is needed to represent and evaluate decomposable type equalities? Haskell implements type equality using built-in type equality coercions [36]. These support decomposition, but have complex typing rules and evaluation semantics that make representation and evaluation difficult. On the other hand, Leibniz equality proofs [5, 28, 39] can be encoded in λ -terms typeable in pure F_{ω} . This means that representing and evaluating Leibniz equality proofs is no harder than representing and evaluating F_{ω} . However, Leibniz equality proofs are not decomposable in F_{ω} . Our goal is to implement a theory of type equality that is decomposable like Haskell’s type equality coercions, but that is also easily represented and evaluated, like Leibniz equality proofs.

We achieve our goal by implementing type equality in a new way, by combining Leibniz equality proofs with *intensional type functions* that can depend on the intensional structure of their inputs. The result is an expressive theory of type equality with a simple semantics. This innovation is the key to defining our typed self-representation and self-evaluators.

Our intensional type functions are defined using a Typecase operator that is inspired by previous work on intensional type analysis (ITA) [13, 21, 34, 37, 42], but is simpler in three ways:

- We support ITA at the type level only, while previous work supports ITA in types and terms.
- Our Typecase operator is not recursive. Previous work used a recursive Typerec operator for type-level ITA.
- We support ITA of quantified types without using kind polymorphism.

We present a self-representation of $F_{\omega}^{\mu i}$ and three self-evaluators with type (2) that operate upon it: one that evaluates terms to weak head normal form, one that performs a single step of left-most reduction, and an implementation of Normalization by Evaluation (NbE) that reduces to β -normal form. The first only reduces closed terms, while the others may reduce under abstractions. We also implement a self-recognizer unquote with type (1), and all the benchmark meta-programs from our previous work on typed self-representation [8]. We have proved that the weak head self-evaluator is correct, and we have implemented and tested our other self-evaluators and meta-programs. Available from our website [1] are the implementations of $F_{\omega}^{\mu i}$ and our meta-programs, as well as an appendix containing proofs of the theorems stated in this paper.

```
data Exp t where
  Abs :: (Exp t1 → Exp t2) → Exp (t1 → t2)
  App :: Exp (t1 → t2) → Exp t1 → Exp t2

eval :: Exp t → Exp t
eval (App e1 e2) =
  let e1' = eval e1 in
  case e1' of
    Abs f → eval (f e2)
    _     → App e1' e2
eval e = e
```

Figure 2: A typed representation of STLC using Haskell GADTs

Rest of the paper. In Section 2 we show how type equality proofs can be used to implement a typed evaluator for STLC in Haskell. In Section 3 we define our calculus $F_{\omega}^{\mu i}$. In Section 4 we first implement type equality proofs for simple types in $F_{\omega}^{\mu i}$ and use them to program a typed STLC evaluator. Then we move beyond simple types and extend our type equality proofs to work with quantified and recursive types. In Section 5 we define our self-representation, in Section 6 we present our self-evaluators, in Section 7 we describe our other benchmark meta-programs and our experiments, and in Section 8 we discuss related work.

2. From GADTs to Type Equality Proofs

In this section, we will show a series of four evaluators for STLC, all written in Haskell. The idea is for each version to use lower-level constructs than the previous ones, and to use constructs with F_{ω} types as much as possible. Along the way, we will highlight the techniques needed to typecheck the evaluators.

GADTs. Figure 2 shows a representation of Simply-Typed λ -Calculus (STLC) terms in Haskell using GADTs. The representation is Higher-Order Abstract Syntax (HOAS), which means that STLC variables are represented as Haskell variables that range over representations, and we use Haskell functions to bind STLC variables. In the Abs constructor, the function type $(\text{Exp } t1 \rightarrow \text{Exp } t2)$ corresponds to a STLC term of type $\text{Exp } t2$ that includes a free variable of type $\text{Exp } t1$.

Also in Figure 2 is a meta-circular evaluator with type (2). That type guarantees that eval preserves the type of its input – that the result has the same type. It is meta-circular because it implements STLC features using the corresponding features in the meta-language (Haskell). In particular, we use Haskell β -reduction (function application) to implement STLC β -reduction.

The evaluator eval implements weak head-normal evaluation. This means that it reduces the left-most β -redex, but does not evaluate under λ -abstractions or in the argument position of applications. If $e = \text{Abs } f$, then e is already in weak head-normal form, and $\text{eval } e = e$. If $e = \text{App } e1 \ e2$, we first recursively evaluate $e1$, letting $e1'$ be value of $e1$. If $e1'$ is an abstraction $\text{Abs } f$, then $\text{App } e1' \ e2$ is a redex. We reduce it by applying f to $e2$, and then we recursively evaluate the result. If $e1'$ is not an abstraction, then we return $\text{App } e1' \ e2$.

We now consider how Haskell type checks eval. First, the type annotation on eval determines that $\text{App } e1 \ e2$ has type $\text{Exp } t$. According to the type of App, $e1$ has type $\text{Exp } (t1 \rightarrow t)$ and $e2$ has type $\text{Exp } t1$, for some type $t1$. Since eval preserves the type of its

```

data Exp t =
  forall t1 t2. (t1 → t2) ~ t ⇒ Abs (Exp t1 → Exp t2)
  | forall t1. App (Exp (t1 → t)) (Exp t1)

eval :: Exp t → Exp t
eval (App e1 e2) =
  let e1' = eval e1 in
  case e1' of
    Abs f → eval (f e2)
    _     → App e1' e2
eval e = e

```

Figure 3: STLC using ADTs and equality coercions

```

refl  :: Eq t t
sym   :: Eq t1 t2 → Eq t2 t1
trans :: Eq t1 t2 → Eq t2 t3 → Eq t1 t3
eqApp :: Eq t1 t2 → Eq (f t1) (f t2)
arrL  :: Eq (t1 → t2) (s1 → s2) → Eq t1 s1
arrR  :: Eq (t1 → t2) (s1 → s2) → Eq t2 s2

coerce :: Eq t1 t2 → t1 → t2

```

Figure 4: Interface of explicit type equality proofs

argument, $e1'$ also has type $\text{Exp } (t1 \rightarrow t)$. If case analysis finds that $e1'$ is of the form $\text{Abs } f$, then the type of Abs tells us that f has the type $\text{Exp } t1 \rightarrow \text{Exp } t$.

We can see that Haskell’s type checker does some nontrivial work to typecheck code with GADTs. Pattern matching $\text{App } e1\ e2$ introduced the existentially quantified type $t1$. When pattern matching determined that $e1'$ is of the form $\text{Abs } f$, the type checker aligned the types of $e1'$ and f so that f could be applied to $e2$.

ADTs and equality constraints. GADTs can be understood and implemented as a combination of algebraic data types (ADTs) and equality between types. Figure 3 reimplements STLC in this style, using ADTs and Haskell’s type equality constraints. In this version, the result type of each constructor of Exp is implicitly $\text{Exp } t$, while in the GADT version the result type of Abs is $\text{Exp } (t1 \rightarrow t2)$. The type equality constraint $(t1 \rightarrow t2) \sim t$ makes up this difference. Haskell implements GADTs using ADTs and equality constraints [36], so the definitions of $\text{Exp } t$ in Figures 2 and 3 are effectively the same. In particular, in both versions the constructors Abs and App have the same types, and the implementation of eval is the same.

Haskell’s type equality coercions are reflexive, symmetric, and transitive, and support a number of other rules for deriving equalities. The type checker automatically derives new equalities based on existing ones and inserts coercions based on known equalities. This is how it is able to typecheck eval . We refer the interested reader to Sulzmann et al. [36].

Explicit type equality proofs. Figure 4 defines an explicit theory of type equality that allows us to derive type equalities and perform coercions manually. We can implement the functions in Figure 4 using Haskell, as we show in the appendix, or we can implement them in $F_{\omega}^{\mu i}$, as we show in Section 4.

The basic properties of type equality, namely reflexivity, symmetry, and transitivity, are encoded by refl , sym , and trans , respectively. The only way to introduce a new type equality proof is by using refl . eqApp shows that equal types are equal in any context. For example, given an equality proof of type $\text{Eq } t1\ t2$, eqApp can derive a proof that $\text{Exp } t1$ is equal to $\text{Exp } t2$ by instantiating f with Exp . The operators arrL and arrR allow type equality proofs

```

data Exp t =
  forall t1 t2. Abs (Eq (t1 → t2) t) (Exp t1 → Exp t2)
  | forall t1. App (Exp (t1 → t)) (Exp t1)

eval :: Exp t → Exp t
eval (App e1 e2) =
  let e1' = eval e1 in
  case e1' of
    Abs eq f →
      let eqL = eqApp (sym (arrL eq))
          eqR = eqApp (arrR eq)
          f' = coerce eqR . f . coerce eqL
      in eval (f' e2)
      → App e1' e2
eval e = e

```

Figure 5: STLC using explicit type equality proofs

about arrow types to be decomposed into proofs about the domain and codomain types, respectively. We have highlighted them to emphasize their importance in type checking eval and in motivating the design of $F_{\omega}^{\mu i}$. Given a proof of $\text{Eq } t1\ t2$, coerce can change the type of a term from $t1$ into $t2$.

Given a closed proof p of type $\text{Eq } t1\ t2$, we expect (1) that it is true that $t1$ and $t2$ are equal types, and (2) that $\text{coerce } p\ e$ evaluates to e for all e . Open proofs include variables of equality proof type, which can be thought of as type equality hypotheses. Until these hypotheses are discharged, $\text{coerce } p\ e$ should not be reducible to e .

ADTs and explicit type equality proofs. Figure 5 shows a version of $\text{Exp } t$ and an evaluator that uses ADTs and explicit type equality proofs. The only difference between this definition of $\text{Exp } t$ and the one in Figure 3 is that we have replaced the type equality constraint $(t1 \rightarrow t2) \sim t$ with a type equality proof of type $\text{Eq } (t1 \rightarrow t2)\ t^1$, in order to clarify the role of type equality in type checking eval .

As before, we know from the type of eval that its argument has type $\text{Exp } t$, and $e1$ has type $\text{Exp } (t1 \rightarrow t)$ and $e2$ has type $\text{Exp } t1$, for some type $t1$. Since eval preserves type, $e1'$ also has type $\text{Exp } (t1 \rightarrow t)$.

The differences begin with the pattern match on $e1'$. If $e1'$ is of the form $\text{Abs } eq\ f$, then there exist types $s1$ and $s2$ such that eq has the type $\text{Eq } (s1 \rightarrow s2)\ (t1 \rightarrow t)$ and f has the type $\text{Exp } s1 \rightarrow \text{Exp } s2$. We use arrL , sym , and eqApp (with f instantiated with Exp) to derive eqL , which has the type $\text{Eq } (\text{Exp } t1)\ (\text{Exp } s1)$. Similarly, we use arrR and eqApp to derive eqR with the type $\text{Eq } (\text{Exp } s2)\ (\text{Exp } t)$. Finally, we use coercions based on eqL and eqR to cast f from the type $\text{Exp } s1 \rightarrow \text{Exp } s2$ to the type $\text{Exp } t1 \rightarrow \text{Exp } t$. Thus, f' can be applied to $e2$, and its result has type $\text{Exp } t$, as required by the type of eval .

Mogensen-Scott encoding. By using a typed Mogensen-Scott encoding [24], we can represent STLC using only functions, type equality proofs, and Haskell’s `newtype`, a special case of an ADT with only one constructor that has a single field. This version is shown in Figure 6. The field of $\text{Exp } t$ defines a simple pattern-matching interface for STLC representations: given case functions for abstraction and application, each producing a result of type r , we can produce an r . We manually define constructors abs and app for $\text{Exp } t$ by their pattern matching behavior. For example, the arguments to app are the two subexpressions of an application node. Given case functions for abstraction and application, app calls the case function for application, and passes along its subexpressions.

¹ Not to be confused with the type class Eq defined in Haskell’s Prelude

```

newtype Exp t = Exp {
  matchExp ::
    forall r.
      (forall a b. Eq t (a → b) → (Exp a → Exp b) → r) →
      (forall s. Exp (s → t) → Exp s → r) →
      r
}

abs :: (Exp t1 → Exp t2) → Exp (t1 → t2)
abs f = Exp (\fAbs fApp → fAbs refl f)

app :: Exp (t1 → t2) → Exp t1 → Exp t2
app e1 e2 = Exp (\fAbs fApp → fApp e1 e2)

eval :: Exp t → Exp t
eval e =
  matchExp e
    (\_ _ → e)
    (\e1 e2 →
      let e1' = eval e1 in
      matchExp e1'
        (\eq f →
          let eqL = eqApp (sym (arrL eq))
              eqR = eqApp (arrR eq)
              f' = coerce eqR . f . coerce eqL
          in f' e2)
        (\_ _ → app e1' e2))

```

Figure 6: Mogensen-Scott encoding of STLC

The `abs` constructor is similar, except that it takes one argument, while the case function `fAbs` for abstractions takes two. The first argument to `fAbs` is a type equality proof that `abs` supplies itself.

The function `matchExp` maps representations to their pattern matching interface, and the constructor `Exp` goes the opposite direction. These establish an isomorphism between `Exp t` and its pattern matching interface. In particular, `matchExp (Exp f) = f`. The type `Exp` is recursive because `Exp` occurs in the type of its field `matchExp`.

The Mogensen-Scott encoding of STLC uses higher order (F_ω) types, recursive types, and type equality proofs. In the next section we present $F_\omega^{\mu i}$, which supports each of these features. It extends F_ω with iso-recursive types and intensional type functions that we use to implement the type equality proof interface in Figure 4. In Section 4 we define a representation and evaluator for STLC in $F_\omega^{\mu i}$, which are similar to Figure 6. Then we go beyond STLC and implement our self-representation and self-evaluator for $F_\omega^{\mu i}$.

3. System $F_\omega^{\mu i}$

System $F_\omega^{\mu i}$ is defined in Figure 7. It extends F_ω with iso-recursive μ types and a type operator `Typecase` that is used to define intensional type functions. The kinds are the same as in F_ω . The kind $*$ classifies base types (the types of terms), and arrow kinds classify type level functions. The types are those of F_ω , plus μ and `Typecase`. The rules of type formation are those of F_ω , plus axioms for μ and `Typecase`. The terms are those of F_ω , plus `fold` and `unfold` that respectively contract or expand a recursive type. The rules of term formation are those of F_ω , plus rules for `fold` and `unfold`. Notably, there are no new terms that are type checked by `Typecase`. This is different than in previous work on intensional type analysis (ITA), where a type-level ITA operator is used to typecheck a term-level ITA operator. Type equivalence is the same as for F_ω , plus the three reduction rules for `Typecase`. The semantics is full F_ω β -reduction, plus a congruence rule for each of `fold` and `unfold` and a reduction rule for

`unfold` combined with `fold`. The normal form terms are those that cannot be reduced. Following Girard et al. [19], we define normal forms simultaneously with neutral terms, which are the normal forms other than abstractions or `fold`. Intuitively, a neutral term can replace a variable in a normal form without introducing a redex.

Capital letters and capitalized words such as `F`, `Exp`, `Bool` range over types. We will often use `F` for higher-kinded types (type functions), and `A`, `B`, `S`, `T`, `X`, `Y` for type variables of kind $*$. Lower case letters and uncapitalized words range over terms.

Recursive types can be used to define recursive functions and data types defined in terms of themselves. For example, each of the three versions of `Exp` defined in Figures 2, 3, and 6 is recursive. An iso-recursive type is not equal (or equivalent) to its definition, but rather is isomorphic to it, and `fold` and `unfold` form the isomorphism: `unfold` maps a recursive type to its definition, and `fold` is the inverse. Intuitively, `fold` generalizes the `Exp` new-type constructor from Figure 6 to work for many data types. Similarly, `unfold` generalizes `matchExp`. Using iso-recursive types is important for making type checking decidable. For more information about iso-recursive μ types, we refer the interested reader to Pierce’s book [30].

To simplify the language and our self-representation, we only support recursive types of kind $* \rightarrow *$ (type functions). This is sufficient for our needs, which are to encode recursive data types in the style seen in the previous section, and to define recursive functions. We can encode recursive base types (types of kind $*$) using a constant type function.

We will discuss `Typecase` in detail in Section 3.3.

3.1 Metatheory

System $F_\omega^{\mu i}$ is type safe and type checking is decidable. Proofs are included in the appendix. For type safety, we use a standard Progress and Preservation proof [45]. For decidability of type checking, we show that reduction of types is confluent and strongly normalizing [25].

Theorem 3.1. [Type Safety]

If $\langle \rangle \vdash e : T$, then either e is a normal form, or there exists an e' such that $\langle \rangle \vdash e' : T$ and $e \rightarrow e'$.

Theorem 3.2. Type checking is decidable.

3.2 Syntactic Sugar and Abbreviations

System $F_\omega^{\mu i}$ is a low-level calculus, more suitable for theory than for real-world programming. We use the following syntactic sugar to make our code more readable. We highlight the syntactic sugar to distinguish it from the core language.

- `let x : T = e1 in e2` desugars to $(\lambda x:T. e2) \ e1$, as usual.
- `let rec x : T1 = e1 in e2` desugars to `let x : T1 = fix T1 ( $\lambda x:T1. e1$ ) in e2`. Here `fix` is a standard fixpoint combinator of type $\forall T:*. (T \rightarrow T) \rightarrow T$.
- `decl X : K = T`; defines a new type abbreviation. `T` is inlined at every subsequent occurrence of `X`. Similarly, `decl x : T = e`; defines an inlined term abbreviation.
- `decl rec x : T = e`; declares a recursive term. It uses `fix` like `let rec`, and inlines like `decl`.

For further brevity, we sometimes omit the type annotations on abstractions, let bindings or declarations, when the type can be easily inferred from context. For example, we will write $(\lambda x. e)$ instead of $(\lambda x:T. e)$. We use $f \circ g$ to denote the composition of (type or term) functions f and g . This desugars to $(\lambda x. f (g x))$, where x is fresh.

(kinds) $K ::= *$ $| K_1 \rightarrow K_2$
 (types) $T ::= X$ $| T_1 \rightarrow T_2$ $| \forall X:K. T$ $| \lambda X:K. T$ $| T_1 T_2$ $| \mu$ $| \text{Typecase}$
 (terms) $e ::= x$ $| \lambda x:T. e$ $| e_1 e_2$ $| \Lambda X:K. e$ $| e T$ $| \text{fold } T_1 T_2 e$ $| \text{unfold } T_1 T_2 e$
 (environments) $\Gamma ::= \langle \rangle$ $| \Gamma, (x:T)$ $| \Gamma, (X:K)$

(normal form terms) $v ::= n$ $| (\lambda x:T. v)$ $| (\Lambda X:K. v)$ $| \text{fold } T_1 T_2 v$
 (neutral terms) $n ::= x$ $| n v$ $| n T$ $| \text{unfold } T_1 T_2 n$

Grammar

$$\frac{(X:K) \in \Gamma}{\Gamma \vdash X : K}$$

$$\frac{\Gamma \vdash T_1 : * \quad \Gamma \vdash T_2 : *}{\Gamma \vdash T_1 \rightarrow T_2 : *}$$

$$\frac{\Gamma, (X:K_1) \vdash T : K_2}{\Gamma \vdash (\lambda X:K_1. T) : K_1 \rightarrow K_2}$$

$$\frac{\Gamma \vdash T_1 : K_2 \rightarrow K \quad \Gamma \vdash T_2 : K_2}{\Gamma \vdash T_1 T_2 : K}$$

$$\Gamma \vdash \mu : ((* \rightarrow *) \rightarrow * \rightarrow *) \rightarrow * \rightarrow *$$

$$\Gamma \vdash \text{Typecase} : (* \rightarrow * \rightarrow *) \rightarrow$$

$$(* \rightarrow *) \rightarrow (* \rightarrow *) \rightarrow$$

$$(((* \rightarrow *) \rightarrow * \rightarrow *) \rightarrow * \rightarrow *) \rightarrow * \rightarrow *) \rightarrow *$$

Type Formation

$$T \equiv T \quad \frac{T_1 \equiv T_2}{T_2 \equiv T_1} \quad \frac{T_1 \equiv T_2 \quad T_2 \equiv T_3}{T_1 \equiv T_3}$$

$$\frac{T_1 \equiv T_1' \quad T_2 \equiv T_2'}{T_1 \rightarrow T_2 \equiv T_1' \rightarrow T_2'}$$

$$\frac{T \equiv T'}{(\lambda X:K. T) \equiv (\lambda X:K. T')}$$

$$\frac{T \equiv T' \quad T_1 \equiv T_1' \quad T_2 \equiv T_2'}{T_1 T_2 \equiv T_1' T_2'}$$

$$(\lambda X:K. T_1) T_2 \equiv (T_1[X := T_2])$$

$$(\forall X:K. T_2) \equiv (\forall X':K. T_2[X := X'])$$

$$(\lambda X:K. T) \equiv (\lambda X':K. T[X := X'])$$

$$\text{Typecase } F_1 F_2 F_3 F_4 (T_1 \rightarrow T_2) \equiv F_1 T_1 T_2$$

$$\text{Typecase } F_1 F_2 F_3 F_4 (\mu T_1 T_2) \equiv F_4 T_1 T_2$$

$$\frac{X \notin \text{FV}(F_3)}{\text{Typecase } F_1 F_2 F_3 F_4 (\forall X:K. T) \equiv F_2 (\forall X:K. F_3 T)}$$

Type Equivalence

$$\frac{(x:T) \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash T_1 : * \quad \Gamma, (x:T_1) \vdash e : T_2}{\Gamma \vdash (\lambda x:T_1. e) : T_1 \rightarrow T_2}$$

$$\frac{\Gamma \vdash e_1 : T_2 \rightarrow T \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 e_2 : T}$$

$$\frac{\Gamma, (X:K) \vdash e : T}{\Gamma \vdash (\Lambda X:K. e) : (\forall X:K. T)}$$

$$\frac{\Gamma \vdash e : (\forall X:K. T_1) \quad \Gamma \vdash T_2 : K}{\Gamma \vdash e : T_1[X:=T_2]}$$

$$\frac{\Gamma \vdash F : (* \rightarrow *) \rightarrow * \rightarrow * \quad \Gamma \vdash T : *}{\Gamma \vdash e : F (\mu F) T}$$

$$\Gamma \vdash \text{fold } F T e : \mu F T$$

$$\frac{\Gamma \vdash F : (* \rightarrow *) \rightarrow * \rightarrow * \quad \Gamma \vdash T : *}{\Gamma \vdash e : \mu F T}$$

$$\frac{\Gamma \vdash \text{unfold } F T e : F (\mu F) T}{\Gamma \vdash e : T_1 \quad T_1 \equiv T_2 \quad \Gamma \vdash T_2 : *}{\Gamma \vdash e : T_2}$$

Term Formation

$$(\lambda x:T. e) e_1 \longrightarrow e[x := e_1]$$

$$(\Lambda X:K. e) T \longrightarrow e[X := T]$$

$$\text{unfold } F T (\text{fold } F' T' e) \longrightarrow e$$

$$\frac{e_1 \longrightarrow e_2}{e_1 e_3 \longrightarrow e_2 e_3}$$

$$e_3 e_1 \longrightarrow e_3 e_2$$

$$e_1 T \longrightarrow e_2 T$$

$$(\lambda x:T. e_1) \longrightarrow (\lambda x:T. e_2)$$

$$(\Lambda X:K. e_1) \longrightarrow (\Lambda X:K. e_2)$$

$$\text{fold } F T e_1 \longrightarrow \text{fold } F T e_2$$

$$\text{unfold } F T e_1 \longrightarrow \text{unfold } F T e_2$$

Reduction

Figure 7: Definition of $F_{\omega}^{\mu i}$


```

decl  $\perp$  : *  $\rightarrow$  * = ( $\forall T$  : *. T);
decl ArrL : *  $\rightarrow$  * =
  Typecase ( $\lambda A$  : *.  $\lambda B$  : *. A) ( $\lambda A$  : *.  $\perp$ ) ( $\lambda A$  : *.  $\perp$ )
    ( $\lambda F$  : (*  $\rightarrow$  *)  $\rightarrow$  *  $\rightarrow$  *.  $\lambda A$  : *.  $\perp$ );
decl ArrR : *  $\rightarrow$  * =
  Typecase ( $\lambda A$  : *.  $\lambda B$  : *. B) ( $\lambda A$  : *.  $\perp$ ) ( $\lambda A$  : *.  $\perp$ )
    ( $\lambda F$  : (*  $\rightarrow$  *)  $\rightarrow$  *  $\rightarrow$  *.  $\lambda A$  : *.  $\perp$ );
decl All : (*  $\rightarrow$  *)  $\rightarrow$  (*  $\rightarrow$  *)  $\rightarrow$  *  $\rightarrow$  * =
   $\lambda Out$  : *  $\rightarrow$  *.  $\lambda In$  : *  $\rightarrow$  *.
  Typecase ( $\lambda A$  : *.  $\lambda B$  : *. B) Out In
    ( $\lambda F$  : (*  $\rightarrow$  *)  $\rightarrow$  *  $\rightarrow$  *.  $\lambda A$  : *.  $\perp$ );
decl Unfold : *  $\rightarrow$  * =
  Typecase ( $\lambda A$  : *.  $\lambda B$  : *.  $\perp$ ) ( $\lambda A$  : *.  $\perp$ ) ( $\lambda A$  : *.  $\perp$ )
    ( $\lambda F$  : (*  $\rightarrow$  *)  $\rightarrow$  *  $\rightarrow$  *.  $\lambda A$  : *. F ( $\mu$  F) A);

```

Figure 8: Intensional type functions

We use $S \times T$ for pair types, which can be easily encoded in System $F_{\omega}^{\mu i}$. Intuitively, $\times$ is an infix type function of kind $* \rightarrow * \rightarrow *$. We use (x, y) to construct the pair of x and y . `fst` and `snd` project the first and second component from a pair, respectively.

3.3 Intensional Type Functions

Our `Typecase` operator allows us to define type functions that depend on the top-level structure of a base type. It is parameterized by four case functions, one for arrow types, two for quantified types, and one for recursive types. When applied to an arrow type or a recursive type, `Typecase` decomposes the input type and applies the corresponding case function to the components. For example, when applied to an arrow type $T_1 \rightarrow T_2$, `Typecase` applies the case function for arrows to T_1 and T_2 . When applied to a recursive type μF , `Typecase` applies the case function for recursive types to F and T .

We have two functions for the case of quantified types because they cannot be easily decomposed in $F_{\omega}^{\mu i}$. Previous work on ITA for quantified types [37] would decompose a quantified type $\forall X : K. T$ into the kind K and a type function of kind $K \rightarrow *$. The components would then be passed as arguments to a case function for quantified types. This approach requires kind polymorphism in types, which is outside of $F_{\omega}^{\mu i}$. Our solution uses two functions for quantified types. `Typecase` applies one function inside the quantified type (under the quantifier), and the other outside.

For example, let F be an intensional type function defined by $F = \text{Typecase } \text{Arr } \text{Out } \text{In } \mu$. Here, `Arr` and `$\mu$` are the case functions for arrow types and recursive types, respectively. `Out` and `In` are the case functions for quantified types, with `Out` being applied outside the type, and `In` inside the type. Then $F (\forall X : K. T) \equiv \text{Out } (\forall X : K. \text{In } T)$. Note that to avoid variable capture, we require that X not occur free in `In` (which can be ensured by renaming X).

Figure 8 defines four intensional type functions. Each expects its input type to be of a particular form: `ArrL` and `ArrR` expect an arrow type, `All` expects a quantified type, and `Unfold` expects a recursive type. On types not of the expected form, each function returns the type $\perp = (\forall T : *. T)$, which we use to indicate an error. The type $\perp$ is only inhabited by non-normalizing terms.

`ArrL` and `ArrR` return the domain and codomain of an arrow type, respectively. More precisely, the specification of `ArrL` is as follows (`ArrR` is similar):

$$\text{ArrL } T \equiv \begin{cases} T_1 & \text{if } T \equiv T_1 \rightarrow T_2 \\ \perp & \text{otherwise} \end{cases}$$

`All` takes two type functions `Out` and `In`, and applies them outside and inside a $\forall$ quantifier, respectively.

$$\text{All } \text{Out } \text{In } T \equiv \begin{cases} \text{Out } (\forall X : K. \text{In } T) & \text{if } T \equiv \forall X : K. T \\ \perp & \text{otherwise} \end{cases}$$

```

decl Eq : *  $\rightarrow$  * =
   $\lambda A$  : *.  $\lambda B$  : *.  $\forall F$  : *  $\rightarrow$  *. F A  $\rightarrow$  F B;

decl refl : ( $\forall A$  : *. Eq A A) =
   $\lambda A$  : *.  $\lambda F$  : *  $\rightarrow$  *.  $\lambda x$  : F A. x;

decl sym : ( $\forall A$  : *.  $\forall B$  : *. Eq A B  $\rightarrow$  Eq B A) =
   $\lambda A$  : *.  $\lambda B$  : *.  $\lambda eq$  : Eq A B.
  let p : Eq A A = refl A in
  eq ( $\lambda T$  : *. Eq T A) p;

decl trans : ( $\forall A$  : *.  $\forall B$  : *.  $\forall C$  : *.
  Eq A B  $\rightarrow$  Eq B C  $\rightarrow$  Eq A C) =
   $\lambda A$  : *.  $\lambda B$  : *.  $\lambda C$  : *.  $\lambda eqAB$  : Eq A B.  $\lambda eqBC$  : Eq B C.
   $\lambda F$  : *  $\rightarrow$  *.  $\lambda x$  : F A. eqBC F (eqAB F x);

decl eqApp : ( $\forall A$  : *.  $\forall B$  : *.  $\forall F$  : *  $\rightarrow$  *.
  Eq A B  $\rightarrow$  Eq (F A) (F B)) =
   $\lambda A$  : *.  $\lambda B$  : *.  $\lambda F$  : *  $\rightarrow$  *.  $\lambda eq$  : Eq A B.
  let p : Eq (F A) (F A) = refl (F A) in
  eq ( $\lambda T$  : *. Eq (F A) (F T)) p;

decl arrL : ( $\forall A1$  : *.  $\forall A2$  : *.  $\forall B1$  : *.  $\forall B2$  : *.
  Eq (A1  $\rightarrow$  A2) (B1  $\rightarrow$  B2)  $\rightarrow$ 
  Eq A1 B1) =
   $\lambda A1$  A2 B1 B2. eqApp (A1  $\rightarrow$  A2) (B1  $\rightarrow$  B2) ArrL;

decl arrR : ( $\forall A1$  : *.  $\forall A2$  : *.  $\forall B1$  : *.  $\forall B2$  : *.
  Eq (A1  $\rightarrow$  A2) (B1  $\rightarrow$  B2)  $\rightarrow$ 
  Eq A2 B2) =
   $\lambda A1$  A2 B1 B2. eqApp (A1  $\rightarrow$  A2) (B1  $\rightarrow$  B2) ArrR;

decl coerce : ( $\forall A$  : *.  $\forall B$  : *. Eq A B  $\rightarrow$  A  $\rightarrow$  B) =
   $\lambda A$  : *.  $\lambda B$  : *.  $\lambda eq$  : Eq A B. eq Id;

```

Figure 9: Implementation of type equality proofs in $F_{\omega}^{\mu i}$.

`Unfold` returns the result of unfolding a recursive type one time:

$$\text{Unfold } T \equiv \begin{cases} F (\mu F) A & \text{if } T \equiv \mu F A \\ \perp & \text{otherwise} \end{cases}$$

In the next section, we will use these intensional type functions to define type equality proofs that are useful for defining GADT-style typed representations and polymorphically-typed self evaluators.

4. Type Equality Proofs in $F_{\omega}^{\mu i}$

In Section 4.1 we implement decomposable type equality proofs in $F_{\omega}^{\mu i}$ and use them to represent and evaluate STLC. Then in Section 4.2 we go beyond simple types to quantified and recursive types in preparation for our $F_{\omega}^{\mu i}$ self-representation and self-evaluators.

4.1 Equality Proofs for Simple Types

Figure 9 shows the $F_{\omega}^{\mu i}$ implementation of the type equality proofs from Figure 4. The foundation of our encoding is Leibniz equality, which encodes that two types are indistinguishable in all contexts. This is a standard technique for encoding type equality in F_{ω} [5, 28, 39]. The type `Eq A B` is defined as $\forall F : * \rightarrow *. F A \rightarrow F B$. Intuitively, the type function F ranges over type contexts, and a Leibniz equality proof can replace the type A with B in any context F .

The only way to introduce a new type equality proof is by `refl`, which constructs an identity function to witness that a type is equal to itself. Symmetry is encoded by `sym`, which uses an equality proof of type `Eq A B` to coerce another proof of type `Eq A A`, replacing the first

```

decl ExpF : (* → *) → * → * =
  λExp : * → *. λT : *. ∀R : *.
    (∀A:*. ∀B:*. Eq (A → B) T → (Exp A → Exp B) → R) →
    (∀S:*. Exp (S → T) → Exp S → R) →
    R;

decl Exp : * → * = μ ExpF;

decl abs:(∀A:*. ∀B:*. (Exp A → Exp B) → Exp (A → B)) =
  ΛA:*. ΛB:*. λf:Exp A → Exp B.
  fold ExpF (A → B)
  (ΛR. λfAbs. λfApp. fAbs A B (refl (A → B)) f);

decl app:(∀A:*. ∀B:*. Exp (A → B) → Exp A → Exp B) =
  ΛA:*. ΛB:*. λe1 : Exp (A → B). λe2 : Exp A.
  fold ExpF B (ΛR. λfAbs. λfApp. fApp A e1 e2);

decl matchExp : (∀T:*. Exp T → ExpF Exp T) =
  ΛT : *. λe : Exp T. unfold ExpF T e;

decl rec eval : (∀T:*. Exp T → Exp T) =
  ΛT:*. λe : Exp T.
  matchExp T e (Exp T)
  (ΛT1 T2. λeq f. e)
  (ΛS:*. λe1 : Exp (S → T). λe2 : Exp S.
    let e1':Exp (S → T) = eval (S → T) e1 in
    matchExp (S → T) e1' (Exp T)
    (ΛA B. λeq:Eq (A → B) (S → T). λf:Exp A → Exp B.
      let eqL:Eq (Exp S) (Exp A) =
        eqApp S A Exp (sym A S (arrL A B S T eq)) in
      let eqR:Eq (Exp B) (Exp T) =
        eqApp B T Exp (arrR A B S T eq) in
      let f':Exp S → Exp T = λx : Exp S.
        let x' : Exp A = coerce (Exp S) (Exp A) eqL x in
        coerce (Exp B) (Exp T) eqR (f x')
      in
      eval T (f' e2))
  (ΛT2. λe3 e4. app S T e1' e2));

```

Figure 10: Encoding and evaluation of STLC in $F_{\omega}^{\mu i}$

A with B and resulting in the type $\text{Eq } B \ A$. Transitivity is encoded by trans , which uses function composition to combine two coercions. A proof of type $\text{Eq } A \ B$ is effectively a coercion – it can coerce any term of type $F \ A$ to $F \ B$. Thus, coerce simply instantiates the proof with the identity function on types. For brevity we will sometimes omit coerce and use equality proofs as coercions directly.

Each of Eq , refl , sym , trans , eqApp , and coerce are definable in the pure F_{ω} subset of $F_{\omega}^{\mu i}$. The addition of intensional type functions allows $F_{\omega}^{\mu i}$ to decompose Leibniz equality proofs. The key is that eqApp is stronger in $F_{\omega}^{\mu i}$ than in F_{ω} because the type function F can be intensional. In particular, arrL and arrR are defined using eqApp with the intensional type functions ArrL and ArrR , respectively.

Figure 10 shows the STLC representation and evaluator in $F_{\omega}^{\mu i}$. It uses a Mogensen-Scott encoding similar to the one in Figure 6, with a few notable differences. The type ExpF is a stratified version of Exp . In particular, it uses a λ -abstraction to untie the recursive knot. Exp is defined as μExpF , which re-ties the knot. Now $\text{Exp } T$ and $\text{ExpF } \text{Exp } T$ are isomorphic, with $\text{unfold } \text{ExpF } T$ converting from $\text{Exp } T$ to $\text{ExpF } \text{Exp } T$, and $\text{fold } \text{ExpF } T$ converting from $\text{ExpF } \text{Exp } T$ back to $\text{Exp } T$. We define matchExp as a convenience and to align with Figure 6, but we could as well use $\text{unfold } \text{ExpF } T$ instead of $\text{matchExp } T$. This version of eval is similar to the previous version. The main difference is in the increased amount of type annotations.

```

decl TcAll : * → * =
  λT. ∀Arr. ∀Out. ∀In. ∀Mu.
    Eq (Typecase Arr Out In Mu T) (All Out In T);
decl UnAll : * → * =
  λT. ∀Out. Eq (All Out (λA:*. A) T) (Out T);
decl IsAll : * → * = λT. (TcAll T × UnAll T);

tcAllx,k,t = ΛArr. ΛOut. ΛIn. ΛMu. refl (Out (∀X:K. In T))
unAllx,k,t = ΛOut. refl (Out (∀X:K. T))
isAllx,k,t = (tcAllx,k,t, unAllx,k,t)

```

Figure 11: IsAll proofs.

4.2 Beyond Simple Types

In this section, we move beyond STLC in preparation for our self-representation of $F_{\omega}^{\mu i}$. We will focus on the question: How can we establish that an unknown type is a quantified or recursive type?

In Figure 10, we establish that a type T is an arrow type by abstracting over types $T1$ and $T2$ of kind $*$ and a proof of type $\text{Eq } (T1 \rightarrow T2) \ T$. This will work for any arrow type because $T1$ and $T2$ must have kind $*$ in order for $T1 \rightarrow T2$ to kind check. The case for recursive types is similar. In $F_{\omega}^{\mu i}$, a recursive type $\mu F \ A$ kind checks only if F has kind $(* \rightarrow *) \rightarrow * \rightarrow *$ and A has kind $*$. Therefore, we can establish that some type T is a recursive type by abstracting over F and A and a proof of type $\text{Eq } (\mu F \ A) \ T$. Given a proof that one recursive type $\mu F1 \ A1$ is equal to another $\mu F2 \ A2$, we know that their unfoldings are equal as well. This can be proved using eqApp and the intensional type function Unfold :

```

decl eqUnfold : (∀F1:(* → *) → * → *. ∀A1:*.
  ∀F2:(* → *) → * → *. ∀A2:*.
  Eq (μ F1 A1) (μ F2 A2) →
  Eq (F1 (μ F1 A1) A1) (F2 (μ F2 A2) A2) =
  ΛF1 A1 F2 A2. eqApp (μ F1 A1) (μ F2 A2) Unfold);

```

We establish that a type is a quantified type in a different way, by proving equalities about the behavior of Typecase on that type. We do this because unlike arrow types and recursive types, we can't abstract over the components of an arbitrary quantified type in $F_{\omega}^{\mu i}$, as was discussed earlier in Section 3.3. Figure 11 defines our IsAll proofs that a type is a quantified type. A proof of type $\text{IsAll } T$ consists of a pair of *polymorphic* equality proofs about Typecase . The first is of type $\text{TcAll } T$ and proves that because T is a quantified type, $\text{Typecase } \text{Arr } \text{Out } \text{In } \text{Mu } T$ is equal to $\text{All } \text{Out } \text{In } T$. The proof is polymorphic because it proves the equality for any Arr and Mu . In other words, Arr and Mu are irrelevant: since T is a quantified type, they can be replaced by the constant $\perp$ functions used by All . The second polymorphic equality proof, of type $\text{UnAll } T$, shows that $\text{All } \text{Out } (\lambda A:*. A) \ T$ is equal to $\text{Out } T$ for any Out . This is true because applying the identity function under the quantifier has no effect. These proofs are orthogonal to each other, and each is useful for some of our operations, as we discuss in Section 7.

We define IsAll proofs using indexed abbreviations $\text{tcAll}_{x,k,t}$, $\text{unAll}_{x,k,t}$, and $\text{isAll}_{x,k,t}$. These are meta-level abbreviations, not part of $F_{\omega}^{\mu i}$. The type of $\text{tcAll}_{x,k,t}$ is $\text{TcAll } (\forall X:K. T)$, the type of $\text{unAll}_{x,k,t}$ is $\text{UnAll } (\forall X:K. T)$, and the type of $\text{isAll}_{x,k,t}$ is $\text{IsAll } (\forall X:K. T)$. Note that $\text{tcAll}_{x,k,t}$ and $\text{unAll}_{x,k,t}$ use refl to create the equality proof. In the case of $\text{unAll}_{x,k,t}$, the proof $\text{refl } (\text{Out } (\forall X:K. T) \text{ has type } \text{Eq } (\text{Out } (\forall X:K. T)) (\text{Out } (\forall X:K. T)))$, which is equivalent to the type $\text{Eq } (\text{All } \text{Out } (\lambda A:*. A) (\forall X:K. T)) (\text{Out } (\forall X:K. T))$.

Impossible Cases. It is sometimes impossible to establish that a type is of a particular form, in particular, if it is already known to be of a different form. This sometimes happens when pattern matching

on a GADT. For example, suppose we added integers to our Haskell representation of STLC. When matching on a representation of type Exp Int , the Abs case would provide a proof that Int is equal to an arrow type $t_1 \rightarrow t_2$, which is impossible. Haskell's type checker can detect that such cases are unreachable, and therefore those cases need not be covered in order for a pattern match expression to be exhaustive.

Our equality proofs support similar reasoning about impossible cases, which we use in some of our meta-programs. In particular, given an impossible type equality proof (which must be hypothetical), we can derive a (strongly normalizing) term of type $\perp$:

```
eqArrMu  :  $\forall A B F T. \text{Eq } (A \rightarrow B) (\mu F T) \rightarrow \perp$ 
arrIsAll :  $\forall A B. \text{IsAll } (A \rightarrow B) \rightarrow \perp$ 
muIsAll  :  $\forall F T. \text{IsAll } (\mu F T) \rightarrow \perp$ 
```

There are three kinds of contradictory equality proofs in $F_\omega^{\mu i}$: a proof that an arrow type is equal to a recursive type (eqArrMu), that an arrow type is a quantified type (arrIsAll), or that a recursive type is a quantified type (muIsAll). Definitions of eqArrMu , arrIsAll , and muIsAll are provided in the appendix.

5. Our Representation Scheme

The self-representation of System $F_\omega^{\mu i}$ is shown in Figure 12. Like the STLC representation in Figure 10, we use a typed Mogensen-Scott encoding, though there are several important differences. Following previous work on typed self-representation [7, 8, 31], we use Parametric Higher-Order Abstract Syntax (PHOAS) [12, 40] to give our representation more expressiveness. The type PExp is parametric in V , which determines the type of free variables in a representation. Intuitively, PExp can be understood as the type of representations that may contain free variables. The type Exp quantifies over V , which ensures that the representation is closed. Our quoter assumes that the designated variable V is globally fresh.

Our quotation procedure is similar to previous work on typed self-representation [7, 8, 31]. The quotation function $\bar{\cdot}$ is defined only on closed terms, and depends on a pre-quotation function $\triangleright$ from type derivations to terms. In the judgment $\Gamma \vdash e : T \triangleright q$, the input is the type derivation $\Gamma \vdash e : T$, and the output is a term q . We call q the pre-representation of e .

We represent variables meta-circularly, that is, by other variables. In particular, a variable of type T is represented by a variable of the same name with type $\text{PExp } V T$. The cases for quoting λ -abstraction, application, fold and unfold are similar. In each case, we recursively quote the subterm and apply the corresponding constructor. The constructors for these cases create the necessary type equality proofs.

To represent type abstraction and application, the quoter generates IsAll proofs itself, since they depend on the kind of the type (which cannot be passed as an argument to the constructors in $F_\omega^{\mu i}$). The quoter also generates utility functions stripAll_K , $\text{underAll}_{X,K,T}$, and $\text{inst}_{X,K,T,S}$ that are useful to meta-programs for operating on type abstractions and applications. These utility functions comprise an extensional representation of polymorphism that is similar to one we developed for F_ω in previous work [8]. The purpose of the extensional representation is to represent polymorphic terms in languages like F_ω and $F_\omega^{\mu i}$ that do not include kind polymorphism.

The function stripAll_K has the type $\text{StripAll } (\forall X:K. T)$, as long as $(\forall X:K. T)$ is well-typed. For any type A in which X does not occur free, stripAll_K can map $\text{All Id } (\lambda B:*. A) (\forall X:K. T) \equiv (\forall X:K. A)$ to A . Note that the quantification of X is redundant, since it does not occur free in the type A . Therefore, any instantiation of X will result in A . We use the fact that all kinds in $F_\omega^{\mu i}$ are inhabited to define stripAll_K . It uses the kind inhabitant T_K for the instantiation. For each kind K , T_K is a closed type of kind K .

The function $\text{underAll}_{X,K,T}$ has the type $\text{UnderAll } (\forall X:K. T)$. It can apply a function under the quantifier of a type $\text{All Id } F_1$

$(\forall X:K. T) \equiv (\forall X:K. F_1 T)$ to produce a result of type $\text{All Id } F_2$ $(\forall X:K. T) \equiv (\forall X:K. F_2 T)$. In particular, our evaluators use $\text{underAll}_{X,K,T}$ to make recursive calls on the body of a type abstraction. The representation of a type abstraction $(\lambda X:K. e)$ of type $(\forall X:K. T)$ contains the term $(\lambda X:K. q)$. Here, q is the representation of e , in which the type variable X can occur free. The type of $(\lambda X:K. q)$ is $\text{All Id } (\text{PExp } V) (\forall X:K. T) \equiv (\forall X:K. \text{PExp } V T)$.

We can use $\text{underAll}_{X,K,T}$ and stripAll_K together in operations that always produce results of a particular type. For example, our measure of the size of a representation always returns a Nat . We use $\text{underAll}_{X,K,T}$ to make the recursive call to size under the quantifier. The result of $\text{underAll}_{X,K,T}$ has the type $\text{All Id } (\lambda Y:*. \text{Nat}) (\forall X:K. T) \equiv (\forall X:K. \text{Nat})$ where the quantification of X is redundant. We can then use strip_K to strip away the redundant quantifier, enabling us to access the Nat underneath.

An instantiation function $\text{inst}_{X,K,T,S}$ has the type $\text{Inst } (\forall X:K. T) (T[X:=S])$. It can be used to instantiate types of the form $(\forall X:K. F T)$, producing instantiations of the form $F (T[X:=S])$.

The combination of IsAll proofs and the utility functions stripAll_K , $\text{underAll}_{X,K,T}$, and $\text{inst}_{X,K,T,S}$ allows us to represent higher-kinded polymorphism without kind polymorphism. Notice that in the types of tabs and tapp (shown in Figure 12), the type variables A range over arbitrary quantified types. The IsAll proofs and utility functions witness that A is a quantified type and provide an interface for working on quantified types that is expressive enough to support a variety of meta-programs.

Properties. Every closed and well-typed term has a unique representation that is also closed and well-typed.

Theorem 5.1. *If $\langle \rangle \vdash e : T$, then $\langle \rangle \vdash \bar{e} : \text{Exp } T$.*

The proof is by induction on the derivation of the typing judgment $\langle \rangle \vdash e : T$. It relies on the fact that we can always produce the proof terms and utility functions needed for each constructor.

All representations are strongly normalizing, even those that represent non-normalizing terms.

Theorem 5.2. *If $\langle \rangle \vdash e : T$, then $\bar{e}$ is strongly normalizing.*

6. Our Self-Evaluators

In this section we discuss our three self-evaluators, which implement weak head normal form evaluation, single-step left-most β -reduction, and normalization by evaluation (NbE).

Weak head-normal evaluation. Figure 13 shows our first evaluator, which reduces terms to their weak head-normal form. The closed and well-typed weak head-normal forms of $F_\omega^{\mu i}$ are λ and Λ abstractions, and fold expressions. There is no evaluation under abstractions or fold expressions, and function arguments are not evaluated before β -reduction.

The function eval evaluates closed representations, which have Exp types. The main evaluator is evalV , which operates on PExp types. If the input is a variable, a λ or Λ abstraction, or a fold expression, it is already a weak head-normal form. We use constant case functions constVar , constAbs , etc. to return the input in these cases. The case for application is similar to that for STLC from Figure 10, except for the use of the utility function matchAbs . This is a specialized version of matchExp that takes a only one case function, for λ -abstractions, and a default value that is returned in all other cases. The types and definitions of the constant case functions and specialized match functions are given in the appendix. We now turn to the interesting new cases, for reducing type applications and $\text{unfold}/\text{fold}$ expressions.

When the input represents a type application, we get a proof that A is an instance of some quantified type B , and the head position subexpression e_1 has type $\text{PExp } V B$. If e_1 evaluates to a type abstraction, we

$$T_* = (\forall X:*.X) \quad T_{K1 \rightarrow K2} = \lambda X:K1. T_{K2}$$

Kind Inhabitants

```

decl Id : * → * = λA:*. A;

decl UnderAll : * → * =
  λT:*. ∀F1:* → *. ∀F2:* → *.
  (∀A:*. F1 A → F2 A) →
  All Id F1 A → All Id F2 A;
decl StripAll : * → * =
  λT:*. ∀A:*. All Id (λB:*. A) T → A;
decl Inst : * → * → * =
  λA:*. λB:*. ∀F:* → *. All Id F A → F B;

underAllX,K,T =
  λF1. λF2. λf : (∀A:*. F1 A → F2 A).
  λe : (∀X:K. F1 T). λX:K. f T (e X);
stripAllK = λA. λe:(∀X:K.A). e TK
instX,K,T,S = λF. λf:(∀X:K.F T). f S

```

Operators on quantified types.

```

decl PExpF : (* → *) → (* → *) → * → * =
  λV:* → *. λPEXPV:* → *. λA:*. ∀R:*.
  (V A → R) →
  (∀S T. Eq (S → T) A → (PEXP V S → PEXP V T) → R) →
  (∀B. PEXP V (B → A) → PEXP V B → R) →
  (IsAll A → StripAll A → UnderAll A →
  All Id (PEXP V) A → R) →
  (∀B:*. IsAll B → Inst B A → PEXP V B → R) →
  (∀F B. Eq (μ F B) A → PEXP V (F (μ F) B) → R) →
  (∀F B. Eq (F (μ F) B) A → PEXP V (μ F B) → R) →
  R

```

```

decl PExp : (* → *) → * → * = λV:* → *. μ (PExpF V);
decl Exp : * → * = λA:*. ∀V:* → *. PExp V A;

```

Definitions of PExp and Exp

```

decl var: (∀V A. V A → PExp V A) =
  λV A. λx:V A. fold (PEXP V) A (
  λR. λvar. λabs. λapp. λtabs. λtapp. λfld. λunfld.
  var x);
decl abs: (∀V A B. (PEXP V A → PEXP V B) → PEXP V (A → B)) =
  λV A B. λf: (PEXP V A → PEXP V B). fold (PEXP V) (A → B) (
  λR. λvar. λabs. λapp. λtabs. λtapp. λfld. λunfld.
  abs A B (refl (A → B)) f);
decl app: (∀V A B. PEXP V (B → A) → PEXP V B → PEXP V A) =
  λV A B. λf: PEXP V (B → A). λx: PEXP V B. fold (PEXP V) A (
  λR. λvar. λabs. λapp. λtabs. λtapp. λfld. λunfld.
  app B f x);
decl tabs: (∀V A. IsAll A → StripAll A → UnderAll A →
  All Id (PEXP V) A → PEXP V A) =
  λV A. λp: IsAll A. λs: StripAll A. λu: UnderAll A.
  λe: All Id (PEXP V) A. fold (PEXP V) A (
  λR. λvar. λabs. λapp. λtabs. λtapp. λfld. λunfld.
  tabs p s u e);
decl tapp: (∀V A B. IsAll A → Inst A B → PEXP V A → PEXP V B) =
  λV A B. λp: IsAll A. λi: Inst A B. λe: PEXP V A.
  fold (PEXP V) B (
  λR. λvar. λabs. λapp. λtabs. λtapp. λfld. λunfld.
  tapp A p i e);
decl fld: (∀V F A. PEXP V (F (μ F) A) → PEXP V (μ F A)) =
  λV F A. λe: PEXP V (F (μ F) A). fold (PEXP V) (μ F A) (
  λR. λvar. λabs. λapp. λtabs. λtapp. λfld. λunfld.
  fld F A (refl (μ F A)) e);
decl unfld: (∀V F A. PEXP V (μ F A) → PEXP V (F (μ F) A)) =
  λV F A. λe : PEXP V (μ F A). fold (PEXP V) (F (μ F) A) (
  λR. λvar. λabs. λapp. λtabs. λtapp. λfld. λunfld.
  unfld F A (refl (F (μ F) A)) e);

decl matchExp : (∀V:* → *. ∀A:*. PEXP V A → PEXP F (PEXP V) A) =
  λV:* → *. λA:*. λe: PEXP V A. unfold (PEXP V) A e

```

Constructors and match for PExp

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T \triangleright x}$$

$$\frac{\Gamma \vdash T1 : * \quad \Gamma, (x: T1) \vdash e : T2 \triangleright q}{\Gamma \vdash (\lambda x: T1. e) : T1 \rightarrow T2 \triangleright \text{abs } V \ T1 \ T2 \ (\lambda x: \text{PEXP } V \ T1. q)}$$

$$\frac{\Gamma \vdash e_1 : T_2 \rightarrow T \triangleright q_1 \quad \Gamma \vdash e_2 : T_2 \triangleright q_2}{\Gamma \vdash e_1 \ e_2 : T \triangleright \text{app } V \ T_2 \ T \ q_1 \ q_2}$$

$$\frac{\Gamma, (X: K) \vdash e : T \triangleright q \quad \begin{array}{l} \text{isAll}_{X,K,T} = p \\ \text{stripAll}_K = s \\ \text{underAll}_{X,K,T} = u \end{array}}{\Gamma \vdash (\lambda X: K. e) : (\forall X: K. T) \triangleright \text{tabs } V \ (\forall X: K. T) \ p \ s \ u \ (\lambda X: K. q)}$$

$$\frac{\Gamma \vdash e : (\forall X: K. T) \triangleright q \quad \begin{array}{l} \text{isAll}_{X,K,T} = p \\ \text{inst}_{X,K,T,A} = i \end{array}}{\Gamma \vdash e \ A : T[X:=A] \triangleright \text{tapp } V \ (\forall X: K. T) \ (T[X:=A]) \ p \ i}$$

$$\frac{\Gamma \vdash F : (* \rightarrow *) \rightarrow * \rightarrow * \quad \Gamma \vdash T : * \quad \Gamma \vdash e : F (\mu F) \ T \triangleright q}{\Gamma \vdash \text{fold } F \ T \ e : \mu F \ T \triangleright \text{fld } V \ F \ T \ q}$$

$$\frac{\Gamma \vdash F : (* \rightarrow *) \rightarrow * \rightarrow * \quad \Gamma \vdash T : * \quad \Gamma \vdash e : \mu F \ T \triangleright q}{\Gamma \vdash \text{unfold } F \ T \ e : F (\mu F) \ T \triangleright \text{unfld } V \ F \ T \ q}$$

$$\frac{\langle \rangle \vdash e : T \triangleright q}{\bar{e} = \lambda V: * \rightarrow *. q}$$

Quotation and pre-quotation

Figure 12: Self-representation of $F_{\omega}^{\mu i}$.

```

decl rec evalV : (∀V:*. ∀A:*. PExp V A → PExp V A) =
  λV:*. → *. λA:*. λe:PExp V A.
  matchExp V A e (PExp V A)
  (constVar V A (PExp V A) e)
  (constAbs V A (PExp V A) e)
  (λB:*. λf : PExp V (B → A). λx : PExp V B.
   let f1 : PExp V (B → A) = evalV V (B → A) f in
   let def : PExp V A = app V B A f1 x in
   matchAbs V (B → A) (PExp V A) f1 def
   (λB1:*. λA1:*. λeq : Eq (B1 → A1) (B → A).
    λf : PExp V B1 → PExp V A1.
    let eqL : Eq B B1 = sym B1 B (arrL B1 A1 B A eq) in
    let eqR : Eq A1 A = arrR B1 A1 B A eq in
    let f' : PExp V B → PExp V A =
      eqR (PExp V) ∘ f ∘ eqL (PExp V)
    in evalV V A (f' x)))
  (constTAbs V A (PExp V A) e)
  (λB : *. λp : IsAll B. λi : Inst B A. λe1 : PExp V B.
   let e2 : PExp V B = evalV V B e1 in
   let def : PExp V A = tapp V B A p i e2 in
   matchTAbs V B (PExp V A) e2 def
   (λp : IsAll B. λs : StripAll B. λu : UnderAll B.
    λe3 : All Id (PExp V) B. evalV V A (i (PExp V) e3)))
  (constFold V A (PExp V A) e)
  (λF : (* → *) → * → *. λB : *.
   λeq : Eq (F (μ F) B) A. λe1 : PExp V (μ F B).
   let e2 : PExp V (μ F B) = evalV V (μ F B) e1 in
   let def:PExp V A = eq (PExp V) (unfld V F B e2) in
   matchFold V (μ F B) (PExp V A) e2 def
   (λF1 : (* → *) → * → *. λB1 : *.
    λeq1 : Eq (μ F1 B1) (μ F B).
    λe3 : PExp V (F1 (μ F1) B1).
    let eq2 : Eq (F1 (μ F1) B1) A =
      trans (F1 (μ F1) B1) (F (μ F) B) A
      (eqUnfold F1 B1 F B eq1) eq
    in evalV V A (eq2 (PExp V) e3)))

decl eval : (∀A:*. Exp A → Exp A) =
  λA:*. λe:Exp A. λV:*. → *. evalV V A (e V);

```

Figure 13: Weak head normal self-evaluator for $F_{\omega}^{\mu i}$

get e3 of type All Id (PExp V) B , where B is some quantified type. We also know that A is an instance of B , witnessed by the instantiation function i of type Inst B A . We use i to reduce the redex, instantiating $e3$ to PExp V A . Then, as before, we continue evaluating the result.

If the input term of type A is an `unfold`, then the head position subexpression $e1$ has the recursive type $\mu F B$, and we get a proof that A is equal to the unfolding of $\mu F B$. If $e1$ evaluates to a `fold`, then we are given proofs that it has a recursive type, which we already knew in this case, and a subexpression $e3$ of the unfolded type. The `unfold` expression reduces to $e3$, and we use transitivity to construct a proof to cast $e3$ to PExp V A , and continue evaluation.

Single-step left-most reduction. Left-most reduction is a restriction of the reduction rules shown in Figure 7. It never evaluates under a λ abstraction, a Λ abstraction, or a fold in a redex, and only evaluates the argument of an application if the function is a normal form and the application is not a redex.

Our implementation of left-most reduction has the same type as our weak head evaluator, but differs in that it reduces at most one redex, possibly under abstractions. The top-level function `step` operates only on closed terms. It has the same type as `eval`, $(\forall T:*. \text{Exp } T \rightarrow \text{Exp } T)$. The main driver is `stepV`, which has the type $(\forall V:*. \rightarrow$

```

PNeExp : (* → *) → * → *
PNfExp : (* → *) → * → *

Sem : (* → *) → * → *

decl NfExp : * → * = λT:*. ∀V:*. → *. PNfExp V T;

sem : (∀V:*. → *. ∀T:*. Exp T → Sem V T)
reify : (∀V:*. → *. ∀T:*. Sem V T → PNfExp V T)

decl nbe : (∀T:*. Exp T → NfExp T) =
  λT:*. λe:Exp T. λV:*. → *.
  reify V T (sem V T e);

unNf : (∀T:*. NfExp T → Exp T)

decl norm : (∀T:*. Exp T → Exp T) =
  λT:*. λe:Exp T. unNf T (nbe T e);

```

Figure 14: Highlights of our NbE implementation.

$\forall T:*. \text{PExp (PExp V) } T \rightarrow \text{PExp V } T$). Its input is a representation of type $\text{PExp (PExp V) } T$, which can contain free variables of PExp V types. In other words, free variables are themselves representations. This is a key to evaluating under abstractions. When going under an abstraction, we use the `var` constructor to tag the variables, so `stepV` can detect them. As `stepV` walks back out of the representation, it removes the `var` tags.

When evaluating an application, there are three possibilities: either the head subexpression is a λ -abstraction, in which case `stepV` reduces the β -redex, or the head subexpression can take a step, or the head expression is normal, in which case `stepV` steps the argument. `stepV` relies on a normal-form checker to decide whether to step the head or the argument subexpression.

Normalization by evaluation. We can use `step` and a normal-form checker to define a normalizer, by repeatedly stepping a representation until a normal form is reached. This is quite inefficient, though, so we also implement an efficient normalizer using the technique of Normalization by Evaluation (NbE). The implementation of NbE is outlined in Figure 14. The top-level function `norm` has the same type $(\forall T:*. \text{Exp } T \rightarrow \text{Exp } T)$ as `eval` and `step`. The main driver is `nbe`, which maps closed terms to closed normal forms of type NfExp T . The type NfExp is a PHOAS representation similar to `Exp`, except that it only represents normal form terms. The type PNfExp is defined by mutual recursion with PNeExp , which represents normal and neutral terms – normal form terms that can be used in head position without introducing a redex. For example, if $f : A \rightarrow B$ is normal and neutral, and x is normal, then $f x$ is normal and neutral. See Figure 7 for a grammar of normal and neutral terms.

We also define a semantic domain Sem V T and a function `sem` that `nbe` uses to map representations into the semantic domain. The semantic domain has the property that, if $e1 \equiv e2$, and $q1$ and $q2$ are pre-representations of $e1$ and $e2$ respectively, then $\text{sem V T } q1 \equiv \text{sem V T } q2$. The function `reify` maps semantic terms of type Sem V T to normal form representations of type PNfExp V T . Since normal forms are a subset of expressions, the function `unNf` can convert normal form representations of type NfExp T to representations of type Exp T .

The type of `nbe` ensures that it maps normalizing terms to their normal form and preserves types. Our `nbe` is not type directed, so it does not produce η -long normal forms. This is sometimes called “untyped normalization by evaluation” [4, 18], though this conflicts with our nomenclature of calling a meta-program typed or untyped to

```

foldExp :
  ∀R : * → *.
  (∀A:*. ∀B:*. (R A → R B) → R (A → B)) →
  (∀A:*. ∀B:*. R (A → B) → R A → R B) →
  (∀A:*. IsAll A → StripAll A → UnderAll A →
   All Id R A → R A) →
  (∀A:*. ∀B:*. IsAll A → Inst A B → R A → R B) →
  (∀F: (*→*)→*→*. ∀A:*. R (F (μ F) A) → R (μ F A)) →
  (∀F: (*→*)→*→*. ∀A:*. R (μ F A) → R (F (μ F) A)) →
  ∀A:*. Exp A → R A

```

Figure 15: Interface for defining folds over representations.

indicate whether it operates on typed or untyped abstract syntax. We call our NbE typed, but not type-directed.

7. Benchmarks and Experiments

In this section we discuss our benchmark meta-programs, our implementation, and our experiments.

To evaluate the expressive power of our language and representation, we reimplemented the meta-programs from our previous work [8] in $F_{\omega}^{\mu i}$. We type check and test our evaluators and benchmark meta-programs using an implementation of $F_{\omega}^{\mu i}$ in Haskell. The implementation includes a parser, type checker, evaluator, and equivalence checker. In particular, we tested that our self-evaluators are self-applicable – they can be applied to themselves.

Benchmark meta-programs. In previous work [8], we implemented a suite of self-applicable meta-programs for F_{ω} , including a self-interpreter and a continuation-passing-style transformation. We reimplemented all of their meta-programs for $F_{\omega}^{\mu i}$. They are defined as folds over the representation, so in order to align our reimplementation as closely as possible to the originals, we also implemented a general fold function for our representation.

Figure 15 shows the type of our general purpose foldExp function. It is a recursive function that takes six fold functions, one for each form of expression other than variables, which are applied uniformly throughout the representation. The type R determines the result type of the fold. We also instantiate V to R, so we can use var to embed partial results of the fold into the representation.

The type of foldExp is reminiscent of the Exp type used in our previous work [8], which is defined by its fold. Notable differences are the addition of fold functions for fold and unfold, and our improved treatment of polymorphic types using Typecase.

We implemented a self-recognizer unquote that recovers a term from its representation. It has the type $\forall A:*. \text{Exp } A \rightarrow A$, and is defined by a fold with $R = \text{Id}$, the identity function on types. unquote uses IsAll proofs in a way we haven’t seen so far. The fold function for type abstractions gets a term of type $\text{All Id Id } A$. When A is a concrete quantified type $\forall X:K. T$, this is equivalent to A. However, the fold function is defined for an abstract quantified type A. It uses the UnAll A component of an IsAll A proof to convert $\text{All Id Id } A$ to A.

A continuation-passing-style (CPS) transformation makes evaluation order explicit and gives a name to each intermediate value in the computation. It also transforms the type of the term in a nontrivial way – the result type is expressed as a recursive function on the input type. The type of our typed call-by-name CPS transformation is shown in Figure 16. Previous implementations of typed CPS transformation [7, 8, 31] use type-level representations of types in order to express this relationship. The type representations were designed to support the kind of function needed to typecheck CPS. A challenge of this approach is that the encoded types should have the same equivalences as regular types. That is, if two types A and B are equivalent,

then their encodings should be as well. In previous work, we used a nonstandard encoding of types to ensure this [8].

In this work, we do not encode types. Instead we combine recursive types and Typecase in a new way to express the type of our CPS transformation. Intuitively, CPS is an iso-recursive intensional type function. The specification for the type CPS is given below, and its definition in $F_{\omega}^{\mu i}$ is shown in Figure 16. $T_1 \cong T_2$ denotes that the types T1 and T2 are isomorphic, witnessed by unfold and fold. A value of type Ct T is function that takes a continuation and calls that continuation with an argument of type T.

$$\begin{aligned}
 \text{CPS } (A \rightarrow B) &\cong \text{Ct } (\text{CPS } A \rightarrow \text{CPS } B) \\
 \text{CPS } (\forall X:K. T) &\cong \text{Ct } (\forall X:K. \text{CPS } T) \\
 \text{CPS } (\mu F A) &\cong \text{Ct } (\text{CPS } (F (\mu F) A))
 \end{aligned}$$

Like unquote, cps uses IsAll proofs in an interesting new way. It is defined as a fold, and the case function for type abstractions is given an $\text{All Id CPS } A$, which it needs to cast to $\text{CPS1F CPS1 } A$, the unfolding of $\text{CPS1 } A$. $\text{All Id CPS } A$ and $\text{CPS1F CPS1 } A$ are both Type-case types, and while their cases for quantified types are the same, the cases for arrow types and recursive types are different. This is where the $\text{TcAll } A$ component of the $\text{IsAll } A$ proof is useful. Since we know A is a quantified types, the Typecase cases for arrow types and recursive types are irrelevant. The function eqCPSAll uses TcAll A to prove $\text{CPS1F CPS1 } A$ and $\text{All Id CPS } A$ are equal.

```

decl eqCPSAll : (∀A:*. IsAll A →
  Eq (CPS1F CPS1 A) (All Id CPS A)) =
  ΛA:*. λp : IsAll A.
  fst p (λA1:*. λA2:*. CPS A1 → CPS A2)
  Id CPS
  (λF: (*→*)→*→*. λB:*. CPS (F (μ F) B));

```

We also implement the other meta-programs from our previous work: a size measure, a normal form checker, and a top-level syntactic form checker. The complete code for all our meta-programs is provided in the appendix. The size measure demonstrates the use of our strip functions to remove redundant quantifiers. Below is the fold function given to foldExp for type abstractions:

```

decl sizeTAb : FoldTAb (λT:*. Nat) =
  ΛA:*. λp:IsAll A. λs:StripAll A.
  λu:UnderAll A. λf:All Id (λT:*.Nat) A.
  succ (s Nat f);

```

Here, A is some unknown quantified type, and f holds the result of the recursive call to size on the body of the type abstraction. The size of the type abstraction is one more than the size of its body, so sizeTAb needs to apply the successor function to the result of the recursive call. However, its type $\text{All Id } (\lambda T:*. \text{Nat})$ A is different than Nat. For example, if $A = (\forall X:K. A')$, then $\text{All Id } (\lambda T:*. \text{Nat}) A \equiv (\forall X:K. \text{Nat})$. The quantifier on X is redundant, and blocks sizeTAb from accessing the result of the recursive call. By removing the redundant quantifier, the strip function s is instrumental in programming size on representations of polymorphic terms.

Implementation. We have implemented System $F_{\omega}^{\mu i}$ in Haskell. The implementation includes a parser, type checker, quoter, evaluator (which does the *evaluation* in Figure 1), and an equivalence checker. Our evaluator is based on NbE similar to Figure 14, except that it operates on untyped first-order abstract syntax based on De-Brujn indices. Our self-evaluators and other meta-programs have been implemented, type checked and tested. Our parser includes special syntax for building quotations and normalizing terms, which is useful for testing. We use [e] to denote the representation of e, and <e> to denote the normal form of e. The normalization of <e> expressions occurs after type checking, but before quotation. Thus [<e>] denotes the representation of the normal form of e.

```

decl Ct : * → * = λA:*. ∀B:*. (A → B) → B;
decl CPS1 : * → * =
  μ (λCPS1:*. → *. λT:*.
    Typecase
      (λA:*. λB:*. Ct (CPS1 A) → Ct (CPS1 B))
      Id (λT:*. Ct (CPS1 T))
      (λF: (*→*)→*→*. λT:*. Ct (CPS1 (F (μ F) T)))
      T);
decl CPS : * → * = λT:*. Ct (CPS1 T);

cps : (∀T:*. Exp T → CPS T)

```

Figure 16: Type of our CPS transformation

We test our meta-programs using functions on natural numbers, which use all the features of the language: recursive types, recursive functions, and polymorphism. We encode natural numbers using a typed Scott encoding [2, 43] that is similar to our encoding of $F_{\omega}^{\mu_i}$ terms. Compared to other encodings, Scott-encoded natural numbers support natural implementations of functions like predecessor, equality, and factorial.

We use our equivalence checker to test our meta-programs. It works by normalizing the two terms, and checking the results for syntactic equality up to renaming. For example, we test that our implementation of NbE normalizes the representation `[fact five]` to the representation of its normal form, `[<fact five>]`:

```
norm Nat [fact five] ≡ [<fact five>]
```

Self-application. Each of our evaluators is *self-applicable*, meaning that it can be applied to its own representation. In particular, the self-application of `eval` is written `eval (∀T:*. Exp T → Exp T) [eval]`. We have self-applied each of our evaluators, and tested the result. Here is an example, specifically for our weak head normal form evaluator:

```

decl eval' = unquote (∀T:*. Exp T → Exp T)
  (eval (∀T:*. Exp T → Exp T) [eval]);

eval' Nat [fact five] ≡ eval Nat [fact five]

```

We define `eval'` by applying `eval` to its representation `[eval]`, and then unquoting the result. In terms of Figure 1, we start with `eval` at the bottom-left corner, then move up to its representation `[eval]`, then right to the representation of its value (weak head normal form in this case) `(eval (∀T:*. Exp T → Exp T) [eval])`, and `unquote` recovers the value from its representation. Finally test that `eval'` and `eval` have the same behavior by testing that they map equal inputs to equal outputs.

8. Related Work

Typed self-representation. Pfenning and Lee [29] considered whether System F could support a useful notion of a self-interpreter, and concluded that the answer seemed to be “no”. They presented a series of typed representations, of System F in F_{ω} , and of F_{ω} in F_{ω}^+ , which extends F_{ω} with kind polymorphism. Whether typed self-representation is possible remained an open question until 2009, when Rendel, Ostermann and Hofer [31] presented the first typed-self representation. Their language was a typed λ -calculus F_{ω}^* that has undecidable type checking. They implemented a self-recognizer, but not a self-evaluator. Jay and Palsberg [22] presented a typed self-representation for a combinator calculus that also has undecidable type checking. Their representation supports a self-recognizer and a self-evaluator, but not with the types described in Section 1. In their

representation scheme, terms have the same type as their representations, and both their interpreters have the type $\forall T. T \rightarrow T$. In previous work we presented self-representations for System U [7], the first for a language with decidable type checking, and for F_{ω} [8], the first for a strongly normalizing language. Each of these supported self-recognizers and CPS transformations, but not self-evaluators.

There is some evidence that the problem of implementing a typed self-evaluator is more difficult than that of implementing a typed self-recognizer. For example, self-recognizers have been implemented in simpler languages than $F_{\omega}^{\mu_i}$, and based on simpler representation techniques. A self-recognizer implemented as a fold relies entirely on meta-level evaluation. The fact that meta-level evaluation is guaranteed to be type-preserving simplifies the implementation of a typed self-recognizer, but the evaluation strategy can only be what the meta-level implements. On the other hand, self-evaluators can control the evaluation strategy, but this requires more work to convince the type checker that the evaluation is type-preserving (e.g. by deriving type equality proofs).

Typed self-evaluation is an important step in the area of typed self-representation. It lays the foundation for other verifiably type-preserving program transformations, like partial evaluators or program optimizers. Our representation techniques can be used to explore for other applications such as typed Domain Specific Languages (DSLs), typed reflection, or multi-stage programming.

It remains an open problem to implement a self-evaluator for a strongly normalizing language without recursion. We use recursion in two ways in our evaluators: first, we use a recursive type for our representation, which has a negative occurrence in its `abs` constructor. Second, we use the fixpoint combinator to control the order of evaluation. This allows our evaluators to select a particular redex in a term to reduce. Previous work on typed-self representation only supported folds, which treat all parts of a representation uniformly.

Intensional type analysis. Intensional type analysis (ITA) was pioneered by Harper and Morrisett [21] for efficient implementation of ad-hoc polymorphism. Previous work on intensional type analysis has included an ITA operator in terms as well as types. Term-level ITA enables runtime type introspection (RTTI), and the primary role of type-level ITA has traditionally been to typecheck RTTI. RTTI is useful for dynamic typing [41], typed compilation [14, 25], garbage collection [37], and marshalling data [16]. ITA has been shown to support type-erasure semantics [14, 15], user-defined types [38], and a kind of parametricity theorem [27].

Early work on ITA was restricted to monotypes – base types, arrows, and products [21]. Subsequently, it was extended to handle polymorphic types [14], higher-order types [42], and recursive types [13]. Trifonov et al. presented λ_i^Q [37], which supports *fully-reflexive* ITA – analysis of all types of kind $*$, including quantified and recursive types.

The most notable difference between $F_{\omega}^{\mu_i}$ and previous languages with ITA is that $F_{\omega}^{\mu_i}$ does not include a term-level ITA operator, and thus does not support runtime type introspection. Our type-level Typecase operator is fully-reflexive, but we restrict the analysis on quantified types to avoid kind-polymorphism, which was used in λ_i^Q . Unlike our Typecase operator, the type-level ITA operator in λ_i^Q is recursive, which requires more complex machinery to keep type checking decidable.

Our Typecase operator is simpler than those from previous work on intensional type analysis. Also, by omitting the term-level ITA operator, we retain a simple semantics of $F_{\omega}^{\mu_i}$. In particular, the reduction of terms does not depend on types. This in turn simplifies our presentation, our self-evaluators and the proofs of our meta-theorems.

GADTs. Generalized algebraic data types (GADTs) were introduced independently by Cheney and Hinze [11] and Xi, Chen and Chen [46]. They applications include intensional type analysis [11,

39, 46] and typed meta-programming [20]. Traditional formulations of GADTs are designed to support efficient encodings, pattern matching, type inference, and/or type erasure semantics [36, 46]. In this work our focus has been to identify a core calculus and representation techniques that can support typed self-evaluation. While our representation is conceptually similar to a GADT, it is meta-theoretically much simpler than a traditional GADT. More work is needed to achieve self-representation and self-evaluation for a full language that includes efficient implementations of GADTs. One important question that needs to be answered is how to represent and evaluate programs that involve user-defined GADTs. For example, if we used a GADT for our self-representation, how would we represent the self-evaluators that operate on it?

Type equality. Type equality has been used to encode GADTs [11, 23, 36, 46], and for generic programming [10, 47], dynamic typing [5, 10, 41], typed meta-programming [28, 35], and simulating dependent types [9]. Some formulations of type equality are built-into the language in order to support type-erasure semantics [36] and type inference [35, 36, 46]. This comes at a cost of a larger and more complex language, which makes self-interpretation more difficult.

The use of polymorphism to encode Leibniz equality [5, 10, 41] is perhaps the simplest encoding technique, though it lacks support for erasure (leading to some runtime overhead) and type inference. Furthermore, without intensional type functions Leibniz equality is not expressive enough for defining typed evaluators, a limitation we have addressed in this paper. Our formulation of type equality has essentially no impact on the semantics, because the heavy lifting is done at the type level by Typecase.

9. Conclusion

We have presented $F_{\omega}^{\mu_i}$, a typed λ -calculus with decidable type checking, and the first language known to support typed self-evaluation. We use intensional type functions to implement type equality proofs, which we then use to define a typed self-representation in the style of Generalized Algebraic Data Types (GADTs). Our three polymorphically-typed self-evaluators implement weak head normal form evaluation, single-step left-most β -reduction, and normalization by evaluation (NbE). Our self-representation also supports all the benchmark meta-programs from previous work on typed self-representation.

We leave for future work the question of whether typed self-evaluation is possible for a language with support for efficient user-defined types.

Acknowledgments

We thank John Bender, Iris Cong, Christian Kalhauge, Oleg Kiselyov, Todd Millstein, and the POPL reviewers for helpful comments, discussions, and suggestions. This material is based upon work supported by the National Science Foundation under Grant Number 1219240.

References

- [1] The webpage accompanying this paper is available at <http://compilers.cs.ucla.edu/pop117/>. The full paper with the appendix is available there, as is the source code for our implementation of System $F_{\omega}^{\mu_i}$ and our operations.
- [2] Martin Abadi, Luca Cardelli, and Gordon Plotkin. Types for the scott numerals, 1993.
- [3] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.
- [4] Klaus Aehlig and Felix Joachimski. Operational aspects of untyped normalisation by evaluation. *Mathematical Structures in Computer Science*, 14:587–611, 8 2004.
- [5] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *ICFP '02: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 157–166. ACM Press, 2002.
- [6] Reg Braithwaite. The significance of the meta-circular interpreter. http://weblog.raganwald.com/2006/11/significance-of-meta-circular_22.html, November 2006.
- [7] 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.
- [8] Matt Brown and Jens Palsberg. Breaking through the normalization barrier: A self-interpreter for F-omega. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 5–17, New York, NY, USA, 2016. ACM.
- [9] Chiyen Chen, Dengping Zhu, and Hongwei Xi. *Implementing Cut Elimination: A Case Study of Simulating Dependent Types in Haskell*, pages 239–254. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
- [10] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 90–104, New York, NY, USA, 2002. ACM.
- [11] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.
- [12] 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.
- [13] Gregory D. Collins and Zhong Shao. Intensional analysis of higher-kinded recursive types. Technical report, Yale University, 2002.
- [14] Karl Crary and Stephanie Weirich. Flexible type analysis. In *In 1999 ACM International Conference on Functional Programming*, pages 233–248. ACM Press, 1999.
- [15] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. *SIGPLAN Not.*, 34(1):301–312, September 1998.
- [16] Dominic Duggan. *A type-based semantics for user-defined marshallings in polymorphic languages*, pages 273–297. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.
- [17] Brendan Eich. Narcissus. <http://mxr.mozilla.org/mozilla/source/js/narcissus/jsexec.js>, 2010.
- [18] Andrzej Filinski and Henning Korsholm Rohde. *A Denotational Account of Untyped Normalization by Evaluation*, pages 167–181. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
- [19] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
- [20] Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in Haskell. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 75–86, New York, NY, USA, 2008. ACM.
- [21] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 130–141, New York, NY, USA, 1995. ACM.
- [22] 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.
- [23] Arie Middelkoop, Atze Dijkstra, and S. Doaitse Swierstra. A lean specification for GADTs: System F with first-class equality proofs. *Higher Order Symbol. Comput.*, 23(2):145–166, June 2010.
- [24] Torben A.E. Mogensen. Efficient self-interpretations in lambda calculus. *Journal of Functional Programming*, 2(3):345–363, 1992. See also DIKU Report D-128, Sep 2, 1994.
- [25] Greg Morrisett. Compiling with types. Technical report, 1995.

- [26] Matthew Naylor. Evaluating Haskell in Haskell. *The Monad.Reader*, 10:25–33, 2008.
- [27] Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 135–148, New York, NY, USA, 2009. ACM.
- [28] Emir Pasalic. *The Role of Type Equality in Meta-programming*. PhD thesis, 2004. AAI3151199.
- [29] Frank Pfenning and Peter Lee. Metacircularity in the polymorphic λ -calculus. *Theoretical Computer Science*, 89(1):137–159, 1991.
- [30] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.
- [31] 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.
- [32] Armin Rigo and Samuele Pedroni. Pypy's approach to virtual machine construction. In *OOPSLA Companion*, pages 044–953, 2006.
- [33] Andreas Rossberg. HaMLet. <http://www.mpi-sws.org/~rossberg/hamlet>, 2010.
- [34] Bratin Saha, Valery Trifonov, and Zhong Shao. Intensional analysis of quantified types. *ACM Trans. Program. Lang. Syst.*, 25(2):159–209, 2003.
- [35] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. *Electron. Notes Theor. Comput. Sci.*, 199:49–65, February 2008.
- [36] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI'07, ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2007.
- [37] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. *SIGPLAN Not.*, 35(9):82–93, September 2000.
- [38] Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In *Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '05, pages 13–24, New York, NY, USA, 2005. ACM.
- [39] Dimitrios Vytiniotis and Stephanie Weirich. Parametricity, type equality, and higher-order polymorphism. *Journal of Functional Programming*, 20(02):175–210, 2010.
- [40] 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.
- [41] Stephanie Weirich. Type-safe cast: (functional pearl). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 58–67, New York, NY, USA, 2000. ACM.
- [42] Stephanie Weirich. Higher-order intensional type analysis. In *Proceedings of the 11th European Symposium on Programming Languages and Systems*, ESOP '02, pages 98–114, London, UK, UK, 2002. Springer-Verlag.
- [43] Wikipedia. Mogensen-Scott encoding. https://en.wikipedia.org/wiki/Mogensen-Scott_encoding.
- [44] Wikipedia. Rubinius. <http://en.wikipedia.org/wiki/Rubinius>, 2010.
- [45] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
- [46] Hongwei Xi, Chiyen Chen, and Gang Chen. Guarded recursive datatype constructors. In *ACM SIGPLAN Notices*, volume 38, pages 224–235. ACM, 2003.
- [47] Zhe Yang. Encoding types in ML-like languages. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 289–300, New York, NY, USA, 1998. ACM.