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 $F_\omega^{\mu i}$, 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 metaprogramming [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, (String $\rightarrow$ 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.

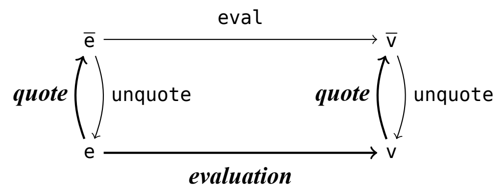


Figure 1: Self-recognizers and self-evaluators.

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:

$$\text{(self-recognizer)} \quad \texttt{unquote} : \forall \texttt{T}.\ \texttt{Exp T} \rightarrow \texttt{T} \qquad (1)$$

$$\text{(self-evaluator)} \quad \texttt{eval} : \forall \texttt{T}.\ \texttt{Exp T} \rightarrow \texttt{Exp T} \qquad (2)$$

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 $\overline{\texttt{e}}$. 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 $\overline{\texttt{e}}$ 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, $F_\omega^{\mu i}$, 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 → B) = (S → 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:

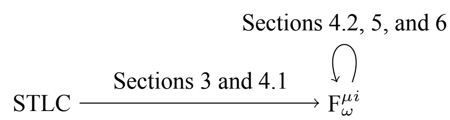
- We support ITA at the type level only, while previous work supports ITA in types and terms.
- Our `Typecase` operator is not recursive. Previous work used a recursive `Typerec` operator for type-level ITA.
- We support ITA of quantified types without using kind polymorphism.

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 `(Exp t1 → Exp t2)` corresponds to a STLC term of type `Exp t2` that includes a free variable of type `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 = Abs f`, then `e` is already in weak head-normal form, and `eval e = e`. If `e = App e1 e2`, we first recursively evaluate `e1`, letting `e1'` be value of `e1`. If `e1'` is an abstraction `Abs f`, then `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 `App e1' e2`.

We now consider how Haskell type checks `eval`. First, the type annotation on `eval` determines that `App e1 e2` has type `Exp t`. According to the type of `App`, `e1` has type `Exp (t1 → t)` and `e2` has type `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 `Exp (t1 → t)`. If case analysis finds that `e1'` is of the form `Abs f`, then the type of `Abs` tells us that `f` has the type `Exp t1 → Exp t`.

We can see that Haskell's type checker does some nontrivial work to typecheck code with GADTs. Pattern matching `App e1 e2` introduced the existentially quantified type `t1`. When pattern matching determined that `e1'` is of the form `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 `Exp t`, while in the GADT version the result type of `Abs` is `Exp (t1 → t2)`. The type equality constraint `(t1 → t2) ~ t` makes up this difference. Haskell implements GADTs using ADTs and equality constraints [36], so the definitions of `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 `Eq t1 t2`, `eqApp` can derive a proof that `Exp t1` is equal to `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 `Eq t1 t2`, `coerce` can change the type of a term from `t1` into `t2`.

Given a closed proof `p` of type `Eq t1 t2`, we expect (1) that it is true that `t1` and `t2` are equal types, and (2) that `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 hypothesis are discharged, `coerce p e` should not be reducible to `e`.

ADTs and explicit type equality proofs. Figure 5 shows a version of `Exp t` and an evaluator that uses ADTs and explicit type equality proofs. The only difference between this definition of `Exp t` and the one in Figure 3 is that we have replaced the type equality constraint `(t1 → t2) ~ t` with a type equality proof of type `Eq (t1 → 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 `Exp t`, and `e1` has type `Exp (t1 → t)` and `e2` has type `Exp t1`, for some type `t1`. Since `eval` preserves type, `e1'` also has type `Exp (t1 → t)`.

The differences begin with the pattern match on `e1'`. If `e1'` is of the form `Abs eq f`, then there exist types `s1` and `s2` such that `eq` has the type `Eq (s1 → s2) (t1 → t)` and `f` has the type `Exp s1 → Exp s2`. We use `arrL`, `sym`, and `eqApp` (with `f` instantiated with `Exp`) to derive `eqL`, which has the type `Eq (Exp t1) (Exp s1)`. Similarly, we use `arrR` and `eqApp` to derive `eqR` with the type `Eq (Exp s2) (Exp t)`. Finally, we use coercions based on `eqL` and `eqR` to cast `f` from the type `Exp s1 → Exp s2` to the type `Exp t1 → Exp t`. Thus, `f'` can be applied to `e2`, and its result has type `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 `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 `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` newtype 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` $\longrightarrow$ `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 `(λx:T.e2) e1`, as usual.
- `let rec x : T1 = e1 in e2` desugars to `let x : T1 = fix T1 (λx:T1. e1) in e2`. Here `fix` is a standard fixpoint combinator of type `∀T:*. (T → T) → 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 `(λx.e)` instead of `(λx:T.e)`. We use `f ∘ g` to denote the composition of (type or term) functions `f` and `g`. This desugars to `(λx. f (g x))`, where `x` is fresh.

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

$$
\begin{array}{rl}
\text{(normal form terms)} & v ::= n \mid (\lambda x{:}T.v) \mid (\Lambda X{:}K.v) \mid \texttt{fold}\ T_1\ T_2\ v \\
\text{(neutral terms)} & n ::= x \mid n\ v \mid n\ T \mid \texttt{unfold}\ T_1\ T_2\ n \\
\end{array}
$$

Grammar

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

$$\frac{\Gamma \vdash T_1 : * \qquad \Gamma \vdash T_2 : *}{\Gamma \vdash T_1 \rightarrow T_2 : *} \qquad \frac{\Gamma,(X{:}K) \vdash T : *}{\Gamma \vdash (\forall X{:}K.T) : *}$$

$$\frac{\Gamma,(X{:}K_1) \vdash T : K_2}{\Gamma \vdash (\lambda X{:}K_1.T) : K_1 \rightarrow K_2} \qquad \frac{\Gamma \vdash T_1 : K_2 \rightarrow K \qquad \Gamma \vdash T_2 : K_2}{\Gamma \vdash T_1\ T_2 : K}$$

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

$$
\begin{array}{l}
\Gamma \vdash \texttt{Typecase} : (* \rightarrow * \rightarrow *) \rightarrow \\
\quad (* \rightarrow *) \rightarrow (* \rightarrow *) \rightarrow \\
\quad (((* \rightarrow *) \rightarrow * \rightarrow *) \rightarrow * \rightarrow *) \rightarrow \\
\quad *
\end{array}
$$

Type Formation

$$T \equiv T \qquad \frac{T1 \equiv T2}{T2 \equiv T1} \qquad \frac{T1 \equiv T2 \qquad T2 \equiv T3}{T1 \equiv T3}$$

$$\frac{T1 \equiv T1' \qquad T2 \equiv T2'}{T1 \rightarrow T2 \equiv T1' \rightarrow T2'} \qquad \frac{T \equiv T'}{(\forall X{:}K.T) \equiv (\forall X{:}K.T')}$$

$$\frac{T \equiv T'}{(\lambda X{:}K.T) \equiv (\lambda X{:}K.T')} \qquad \frac{T1 \equiv T1' \qquad T2 \equiv T2'}{T1\ T2 \equiv T1'\ T2'}$$

$$
\begin{array}{r}
(\lambda X{:}K.T1)\ T2 \equiv (T1[X := T2]) \\
(\forall X{:}K.T2) \equiv (\forall X'{:}K.T2[X := X']) \\
(\lambda X{:}K.T) \equiv (\lambda X'{:}K.T[X := X'])
\end{array}
$$

$$
\begin{array}{r}
\texttt{Typecase}\ F1\ F2\ F3\ F4\ (T1 \rightarrow T2) \equiv F1\ T1\ T2 \\
\texttt{Typecase}\ F1\ F2\ F3\ F4\ (\mu\ T1\ T2) \equiv F4\ T1\ T2
\end{array}
$$

$$\frac{X \notin FV(F3)}{\texttt{Typecase}\ F1\ F2\ F3\ F4\ (\forall X{:}K.T) \equiv F2\ (\forall X{:}K.\ F3\ T)}$$

Type Equivalence

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash e : T_1 \qquad T_1 \equiv T_2 \qquad \Gamma \vdash T_2 : *}{\Gamma \vdash e : T_2}$$

Term Formation

$$
\begin{array}{r}
(\lambda x{:}T.e)\ e_1 \longrightarrow e[x := e_1] \\
(\Lambda X{:}K.e)\ T \longrightarrow e[X := T] \\
\texttt{unfold}\ F\ T\ (\texttt{fold}\ F'\ T'\ e) \longrightarrow e
\end{array}
$$

$$\frac{e_1 \longrightarrow e_2}{\begin{array}{c} e_1\ e_3 \longrightarrow e_2\ e_3 \\ e_3\ e_1 \longrightarrow e_3\ e_2 \\ e_1\ T \longrightarrow e_2\ T \\ (\lambda x{:}T.e_1) \longrightarrow (\lambda x{:}T.e_2) \\ (\Lambda X{:}K.e_1) \longrightarrow (\Lambda X{:}K.e_2) \\ \texttt{fold}\ F\ T\ e_1 \longrightarrow \texttt{fold}\ F\ T\ e_2 \\ \texttt{unfold}\ F\ T\ e_1 \longrightarrow \texttt{unfold}\ F\ T\ e_2 \end{array}}$$

Reduction

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

```
decl ⊥ : * = (∀T:*. T);
decl ArrL : ∗ → ∗ =
 Typecase (λA:∗. λB:∗. A) (λA:∗.⊥) (λA:∗.⊥)
          (λF:(∗ → ∗) → ∗ → ∗. λA:∗. ⊥);
decl ArrR : ∗ → ∗ =
 Typecase (λA:∗. λB:∗. B) (λA:∗.⊥) (λA:∗.⊥)
          (λF:(∗ → ∗) → ∗ → ∗. λA:∗. ⊥);
decl All : (∗ → ∗) → (∗ → ∗) → ∗ → ∗ =
 λOut:∗ → ∗. λIn:∗ → ∗.
 Typecase (λA:∗. λB:∗. B) Out In
          (λF:(∗ → ∗) → ∗ → ∗. λA:∗. ⊥);
decl Unfold : ∗ → ∗ =
 Typecase (λA:∗. λB:∗. ⊥) (λA:∗.⊥) (λA:∗.⊥)
          (λF:(∗ → ∗) → ∗ → ∗. λA:∗. F (μ F) A);
```

Figure 8: Intensional type functions

We use `S × T` for pair types, which can be easily encoded in System $F_\omega^{\mu i}$. Intuitively, `×` is an infix type function of kind $* \to * \to *$. 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 `T1` → `T2`, `Typecase` applies the case function for arrows to `T1` and `T2`. When applied to a recursive type `μ F T`, `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 `∀X:K.T` into the kind `K` and a type function of kind `K` → ∗. 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 = Typecase Arr Out 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 (∀X:K.T)` ≡ `Out (∀X:K. 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 `⊥=(∀T:*.T)`, which we use to indicate an error. The type ⊥ 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):

$$\mathtt{ArrL\ T} \equiv \begin{cases} \mathtt{T1} & \text{if } \mathtt{T} \equiv \mathtt{T1} \to \mathtt{T2} \\ \bot & \text{otherwise} \end{cases}$$

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

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

```
decl Eq : ∗ → ∗ =
 λA:∗. λB:∗. ∀F:∗ → ∗. F A → F B;

decl refl : (∀A:∗. Eq A A) =
 ΛA:∗. ΛF:∗ → ∗. λx : F A. x;

decl sym : (∀A:∗. ∀B:∗. Eq A B → Eq B A) =
 ΛA:∗. ΛB:∗. λeq : Eq A B.
 let p : Eq A A = refl A in
 eq (ΛT:∗. Eq T A) p;

decl trans : (∀A:∗. ∀B:∗. ∀C:∗.
               Eq A B → Eq B C → Eq A C) =
 ΛA:∗. ΛB:∗. ΛC:∗. λeqAB:Eq A B. λeqBC:Eq B C.
 ΛF:∗ → ∗. λx:F A. eqBC F (eqAB F x);

decl eqApp : (∀A:∗. ∀B:∗. ∀F:∗ → ∗.
              Eq A B → Eq (F A) (F B)) =
  ΛA:∗. ΛB:∗. ΛF:∗ → ∗. λeq : Eq A B.
  let p : Eq (F A) (F A) = refl (F A) in
  eq (λT:∗. Eq (F A) (F T)) p;

decl arrL : (∀A1:∗. ∀A2:∗. ∀B1:∗. ∀B2:∗.
              Eq (A1 → A2) (B1 → B2) →
              Eq A1 B1) =
 ΛA1 A2 B1 B2. eqApp (A1→A2) (B1→B2) ArrL;

decl arrR : (∀A1:∗. ∀A2:∗. ∀B1:∗. ∀B2:∗.
              Eq (A1 → A2) (B1 → B2) →
              Eq A2 B2) =
 ΛA1 A2 B1 B2. eqApp (A1→A2) (B1→B2) ArrR;

decl coerce : (∀A:∗. ∀B:∗. Eq A B → A → B) =
 ΛA:∗. ΛB:∗. λeq: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:

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

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 `∀F:∗ → ∗. F A → 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 `Eq B A`. Transitivity is encoded by `trans`, which uses function composition to combine two coercions. A proof of type `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 `μ ExpF`, which re-ties the knot. Now `Exp T` and `ExpF Exp T` are isomorphic, with `unfold ExpF T` converting from `Exp T` to `ExpF Exp T`, and `fold ExpF T` converting from `ExpF Exp T` back to `Exp T`. We define `matchExp` as a convenience and to align with Figure 6, but we could as well use `unfold ExpF T` instead of `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);
```

$\texttt{tcAll}_{X,K,T}$ = `ΛArr.ΛOut.ΛIn.ΛMu.refl (Out(∀X:K.In T))`
$\texttt{unAll}_{X,K,T}$ = `ΛOut.refl (Out (∀X:K.T))`
$\texttt{isAll}_{X,K,T}$ = ($\texttt{tcAll}_{X,K,T}$, $\texttt{unAll}_{X,K,T}$)

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 `Eq (T1 → T2) T`. This will work for any arrow type because `T1` and `T2` must have kind ∗ in order for `T1 → T2` to kind check. The case for recursive types is similar. In $F_\omega^{\mu i}$, a recursive type `μ F A` kind checks only if `F` has kind `(∗ → ∗) → ∗ → ∗` 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 `Eq (μ F A) T`. Given a proof that one recursive type `μ F1 A1` is equal to another `μ 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) (F2 (μ F2) 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 `IsAll T` consists of a pair of *polymorphic* equality proofs about `Typecase`. The first is of type `TcAll T` and proves that because `T` is a quantified type, `Typecase Arr Out In Mu T` is equal to `All Out 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 ⊥ functions used by `All`. The second polymorphic equality proof, of type `UnAll T`, shows that `All Out (λA:∗.A) T` is equal to `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 $\texttt{tcAll}_{X,K,T}$, $\texttt{unAll}_{X,K,T}$, and $\texttt{isAll}_{X,K,T}$. These are meta-level abbreviations, not part of $F_\omega^{\mu i}$. The type of $\texttt{tcAll}_{X,K,T}$ is `TcAll (∀X:K.T)`, the type of $\texttt{unAll}_{X,K,T}$ is `UnAll (∀X:K.T)`, and the type of $\texttt{isAll}_{X,K,T}$ is `IsAll (∀X:K.T)`. Note that $\texttt{tcAll}_{X,K,T}$ and $\texttt{unAll}_{X,K,T}$ use `refl` to create the equality proof. In the case of $\texttt{unAll}_{X,K,T}$, the proof `refl (Out (∀X:K.T))` has type `Eq (Out (∀X:K.T)) (Out (∀X:K.T))`, which is equivalent to the type `Eq (All Out (λA:∗.A) (∀X:K.T)) (Out (∀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 `t1` $\to$ `t2`, 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 $\bot$:

```
eqArrMu  : ∀A B F T. Eq (A → B) (μ F T) → ⊥
arrIsAll : ∀A B. IsAll (A → B) → ⊥
muIsAll  : ∀F T. IsAll (μ F T) → ⊥
```

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 `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 $\mathtt{stripAll}_\mathtt{K}$, $\mathtt{underAll}_\mathtt{X,K,T}$, and $\mathtt{inst}_\mathtt{X,K,T,S}$ 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 $\mathtt{stripAll}_\mathtt{K}$ has the type `StripAll (∀X:K.T)`, as long as `(∀X:K.T)` is well-typed. For any type `A` in which `X` does not occur free, $\mathtt{stripAll}_\mathtt{K}$ can map `All Id (λB:*.A) (∀X:K.T)` $\equiv$ `(∀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 $\mathtt{stripAll}_\mathtt{K}$. It uses the kind inhabitant $\mathtt{T}_\mathtt{K}$ for the instantiation. For each kind `K`, $\mathtt{T}_\mathtt{K}$ is a closed type of kind `K`.

The function $\mathtt{underAll}_\mathtt{X,K,T}$ has the type `UnderAll (∀X:K.T)`. It can apply a function under the quantifier of a type `All Id F1 (∀X:K.T)` $\equiv$ `(∀X:K. F1 T)` to produce a result of type `All Id F2 (∀X:K.T)` $\equiv$ `(∀X:K. F2 T)`. In particular, our evaluators use $\mathtt{underAll}_\mathtt{X,K,T}$ to make recursive calls on the body of a type abstraction. The representation of a type abstraction `(ΛX:K.e)` of type `(∀X:K.T)` contains the term `(ΛX:K.q)`. Here, `q` is the representation of `e`, in which the type variable `X` can occur free. The type of `(ΛX:K.q)` is `All Id (PExp V) (∀X:K.T)` $\equiv$ `(∀X:K. PExp V T)`.

We can use $\mathtt{underAll}_\mathtt{X,K,T}$ and $\mathtt{stripAll}_\mathtt{K}$ 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 $\mathtt{underAll}_\mathtt{X,K,T}$ to make the recursive call to `size` under the quantifier. The result of $\mathtt{underAll}_\mathtt{X,K,T}$ has the type `All Id (λY:*. Nat) (∀X:K.T)` $\equiv$ `(∀X:K.Nat)` where the quantification of `X` is redundant. We can then use $\mathtt{strip}_\mathtt{K}$ to strip away the redundant quantifier, enabling us to access the `Nat` underneath.

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

The combination of `IsAll` proofs and the utility functions $\mathtt{stripAll}_\mathtt{K}$, $\mathtt{underAll}_\mathtt{X,K,T}$, and $\mathtt{inst}_\mathtt{X,K,T,S}$ 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* $\langle\rangle \vdash \bar{\mathtt{e}}$ `: 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* $\bar{\mathtt{e}}$ *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 unfold/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 `e1` has type `PExp V B`. If `e1` evaluates to a type abstraction, we

```
T∗ = (∀X:∗.X)        T_{K1→K2} = λX: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;

underAll_{X,K,T} =
 ΛF1. ΛF2. λf : (∀A:∗. F1 A → F2 A).
 λe : (∀X:K. F1 T). ΛX:K. f T (e X);
stripAll_K = ΛA. λe:(∀X:K.A). e T_K
inst_{X,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 (PExpF 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 (PExpF 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 (PExpF 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 (PExpF 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 (PExpF 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 (PExpF 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 (PExpF 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 → PExpF (PExp V) A) =
 ΛV:∗→∗.ΛA:∗.λe:PExp V A. unfold (PExpF V) A e
```

Constructors and match for PExp

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

$$\frac{\Gamma \vdash \mathtt{T1} : * \qquad \Gamma,(\mathtt{x}:\mathtt{T1}) \vdash \mathtt{e} : \mathtt{T2} \rhd \mathtt{q}}{\Gamma \vdash (\lambda \mathtt{x}:\mathtt{T1.e}) : \mathtt{T1} \to \mathtt{T2} \rhd \mathtt{abs\ V\ T1\ T2}\ (\lambda \mathtt{x}:\mathtt{PExp\ V\ T1.\ q})}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_2 \to \mathtt{T} \rhd \mathtt{q}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2 \rhd \mathtt{q}_2}{\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{T} \rhd \mathtt{app\ V\ T}_2\ \mathtt{T\ q}_1\ \mathtt{q}_2}$$

$$\frac{\Gamma,(\mathtt{X}:\mathtt{K}) \vdash \mathtt{e} : \mathtt{T} \rhd \mathtt{q} \qquad \mathtt{isAll}_{X,K,T} = \mathtt{p} \quad \mathtt{stripAll}_K = \mathtt{s} \quad \mathtt{underAll}_{X,K,T} = \mathtt{u}}{\Gamma \vdash (\Lambda \mathtt{X}:\mathtt{K.e}) : (\forall \mathtt{X}:\mathtt{K.T}) \rhd \mathtt{tabs\ V}\ (\forall \mathtt{X}:\mathtt{K.T})\ \mathtt{p\ s\ u}\ (\Lambda \mathtt{X}:\mathtt{K.q})}$$

$$\frac{\Gamma \vdash \mathtt{e} : (\forall \mathtt{X}:\mathtt{K.T}) \rhd \mathtt{q} \quad \mathtt{isAll}_{X,K,T} = \mathtt{p} \quad \Gamma \vdash \mathtt{A} : \mathtt{K} \quad \mathtt{inst}_{X,K,T,A} = \mathtt{i}}{\Gamma \vdash \mathtt{e\ A} : \mathtt{T[X:=A]} \rhd \mathtt{tapp\ V}\ (\forall \mathtt{X}:\mathtt{K.T})\ (\mathtt{T[X:=A]})\ \mathtt{p\ i}}$$

$$\frac{\Gamma \vdash \mathtt{F} : (* \to *) \to * \to * \quad \Gamma \vdash \mathtt{T} : * \quad \Gamma \vdash \mathtt{e} : \mathtt{F}\ (\mu\ \mathtt{F})\ \mathtt{T} \rhd \mathtt{q}}{\Gamma \vdash \mathtt{fold\ F\ T\ e} : \mu\ \mathtt{F\ T} \rhd \mathtt{fld\ V\ F\ T\ q}}$$

$$\frac{\Gamma \vdash \mathtt{F} : (* \to *) \to * \to * \quad \Gamma \vdash \mathtt{T} : * \quad \Gamma \vdash \mathtt{e} : \mu\ \mathtt{F\ T} \rhd \mathtt{q}}{\Gamma \vdash \mathtt{unfold\ F\ T\ e} : \mathtt{F}\ (\mu\ \mathtt{F})\ \mathtt{T} \rhd \mathtt{unfld\ V\ F\ T\ q}}$$

$$\frac{\langle\rangle \vdash \mathtt{e} : \mathtt{T} \rhd \mathtt{q}}{\overline{\mathtt{e}} = \Lambda \mathtt{V}:* \to *.\ \mathtt{q}}$$

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 `μ F B`, and we get a proof that `A` is equal to the unfolding of `μ 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`, `(∀T:*. Exp T → Exp T)`. The main driver is `stepV`, which has the type `(∀V:* → *. ∀T:*. PExp (PExp V) T → PExp V T)`. Its input is a representation of type `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.

```
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.

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 `(∀T:*. Exp T → 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 → 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` ≡ `e2`, and `q1` and `q2` are pre-representations of `e1` and `e2` respectively, then `sem V T q1` ≡ `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 `∀A:∗. Exp A → A`, and is defined by a fold with `R=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 `All Id Id A`. When `A` is a concrete quantified type `∀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 `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. `T1` ≅ `T2` 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`.

```
CPS (A → B)  ≅ Ct (CPS A → CPS B)
CPS (∀X:K.T) ≅ Ct (∀X:K. CPS T)
CPS (μ F A)  ≅ Ct (CPS (F (μ F) A))
```

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 `All Id CPS A`, which it needs to cast to `CPS1F CPS1 A`, the unfolding of `CPS1 A`. `All Id CPS A` and `CPS1F CPS1 A` are both `Typecase` 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 `TcAll A` component of the `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 `CPS1F CPS1 A` and `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 sizeTAbs : FoldTAbs (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 `sizeTAbs` needs to apply the successor function to the result of the recursive call. However, its type `All Id (λT:∗.Nat) A` is different than `Nat`. For example, if `A = (∀X:K.A′)`, then `All Id (λT:∗.Nat) A ≡ (∀X:K.Nat)`. The quantifier on `X` is redundant, and blocks `sizeTAbs` 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 DeBruijn 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 `∀T. T → 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/popl17/. 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] Chiyan 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 marshalling 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 Æ. 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, Chiyan 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.

A. Haskell Implementation of Figure 4

In this section we show how to encode our explicit type equality proofs from Figure 4 in Haskell, using type equality constraints.

```
{-# LANGUAGE GADTs, ExistentialQuantification #-}

import Prelude hiding (Eq)

data Eq t1 t2 = t1 ~ t2 => Eq

refl :: Eq t t
refl = Eq

sym :: Eq t1 t2 -> Eq t2 t1
sym Eq = Eq

trans :: Eq t1 t2 -> Eq t2 t3 -> Eq t1 t3
trans Eq Eq = Eq

eqApp :: Eq t1 t2 -> Eq (f t1) (f t2)
eqApp Eq = Eq

arrL :: Eq (t1 -> t2) (s1 -> s2) -> Eq t1 s1
arrL Eq = Eq

arrR :: Eq (t1 -> t2) (s1 -> s2) -> Eq t2 s2
arrR Eq = Eq

coerce :: Eq t1 t2 -> f t1 -> f t2
coerce Eq x = x
```

A type equality proof `Eq a b` simply wraps a type equality coercion `a ~ b`. Haskell's type inference automatically derives new coercions based from the axioms symmetry, transitivity, and decomposition. Thus we can implement the corresponding rule for our explicit type equality proofs by pattern matching on the input proofs, which introduces the coercions into the typing context, and then constructing a new proof. Type inference will automatically derive the coercion for the new proof from those of the input proofs. If we didn't pattern match, type checking would fail. For example `trans x y = Eq` would not type check, because the coercions for the input proofs are not introduced into the typing context.

B. Section 3 Proofs

B.1 Type Safety

Lemma B.1 (Inversion of kindings)**.** *Suppose* $\Gamma \vdash \mathtt{T} : \mathtt{K}$*. Then:*

1. *If* $\mathtt{T}$=$\mathtt{X}$*, then* $(\mathtt{X}:\mathtt{K}) \in \Gamma$.
2. *If* $\mathtt{T}=(\mathtt{T}_1 \to \mathtt{T}_2)$*, then* $\mathtt{K}=*$ *and* $\Gamma \vdash \mathtt{T}_1 : *$ *and* $\Gamma \vdash \mathtt{T}_2 : *$.
3. *If* $\mathtt{T}=(\forall \mathtt{X}:\mathtt{K}_1.\mathtt{T}')$*, then* $\mathtt{K}=*$ *and* $\Gamma,(\mathtt{X}:\mathtt{K}_1) \vdash \mathtt{T}' : *$.
4. *If* $\mathtt{T}=(\lambda \mathtt{X}:\mathtt{K}_1.\mathtt{T}')$*, then* $\mathtt{K}=(\mathtt{K}_1 \to \mathtt{K}_2)$ *and* $\Gamma,(\mathtt{X}:\mathtt{K}_1) \vdash \mathtt{T}' : \mathtt{K}_2$.
5. *If* $\mathtt{T}=(\mathtt{T}_1\ \mathtt{T}_2)$*, then* $\Gamma \vdash \mathtt{T}_1 : \mathtt{K}_2 \to \mathtt{K}$ *and* $\Gamma \vdash \mathtt{T}_2 : \mathtt{K}_2$.
6. *If* $\mathtt{T}=(\mu\ \mathtt{T}_1\ \mathtt{T}_2)$*, then* $\mathtt{K}=*$ *and* $\Gamma \vdash \mathtt{T}_1 : (* \to *) \to * \to *$ *and* $\Gamma \vdash \mathtt{T}_2 : *$.
7. *If* $\mathtt{T}=(\mathtt{Typecase}\ \mathtt{T}_1\ \mathtt{T}_2\ \mathtt{T}_3\ \mathtt{T}_4\ \mathtt{T}_5)$*, then* $\mathtt{K}=*$ *and* $\Gamma \vdash \mathtt{T}_1 : * \to * \to *$ *and* $\Gamma \vdash \mathtt{T}_2 : * \to *$ *and* $\Gamma \vdash \mathtt{T}_3 : * \to *$ *and* $\Gamma \vdash \mathtt{T}_4 : ((* \to *) \to * \to *) \to * \to *$ *and* $\Gamma \vdash \mathtt{T}_5 : *$.

Proof. Straightforward. □

Lemma B.2 (Inversion of typings)**.** *Suppose* $\Gamma \vdash \mathtt{e} : \mathtt{T}$*. Then* $\Gamma \vdash \mathtt{T} : *$ *and:*

1. *If* $\mathtt{e} = \mathtt{x}$*, then* $(\mathtt{x}:\mathtt{T}') \in \Gamma$ *and* $T' \equiv T$.
2. *If* $\mathtt{e} = (\lambda \mathtt{x}:\mathtt{T}_1.\mathtt{e}_1)$*, then* $\mathtt{T}\equiv(\mathtt{T}_1 \to \mathtt{T}_2)$ *and* $\Gamma,(\mathtt{x}:\mathtt{T}_1) \vdash \mathtt{e}_1 : \mathtt{T}_2$.
3. *If* $\mathtt{e} = (\mathtt{e}_1\ \mathtt{e}_2)$*, then* $\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_2 \to \mathtt{T}$ *and* $\Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2$.
4. *If* $\mathtt{e} = (\Lambda \mathtt{X}:\mathtt{K}.\mathtt{e}_1)$*, then* $\mathtt{T}\equiv(\forall \mathtt{X}:\mathtt{K}.\mathtt{T}')$ *and* $\Gamma,(\mathtt{X}:\mathtt{K}) \vdash \mathtt{e}_1 : \mathtt{T}'$.
5. *If* $\mathtt{e} = \mathtt{e}_1\ \mathtt{T}_2$*, then* $\mathtt{T}\equiv\mathtt{T}_1[\mathtt{X}{:=}\mathtt{T}_2]$ *and* $\Gamma \vdash \mathtt{e}_1 : (\forall \mathtt{X}:\mathtt{K}.\mathtt{T}_1)$.
6. *If* $\mathtt{e} = \mathtt{fold}\ \mathtt{T}_1\ \mathtt{T}_2\ \mathtt{e}'$*, then* $\mathtt{T}\equiv(\mu\ \mathtt{T}_1\ \mathtt{T}_2)$ *and* $\Gamma \vdash \mathtt{e}' : \mathtt{T}_1\ (\mu\ \mathtt{T}_1)\ \mathtt{T}_2$.
7. *If* $\mathtt{e} = \mathtt{unfold}\ \mathtt{T}_1\ \mathtt{T}_2\ \mathtt{e}'$*, then* $\mathtt{T}\equiv(\mathtt{T}_1\ (\mu\ \mathtt{T}_1)\ \mathtt{T}_2)$ *and* $\Gamma \vdash \mathtt{e}' : \mu\ \mathtt{T}_1\ \mathtt{T}_2$.

Proof. By induction on the derivation $\Gamma \vdash \mathtt{e} : \mathtt{T}$.

Each case can be derived either by the corresponding typing rule, or by the type conversion rule:

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}_1 \qquad \mathtt{T}_1 \equiv \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{T}_2 : *}{\Gamma \vdash \mathtt{e} : \mathtt{T}_2}$$

By induction, the result holds for $\Gamma \vdash \mathtt{e} : \mathtt{T}_1$. Since $\mathtt{T}_1 \equiv \mathtt{T}_2$, the result holds for T also. □

Lemma B.3 (Canonical β-normal forms)**.** *Suppose* $\Gamma \vdash \mathtt{v} : \mathtt{T}$ *and* $\mathtt{v}$ *is a* β*-normal form. Then either* $\mathtt{v}$ *is neutral, or:*

1. $\mathtt{T} \equiv \mathtt{T}_1 \to \mathtt{T}_2$ *and* $\mathtt{v} = (\lambda \mathtt{x}:\mathtt{T}.\mathtt{e})$.
2. $\mathtt{T} \equiv (\forall \mathtt{X}:\mathtt{K}.\mathtt{T}')$ *and* $\mathtt{v} = (\Lambda \mathtt{X}:\mathtt{K}.\mathtt{e})$.
3. $\mathtt{T} \equiv \mu\ \mathtt{T}_1\ \mathtt{T}_2$ *and* $\mathtt{v} = \mathtt{fold}\ \mathtt{T}_1\ \mathtt{T}_2\ \mathtt{e}$.

Proof. By induction on the derivation $\Gamma \vdash \mathtt{v} : \mathtt{T}$.

If $\mathtt{v} = \mathtt{x}$, then v is neutral.

If $\mathtt{v} = (\lambda \mathtt{x}:\mathtt{T}_1.\mathtt{e})$, then by by Lemma B.2, $\mathtt{T} \equiv \mathtt{T}_1 \to \mathtt{T}_2$, so $\mathtt{T} \equiv \mathtt{T}_1 \to \mathtt{T}_2$ as required.

If $\mathtt{v} = \mathtt{e}_1\ \mathtt{e}_2$, then v must be neutral as required.

If $\mathtt{v} = (\Lambda \mathtt{X}:\mathtt{K}.\mathtt{e})$, then by Lemma B.2, $\mathtt{T} = (\forall \mathtt{X}:\mathtt{K}.\mathtt{T}')$, so $\mathtt{T} \equiv (\forall \mathtt{X}:\mathtt{K}.\mathtt{T}')$ as required.

If $\mathtt{v} = \mathtt{e}\ \mathtt{T}_1$, then v must be neutral as required.

If $\mathtt{v} = \mathtt{fold}\ \mathtt{T}_1\ \mathtt{T}_2\ \mathtt{e}$, then by Lemma B.2, $\mathtt{T}=(\mu\ \mathtt{T}_1\ \mathtt{T}_2)$, so $\mathtt{T}\equiv(\mu\ \mathtt{T}_1\ \mathtt{T}_2)$ as required.

If $\mathtt{v} = \mathtt{unfold}\ \mathtt{T}_1\ \mathtt{T}_2\ \mathtt{e}$, then v must be neutral as required. □

Lemma B.4 (Progress)**.** *If* $\Gamma \vdash \mathtt{e} : \mathtt{T}$*, then either* e *is* β*-normal or there exists a term* $\mathtt{e}'$ *such that* $\mathtt{e} \longrightarrow \mathtt{e}'$.

Proof. By induction on the typing derivation $\Gamma \vdash \mathtt{e} : \mathtt{T}$.

Suppose the derivation is by the first rule (for variables). Then e is a variable, which is a β-normal form.

Suppose the derivation is by the second rule (for λ abstraction). Then $\mathtt{e}=(\lambda \mathtt{x}:\mathtt{T}_1.\mathtt{e}_1)$. By Lemma B.2, we have that $\mathtt{T}=\mathtt{T}_1 \to \mathtt{T}_2$ and $\Gamma,(\mathtt{x}:\mathtt{T}_1) \vdash \mathtt{e}_1 : \mathtt{T}_2$. By induction, either $\mathtt{e}_1$ is a β-normal form, or there exists an $\mathtt{e}_1{}'$ such that $\mathtt{e}_1 \longrightarrow \mathtt{e}_1{}'$. In the former case, $(\lambda \mathtt{x}:\mathtt{T}_1.\mathtt{e}_1)$ is alos a β-normal form as required. In the latter case, $(\lambda \mathtt{x}:\mathtt{T}_1.\mathtt{e}_1) \longrightarrow (\lambda \mathtt{x}:\mathtt{T}_1.\mathtt{e}_1{}')$ as required.

The cases for type abstraction and fold are similar to the previous case for λ abstraction.

Suppose the derivation is by the third rule (for application). Then $\mathtt{e}=(\mathtt{e}_1\ \mathtt{e}_2)$. By Lemma B.2, we have that $\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_2 \to \mathtt{T}$ and $\Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2$. By induction, $\mathtt{e}_1$ is either β-normal form or there exists an $\mathtt{e}_1{}'$ such that $\mathtt{e}_1 \longrightarrow \mathtt{e}_1{}'$. Similarly, $\mathtt{e}_2$ is either β-normal form or there exists an $\mathtt{e}_2{}'$ such that $\mathtt{e}_2 \longrightarrow \mathtt{e}_2{}'$. If $\mathtt{e}_1 \longrightarrow \mathtt{e}_1{}'$, then $\mathtt{e}_1\ \mathtt{e}_2 \longrightarrow \mathtt{e}_1{}'\ \mathtt{e}_2$. If $\mathtt{e}_1 \longrightarrow \mathtt{e}_1{}'$, then $\mathtt{e}_1\ \mathtt{e}_2 \longrightarrow \mathtt{e}_1{}'\ \mathtt{e}_2$. The remaining case is when $\mathtt{e}_1$ and $\mathtt{e}_2$ are both β-normal. Since $\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_2 \to \mathtt{T}$, by Lemma B.3 either $\mathtt{e}_1$ is neutral or $\mathtt{e}_1 = (\lambda \mathtt{x}:\mathtt{T}_2.\mathtt{e}_1{}')$. If $\mathtt{e}_1$ is neutral then $\mathtt{e}_1\ \mathtt{e}_2$ is β-normal as required. If $\mathtt{e}_1 = (\lambda \mathtt{x}:\mathtt{T}_2.\mathtt{e}_1{}')$, then $\mathtt{e} = (\mathtt{e}_1\ \mathtt{e}_2) = (\lambda \mathtt{x}:\mathtt{T}_2.\mathtt{e}_1{}') \longrightarrow \mathtt{e}_1{}'[\mathtt{x}{:=}\ \mathtt{e2}]$ as required.

The cases for when the derivation is by the fifth rule (type application) and seventh rule (unfold) are similar to the previous case for application.

The case for when the derivation is by the eigth rule (type conversion) is by straightforward induction. □

Lemma B.5 (Weakening of kindings)**.**

1. *If* $\Gamma \vdash \mathtt{T_1 : K}$ *and* $\mathtt{x} \notin \mathtt{dom}(\Gamma)$ *and* $\Gamma \vdash \mathtt{T_2 : *}$*, then* $\Gamma\mathtt{,(x{:}T_2)} \vdash \mathtt{T_1 : K}$*.*
2. *If* $\Gamma \vdash \mathtt{T : K}$ *and* $\mathtt{X} \notin \mathtt{dom}(\Gamma)$*, then* $\Gamma\mathtt{,(X{:}K_1)} \vdash \mathtt{T : K}$*.*

Proof. 1) Is trivial, since $\mathtt{x} \notin FV(T)$. 2) Is by straightforward induction on kinding derivations. □

Definition B.1 (Permutation of context)**.** *A context* Γ_1 *is a permutation of another context* Γ_2 *if* $\Gamma_1 \approx \Gamma_2$ *can be derived from the following rules:*

$$\Gamma \approx \Gamma$$

$$\frac{\Gamma_1 \approx \Gamma_2}{\Gamma_2 \approx \Gamma_1}$$

$$\frac{\Gamma_1 \approx \Gamma_2 \qquad \Gamma_2 \approx \Gamma_3}{\Gamma_1 \approx \Gamma_3}$$

$$\frac{}{\Gamma\mathtt{,(x{:}T_1),(y{:}T_2)} \approx \Gamma\mathtt{,(y{:}T_2),(x{:}T_1)}}$$

$$\frac{}{\Gamma\mathtt{,(X{:}K_1),(}\beta\mathtt{{:}K_2)} \approx \Gamma\mathtt{,(}\beta\mathtt{{:}K_2),(X{:}K_1)}}$$

$$\frac{\mathtt{X} \notin \mathtt{FV(T)}}{\Gamma\mathtt{,(X{:}K),(x{:}T)} \approx \Gamma\mathtt{,(x{:}T),(X{:}K)}}$$

Lemma B.6 (Preservation of kindings under permutation of context)**.** *If* $\Gamma_1 \vdash \mathtt{T : K}$ *and* $\Gamma_1 \approx \Gamma_2$*, then* $\Gamma_2 \vdash \mathtt{T : K}$*.*

Proof. By straightforward induction on the kinding derivation. □

Lemma B.7 (Preservation of typings under permutation of context)**.** *If* $\Gamma_1 \vdash \mathtt{e : T}$ *and* $\Gamma_1 \approx \Gamma_2$*, then* $\Gamma_2 \vdash \mathtt{e : T}$*.*

Proof. By straightforward induction on the typing derivation. □

Lemma B.8 (Preservation of kinds under type substitution)**.** *If* $\Gamma\mathtt{,X{:}K_2} \vdash \mathtt{T_1 : K_1}$ *and* $\Gamma \vdash \mathtt{T_2 : K_2}$*, then* $\Gamma \vdash \mathtt{T_1[X{:=}T_2] : K_1}$*.*

Proof. By straightforward induction on kinding derivations. □

Lemma B.9 (Substitution of types on typings)**.** *If* $\Gamma\mathtt{,X{:}K} \vdash \mathtt{e : T_1}$ *and* $\Gamma \vdash \mathtt{T_2 : K}$*, then* $\Gamma \vdash \mathtt{e[X{:=}T_2] : T_1[X{:=}T_2]}$*.*

Proof. By Lemma B.8 and straightforward induction on kinding derivations. □

Lemma B.10 (Preservation of types under term substitution)**.** *If* $\Gamma\mathtt{,x{:}T_2} \vdash \mathtt{e_1 : T_1}$ *and* $\Gamma \vdash \mathtt{e_2 : T_2}$*, then* $\Gamma \vdash \mathtt{e_1[x{:=}e_2] : T_1}$*.*

Proof. By straightforward induction on typing derivations. □

Lemma B.11 (Preservation of type equivalence under type substitution)**.** *If* $\mathtt{T_1} \equiv \mathtt{T_2}$*, then* $\mathtt{T_1[X{:=}T]} \equiv \mathtt{T_2[X{:=}T]}$

Proof. By straightforward induction on equivalence derivations. □

Lemma B.12 (Type Preservation)**.** *If* $\Gamma \vdash \mathtt{e : T}$ *and* $\mathtt{e} \longrightarrow \mathtt{e'}$*, then* $\Gamma \vdash \mathtt{e' : T}$*.*

Proof. By induction on the derivation of $\Gamma \vdash \mathtt{e : T}$.

Suppose $\mathtt{e} \longrightarrow \mathtt{e'}$ is by the first rule. Then $\mathtt{e = (}\lambda\mathtt{x{:}T_1.e_1)e_2}$ and $\mathtt{e' = e_1[x{:=}e_2]}$. By Lemma B.2, we have that $\Gamma \vdash \mathtt{(}\lambda\mathtt{x{:}T_1.e_1) : T_1 \rightarrow T}$ and $\Gamma \vdash \mathtt{e_2 : T_1}$. By Lemma B.2 again, we have that $\Gamma\mathtt{,(x{:}T_1)} \vdash \mathtt{e_1 : T}$. By Lemma B.10, we have that $\Gamma \vdash \mathtt{e_1[x{:=}e_2] : T}$ as required.

Suppose $\mathtt{e} \longrightarrow \mathtt{e'}$ is by the second rule. Then $\mathtt{e = (}\Lambda\mathtt{X{:}K.e_1)T_2}$ and $\mathtt{e' = e_1[X{:=}T_2]}$. By Lemma B.2, we have that $\Gamma \vdash \mathtt{(}\Lambda\mathtt{X{:}K.e_1) : (}\forall\mathtt{X{:}K.T_1)}$ and $\mathtt{T} \equiv \mathtt{T_1[X{:=}T_2]}$. By Lemma B.2 again, we have that $\Gamma\mathtt{,(X{:}K)} \vdash \mathtt{e_1 : T_1}$. By Lemma B.9, we have that $\Gamma \vdash \mathtt{e_1[X{:=}T_2] : T_1[X{:=}T_2]}$ as required.

Suppose $\mathtt{e} \longrightarrow \mathtt{e'}$ is by the third rule. Then $\mathtt{e = unfold\ T_1\ T_2\ (fold\ T_1'\ T_2'\ e_1)}$ and $\mathtt{e' = e_1}$. By Lemma B.2, we have that $\mathtt{T} \equiv \mathtt{T_1\ (\mu\ T_1)\ T_2}$ and $\Gamma \vdash \mathtt{fold\ T_1'\ T_2'\ e_1 : \mu\ T_1\ T_2}$. By Lemma B.2 again, we have that $\mathtt{\mu\ T_1\ T_2} \equiv \mathtt{\mu\ T_1'\ T_2'}$ and $\Gamma \vdash \mathtt{e_1 : T_1'\ (\mu\ T_1')\ T_2'}$. Since $\mathtt{\mu\ T_1\ T_2} \equiv \mathtt{\mu\ T_1'\ T_2'}$, we have that $\mathtt{T_1} \equiv \mathtt{T_1'}$ and $\mathtt{T_2} \equiv \mathtt{T_2'}$. Therefore $\Gamma \vdash \mathtt{e_1 : T_1'\ (\mu\ T_1')\ T_2'}$ as required.

The remaining cases are by straightforward induction. □

Theorem 3.1. *[Type Safety]*
If $\langle\rangle \vdash \mathtt{e : T}$*, then either* $\mathtt{e}$ *is a normal form, or there exists an* $\mathtt{e'}$ *such that* $\langle\rangle \vdash \mathtt{e' : T}$ *and* $\mathtt{e} \longrightarrow \mathtt{e'}$*.*

Proof. By Lemmas B.4 and B.12. □

B.2 Type Reduction

Definition B.2 (Reduction on types)**.** *Type reduction is a directed variant of the type equivalence rules in Figure 7, without rules for reflexivity, symmetry, or* α*-conversion.*

$$\frac{\mathtt{T1} \longrightarrow \mathtt{T1'}}{\mathtt{(T1 \rightarrow T2)} \longrightarrow \mathtt{(T1' \rightarrow T2)}} \qquad \frac{\mathtt{T2} \longrightarrow \mathtt{T2'}}{\mathtt{(T1 \rightarrow T2)} \longrightarrow \mathtt{(T1 \rightarrow T2')}}$$

$$\frac{\mathtt{T} \longrightarrow \mathtt{T'}}{\mathtt{(}\forall\mathtt{X{:}K.T)} \longrightarrow \mathtt{(}\forall\mathtt{X{:}K.T')}} \qquad \frac{\mathtt{T} \longrightarrow \mathtt{T'}}{\mathtt{(}\lambda\mathtt{X{:}K.T)} \longrightarrow \mathtt{(}\lambda\mathtt{X{:}K.T')}}$$

$$\frac{\mathtt{T1} \longrightarrow \mathtt{T1'}}{\mathtt{T1\ T2} \longrightarrow \mathtt{T1'\ T2}} \qquad \frac{\mathtt{T2} \longrightarrow \mathtt{T2'}}{\mathtt{T1\ T2} \longrightarrow \mathtt{T1\ T2'}}$$

$$\mathtt{(}\lambda\mathtt{X{:}K.T1)\ T2} \longrightarrow \mathtt{(T1[X := T2])}$$

$$\mathtt{Typecase\ F1\ F2\ F3\ F4\ (T1 \rightarrow T2)} \longrightarrow \mathtt{F1\ T1\ T2}$$
$$\mathtt{Typecase\ F1\ F2\ F3\ F4\ (\mu\ T1\ T2)} \longrightarrow \mathtt{F4\ T1\ T2}$$

$$\frac{\mathtt{X} \notin \mathtt{FV(F3)}}{\mathtt{Typecase\ F1\ F2\ F3\ F4\ (}\forall\mathtt{X{:}K.T)} \longrightarrow \mathtt{F2\ (}\forall\mathtt{X{:}K.\ F3\ T)}}$$

We use $\longrightarrow^*$ to denote the reflexive transitive closure of $\longrightarrow$. Also, we use $\mathtt{T1 = T2}$ to denote that $\mathtt{T1}$ and $\mathtt{T2}$ are syntactically equal types, up to renaming (i.e. they are α-equivalent).

Lemma B.13 (Preservation of kinds under type reduction)**.** *If* $\Gamma \vdash \mathtt{T1 : K}$ *and* $\mathtt{T1} \longrightarrow \mathtt{T2}$*, then* $\Gamma \vdash \mathtt{T2 : K}$*.*

Proof. By induction on the derivation of $\mathtt{T1} \longrightarrow \mathtt{T2}$.

We consider only the cases for β-reduction and the three Typecase eliminations. The others are straightforward by induction.

Suppose the equivalence is by β-reduction. Then $\mathtt{T1=(}\lambda\mathtt{X{:}K1.A)B}$ and $\mathtt{T2=A[X{:=}B]}$. By Lemma B.1, we have that $\Gamma\mathtt{,(X{:}K1)} \vdash \mathtt{A : K}$ and $\Gamma \vdash \mathtt{B : K1}$. By Lemma B.8, we have that $\Gamma \vdash \mathtt{A[X{:=}B] : K}$ as required.

Suppose the equivalence is by the first $\mathtt{Typecase}$ reduction rule. Then $\mathtt{K=*}$ and $\mathtt{T1=Typecase\ F1\ F2\ F3\ F4\ (A \rightarrow B)}$ and $\mathtt{T2=F1\ A\ B}$. By Lemma B.1, we have that $\Gamma \vdash \mathtt{F1 : * \rightarrow * \rightarrow *}$ and $\Gamma \vdash \mathtt{A : *}$ and $\Gamma \vdash \mathtt{B : *}$. Therefore, $\Gamma \vdash \mathtt{F1\ A\ B : *}$, as required.

The remaining $\mathtt{Typecase}$ reduction cases are similar. □

B.3 Type Reduction is Strongly Normalizing

Our proof of strong normalization of types is based on the technique from Girard, Taylor and Lafont (GTL) [19]. The types of $F_\omega^{\mu i}$ consist of the simply-typed λ-calculus, extended with constructors for arrow, quantified, and recursive types, and `Typecase`.

Lemma B.14. *A type* $\mathtt{T}$ *is strongly normalizing iff there is there is a number* $\nu(\mathtt{T})$ *that bounds the length of reduction sequences starting from* $\mathtt{T}$.

Proof. Straightforward. □

Definition B.3 (Reducibility)**.**

1. *If* $\Gamma \vdash \mathtt{T} : *$ *and* $\mathtt{T}$ *is SN, then* $\mathtt{T} \in \mathtt{RED}_*$.
2. *If* $\Gamma \vdash \mathtt{T} : \mathtt{K1} \to \mathtt{K2}$ *and for all* $\mathtt{T1} \in \mathtt{RED}_{\mathtt{K1}}$, $\mathtt{T\ T1} \in \mathtt{RED}_{\mathtt{K2}}$*, then* $\mathtt{T} \in RED_{\mathtt{K1}\to\mathtt{K2}}$.
3. (a) *If* $(\mathtt{T1} \to \mathtt{T2}) \in \mathtt{RED}_*$*, then* $\mathtt{T1} \in \mathtt{RED}_*$ *and* $\mathtt{T2} \in \mathtt{RED}_*$.
   (b) *If* $(\forall \mathtt{X{:}K.T}) \in \mathtt{RED}_*$*, then* $\mathtt{T} \in \mathtt{RED}_*$.
   (c) *If* $(\mu\ \mathtt{F}) \in \mathtt{RED}_{*\to*}$*, then* $\mathtt{F} \in \mathtt{RED}_{(*\to*)\to*\to*}$.
   (d) *If* $(\mu\ \mathtt{F\ A}) \in \mathtt{RED}_*$*, then* $\mathtt{F} \in \mathtt{RED}_{(*\to*)\to*\to*}$ *and* $\mathtt{A} \in \mathtt{RED}_*$.

Definition B.4 (Neutral Types)**.** *All types other than abstractions are neutral.*

Definition B.5 (Conditions of Reducibility)**.** *We will prove by induction on kinds that all types satisfy the following conditions of reducibility:*

(CR 1) *If* $\mathtt{T} \in \mathtt{RED}_\mathtt{K}$*, then* $\mathtt{T}$ *is SN.*
(CR 2) *If* $\mathtt{T} \in \mathtt{RED}_\mathtt{K}$ *and* $\mathtt{T} \longrightarrow^* \mathtt{T}'$*, then* $\mathtt{T}' \in \mathtt{RED}_\mathtt{K}$.
(CR 3) *If* $\mathtt{T}$ *is neutral and for all* $\mathtt{T}'$, $\mathtt{T} \longrightarrow \mathtt{T}'$ *implies* $\mathtt{T}' \in \mathtt{RED}_\mathtt{K}$*, then* $\mathtt{T} \in \mathtt{RED}_\mathtt{K}$.

A consequence of **(CR 3)** will be that if $\Gamma \vdash \mathtt{T} : \mathtt{K}$ and $\mathtt{T}$ neutral and normal, then $\mathtt{T} \in \mathtt{RED}_\mathtt{K}$. This is called (CR 4) by GTL [19].

Lemma B.15. *For all kinds* $\mathtt{K}$*, the conditions of reducibility hold for* $\mathtt{RED}_\mathtt{K}$.

Proof. By induction on $\mathtt{K}$, following the same argument as GTL [19]. □

Lemma B.16 (Reducibility of $\to$)**.** *If* $\mathtt{T1,T2} \in \mathtt{RED}_*$*, then* $(\mathtt{T1} \to \mathtt{T2}) \in \mathtt{RED}_*$.

Proof. It is clear that if $\mathtt{T1}$ and $\mathtt{T2}$ are SN, then so is $(\mathtt{T1} \to \mathtt{T2})$. That satisfies requirement 1 of reducibility. Requirement 3a is by assumption. □

Lemma B.17 (Reducibility of $\forall$)**.** *If* $\mathtt{T} \in \mathtt{RED}_*$*, then* $(\forall \mathtt{X{:}K.T}) \in \mathtt{RED}_*$.

Proof. Similar to Lemma B.16. □

Lemma B.18 (Reducibility of μ)**.** $\mu \in \mathtt{RED}_{((*\to*)\to*\to*)\to*\to*}$

Proof. It suffices to show that for all $\mathtt{F}$ and $\mathtt{A}$, if $\mathtt{F} \in \mathtt{RED}_{(*\to*)\to*\to*}$ and $\mathtt{A} \in \mathtt{RED}_*$, then $\mu\ \mathtt{F} \in \mathtt{RED}_{*\to*}$ and $\mu\ \mathtt{F\ A} \in \mathtt{RED}_*$.

By (CR 1), $\nu(\mathtt{F})$ and $\nu(\mathtt{A})$ are defined. We proceed by induction on $\nu(\mathtt{F}) + \nu(\mathtt{A})$.

We only consider the codes for $\mu\ \mathtt{F\ A}$. The case for $\mu\ \mathtt{F}$ is similar.

If $\nu(\mathtt{F}) + \nu(\mathtt{A}) = 0$, then $\mathtt{F}$ and $\mathtt{A}$ are normal, and $\mu\ \mathtt{F\ A}$ is neutral and normal. Therefore, $\mu\ \mathtt{F\ A} \in \mathtt{RED}_*$ follows by (CR 3).

Suppose $\nu(\mathtt{F}) + \nu(\mathtt{A}) > 0$. Then $\mu\ \mathtt{F\ A}$ steps. Consider the step:

Suppose $\mu\ \mathtt{F\ A} \longrightarrow \mu\ \mathtt{F'\ A}$ and $\mathtt{F} \longrightarrow \mathtt{F}'$. Then by (CR 2) $\mathtt{F}'$ is reducible, and $\nu(\mathtt{F}') < \nu(\mathtt{F})$. By induction, $\mu\ \mathtt{F'\ A}$ is reducible.

Suppose $\mu\ \mathtt{F\ A} \longrightarrow \mu\ \mathtt{F\ A'}$ and $\mathtt{A} \longrightarrow \mathtt{A}'$. Then $\mathtt{A}'$ is reducible and $\nu(\mathtt{F}') < \nu(\mathtt{F})$. By induction, $\mu\ \mathtt{F\ A'}$ is reducible.

Since $\mu\ \mathtt{F\ A}$ always steps to reducible types, so by (CR 3) and requirement 3(c) of reducibility, $\mu\ \mathtt{F\ A}$ is reducible. □

Lemma B.19 (Reducibility of `Typecase`)**.** $\mathtt{Typecase} \in \mathtt{RED}_{(*\to*\to*)\to(*\to*)\to(*\to*)\to(((*\to*)\to*\to*)\to*\to*)\to*\to*}$.

Proof. It suffices to show that for all $\mathtt{F1} \in \mathtt{RED}_{*\to*\to*}$, $\mathtt{F2} \in \mathtt{RED}_{*\to*}$, $\mathtt{F3} \in \mathtt{RED}_{*\to*}$, $\mathtt{F4} \in \mathtt{RED}_{((*\to*)\to*\to*)\to*\to*}$, and $\mathtt{S} \in \mathtt{RED}_*$, it is true that $\mathtt{Typecase\ F1\ F2\ F3\ F4\ S} \in \mathtt{RED}_*$.

We proceed by induction on $\nu(\mathtt{F1}) + \nu(\mathtt{F2}) + \nu(\mathtt{F3}) + \nu(\mathtt{F4}) + \nu(\mathtt{S})$.

Suppose $\mathtt{F1} \in \mathtt{RED}_{*\to*\to*}$, $\mathtt{F2} \in \mathtt{RED}_{*\to*}$, $\mathtt{F3} \in \mathtt{RED}_{*\to*}$, $\mathtt{F4} \in \mathtt{RED}_{((*\to*)\to*\to*)\to*\to*}$, and $\mathtt{S} \in \mathtt{RED}_*$.

Let $\mathtt{T} = \mathtt{Typecase\ F1\ F2\ F3\ F4\ S}$. Consider the possible reduction steps from $\mathtt{T}$:

- $\mathtt{T} \longrightarrow \mathtt{Typecase\ F1'\ F2\ F3\ F4\ S}$ and $\mathtt{F1} \longrightarrow \mathtt{F1}'$. By (CR 2) $\mathtt{F1}'$ is reducible and $\nu(\mathtt{F1}') < \nu(\mathtt{F1})$, so by induction $\mathtt{Typecase\ F1'\ F2\ F3\ F4\ S} \in \mathtt{RED}_*$. The cases for steps in one of $\mathtt{F2,F3,F4,S}$ are similar.
- $\mathtt{T} \longrightarrow \mathtt{F1\ S1\ S2}$ and $\mathtt{S} = \mathtt{S1} \to \mathtt{S2}$. By definition 3(a) of reducibility, $\mathtt{S1,\ S2} \in \mathtt{RED}_*$, so $\mathtt{F1\ S1\ S2} \in \mathtt{RED}_*$.
- $\mathtt{T} \longrightarrow \mathtt{F2}\ (\forall \mathtt{X{:}K.\ F3\ S1})$ and $\mathtt{S} = (\forall \mathtt{X{:}K.S1})$. By definition 3(b) of reducibility, $\mathtt{S1} \in \mathtt{RED}_*$, so $\mathtt{F3\ S1}$ is reducible, so by Lemma B.17 $(\forall \mathtt{X{:}K.\ F3\ S1}) \in \mathtt{RED}_*$, so $\mathtt{F2}\ (\forall \mathtt{X{:}K.\ F3\ S1}) \in \mathtt{RED}_*$.
- $\mathtt{T} \longrightarrow \mathtt{F4\ S1\ S2}$ and $\mathtt{S} = \mu\ \mathtt{S1\ S2}$. By definition 3(d) of reducibility, $\mathtt{S1} \in \mathtt{RED}_{(*\to*)\to*\to*}$ and $\mathtt{S2} \in \mathtt{RED}_*$, so $\mathtt{F4\ S1\ S2} \in \mathtt{RED}_*$.

In all cases, the result of stepping $\mathtt{T}$ is contained in RED_*, so by (CR 3) we have that $\mathtt{T} \in \mathtt{RED}_*$. □

Definition B.6 (Reducible substitution)**.** *Let* $\Gamma = \langle\rangle$, $\mathtt{X}_1{:}\mathtt{K}_1$, $\mathtt{X}_2{:}\mathtt{K}_2$, $\ldots$, $\mathtt{X}_n{:}\mathtt{K}_n$*, and let* σ *be a substitution* $[\mathtt{X}_1{:=}\mathtt{T}_1, \mathtt{X}_2{:=}\mathtt{T}_2, \ldots, \mathtt{X}_n{:=}\mathtt{T}_n]$. *If every* $\mathtt{T}_i \in \mathtt{RED}_{\mathtt{K}i}$*, then we say that* σ *is a reducible substitution for* Γ.

Lemma B.20. *If* $\Gamma \vdash \mathtt{T} : \mathtt{K}$*, and* σ *is a reducible substitution for* Γ*, then* $\mathtt{T}\sigma \in \mathtt{RED}_\mathtt{K}$.

Proof. By induction on the derivation of $\Gamma \vdash \mathtt{T} : \mathtt{K}$.

Suppose $\Gamma \vdash \mathtt{T} : \mathtt{K}$ is by the rule for variables. Since σ is a reducible substitution, $\mathtt{T}\sigma$ is equal to some type $\in \mathtt{RED}_\mathtt{K}$.

Suppose $\Gamma \vdash \mathtt{T} : \mathtt{K}$ is by the rule for arrow types. Then $\mathtt{T} = (\mathtt{T1} \to \mathtt{T2})$, and $\mathtt{K} = *$ and $\Gamma \vdash \mathtt{T1} : *$ and $\Gamma \vdash \mathtt{T2} : *$. By induction, $\mathtt{T1}\sigma, \mathtt{T2}\sigma \in \mathtt{RED}_*$, and so are SN. Therefore $\mathtt{T}\sigma$ is SN, and the requirements 1 and 3(a) of reducibility are satisfied, so $\mathtt{T}\sigma \in \mathtt{RED}_*$.

Suppose $\Gamma \vdash \mathtt{T} : \mathtt{K}$ is by the rule for quantified types. Then $\mathtt{T} = (\forall \mathtt{X{:}K.T'})$ and $\mathtt{K} = *$ and $\Gamma, \mathtt{X{:}K} \vdash \mathtt{T}' : *$. Without loss of generality (by renaming), assume that $\mathtt{X}$ does not occur in Γ. Since the type variable $\mathtt{X}$ is a neutral and normal, $\mathtt{X} \in \mathtt{RED}_\mathtt{K}$ and so $\sigma[\mathtt{X{:=}X}]$ is a reducible substitution for $\Gamma, \mathtt{X{:}K}$. By induction, $\mathtt{T}'\sigma[\mathtt{X{:=}X}] = \mathtt{T}'\sigma \in \mathtt{RED}_*$, so by Lemma B.17 $(\forall \mathtt{X{:}K.\ T}'\sigma) = \mathtt{T}\sigma \in \mathtt{RED}_*$.

Suppose $\Gamma \vdash \mathtt{T} : \mathtt{K}$ is by the rule for λ-abstraction. Then $\mathtt{T} = (\lambda \mathtt{X{:}K1.T'})$ and $\mathtt{K} = \mathtt{K1} \to \mathtt{K2}$ and $\Gamma \vdash (\lambda \mathtt{X{:}K1.T'}) : \mathtt{K1} \to \mathtt{K2}$ and $\Gamma, \mathtt{X{:}K} \vdash \mathtt{T}' : \mathtt{K2}$. Without loss of generality (by renaming), assume that $\mathtt{X}$ does not occur in Γ. Since the type variable $\mathtt{X}$ is a neutral and normal, $\mathtt{X} \in \mathtt{RED}_\mathtt{K}$ and so $\sigma[\mathtt{X{:=}X}]$ is a reducible substitution for $\Gamma, \mathtt{X{:}K}$. By induction, $\mathtt{T}'\sigma[\mathtt{X{:=}X}] = \mathtt{T}'\sigma \in \mathtt{RED}_{\mathtt{K2}}$. Therefore, $(\lambda \mathtt{X{:}K1.\ T}'\sigma) = (\lambda \mathtt{X{:}K1.T'})\sigma \in \mathtt{RED}_{\mathtt{K1}\to\mathtt{K2}}$ as required.

The case for type application is by straightforward induction.
The case for μ is by Lemma B.18.
The case for `Typecase` is by Lemma B.19. □

Lemma B.21 (SN of type reduction)**.** *If* $\Gamma \vdash \mathtt{T} : \mathtt{K}$*, then* $\mathtt{T}$ *is SN.*

Proof. Follows from Lemma B.20 and (CR 1). □

B.4 Types have unique normal forms

Lemma B.22 (Confluence of type reduction)**.** *If* $\Gamma \vdash$ `T : K`*, and* `T` $\longrightarrow$ `T1` *and* `T` $\longrightarrow$ `T2`*, then there exists a type* `T′` *such that* `T1` $\longrightarrow^*$ `T′` *and* `T2` $\longrightarrow^*$ `T′`

Proof. Standard. □

Lemma B.23. *If* `T1` $\equiv$ `T2` *then there exists a type* `T` *such that* `T1` $\longrightarrow^*$ `T` *and* `T2` $\longrightarrow^*$ `T`*.*

Proof. By induction on the derivation of `T1` $\equiv$ `T2`.

If `T1` $\equiv$ `T2` is by reflexivity. Then `T1 = T2`. The result follows by Lemma B.21.

If `T1` $\equiv$ `T2` is by symmetry, then the result follows from the induction hypothesis.

If `T1` $\equiv$ `T2` is by transitivity, there exists a type `T3` such that `T1` $\equiv$ `T3` and `T3` $\equiv$ `T2`. By two uses of induction, there exist types `T` and `T′` such that `T1` $\longrightarrow^*$ `T`, `T3` $\longrightarrow^*$ `T`, `T3` $\longrightarrow^*$ `T′`, and `T2` $\longrightarrow^*$ `T′`. By Lemma B.22, `T3` $\longrightarrow^*$ `T` and `T3` $\longrightarrow^*$ `T′` imply that there exists a `T″` such that `T` $\longrightarrow^*$ `T″` and `T′` $\longrightarrow^*$ `T″`. Therefore `T1` $\longrightarrow^*$ `T″` and `T2` $\longrightarrow^*$ `T″` as required.

If `T1` $\equiv$ `T2` is by the congruence rule for arrows, we have that `T1 = (A1` $\rightarrow$ `B1)`, `T2 = (A2` $\rightarrow$ `B2)`, and `A1` $\equiv$ `A2` and `B1` $\equiv$ `B2`. By induction, there exist normal form types `A` and `B` such that `A1` $\longrightarrow^*$ `A`, `A2` $\longrightarrow^*$ `A`, `B1` $\longrightarrow^*$ `B`, and `B2` $\longrightarrow^*$ `B`. Let `T = (A` $\rightarrow$ `B)`. Construct `T1` $\longrightarrow^*$ `T` and `T2` $\longrightarrow^*$ `T` by first reducing `A1` and `A2` to `A` and then reducing `B1` and `B2` to `B`.

The congruence rules for μ and application are similar.

The remaining rules for type equivalence have corresponding rules for type reduction. □

Lemma B.24. *If* $\Gamma \vdash$ `T : K` *and* `T` $\longrightarrow^*$ `N1` *and* `T` $\longrightarrow^*$ `N2`*, where* `N1,N2` *are normal forms, then* `N1 = N2`*.*

Proof. By Lemma B.22, there exists a type `T′` such that `N1` $\longrightarrow^*$ `T′` and `N2` $\longrightarrow^*$ `T′`. But `N1` and `N2` are normal forms, so it must be the case that `N1 = T′` and `N2 = T′`. So `N1 = N2` as required. □

We say "`T` has a normal form" to mean that there exists a normal form `N` such that `T` $\longrightarrow^*$ `N`.

Lemma B.25 (Types have unique normal forms)**.** *If* $\Gamma \vdash$ `T : K`*, then* `T` *has a normal form that is unique up to renaming.*

Proof. First, by Lemma B.21 we have that there exists a normal form.

Second, suppose that `T` has two normal forms `N1` and `N2`. Then by Lemma B.24 `N1 = N2`. □

Based on Lemma B.25, we use `nf(T)` to denote the unique normal form of `T`. We have that if $\Gamma \vdash$ `T : K`, then `T` $\longrightarrow^*$ `nf(T)`.

B.5 Type equivalence and type checking are decidable

Lemma B.26. *If* `T` $\longrightarrow^*$ `T′`*, then* `T` $\equiv$ `T′`*.*

Proof. Straightforward. □

Lemma B.27. *If* $\Gamma \vdash$ `T : K`*, then* `T` $\equiv$ `nf(T)`*.*

Proof. By Lemma B.25, `nf(T)` exists and `T` $\longrightarrow^*$ `nf(T)`. The result follows by Lemma B.26. □

Lemma B.28. *If* $\Gamma \vdash$ `N1 : K` *and* $\Gamma \vdash$ `N2 : K` *and* `N1,N2` *are normal forms, then* `N1` $\equiv$ `N2` *if and only if* `N1 = N2`*.*

Proof. ($\Leftarrow$) is immediate.

($\Rightarrow$) By Lemma B.23, there exists a type `T` such that `N1` $\longrightarrow^*$ `T` and `N2` $\longrightarrow^*$ `T`. But since `N1` and `N2` are normal forms, `N1 = T` and `N2 = T`, so `N1 = N2` as required. □

Lemma B.29. *If* $\Gamma \vdash$ `T1 : K` *and* $\Gamma \vdash$ `T2 : K`*,* `T1` $\equiv$ `T2` *is decidable.*

Proof. By Lemma B.27, `T1` $\equiv$ `nf(T1)` and `T2` $\equiv$ `nf(T2)`. Therefore, `T1` $\equiv$ `T2` if and only if `nf(T1)` $\equiv$ `nf(T2)`. By Lemma B.28, `nf(T1)` $\equiv$ `nf(T2)` if and only if `nf(T1) = nf(T2)`.

Therefore, we can decide `T1` $\equiv$ `T2` by reducing both to normal form and checking whether those normal forms are equal up to renaming. □

Theorem 3.2. *Type checking is decidable.*

Proof. Type checking is syntax-directed: there is one rule per syntactic form, plus the type conversion rule based on equivalence. Decidability follows from that type equivalence is decidable (Lemma B.29). □

C. Section 5 Proofs

The following definition relates the environment used to typecheck a term with the environment used to typecheck its pre-representation.

Definition C.1 (Environment mapping for pre-representations)**.**

$$\overline{\langle\rangle} = \langle\rangle$$
$$\overline{\Gamma,\texttt{X:K}} = \overline{\Gamma},\texttt{X:K}$$
$$\overline{\Gamma,\texttt{x:T}} = \overline{\Gamma},\texttt{x:PExp V T}$$

Lemma C.1. *If* $\Gamma \vdash$ `T : K`*, then* $\overline{\Gamma} \vdash$ `T : K`

Proof. Straightforward, since $\bar{\cdot}$ does not affect the presence, order, or kinds of type variables in the environment. □

Lemma C.2. *If* $\Gamma \vdash$ `e : T` *and* `e` *contains no free term variables, then* $\overline{\Gamma} \vdash$ `e : T`

Proof. Straightforward, using Lemma C.1. □

Lemma C.3. *If* $\Gamma \vdash$ `(∀X:K.T)` $: *$*, then*

1. $\Gamma \vdash \texttt{tcAll}_{\texttt{X,K,T}}$ `: TcAll (∀X:K.T)`
2. $\Gamma \vdash \texttt{unAll}_{\texttt{X,K,T}}$ `: UnAll (∀X:K.T)`
3. $\Gamma \vdash \texttt{isAll}_{\texttt{X,K,T}}$ `: IsAll (∀X:K.T)`

Proof. Suppose $\Gamma \vdash$ `(∀X:K.T)` $: *$. Then Γ`,X:K` $\vdash$ `T :` $*$.

1) $\texttt{tcAll}_{\texttt{X,K,T}}$ `=` Λ`Arr:`$* \rightarrow * \rightarrow *$`.` Λ`Out:`$* \rightarrow *$`.` Λ`In:`$* \rightarrow *$`.` Λ`Mu:((`$* \rightarrow *$`)` $\rightarrow * \rightarrow *$`)` $\rightarrow * \rightarrow *$`.refl (Out(∀X:K.In T))`. It is easily checked that $\texttt{tcAll}_{\texttt{X,K,T}}$ has the type `Eq (Out(∀X:K.In T)) (Out(∀X:K.In T))`. But `Typecase Arr Out In Mu (∀X:K.T)` $\equiv$ `Out (∀X:K.In T)`, and also `All Out In (∀X:K.T)` $\equiv$ `Out (∀X:K.In T)`, so we have that $\texttt{tcAll}_{\texttt{X,K,T}}$ has the type `TcAll (∀X:K.T)` as required.

2) Follows from reasoning similar to 1.

3) Follows from 1 and 2. □

Lemma C.4. *If* $\Gamma \vdash$ `(∀X:K.T)` $: *$ *and* $\Gamma \vdash$ `S : K`*, then*

1. $\overline{\Gamma} \vdash \texttt{underAll}_{\texttt{X,K,T}}$ `: UnderAll (∀X:K.T)`
2. $\overline{\Gamma} \vdash \texttt{stripAll}_{\texttt{K}}$ `: StripAll (∀X:K.T)`
3. $\overline{\Gamma} \vdash \texttt{inst}_{\texttt{X,K,T,S}}$ `: Inst (∀X:K.T) (T[X:=S])`

Proof. First, note that $\texttt{underAll}_{\texttt{X,K,T}}$, $\texttt{stripAll}_{\texttt{K}}$, and $\texttt{inst}_{\texttt{X,K,T,S}}$ contain no free term variables. Therefore, by Lemma C.2 it is sufficient to show

1. $\Gamma \vdash \texttt{underAll}_{\texttt{X,K,T}}$ `: UnderAll (∀X:K.T)`
2. $\Gamma \vdash \texttt{stripAll}_{\texttt{K}}$ `: StripAll (∀X:K.T)`
3. $\Gamma \vdash \texttt{inst}_{\texttt{X,K,T,S}}$ `: Inst (∀X:K.T) (T[X:=S])`

For 1, it is easily checked that $\Gamma \vdash$ $\texttt{underAll}_{\texttt{X,K,T}}$: (∀F1:∗ → ∗. ∀F2:∗ → ∗. (∀A:∗. F1 A → F2 A) → (∀X:K. F1 T) → (∀X:K. F2 T). The result follows from the type equivalences All Id F1 (∀X:K.T) ≡ (∀X:K. F1 T) and All Id F2 (∀X:K.T) ≡ (∀X:K. F2 T).

The cases for 2 and 3 are similar. □

Lemma C.5. *If* $\Gamma \vdash$ e : T*, then there exists a unique* q *such that* $\Gamma \vdash$ e : T ▷ q *and* $\overline{\Gamma} \vdash$ q : PExp V T.

Proof. By straightforward induction on the derivation of $\Gamma \vdash$ e : T, using the types of the constructors, Lemma C.3, and Theorem C.4.

Suppose $\Gamma \vdash$ e : T is by the rule for variables. Then e = x and (x:T) ∈ Γ and $\Gamma \vdash$ x : T ▷ x. By the definition of $\bar{\cdot}$, (x:PExp V T) ∈ $\overline{\Gamma}$. Therefore, $\overline{\Gamma} \vdash$ x : PExp V T as required.

Suppose $\Gamma \vdash$ e : T is by the rule for λ-abstractions. Then e = (λx:T1. e1) and Γ,x:T1 ⊢ e1 : T2 and T = T1 → T2. Also, Γ,x:T1 ⊢ e1 : T2 ▷ q1 and Γ ⊢ (λx:T1.e1) ▷ abs V T1 T2 (λx:PExp V T1. q1) By induction, q1 is unique and $\overline{\Gamma\texttt{,x:T1}} \vdash$ q1 : PExp V T2. But $\overline{\Gamma\texttt{,x:T1}}$ = $\overline{\Gamma}$,x:(PExp V T1), so $\overline{\Gamma} \vdash$ (λx:PExp V T1. q1) : PExp V T1 → PExp V T2. Therefore, q = abs V T1 T2 (λx:PExp V T1. q1) is unique and by the type of abs, it is easily checked that $\overline{\Gamma} \vdash$ abs V T1 T2 (λx:PExp V T1. q1) : PExp V (T1 → T2).

The cases for applications, and fold and unfold expressions are similar.

Suppose Γ ⊢ e : T is by the rule for type abstractions. Then e = (ΛX:K.e1) and Γ,X:K ⊢ e1 : T1 and T = (∀X:K.T1). Also, Γ,X:K ⊢ e1 : T1 ▷ q1 and Γ ⊢ (ΛX:K.e1) ▷ tabs V (∀X:K.T1) p s u (ΛX:K.q1), where p = $\texttt{isAll}_{\texttt{X,K,T}}$, s = $\texttt{stripAll}_{\texttt{K}}$ = s, and u = $\texttt{underAll}_{\texttt{X,K,T}}$. By induction, q1 is unique and $\overline{\Gamma\texttt{,X:K}} \vdash$ q1 : PExp V T1. Since $\overline{\Gamma\texttt{,X:K}}$ = $\overline{\Gamma}$,X:K, we have that $\overline{\Gamma} \vdash$ (ΛX:K.q1) : (∀X:K. PExp V T1). It follows from (All Id (PExp V) (∀X:K.T1)) ≡ (∀X:K. PExp V T1) that $\overline{\Gamma} \vdash$ (ΛX:K.q1) : (All Id (PExp V) (∀X:K.T1)). By Theorem C.3, Γ ⊢ p : IsAll (∀X:K.T). But p does not contain any free term variables, so by Lemma C.2 we have that $\overline{\Gamma} \vdash$ p : IsAll (∀X:K.T). By Theorem C.4, we have that $\overline{\Gamma} \vdash$ s : StripAll (∀X:K.T) and $\overline{\Gamma} \vdash$ u : UnderAll (∀X:K.T). Therefore, q = tabs V (∀X:K.T1) p s u (ΛX:K.q1) is unique and by the type of tabs, we have that $\overline{\Gamma} \vdash$ tabs V (∀X:K.T1) p s u (ΛX:K.q1) : PExp V (∀X:K.T1) as required.

The case for type applications is similar.

Suppose Γ ⊢ e : T is by the rule for type conversion. The Γ ⊢ e : S and S ≡ T. By the induction hypothesis, there exists a unique q such that Γ ⊢ e : S ▷ q and $\overline{\Gamma} \vdash$ q : PExp V S. Since PExp V S ≡ PExp V T, we have that $\overline{\Gamma} \vdash$ q : PExp V T as required. □

Theorem 5.1. *If* ⟨⟩ ⊢ e : T*, then* ⟨⟩ ⊢ $\overline{\texttt{e}}$: Exp T.

Proof. Follows straightforwardly from Theorem C.5. □

Lemma C.6 (SN of refl)**.** *For any type* T*,* refl T *is SN.*

Proof. Straightforward. □

Lemma C.7 (SN of $\texttt{isAll}_{\texttt{X,K,T}}$)**.** *If* Γ ⊢ (∀X:K.T) : ∗*, then* $\texttt{isAll}_{\texttt{X,K,T}}$ *is SN.*

Proof. By Lemma C.6, $\texttt{tcAll}_{\texttt{X,K,T}}$ and $\texttt{unAll}_{\texttt{X,K,T}}$ are SN.

The pair ($\texttt{tcAll}_{\texttt{X,K,T}}$, $\texttt{unAll}_{\texttt{X,K,T}}$) is encoded as (ΛR:∗. λf:TcAll (∀X:K.T) → UnAll (∀X:K.T) → R. f $\texttt{tcAll}_{\texttt{X,K,T}}$ $\texttt{unAll}_{\texttt{X,K,T}}$), which is SN since $\texttt{tcAll}_{\texttt{X,K,T}}$ and $\texttt{unAll}_{\texttt{X,K,T}}$ are. □

Lemma C.8 (SN of $\texttt{inst}_{\texttt{X,K,T,S}}$)**.** *If* Γ ⊢ (∀X:K.T) : ∗ *and* Γ ⊢ S : K*, then* $\texttt{inst}_{\texttt{X,K,T,S}}$ *is SN.*

Proof. Immediate, the definition of $\texttt{inst}_{\texttt{X,K,T,S}}$ is a normal form. □

Lemma C.9 (SN of $\texttt{stripAll}_{\texttt{K}}$)**.** *For any kind* K*,* $\texttt{stripAll}_{\texttt{K}}$ *is SN.*

Proof. Immediate, the definition of $\texttt{stripAll}_{\texttt{K}}$ is a normal form. □

Lemma C.10 (SN of $\texttt{underAll}_{\texttt{X,K,T}}$)**.** *If* Γ ⊢ (∀X:K.T) : ∗*, then* $\texttt{underAll}_{\texttt{X,K,T}}$ *is SN.*

Proof. Immediate, the definition of $\texttt{underAll}_{\texttt{X,K,T}}$ is a normal form. □

Lemma C.11 (SN of constructors)**.**

1. *For any* V, A, e*, if* e *is SN and* var V A e *is well-typed, then* var V A e *is SN.*
2. *For any* V, A, B, f*, if* f *is SN and* abs V A f *is well-typed, then* abs V A f *is SN.*
3. *For any* V, A, B, e1, e2*, if* e1 *and* e2 *are SN and* app V A B e1 e2 *is well-typed, then* app V A B e1 e2 *is SN.*
4. *For any* V, A, p, s, u, e*, if* p, s, u*, and* e *are SN and* tabs V A p s u e *is well-typed, then* tabs V A p s u e *is SN.*
5. *For any* V, A, B, p, i, e*, if* p, i*, and* e *are SN and* tapp V A B p i e *is well-typed, then* tapp V A B p i e *is SN.*
6. *For any* V, F, A, e*, if* e *is SN and* fld V F A e *is well-typed, then* fld V F A e *is SN.*
7. *For any* V, F, A, e*, if* e *is SN and* unfld V F A e *is well-typed, then* unfld V F A e *is SN.*

Proof. By Lemma C.6, the type equality proofs created by abs, fld and unfld are SN.

The result holds since each case reduces in a few steps to a term of the form fold (PExpF V) A (λR.λvar.λabs.λapp.λtabs.λtapp. λfld.λunfld. e), where e is an application with a variable in head position, and only SN term arguments. □

Lemma C.12. *If* Γ ⊢ e : T ▷ q*, then* q *is strongly normalizing.*

Proof. By straightforward induction on the derivation of Γ ⊢ e : T ▷ q, using Lemmas C.7, C.8, C.9, C.10, and C.11. □

Theorem 5.2. *If* ⟨⟩ ⊢ e : T*, then* $\overline{\texttt{e}}$ *is strongly normalizing.*

Proof. We have that $\overline{\texttt{e}}$ = ΛV:∗ → ∗. q and ⟨⟩ ⊢ e : T ▷ q. By Lemma C.12, q is SN. Therefore {ole is also. □

D. Code Listings

D.1 Prelude

We define a prelude of some useful data types – pairs, booleans, natural numbers, as well a fixpoint combinator used to define recursive functions.

```
decl Id : * → * = λA:*. A;

-- Pairs
decl Pair : * → * → * =
  λA:*. λB:*. ∀C:*. (A → B → C) → C;
decl pair : ∀A:*. ∀B:*. A → B → Pair A B =
  ΛA:*. ΛB:*. λa:A. λb:B.
  ΛC:*. λf:A → B → C. f a b;
decl fst : ∀A:*. ∀B:*. Pair A B → A =
  ΛA:*. ΛB:*. λp:Pair A B. p A (λa:A. λb:B. a);
decl snd : ∀A:*. ∀B:*. Pair A B → B =
  ΛA:*. ΛB:*. λp:Pair A B. p B (λa:A. λb:B. b);

-- Booleans
decl Bool  : *    = ∀A:*. A → A → A;
decl true  : Bool = ΛA:*. λt:A. λf:A. t;
decl false : Bool = ΛA:*. λt:A. λf:A. f;
decl and   : Bool → Bool → Bool =
  λb1:Bool. λb2:Bool. b1 Bool b2 false;
decl or    : Bool → Bool → Bool =
  λb1:Bool. λb2:Bool. b1 Bool true b2;
decl not   : Bool → Bool =
  λb:Bool. b Bool false true;

-- Fixpoint and Bottom
decl RecF : (* → *) → * → * =
  λRec : * → *. λA:*. Rec A → A;

decl Rec : * → * = μ RecF;

decl fix : (∀A:*. (A → A) → A) =
  ΛA:*. λf:A → A.
  let x : (Rec A) = fold RecF A (λr : Rec A. f (unfold RecF A r r)) in
  f (unfold RecF A x x);

decl Bottom : * = (∀A:*. A);

decl bottom : Bottom =
  ΛA:*. fix A (λx:A. x);

-- Natural numbers
decl NatF : (* → *) → * → * =
  λNatF : * → *. λB:*.
  ∀A:*. A → (NatF Bottom → A) → A;

decl Nat : * = μ NatF Bottom;

decl zero : Nat =
  fold NatF Bottom
    (ΛA:*. λz:A. λs:Nat → A. z);

decl succ : Nat → Nat =
  λn:Nat.
  fold NatF Bottom
    (ΛA:*. λz:A. λs:Nat → A. s n);

decl one : Nat = succ zero;
decl two : Nat = succ one;
decl three : Nat = succ two;
decl four : Nat = succ three;
decl five : Nat = succ four;
```

```
decl six : Nat = succ five;
decl seven : Nat = succ six;
decl eight : Nat = succ seven;
decl nine : Nat = succ eight;
decl ten : Nat = succ nine;

decl pred : Nat → Nat =
  λn:Nat.
  unfold NatF Bottom n Nat
    -- n = 0
    zero
    -- n > 0
    (λm:Nat. m);

decl rec plus : Nat → Nat → Nat =
  λm:Nat. λn:Nat.
  unfold NatF Bottom m Nat
    -- zero
    n
    -- succ
    (λpm : Nat. plus pm (succ n));

decl rec times : Nat → Nat → Nat =
  λm:Nat. λn:Nat.
  unfold NatF Bottom m Nat
    zero
    (λpm : Nat. plus (times pm n) n);

decl rec eqNat : Nat → Nat → Bool =
  λm : Nat. λn : Nat.
  unfold NatF Bottom m Bool
    -- m = 0
    (unfold NatF Bottom n Bool
      -- n = 0
      true
      -- n > 0
      (λpn : Nat. false))
    -- m > 0
    (λpm : Nat.
     unfold NatF Bottom n Bool
       -- n = 0
       false
       -- n > 0
       (λpn : Nat. eqNat pm pn));

decl rec fact : Nat → Nat =
  λn:Nat.
  eqNat n zero Nat
    -- n = 0
    one
    -- n != 0
    (eqNat n one Nat
      -- n = 1
      one
      -- n != 1
      (times n (fact (pred n))));
```

D.2 Intensional Type Functions

```
decl All : (* → *) → (* → *) → * → * =
  λOut : * → *. λIn  : * → *.
  Typecase
    (λA:*. λB:*. Bottom)
    Out
    In
```

```
    (λF:(* → *) → * → *. λA:*. Bottom);

decl Unfold : * → * =
  Typecase
    (λA:*. λB:*. Bottom)
    (λT:*. Bottom)
    (λT:*. Bottom)
    (λF:(* → *) → * → *. λA:*. F (μ F) A);

decl ArrL : * → * =
  Typecase
    (λa:*. λb:*. a)
    (λa:*. Bottom)
    (λa:*. Bottom)
    (λf:(* → *) → * → *. λa:*. Bottom);

decl ArrR : * → * =
  Typecase
    (λa:*. λb:*. b)
    (λa:*. Bottom)
    (λa:*. Bottom)
    (λf:(* → *) → * → *. λa:*. Bottom);
```

D.3 Type Equality

```
decl Eq : * → (* → *) = λA:*. λB:*. ∀F:* → *. F A → F B;

decl refl : (∀A:*. Eq A A) = ΛA:*. ΛF:* → *. λx : F A. x;

decl sym : (∀A:*. ∀B:*. Eq A B → Eq B A) =
  ΛA:*. ΛB:*. λeq : Eq A B.
  let p : Eq A A = refl A in
  eq (λT:*. Eq T A) p;

decl trans : (∀A:*. ∀B:*. ∀C:*. Eq A B → Eq B C → Eq A C) =
  ΛA:*. ΛB:*. ΛC:*. λeqAB : Eq A B. λeqBC : Eq B C.
  ΛF : * → *. λx : F A. eqBC F (eqAB F x);

decl eqApp : (∀A:*. ∀B:*. ∀F:* → *.
              Eq A B → Eq (F A) (F B)) =
  ΛA:*. ΛB:*. ΛF:* → *. λeq : Eq A B.
  let p : Eq (F A) (F A) = refl (F A) in
  eq (λT:*. Eq (F A) (F T)) p;

decl coerce : (∀A:*. ∀B:*. Eq A B → A → B) =
 ΛA:*. ΛB:*. λeq:Eq A B. eq Id;

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

decl UnAll : * → * = λA:*. ∀Out:* → *. Eq (All Out Id A) (Out A);

decl IsAll : * → * = λA:*. Pair (TcAll A) (UnAll A);

decl isAll : (∀A : *. TcAll A → UnAll A → IsAll A) =
  ΛA:*. pair (TcAll A) (UnAll A);

decl tcAll : (∀A:*. IsAll A → TcAll A) = ΛA:*. fst (TcAll A) (UnAll A);

decl unAll : (∀A:*. IsAll A → UnAll A) = ΛA:*. snd (TcAll A) (UnAll A);

decl arrL : (∀A1:*. ∀A2:*. ∀B1:*. ∀B2:*.
             Eq (A1 → A2) (B1 → B2) →
             Eq A1 B1) =
```

```
  ΛA1:*. ΛA2:*. ΛB1:*. ΛB2:*. eqApp (A1 → A2) (B1 → B2) ArrL;

decl arrR : (∀A1:*. ∀A2:*. ∀B1:*. ∀B2:*.
             Eq (A1 → A2) (B1 → B2) →
             Eq A2 B2) =
  ΛA1:*. ΛA2:*. ΛB1:*. ΛB2:*. eqApp (A1 → A2) (B1 → B2) ArrR;

decl coerce : (∀A∗:. ∀B∗:. Eq A B → A → B) =
 ΛA∗:. ΛB∗:. λeq:Eq A B. eq Id;

decl eqUnfold : (∀F1: (* → *) → * → *. ∀A1:*.
                 ∀F2: (* → *) → * → *. ∀A2:*.
                 Eq (μ F1 A1) (μ F2 A2) →
                 Eq (F1 (μ F1) A1) (F2 (μ F2) A2)) =
  ΛF1 : (* → *) → * → *. ΛA1:*.
  ΛF2 : (* → *) → * → *. ΛA2:*.
  eqApp (μ F1 A1) (μ F2 A2) Unfold;

-- Contradictions
decl eqArrMu : (∀A1:*. ∀A2:*. ∀F : (* → *) → * → *. ∀B:*.
                Eq (A1 → A2) (μ F B) → Bottom) =
  ΛA1:*. ΛA2:*. ΛF:(* → *) → * → *. ΛB:*.
  λeq : Eq (A1 → A2) (μ F B). ΛC : *.
  let id : (∀A:*. A → A) = ΛA:*. λx:A. x in
  let id1 : Typecase (λX:*.λY:*.(∀A:*. A → A)) (λX:*.C) (λX:*.C)
                     (λF : (* → *) → * → *. λB : *. C) (A1 → A2) = id in
  let id2 : Typecase (λX:*.λY:*.(∀A:*. A → A)) (λX:*.C) (λX:*.C)
                     (λF : (* → *) → * → *. λB : *. C) (μ F B) =
    eq (λT:*. Typecase (λX:*.λY:*.(∀A:*. A → A)) (λX:*.C) (λX:*.C)
                       (λF : (* → *) → * → *. λB : *. C) T)
       id1
  in id2;

decl arrIsAll : (∀A : *. ∀B : *. IsAll (A → B) → Bottom) =
  ΛA : *. ΛB : *.
  λp : IsAll (A → B).
  let id : (∀A:*. A → A) = ΛA:*. λx:A. x in
  let id1 : Typecase (λX:*. λY:*. (∀A:*. A → A))
                     (λX:*. Bottom) (λX:*. Bottom)
                     (λF: (* → *) → * → *. λB : *. Bottom)
                     (A → B) = id
  in
  let eq : Eq (Typecase (λX:*. λY:*. (∀A:*. A → A))
                        (λX:*. Bottom) (λX:*. Bottom)
                        (λF: (* → *) → * → *. λB : *. Bottom)
                        (A → B))
              (All (λX:*. Bottom) (λX:*. Bottom) (A → B)) =
    tcAll (A → B) p
      (λX:*. λY:*. (∀A:*. A → A))
      (λX:*. Bottom) (λX:*. Bottom)
      (λF: (* → *) → * → *. λB : *. Bottom)
  in
  eq (λX:*. X) id1;

decl muIsAll : (∀F:(* → *) → * → *. ∀A:*. IsAll (μ F A) → Bottom) =
  ΛF:(* → *) → * → *. ΛA:*. λp:IsAll (μ F A).
  let id : (∀A:*. A → A) = ΛA:*. λx:A. x in
  let id1 : Typecase (λX:*. λY:*. Bottom) (λX:*. Bottom) (λX:*. Bottom)
                     (λF: (* → *) → * → *. λB : *. (∀A:*. A → A))
                     (μ F A) = id
  in
  let eq : Eq (Typecase (λX:*. λY:*. Bottom) (λX:*. Bottom) (λX:*. Bottom)
                        (λF: (* → *) → * → *. λB : *. (∀A:*. A → A))
                        (μ F A))
              (All (λX:*. Bottom) (λX:*. Bottom) (μ F A)) =
```

```
    tcAll (µ F A) p
      (λX:*. λY:*. Bottom) (λX:*. Bottom) (λX:*. Bottom)
      (λF: (* → *) → * → *. λB : *. (∀A:*. A → A))
  in
  eq (λX:*. X) id1;
```

D.4 Representation

```
decl StripAll : * → * =
  λA:*. ∀B:*. All Id (λA:*. B) A → B;

decl UnderAll : * → * =
  λA:*. ∀F1:* → *. ∀F2:* → *.
  (∀B:*. F1 B → F2 B) →
  All Id F1 A → All Id F2 A;

decl Inst : * → * → * =
  λA:*. λB:*. (∀F:* → *. All Id F A → F B);

decl PExpF : (* → *) → (* → *) → * → * =
  λV : * → *. λPExpV:* → *. λA:*.
  ∀R:*.
  -- var
  (V A → R) →
  -- abs
  (∀A1:*. ∀A2:*. Eq (A1 → A2) A → (PExpV A1 → PExpV A2) → R) →
  -- app
  (∀B:*. PExpV (B → A) → PExpV B → R) →
  -- tabs
  (IsAll A → StripAll A → UnderAll A → All Id PExpV A → R) →
  -- tapp
  (∀B:*. IsAll B → Inst B A → PExpV B → R) →
  -- fold
  (∀F : (* → *) → * → *. ∀B : *.
   Eq (µ F B) A → PExpV (F (µ F) B) → R) →
  -- unfold
  (∀F : (* → *) → * → *. ∀B : *.
   Eq (F (µ F) B) A → PExpV (µ F B) → R) →
  R
;

decl PExp : (* → *) → * → * = λV : * → *. µ (PExpF V);

decl Exp : * → * = λA:*. ∀V:* → *. PExp V A;

decl VarF : (* → *) → * → * → * =
  λV:* → *. λA:*. λR:*. V A → R;

decl AbsF : (* → *) → * → * → * = λV:* → *. λA:*. λR:*.
  ∀A1:*. ∀A2:*. Eq (A1 → A2) A → (PExp V A1 → PExp V A2) → R;

decl AppF : (* → *) → * → * → * = λV:* → *. λA:*. λR:*.
  ∀B:*. PExp V (B → A) → PExp V B → R;

decl TAbsF : (* → *) → * → * → * = λV:* → *. λA:*. λR:*.
  IsAll A → StripAll A → UnderAll A → All Id (PExp V) A → R;

decl TAppF : (* → *) → * → * → * = λV:* → *. λA:*. λR:*.
  ∀B:*. IsAll B → Inst B A → PExp V B → R;

decl FoldF : (* → *) → * → * → * = λV:* → *. λA:*. λR:*.
  ∀F : (* → *) → * → *. ∀B : *.
  Eq (µ F B) A → PExp V (F (µ F) B) → R;

decl UnfoldF : (* → *) → * → * → * = λV:* → *. λA:*. λR:*.
  ∀F : (* → *) → * → *. ∀B : *.
```

```
  Eq (F (μ F) B) A → PExp V (μ F B) → R;

decl var : (∀V : * → *. ∀A:*. V A → PExp V A) =
  ΛV:* → *. ΛA:*. λx:V A.
  fold (PExpF V) A (
    ΛR:*.
    λvar: VarF V A R.
    λabs: AbsF V A R.
    λapp: AppF V A R.
    λtabs: TAbsF V A R.
    λtapp: TAppF V A R.
    λfld: FoldF V A R.
    λunfld: UnfoldF V A R.
    var x);

decl abs : (∀V : * → *. ∀A:*. ∀B:*.
           (PExp V A → PExp V B) →
           PExp V (A → B)) =
  ΛV:* → *. ΛA:*. ΛB:*. λf:PExp V A → PExp V B.
  fold (PExpF V) (A → B) (
    ΛR:*.
    λvar: VarF V (A → B) R.
    λabs: AbsF V (A → B) R.
    λapp: AppF V (A → B) R.
    λtabs: TAbsF V (A → B) R.
    λtapp: TAppF V (A → B) R.
    λfld: FoldF V (A → B) R.
    λunfld: UnfoldF V (A → B) R.
    abs A B (refl (A → B)) f);


decl app : (∀V : * → *. ∀A:*. ∀B:*.
          PExp V (A → B) →
          PExp V A →
          PExp V B) =
  ΛV:* → *. ΛA:*. ΛB:*.
  λf:PExp V (A → B). λx:PExp V A.
  fold (PExpF V) B (
    ΛR:*.
    λvar: VarF V B R.
    λabs: AbsF V B R.
    λapp: AppF V B R.
    λtabs: TAbsF V B R.
    λtapp: TAppF V B R.
    λfld: FoldF V B R.
    λunfld: UnfoldF V B R.
    app A f x);

decl tabs : (∀V : * → *. ∀A:*.
           IsAll A →
           StripAll A →
           UnderAll A →
           All Id (PExp V) A →
           PExp V A) =
  ΛV:* → *. ΛA:*. λp : IsAll A. λs : StripAll A.
  λu : UnderAll A. λe:All Id (PExp V) A.
  fold (PExpF V) A (
    ΛR:*.
    λvar: VarF V A R.
    λabs: AbsF V A R.
    λapp: AppF V A R.
    λtabs: TAbsF V A R.
    λtapp: TAppF V A R.
    λfld: FoldF V A R.
    λunfld: UnfoldF V A R.
```

```
    tabs p s u e);

decl tapp : (∀V : * → *. ∀A:*. ∀B:*.
             IsAll A → Inst A B → PExp V A →
             PExp V B) =
  ΛV:* → *. ΛA:*. ΛB:*.
  λp : IsAll A. λi : Inst A B. λe : PExp V A.
  fold (PExpF V) B (
    ΛR:*.
    λvar: VarF V B R.
    λabs: AbsF V B R.
    λapp: AppF V B R.
    λtabs: TAbsF V B R.
    λtapp: TAppF V B R.
    λfld: FoldF V B R.
    λunfld: UnfoldF V B R.
    tapp A p i e);

decl fld : (∀V : * → *. ∀F:(* → *) → * → *. ∀A:*.
            PExp V (F (μ F) A) →
            PExp V (μ F A)) =
  ΛV:* → *. ΛF:(* → *) → * → *. ΛA:*.
  λe : PExp V (F (μ F) A).
  fold (PExpF V) (μ F A) (
    ΛR:*.
    λvar: VarF V (μ F A) R.
    λabs: AbsF V (μ F A) R.
    λapp: AppF V (μ F A) R.
    λtabs: TAbsF V (μ F A) R.
    λtapp: TAppF V (μ F A) R.
    λfld: FoldF V (μ F A) R.
    λunfld: UnfoldF V (μ F A) R.
    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:* → *.
  ΛF:(* → *) → * → *. ΛA:*.
  λe : PExp V (μ F A).
  fold (PExpF V) (F (μ F) A) (
    ΛR:*.
    λvar: VarF V (F (μ F) A) R.
    λabs: AbsF V (F (μ F) A) R.
    λapp: AppF V (F (μ F) A) R.
    λtabs: TAbsF V (F (μ F) A) R.
    λtapp: TAppF V (F (μ F) A) R.
    λfld: FoldF V (F (μ F) A) R.
    λunfld: UnfoldF V (F (μ F) A) R.
    unfld F A (refl (F (μ F) A)) e);
```

D.5 Pattern Matching

```
decl constVar : (∀V:* → *. ∀A:*. ∀R : *.
                R → VarF V A R) =
  ΛV:* → *. ΛA:*. ΛR:*. λr:R. λx:V A. r;

decl constAbs : (∀V:* → *. ∀A:*. ∀R : *.
                R → AbsF V A R) =
  ΛV:* → *. ΛA:*. ΛR:*. λr:R.
  ΛA1:*. ΛA2:*. λeq:Eq (A1 → A2) A.
  λf:PExp V A1 → PExp V A2. r;

decl constApp : (∀V:* → *. ∀A:*. ∀R : *.
                R → AppF V A R) =
  ΛV:* → *. ΛA:*. ΛR:*. λr:R.
```

```
  ΛB:*. λe1 : PExp V (B → A). λe2 : PExp V B. r;

decl constTAbs : (∀V:* → *. ∀A:*. ∀R : *.
                   R → TAbsF V A R) =
  ΛV:* → *. ΛA:*. ΛR:*. λr:R.
  λp:IsAll A. λs:StripAll A. λu:UnderAll A.
  λe:All Id (PExp V) A. r;

decl constTApp : (∀V:* → *. ∀A:*. ∀R : *.
                  R → TAppF V A R) =
  ΛV:* → *. ΛA:*. ΛR:*. λr:R.
  ΛB:*. λp:IsAll B. λinst:(∀F:* → *. All Id F B → F A).
  λf : PExp V B. r;

decl constFold : (∀V:* → *. ∀A:*. ∀R : *.
                  R → FoldF V A R) =
  ΛV:* → *. ΛA:*. ΛR:*. λr:R.
  ΛF : (* → *) → * → *. ΛB:*.
  λeqFold : Eq (μ F B) A. λe:PExp V (F (μ F) B). r;

decl constUnfold : (∀V:* → *. ∀A:*. ∀R:*.
                    R → UnfoldF V A R) =
  ΛV:* → *. ΛA:*. ΛR : *. λr : R.
  ΛF : (* → *) → * → *. ΛB : *.
  λeq : Eq (F (μ F) B) A. λe : PExp V (μ F B). r;

decl matchVar : (∀V:* → *. ∀A:*. ∀R : *.
                 PExp V A → R → VarF V A R → R) =
  ΛV : * → *. ΛA : *. ΛR : *.
  λe : PExp V A. λdefault : R. λwhenVar : VarF V A R.
  unfold (PExpF V) A e R
    whenVar
    (constAbs V A R default)
    (constApp V A R default)
    (constTAbs V A R default)
    (constTApp V A R default)
    (constFold V A R default)
    (constUnfold V A R default);

decl matchAbs : (∀V:* → *. ∀A:*. ∀R : *.
                 PExp V A → R → AbsF V A R → R) =
  ΛV : * → *. ΛA : *. ΛR : *.
  λe : PExp V A. λdefault : R. λwhenAbs : AbsF V A R.
  unfold (PExpF V) A e R
    (constVar V A R default)
    whenAbs
    (constApp V A R default)
    (constTAbs V A R default)
    (constTApp V A R default)
    (constFold V A R default)
    (constUnfold V A R default);

decl matchApp : (∀V:* → *. ∀A:*. ∀R : *.
                 PExp V A → R → AppF V A R → R) =
  ΛV : * → *. ΛA : *. ΛR : *.
  λe : PExp V A. λdefault : R. λwhenApp : AppF V A R.
  unfold (PExpF V) A e R
    (constVar V A R default)
    (constAbs V A R default)
    whenApp
    (constTAbs V A R default)
    (constTApp V A R default)
    (constFold V A R default)
    (constUnfold V A R default);
```

```
decl matchTAbs : (∀V : * → *. ∀A:*. ∀R : *.
                  PExp V A → R → TAbsF V A R → R) =
  ΛV : * → *. ΛA : *. ΛR : *.
  λe : PExp V A. λdefault : R. λwhenTAbs : TAbsF V A R.
  unfold (PExpF V) A e R
    (constVar V A R default)
    (constAbs V A R default)
    (constApp V A R default)
    whenTAbs
    (constTApp V A R default)
    (constFold V A R default)
    (constUnfold V A R default);

decl matchTApp : (∀V : * → *. ∀A:*. ∀R : *.
                  PExp V A → R → TAppF V A R → R) =
  ΛV : * → *. ΛA : *. ΛR : *.
  λe : PExp V A. λdefault : R. λwhenTApp : TAppF V A R.
  unfold (PExpF V) A e R
    (constVar V A R default)
    (constAbs V A R default)
    (constApp V A R default)
    (constTAbs V A R default)
    whenTApp
    (constFold V A R default)
    (constUnfold V A R default);

decl matchFold : (∀V : * → *. ∀A:*. ∀R : *.
                  PExp V A → R → FoldF V A R → R) =
  ΛV : * → *. ΛA : *. ΛR : *.
  λe : PExp V A. λdefault : R. λwhenFold : FoldF V A R.
  unfold (PExpF V) A e R
    (constVar V A R default)
    (constAbs V A R default)
    (constApp V A R default)
    (constTAbs V A R default)
    (constTApp V A R default)
    whenFold
    (constUnfold V A R default);

decl matchUnfold : (∀V:* → *. ∀A:*. ∀R : *.
                    PExp V A → R → UnfoldF V A R → R) =
  ΛV : * → *. ΛA : *. ΛR : *.
  λe : PExp V A. λdefault : R. λwhenUnfold : UnfoldF V A R.
  unfold (PExpF V) A e R
    (constVar V A R default)
    (constAbs V A R default)
    (constApp V A R default)
    (constTAbs V A R default)
    (constTApp V A R default)
    (constFold V A R default)
    whenUnfold;

decl matchExp :
  (∀V : * → *. ∀A:*. PExp V A → ∀R : *.
   VarF V A R → AbsF V A R → AppF V A R →
   TAbsF V A R → TAppF V A R →
   FoldF V A R → UnfoldF V A R → R) =
  ΛV : * → *. ΛA:*. λe : PExp V A.
  unfold (PExpF V) A e;
```

D.6 Weak Head Normal Form Evaluator

```
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 f1 : PExp V B → PExp V A =
       λx : PExp V B. eqR (PExp V) (f (eqL (PExp V) x))
     in evalV V A (f1 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);
```

D.7 Single-step Left-most Reduction

The implementation of `step` refers to `foldExpV`, which is defined in Section D.9.1, and `nfV` and `nf`, which are defined in Section D.10.

```
decl outExp : (∀V: * → *. ∀A:*. PExp (PExp V) A → PExp V A) =
  ΛV : * → *. foldExpV (PExp V) (abs V) (app V)
               (tabs V) (tapp V) (fld V) (unfld V);

decl stepAbs : (∀V : * → *.
               (∀A:*. PExp (PExp V) A → PExp V A) →
               ∀A : *. AbsF (PExp V) A (PExp V A)) =
  ΛV : * → *. λstep : (∀A:*. PExp (PExp V) A → PExp V A).
  ΛA : *. ΛA1:*. ΛA2:*. λeq : Eq (A1 → A2) A.
  λf : PExp (PExp V) A1 → PExp (PExp V) A2.
  eq (PExp V)
     (abs V A1 A2 (λx : PExp V A1.
       step A2 (f (var (PExp V) A1 x))));

decl stepApp : (∀V : * → *.
               (∀A:*. PExp (PExp V) A → PExp V A) →
               ∀A:*. AppF (PExp V) A (PExp V A)) =
  ΛV : * → *. λstep : (∀A:*. PExp (PExp V) A → PExp V A).
  ΛA:*. ΛB:*. λf : PExp (PExp V) (B → A). λx : PExp (PExp V) B.
  let default : PExp V A =
    let stepF : PExp V A = app V B A (step (B → A) f) (outExp V B x) in
    let stepX : PExp V A = app V B A (outExp V (B → A) f) (step B x) in
    let f_nf : Bool = nfV (PExp V) (B → A) f in
    f_nf (PExp V A) stepX stepF
  in
  matchAbs (PExp V) (B → A) (PExp V A) f default
    (ΛB1 : *. ΛA1:*.
```

```
    λeq : Eq (B1 → A1) (B → A).
    λf : PExp (PExp V) B1 → PExp (PExp V) A1.
    let eqB : Eq B1 B = arrL B1 A1 B A eq in
    let eqA : Eq A1 A = arrR B1 A1 B A eq in
    let x1 : PExp (PExp V) B1 = sym B1 B eqB (PExp (PExp V)) x in
    eqA (PExp V) (outExp V A1 (f x1)));

decl stepTAbs : (∀V : * → *.
                (∀A:*. PExp (PExp V) A → PExp V A) →
                ∀A : *. TAbsF (PExp V) A (PExp V A)) =
  ΛV : * → *. λstep : (∀A:*. PExp (PExp V) A → PExp V A).
  ΛA : *. λp : IsAll A. λs : StripAll A.
  λu : UnderAll A. λe : All Id (PExp (PExp V)) A.
  tabs V A p s u (u (PExp (PExp V)) (PExp V) step e);

decl stepTApp : (∀V : * → *.
                (∀A:*. PExp (PExp V) A → PExp V A) →
                ∀A:*. TAppF (PExp V) A (PExp V A)) =
  ΛV : * → *. λstep : (∀A:*. PExp (PExp V) A → PExp V A).
  ΛA : *. ΛB : *. λp : IsAll B. λi : Inst B A. λe : PExp (PExp V) B.
  let default : PExp V A = tapp V B A p i (step B e) in
  matchTAbs (PExp V) B (PExp V A) e default
    (λp : IsAll B. λs : StripAll B. λu : UnderAll B. λe : All Id (PExp (PExp V)) B.
     outExp V A (i (PExp (PExp V)) e));

decl stepFold : (∀V : * → *.
                (∀A:*. PExp (PExp V) A → PExp V A) →
                ∀A : *. FoldF (PExp V) A (PExp V A)) =
  ΛV : * → *. λstep : (∀A:*. PExp (PExp V) A → PExp V A).
  ΛA : *. ΛF : (* → *) → * → *. ΛB:*.
  λeq : Eq (μ F B) A. λe : PExp (PExp V) (F (μ F) B).
  eq (PExp V) (fld V F B (step (F (μ F) B) e));

decl stepUnfold : (∀V : * → *.
                  (∀A:*. PExp (PExp V) A → PExp V A) →
                  ∀A:*. UnfoldF (PExp V) A (PExp V A)) =
  ΛV : * → *. λstep : (∀A:*. PExp (PExp V) A → PExp V A).
  ΛA : *. ΛF : (* → *) → * → *. ΛB : *.
  λeq : Eq (F (μ F) B) A. λe : PExp (PExp V) (μ F B).
  let default : PExp V (F (μ F) B) = unfld V F B (step (μ F B) e) in
  eq (PExp V)
     (matchFold (PExp V) (μ F B) (PExp V (F (μ F) B)) e default
       (ΛF1 : (* → *) → * → *. ΛB1:*.
        λeq1 : Eq (μ F1 B1) (μ F B).
        λe : PExp (PExp V) (F1 (μ F1) B1).
        eqUnfold F1 B1 F B eq1 (PExp V)
          (outExp V (F1 (μ F1) B1) e)));

decl rec stepV : (∀V : * → *. ∀A:*. PExp (PExp V) A → PExp V A) =
  ΛV:* → *. ΛA:*. λe:PExp (PExp V) A.
  unfold (PExpF (PExp V)) A e
    (PExp V A) -- result type
    (λx : PExp V A. x)
    (stepAbs V (stepV V) A) (stepApp V (stepV V) A)
    (stepTAbs V (stepV V) A) (stepTApp V (stepV V) A)
    (stepFold V (stepV V) A) (stepUnfold V (stepV V) A);

decl step : (∀A : *. Exp A → Exp A) =
  ΛA:*. λe:Exp A. ΛV:* → *. stepV V A (e (PExp V));

decl rec stepNorm : (∀A:*. Exp A → Exp A) =
  ΛA : *. λe : Exp A.
  let nf : Bool = nf A e in
  nf (Exp A) e (stepNorm A (step A e));
```

D.8 Normalization by Evaluation

The implementation of `nbe` refers to `foldExp`, which is defined in Section D.9.1.

```
load "Repr";
load "Fold";

decl PNfExpF : (* → *) → (* → *) → * → * =
  λNe : * → *. λNf : * → *. λA : *.
  ∀R : *.
  -- neutral
  (Ne A → R) →
  -- abs
  (∀A1:*. ∀A2:*. Eq (A1 → A2) A → (Ne A1 → Nf A2) → R) →
  -- tabs
  (IsAll A → StripAll A → UnderAll A → All Id Nf A → R) →
  -- fold
  (∀F : (* → *) → * → *. ∀B : *.
   Eq (μ F B) A → Nf (F (μ F) B) → R) →
  R;

decl PNfExp1 : (* → *) → * → * = λNe : * → *. μ (PNfExpF Ne);

decl PNeExpF : (* → *) → (* → *) → * → * =
  λV : * → *. λNe : * → *. λA : *.
  ∀R : *.
  -- var
  (V A → R) →
  -- app
  (∀B:*. Ne (B → A) → PNfExp1 Ne B → R) →
  -- tapp
  (∀B:*. IsAll B → Inst B A → Ne B → R) →
  -- unfold
  (∀F:(* → *) → * → *. ∀B:*.
   Eq (F (μ F) B) A → Ne (μ F B) → R) →
  R;

decl PNeExp : (* → *) → * → * = λV : * → *. μ (PNeExpF V);
decl PNfExp : (* → *) → * → * = λV : * → *. μ (PNfExpF (PNeExp V));
decl NfExp  : * → * = λT:*. ∀V:* → *. PNfExp V T;

decl NfNe : (* → *) → * → * → * =
  λV : * → *. λA : *. λR : *. PNeExp V A → R;

decl NfAbs : (* → *) → * → * → * =
  λV : * → *. λA : *. λR : *.
  ∀A1:*. ∀A2:*. Eq (A1 → A2) A → (PNeExp V A1 → PNfExp V A2) → R;

decl NfTAbs : (* → *) → * → * → * =
  λV:* → *. λA:*. λR:*.
  IsAll A → StripAll A → UnderAll A → All Id (PNfExp V) A → R;

decl NfFold : (* → *) → * → * → * =
  λV:* → *. λA:*. λR:*.
  ∀F : (* → *) → * → *. ∀B : *.
  Eq (μ F B) A → PNfExp V (F (μ F) B) → R;

decl NeVar : (* → *) → * → * → * =
  λV : * → *. λA : *. λR : *. V A → R;

decl NeApp : (* → *) → * → * → * =
  λV : * → *. λA : *. λR : *.
  ∀B:*. PNeExp V (B → A) → PNfExp V B → R;

decl NeTApp : (* → *) → * → * → * =
  λV:* → *. λA:*. λR:*.
```

```
  ∀B:*. IsAll B → Inst B A → PNeExp V B → R;

decl NeUnfold : (* → *) → * → * → * =
  λV:* → *. λA:*. λR:*.
  ∀F : (* → *) → * → *. ∀B:*.
  Eq (F (μ F) B) A → PNeExp V (μ F B) → R;

decl mkNfNe : (∀V : * → *. ∀A : *. PNeExp V A → PNfExp V A) =
  ΛV:* → *. ΛA:*. λe : PNeExp V A.
  fold (PNfExpF (PNeExp V)) A
    (ΛR:*.
     λne : NfNe V A R.
     λabs : NfAbs V A R.
     λtabs : NfTAbs V A R.
     λfld : NfFold V A R. ne e);

decl mkNfAbs : (∀V:* → *. ∀A:*. ∀B:*.
                (PNeExp V A → PNfExp V B) →
                PNfExp V (A → B)) =
  ΛV:* → *. ΛA:*. ΛB:*. λf : PNeExp V A → PNfExp V B.
  fold (PNfExpF (PNeExp V)) (A → B)
    (ΛR:*.
     λne : NfNe V (A → B) R.
     λabs : NfAbs V (A → B) R.
     λtabs : NfTAbs V (A → B) R.
     λfld : NfFold V (A → B) R.
     abs A B (refl (A → B)) f);

decl mkNfTAbs : (∀V:* → *. ∀A:*.
                 IsAll A → StripAll A → UnderAll A →
                 All Id (PNfExp V) A → PNfExp V A) =
  ΛV:* → *. ΛA:*.
  λp:IsAll A. λs:StripAll A. λu:UnderAll A. λe:All Id (PNfExp V) A.
  fold (PNfExpF (PNeExp V)) A
    (ΛR:*.
     λne : NfNe V A R.
     λabs : NfAbs V A R.
     λtabs : NfTAbs V A R.
     λfld : NfFold V A R.
     tabs p s u e);

decl mkNfFold : (∀V:* → *. ∀F:(* → *) → * → *. ∀A:*.
                 PNfExp V (F (μ F) A) → PNfExp V (μ F A)) =
  ΛV:* → *. ΛF:(* → *) → * → *. ΛA:*. λe : PNfExp V (F (μ F) A).
  fold (PNfExpF (PNeExp V)) (μ F A)
    (ΛR:*.
     λne : NfNe V (μ F A) R.
     λabs : NfAbs V (μ F A) R.
     λtabs : NfTAbs V (μ F A) R.
     λfld : NfFold V (μ F A) R.
     fld F A (refl (μ F A)) e);

decl mkNeVar : (∀V : * → *. ∀A : *. V A → PNeExp V A) =
  ΛV : * → *. ΛA : *. λx : V A.
  fold (PNeExpF V) A
    (ΛR : *.
     λvar : NeVar V A R.
     λapp : NeApp V A R.
     λtapp : NeTApp V A R.
     λunfld : NeUnfold V A R.
     var x);

decl mkNeApp : (∀V : * → *. ∀A : *. ∀B : *.
                PNeExp V (A → B) → PNfExp V A → PNeExp V B) =
  ΛV : * → *. ΛA : *. ΛB : *. λe1 : PNeExp V (A → B). λe2 : PNfExp V A.
```

```
  fold (PNeExpF V) B
    (ΛR : *.
     λvar : NeVar V B R.
     λapp : NeApp V B R.
     λtapp : NeTApp V B R.
     λunfld : NeUnfold V B R.
     app A e1 e2);

decl mkNeTApp : (∀V : * → *. ∀A : *. ∀B : *.
                 IsAll A → Inst A B → PNeExp V A → PNeExp V B) =
  ΛV : * → *. ΛA : *. ΛB : *. λp : IsAll A. λi : Inst A B. λe : PNeExp V A.
  fold (PNeExpF V) B (ΛR : *.
     λvar : NeVar V B R.
     λapp : NeApp V B R.
     λtapp : NeTApp V B R.
     λunfld : NeUnfold V B R.
     tapp A p i e);

decl mkNeUnfold : (∀V : * → *. ∀F : (* → *) → * → *. ∀A : *.
                   PNeExp V (μ F A) → PNeExp V (F (μ F) A)) =
  ΛV : * → *. ΛF : (* → *) → * → *. ΛA : *. λe : PNeExp V (μ F A).
  fold (PNeExpF V) (F (μ F) A)
    (ΛR : *.
     λvar : NeVar V (F (μ F) A) R.
     λapp : NeApp V (F (μ F) A) R.
     λtapp : NeTApp V (F (μ F) A) R.
     λunfld : NeUnfold V (F (μ F) A) R.
     unfld F A (refl (F (μ F) A)) e);

decl SemF : (* → *) → (* → *) → * → * =
  λV : * → *. λSem : * → *. λA : *.
  ∀R:*.
  (PNeExp V A → R) →
  (∀A1:*. ∀A2:*. Eq (A1 → A2) A → (Sem A1 → Sem A2) → R) →
  (IsAll A → StripAll A → UnderAll A → All Id Sem A → R) →
  (∀F : (* → *) → * → *. ∀B : *.
   Eq (μ F B) A →
   Sem (F (μ F) B) →
   R) →
  R;

decl Sem : (* → *) → * → * = λV : * → *. μ (SemF V);

decl SemNe : (* → *) → * → * → * =
  λV : * → *. λA : *. λR : *. PNeExp V A → R;

decl SemArr : (* → *) → * → * → * =
  λV : * → *. λA : *. λR : *.
  ∀A1:*. ∀A2:*.
  Eq (A1 → A2) A →
  (Sem V A1 → Sem V A2) →
  R;

decl SemAll : (* → *) → * → * → * =
  λV : * → *. λA : *. λR : *.
  IsAll A → StripAll A → UnderAll A → All Id (Sem V) A → R;

decl SemMu : (* → *) → * → * → * =
  λV : * → *. λA : *. λR : *.
  ∀F : (* → *) → * → *. ∀B : *.
  Eq (μ F B) A →
  Sem V (F (μ F) B) →
  R;

decl nbeNe : (∀V : * → *. ∀A : *. PNeExp V A → Sem V A) =
```

```
  ΛV : * → *. ΛA : *. λe : PNeExp V A.
  fold (SemF V) A (ΛR : *.
  λne : SemNe V A R.
  λarr : SemArr V A R.
  λall : SemAll V A R.
  λmu : SemMu V A R.
  ne e);

decl semAbs : (∀V : * → *. ∀A : *. ∀B : *.
              (Sem V A → Sem V B) → Sem V (A → B)) =
  ΛV : * → *. ΛA : *. ΛB : *. λf : Sem V A → Sem V B.
  fold (SemF V) (A → B) (ΛR : *.
  λne : SemNe V (A → B) R.
  λarr : SemArr V (A → B) R.
  λall : SemAll V (A → B) R.
  λmu : SemMu V (A → B) R.
  arr A B (refl (A → B)) f);

decl semTAbs : (∀V : * → *. ∀A : *.
               IsAll A → StripAll A → UnderAll A → All Id (Sem V) A →
               Sem V A) =
  ΛV : * → *. ΛA : *.
  λp : IsAll A. λs : StripAll A. λu : UnderAll A. λe : All Id (Sem V) A.
  fold (SemF V) A (ΛR : *.
  λne : SemNe V A R.
  λarr : SemArr V A R.
  λall : SemAll V A R.
  λmu : SemMu V A R.
  all p s u e);

decl semFold : (∀V : * → *. ∀F : (* → *) → * → *. ∀A : *.
               Sem V (F (μ F) A) → Sem V (μ F A)) =
  ΛV : * → *. ΛF : (* → *) → * → *. ΛA : *. λe : Sem V (F (μ F) A).
  fold (SemF V) (μ F A) (ΛR : *.
  λne : SemNe V (μ F A) R.
  λarr : SemArr V (μ F A) R.
  λall : SemAll V (μ F A) R.
  λmu : SemMu V (μ F A) R.
  mu F A (refl (μ F A)) e);

decl reifyArr : (∀V : * → *.
               (∀A:*. Sem V A → PNfExp V A) →
               ∀A : *. ∀A1 : *. ∀A2 : *.
               Eq (A1 → A2) A →
               (Sem V A1 → Sem V A2) →
               PNfExp V A) =
  ΛV : * → *. λreify : (∀A:*. Sem V A → PNfExp V A).
  ΛA : *. ΛA1:*. ΛA2:*. λeq : Eq (A1 → A2) A. λf : Sem V A1 → Sem V A2.
  eq (PNfExp V)
     (mkNfAbs V A1 A2 (λx : PNeExp V A1.
       reify A2 (f (nbeNe V A1 x))));

decl reifyAll : (∀V : * → *. (∀A:*. Sem V A → PNfExp V A) →
                ∀A : *. IsAll A → StripAll A → UnderAll A →
                All Id (Sem V) A → PNfExp V A) =
  ΛV : * → *. λreify : (∀A:*. Sem V A → PNfExp V A). ΛA : *.
  λp : IsAll A. λs : StripAll A. λu : UnderAll A. λe : All Id (Sem V) A.
  mkNfTAbs V A p s u (u (Sem V) (PNfExp V) reify e);

decl reifyMu : (∀V : * → *.
              (∀A:*. Sem V A → PNfExp V A) →
              ∀A : *. ∀F : (* → *) → * → *. ∀B : *.
              Eq (μ F B) A → Sem V (F (μ F) B) → PNfExp V A) =
  ΛV : * → *. λreify : (∀A:*. Sem V A → PNfExp V A).
  ΛA : *. ΛF : (* → *) → * → *. ΛB : *.
```

```
  λeq : Eq (μ F B) A. λe : Sem V (F (μ F) B).
  let e1 : PNfExp V (μ F B) = mkNfFold V F B (reify (F (μ F) B) e) in
  eq (PNfExp V) e1;

decl rec reify : (∀V : * → *. ∀A:*. Sem V A → PNfExp V A) =
  ΛV : * → *. ΛA : *. λe : Sem V A.
  unfold (SemF V) A e
    (PNfExp V A)      -- out
    (mkNfNe V A)     -- ne
    (reifyArr V (reify V) A)
    (reifyAll V (reify V) A)
    (reifyMu V (reify V) A);

decl semApp : (∀V : * → *. ∀B : *. ∀A : *.
               Sem V (B → A) → Sem V B → Sem V A) =
  ΛV:* → *. ΛB:*. ΛA:*. λf:Sem V (B → A). λx:Sem V B.
  unfold (SemF V) (B → A) f (Sem V A)
    -- ne
    (λf : PNeExp V (B → A). nbeNe V A (mkNeApp V B A f (reify V B x)))
    -- arr
    (ΛA1:*. ΛA2:*. λeq:Eq (A1 → A2) (B → A).
     λf:Sem V A1 → Sem V A2.
     let eqL : Eq B A1 = sym A1 B (arrL A1 A2 B A eq) in
     let eqV : Eq A2 A = arrR A1 A2 B A eq in
     eqV (Sem V) (f (eqL (Sem V) x)))
    -- all
    (λp : IsAll (B → A). λs : StripAll (B → A). λu : UnderAll (B → A).
     λe : All Id (Sem V) (B → A). arrIsAll B A p (Sem V A))
    -- mu
    (ΛF : (* → *) → * → *. ΛT : *.
     λeq : Eq (μ F T) (B → A).
     λe : Sem V (F (μ F) T).
     eqArrMu B A F T (sym (μ F T) (B → A) eq) (Sem V A));

decl semTApp : (∀V : * → *. ∀A : *. ∀B : *.
                IsAll A → Inst A B → Sem V A → Sem V B) =
  ΛV:* → *. ΛA:*. ΛB:*. λp:IsAll A. λi:Inst A B. λe:Sem V A.
  unfold (SemF V) A e (Sem V B)
    -- ne
    (λf : PNeExp V A. nbeNe V B (mkNeTApp V A B p i f))
    -- arr
    (ΛA1:*. ΛA2:*.
     λeq : Eq (A1 → A2) A. λf : Sem V A1 → Sem V A2.
     let p1 : IsAll (A1 → A2) = sym (A1 → A2) A eq IsAll p in
     arrIsAll A1 A2 p1 (Sem V B))
    -- all
    (λp : IsAll A. λs : StripAll A. λu : UnderAll A.
     λe : All Id (Sem V) A. i (Sem V) e)
    -- mu
    (ΛF : (* → *) → * → *. ΛT : *.
     λeq : Eq (μ F T) A.
     λe : Sem V (F (μ F) T).
     let p1 : IsAll (μ F T) = sym (μ F T) A eq IsAll p in
     muIsAll F T p1 (Sem V B));

decl semUnfold : (∀V : * → *. ∀F : (* → *) → * → *. ∀A:*.
                  Sem V (μ F A) → Sem V (F (μ F) A)) =
  ΛV : * → *. ΛF : (* → *) → * → *. ΛA : *.
  λe : Sem V (μ F A).
  unfold (SemF V) (μ F A) e (Sem V (F (μ F) A))
    -- ne
    (λx : PNeExp V (μ F A). nbeNe V (F (μ F) A) (mkNeUnfold V F A x))
    -- arr
    (ΛA1 : *. ΛA2 : *.
     λeq : Eq (A1 → A2) (μ F A). λf : Sem V A1 → Sem V A2.
```

```
     let bot : (∀C:*. C) = eqArrMu A1 A2 F A eq in
     bot (Sem V (F (μ F) A)))
    -- all
    (λpAll : IsAll (μ F A). λs : StripAll (μ F A). λu : UnderAll (μ F A).
     λe : All Id (Sem V) (μ F A).
     muIsAll F A pAll
       (Sem V (F (μ F) A)))
    -- mu
    (ΛF1 : (* → *) → * → *. ΛA1:*.
     λeq : Eq (μ F1 A1) (μ F A).
     λe : Sem V (F1 (μ F1) A1).
     eqUnfold F1 A1 F A eq (Sem V) e);

decl sem : (∀V : * → *. ∀A : *. Exp A → Sem V A) =
  ΛV:* → *.
  foldExp (Sem V) (semAbs V) (semApp V)
    (semTAbs V) (semTApp V) (semFold V) (semUnfold V);

decl nbe : (∀A : *. Exp A → NfExp A) =
  ΛA:*. λe:Exp A. ΛV:* → *. reify V A (sem V A e);

decl rec neToExp : (∀V : * → *.
                     (∀A : *. PNfExp (PExp V) A → PExp V A) →
                     ∀A : *. PNeExp (PExp V) A → PExp V A) =
  ΛV:* → *. λnfToExp:(∀A : *. PNfExp (PExp V) A → PExp V A).
  ΛA : *. λe : PNeExp (PExp V) A.
  let neToExp : (∀A : *. PNeExp (PExp V) A → PExp V A) = neToExp V nfToExp in
  unfold (PNeExpF (PExp V)) A e
    (PExp V A)
    -- var
    (λx : PExp V A. x)
    -- app
    (ΛB : *. λf : PNeExp (PExp V) (B → A). λx : PNfExp (PExp V) B.
     app V B A (neToExp (B → A) f) (nfToExp B x))
    -- tapp
    (ΛB : *. λp : IsAll B. λi : Inst B A. λf : PNeExp (PExp V) B.
     tapp V B A p i (neToExp B f))
    -- unfold
    (ΛF : (* → *) → * → *. ΛB : *.
     λeq : Eq (F (μ F) B) A. λe : PNeExp (PExp V) (μ F B).
     let e1 : PExp V (F (μ F) B) = unfld V F B (neToExp (μ F B) e) in
     eq (PExp V) e1);

decl nfToExpVar : (∀V : * → *.
                   (∀A : *. PNfExp (PExp V) A → PExp V A) →
                   ∀A : *. NfNe (PExp V) A (PExp V A)) =
  ΛV : * → *. λnfToExp : (∀A : *. PNfExp (PExp V) A → PExp V A). ΛA : *.
  neToExp V nfToExp A;

decl nfToExpAbs : (∀V : * → *.
                   (∀A : *. PNfExp (PExp V) A → PExp V A) →
                   ∀A : *. NfAbs (PExp V) A (PExp V A)) =
  ΛV:* → *. λnfToExp:(∀A : *. PNfExp (PExp V) A → PExp V A).
  ΛA:*. ΛA1:*. ΛA2:*. λeq:Eq (A1 → A2) A.
  λf : PNeExp (PExp V) A1 → PNfExp (PExp V) A2.
  eq (PExp V)
     (abs V A1 A2
       (λx : PExp V A1.
        let neX : PNeExp (PExp V) A1 = mkNeVar (PExp V) A1 x in
        nfToExp A2 (f neX)));

decl nfToExpTAbs : (∀V : * → *.
                     (∀A : *. PNfExp (PExp V) A → PExp V A) →
                     ∀A : *. NfTAbs (PExp V) A (PExp V A)) =
  ΛV : * → *. λnfToExp : (∀A : *. PNfExp (PExp V) A → PExp V A).
```

```
  ΛA : *. λp : IsAll A. λs : StripAll A. λu : UnderAll A.
  λe : All Id (PNfExp (PExp V)) A.
  tabs V A p s u (u (PNfExp (PExp V)) (PExp V) nfToExp e);

decl nfToExpFold : (∀V : * → *.
                    (∀A : *. PNfExp (PExp V) A → PExp V A) →
                    ∀A : *. NfFold (PExp V) A (PExp V A)) =
  ΛV : * → *. λnfToExp : (∀A : *. PNfExp (PExp V) A → PExp V A).
  ΛA : *. ΛF : (* → *) → * → *. ΛB : *.
  λeq : Eq (μ F B) A. λe : PNfExp (PExp V) (F (μ F) B).
  let e1 : PExp V (μ F B) = fld V F B (nfToExp (F (μ F) B) e) in
  eq (PExp V) e1;

decl rec nfToExp : (∀V : * → *. ∀A : *. PNfExp (PExp V) A → PExp V A) =
  ΛV : * → *. ΛA : *. λe : PNfExp (PExp V) A.
  unfold (PNfExpF (PNeExp (PExp V))) A e
    (PExp V A)
    (nfToExpVar V (nfToExp V) A) (nfToExpAbs V (nfToExp V) A)
    (nfToExpTAbs V (nfToExp V) A) (nfToExpFold V (nfToExp V) A);

decl unNf : (∀T : *. NfExp T → Exp T) =
  ΛT : *. λe : NfExp T. ΛV : * → *.
  nfToExp V T (e (PExp V));

decl norm : (∀A:*. Exp A → Exp A) = ΛA:*. λe : Exp A. (unNf A (nbe A e));
```

D.9 POPL'16 Meta-Programs

D.9.1 foldExp

```
decl foldAbs : (∀V:* → *. ∀A:*. (∀A:*. PExp V A → V A) →
                (∀A:*. ∀B:*. (V A → V B) → V (A → B)) →
                AbsF V A (V A)) =
  ΛV : * → *. ΛA : *. λfoldExp : (∀A:*. PExp V A → V A).
  λabs : (∀A:*. ∀B:*. (V A → V B) → V (A → B)).
  ΛA1:*. ΛA2:*. λeq:Eq (A1 → A2) A. λf:PExp V A1 → PExp V A2.
  eq V (abs A1 A2 (λx : V A1.
         foldExp A2 (f (var V A1 x))));

decl foldApp : (∀V:* → *. ∀A:*. (∀A:*. PExp V A → V A) →
                (∀A:*. ∀B:*. V (A → B) → V A → V B) →
                AppF V A (V A)) =
  ΛV : * → *. ΛA : *. λfoldExp : (∀A:*. PExp V A → V A).
  λapp : (∀A:*. ∀B:*. V (A → B) → V A → V B).
  ΛB : *. λe1 : PExp V (B → A). λe2 : PExp V B.
  app B A (foldExp (B → A) e1) (foldExp B e2);

decl foldTAbs : (∀V:* → *. ∀A:*. (∀A:*. PExp V A → V A) →
                 (∀A:*. IsAll A → StripAll A → UnderAll A → (All Id V A) → V A) →
                 TAbsF V A (V A)) =
  ΛV : * → *. ΛA : *. λfoldExp : (∀A:*. PExp V A → V A).
  λtabs : (∀A:*. IsAll A → StripAll A → UnderAll A → (All Id V A) → V A).
  λp : IsAll A. λs : StripAll A. λu : UnderAll A. λe : All Id (PExp V) A.
  tabs A p s u (u (PExp V) V foldExp e);

decl foldTApp : (∀V : * → *. ∀A:*. (∀A:*. PExp V A → V A) →
                 (∀A:*. ∀B:*. IsAll A → Inst A B → V A → V B) →
                 TAppF V A (V A)) =
  ΛV : * → *. ΛA : *. λfoldExp : (∀A:*. PExp V A → V A).
  λtapp : (∀A:*. ∀B:*. IsAll A → Inst A B → V A → V B).
  ΛB:*. λp : IsAll B. λi : Inst B A. λe : PExp V B.
  tapp B A p i (foldExp B e);

decl foldFold : (∀V : * → *. ∀A:*. (∀A:*. PExp V A → V A) →
                 (∀F:(* → *) → * → *. ∀A:*. V (F (μ F) A) → V (μ F A)) →
                 FoldF V A (V A)) =
```

```
  ΛV : * → *. ΛA : *. λfoldExp : (∀A:*. PExp V A → V A).
  λfld : (∀F : (* → *) → * → *. ∀A : *. V (F (μ F) A) → V (μ F A)).
  ΛF:(* → *) → * → *. ΛB:*. λeqFold:Eq (μ F B) A. λe:PExp V (F (μ F) B).
  eqFold V (fld F B (foldExp (F (μ F) B) e));

decl foldUnfold : (∀V : * → *. ∀A:*. (∀A:*. PExp V A → V A) →
                   (∀F : (* → *) → * → *. ∀A : *. V (μ F A) → V (F (μ F) A)) →
                   UnfoldF V A (V A)) =
  ΛV : * → *. ΛA : *. λfoldExp : (∀A:*. PExp V A → V A).
  λunfld : (∀F : (* → *) → * → *. ∀A : *. V (μ F A) → V (F (μ F) A)).
  ΛF : (* → *) → * → *. ΛB : *. λeq : Eq (F (μ F) B) A. λe : PExp V (μ F B).
  eq V (unfld F B (foldExp (μ F B) e));

decl FoldAbs : (* → *) → * =
  λV : * → *. ∀A:*. ∀B:*. (V A → V B) → V (A → B);

decl FoldApp : (* → *) → * =
  λV : * → *. ∀A:*. ∀B:*. V (A → B) → V A → V B;

decl FoldTAbs : (* → *) → * =
  λV : * → *. ∀A:*. IsAll A → StripAll A → UnderAll A → (All Id V A) → V A;

decl FoldTApp : (* → *) → * =
  λV : * → *. ∀A:*. ∀B:*. IsAll A → Inst A B → V A → V B;

decl FoldFold : (* → *) → * =
  λV : * → *. ∀F : (* → *) → * → *. ∀A : *. V (F (μ F) A) → V (μ F A);

decl FoldUnfold : (* → *) → * =
  λV : * → *. ∀F : (* → *) → * → *. ∀A : *. V (μ F A) → V (F (μ F) A);

decl rec foldExpV : (∀V : * → *.
                     FoldAbs V → FoldApp V →
                     FoldTAbs V → FoldTApp V →
                     FoldFold V → FoldUnfold V →
                     ∀A:*. PExp V A → V A) =
  ΛV : * → *.
  λabs : FoldAbs V. λapp : FoldApp V.
  λtabs : FoldTAbs V. λtapp : FoldTApp V.
  λfld : FoldFold V. λunfld : FoldUnfold V.
  let foldExp : (∀A:*. PExp V A → V A) =
    foldExpV V abs app tabs tapp fld unfld
  in
  ΛA:*. λe : PExp V A.
  unfold (PExpF V) A e (V A)
    (λx : V A. x)
    (foldAbs V A foldExp abs)
    (foldApp V A foldExp app)
    (foldTAbs V A foldExp tabs)
    (foldTApp V A foldExp tapp)
    (foldFold V A foldExp fld)
    (foldUnfold V A foldExp unfld);

decl foldExp : (∀V : * → *.
                FoldAbs V → FoldApp V →
                FoldTAbs V → FoldTApp V →
                FoldFold V → FoldUnfold V →
                ∀A:*. Exp A → V A) =
  ΛV : * → *.
  λabs : FoldAbs V. λapp : FoldApp V.
  λtabs : FoldTAbs V. λtapp : FoldTApp V.
  λfld : FoldFold V. λunfld : FoldUnfold V.
  ΛA:*. λe:Exp A.
  foldExpV V abs app tabs tapp fld unfld A (e V);
```

D.9.2 unquote

```
decl unquoteAbs : FoldAbs Id = ΛA:*. ΛB:*. λf : A → B. f;

decl unquoteApp : FoldApp Id = ΛA:*. ΛB:*. λf : A → B. f;

decl unquoteTAbs : FoldTAbs Id =
  ΛA:*. λp : IsAll A. λs : StripAll A. λu : UnderAll A. λe : All Id Id A.
  let eq : Eq (All Id Id A) A = unAll A p Id in
  eq Id e;

decl unquoteTApp : FoldTApp Id =
  ΛA:*. ΛB:*. λp : IsAll A. λi : Inst A B. λx : A.
  let eq : Eq A (All Id Id A) = sym (All Id Id A) A (unAll A p Id) in
  i Id (eq Id x);

decl unquoteFold : FoldFold Id =
  ΛF: (* → *) → * → *. ΛA:*. λx : F (μ F) A. fold F A x;

decl unquoteUnfold : FoldUnfold Id =
  ΛF: (* → *) → * → *. ΛA:*. λx : μ F A. unfold F A x;

decl unquote : (∀A:*. Exp A → A) =
  foldExp Id unquoteAbs unquoteApp unquoteTAbs unquoteTApp unquoteFold unquoteUnfold;
```

D.9.3 cps

```
decl Ct : * → * = λA:*. ∀B:*. (A → B) → B;

decl ct : (∀A:*. A → Ct A) =
  ΛA:*. λx:A. ΛB:*. λf:A → B. f x;

decl CPS1F : (* → *) → * → * =
  λCPS1:* → *. λA:*.
  Typecase
    (λX:*. λY:*. Ct (CPS1 X) → Ct (CPS1 Y))
    Id (λX:*. Ct (CPS1 X))
    (λF : (* → *) → * → *. λB : *. Ct (CPS1 (F (μ F) B)))
    A;

decl CPS1 : * → * = μ CPS1F;
decl CPS : * → * = λA:*. Ct (CPS1 A);

decl cpsAbs : FoldAbs CPS =
  ΛA:*. ΛB:*. λf : CPS A → CPS B.
  ΛV:*. λk : CPS1 (A → B) → V.
  k (fold CPS1F (A → B) f);

decl cpsApp : FoldApp CPS =
  ΛA:*. ΛB:*. λe1 : CPS (A → B). λe2 : CPS A.
  ΛV:*. λk : CPS1 B → V.
  e1 V (λf : CPS1 (A → B). unfold CPS1F (A → B) f e2 V k);

decl eqCPSAll : (∀A : *. IsAll A → Eq (CPS1F CPS1 A) (All Id CPS A)) =
  ΛA:*. λp : IsAll A.
  tcAll A p (λX:*. λY:*. CPS X → CPS Y) Id CPS
            (λF:(* → *) → * → *. λB:*. CPS (F (μ F) B));

decl cpsTAbs : FoldTAbs CPS =
  ΛA:*. λp:IsAll A. λs:StripAll A. λu:UnderAll A. λe:All Id CPS A.
  let e1 : CPS1F CPS1 A =
    sym (CPS1F CPS1 A) (All Id CPS A) (eqCPSAll A p) Id e
  in
  let e2 : CPS1 A = fold CPS1F A e1 in
  ΛV : *. λk : CPS1 A → V. k e2;
```

```
decl cpsTApp : FoldTApp CPS =
  ΛA:*. ΛB:*. λp : IsAll A. λi : Inst A B. λe : CPS A.
  ΛV : *. λk : CPS1 B → V.
  e V (λe1 : CPS1 A.
       let e2 : CPS1F CPS1 A = unfold CPS1F A e1 in
       let e3 : All Id CPS A = eqCPSAll A p Id e2 in
       let e4 : CPS B = i CPS e3 in
       e4 V k);

decl cpsFold : FoldFold CPS =
  ΛF : (* → *) → * → *. ΛA : *. λe : CPS (F (μ F) A).
  let e1 : CPS1F CPS1 (μ F A) = e in
  let e2 : CPS1 (μ F A) = fold CPS1F (μ F A) e1 in
  ΛV : *. λk : CPS1 (μ F A) → V. k e2;

decl cpsUnfold : FoldUnfold CPS =
  ΛF:(* → *) → * → *. ΛA:*. λe : CPS (μ F A).
  ΛV : *. λk : CPS1 (F (μ F) A) → V.
  e V (λe1 : CPS1 (μ F A).
       let e2 : CPS1F CPS1 (μ F A) = unfold CPS1F (μ F A) e1 in
       let e3 : CPS (F (μ F) A) = e2 in
       e3 V k);

decl cps : (∀A:*. Exp A → CPS A) =
  foldExp CPS cpsAbs cpsApp cpsTAbs cpsTApp cpsFold cpsUnfold;
```

D.10 Normal form checker

In previous work [8], we implemented a normal form checker as a fold. While this is possible for $F_\omega^{\mu i}$ too, it would only work on closed representations (with types of the form `Exp T`). This normal form checker is not implemented as a fold, which allows it to check open representations (of type `PExp V T` for any `V`, `T`). We use this capability in `step` to determine where in the term the left-most redex is.

```
-- pair of bools: normal, neutral
decl Bools : *    = Pair Bool Bool;
decl bools : Bool → Bool → Bools  = pair Bool Bool;
decl fstBools : Bools → Bool = fst Bool Bool;
decl sndBools : Bools → Bool = snd Bool Bool;

decl nfNeVar : (∀V:* → *. ∀A:*. VarF V A Bools) =
  ΛV : * → *. ΛA:*. λx : V A. bools true true;

decl nfNeAbs : (∀V:* → *. ∀A:*. (∀A:*. PExp V A → Bools) → AbsF V A Bools) =
  ΛV : * → *. ΛA:*. λnfNe : (∀A:*. PExp V A → Bools).
  ΛA1:*. ΛA2:*. λeq : Eq (A1 → A2) A. λf : PExp V A1 → PExp V A2.
  let x : PExp V A1 = var V A1 (bottom (V A1)) in
  bools (fstBools (nfNe A2 (f x))) false;

decl nfNeApp : (∀V:* → *. ∀A:*. (∀A:*. PExp V A → Bools) → AppF V A Bools) =
  ΛV : * → *. ΛA:*. λnfNe : (∀A:*. PExp V A → Bools).
  ΛB:*. λf:PExp V (B → A). λx:PExp V B.
  let f_nfNe : Bools = nfNe (B → A) f in
  let x_nfNe : Bools = nfNe B x in
  let ne : Bool = and (sndBools f_nfNe) (fstBools x_nfNe) in
  bools ne ne;

decl nfNeTAbs : (∀V:* → *. ∀A:*. (∀A:*. PExp V A → Bools) → TAbsF V A Bools) =
  ΛV : * → *. ΛA:*. λnfNe : (∀A:*. PExp V A → Bools).
  λp : IsAll A. λs : StripAll A. λu : UnderAll A. λe : All Id (PExp V) A.
  let e1 : All Id (λA:*. Bools) A = u (PExp V) (λA:*. Bools) nfNe e in
  let bs : Bools = s Bools e1 in
  bools (fstBools bs) false;

decl nfNeTApp : (∀V:* → *. ∀A:*. (∀A:*. PExp V A → Bools) → TAppF V A Bools) =
  ΛV:* → *. ΛA:*. λnfNe : (∀A:*. PExp V A → Bools).
  ΛB:*. λp:IsAll B. λi:Inst B A. λe:PExp V B.
  let ne : Bool = sndBools (nfNe B e) in
  bools ne ne;
```

```
decl nfNeFold : (∀V:* → *. ∀A:*. (∀A:*. PExp V A → Bools) → FoldF V A Bools) =
  ΛV:* → *. ΛA:*. λnfNe : (∀A:*. PExp V A → Bools).
  ΛF:(* → *) → * → *. ΛB:*. λeqFold : Eq (μ F B) A. λe:PExp V (F (μ F) B).
  bools (fstBools (nfNe (F (μ F) B) e)) false;

decl nfNeUnfold : (∀V:* → *. ∀A:*. (∀A:*. PExp V A → Bools) → UnfoldF V A Bools) =
  ΛV:* → *. ΛA:*. λnfNe:(∀A:*. PExp V A → Bools).
  ΛF:(* → *) → * → *. ΛB:*. λeq:Eq (F (μ F) B) A. λe:PExp V (μ F B).
  let ne : Bool = sndBools (nfNe (μ F B) e) in
  bools ne ne;

decl rec nfNe : (∀V : * → *. ∀A:*. PExp V A → Bools) =
  ΛV : * → *. ΛA:*. λe : PExp V A.
  unfold (PExpF V) A e Bools
    (nfNeVar V A)
    (nfNeAbs V A (nfNe V))
    (nfNeApp V A (nfNe V))
    (nfNeTAbs V A (nfNe V))
    (nfNeTApp V A (nfNe V))
    (nfNeFold V A (nfNe V))
    (nfNeUnfold V A (nfNe V));

decl nfV : (∀V : * → *. ∀A:*. PExp V A → Bool) =
  ΛV : * → *. ΛA:*. λe : PExp V A. fstBools (nfNe V A e);

decl nf : (∀A:*. Exp A → Bool) = ΛA:*. λe:Exp A. nfV Id A (e Id);
```

D.11 size

```
decl KNat : * → * = λA:*. Nat;

decl sizeAbs : FoldAbs KNat =
  ΛA:*. ΛB:*. λf:Nat → Nat. succ (f (succ zero));

decl sizeApp : FoldApp KNat =
  ΛA:*. ΛB:*. λf:Nat. λx:Nat. succ (plus f x);

decl sizeTAbs : FoldTAbs KNat =
  ΛA:*. λp : IsAll A. λs:StripAll A. λu : UnderAll A. λf : All Id KNat A.
  succ (s Nat f);

decl sizeTApp : FoldTApp KNat =
  ΛA:*. ΛB:*. λp : IsAll A. λi : Inst A B. λf : Nat. succ f;

decl sizeFold : FoldFold KNat =
  ΛF : (* → *) → * → *. ΛA:*. λn : Nat. succ n;

decl sizeUnfold : FoldUnfold KNat =
  ΛF : (* → *) → * → *. ΛA:*. λn : Nat. succ n;

decl size : (∀A:*. Exp A → Nat) =
  foldExp KNat sizeAbs sizeApp sizeTAbs sizeTApp sizeFold sizeUnfold;
```

D.12 isAbs

```
decl KBool : * → * = λA:*. Bool;

decl isAbsAbs : FoldAbs KBool =
  ΛA:*. ΛB:*. λf:Bool → Bool. true;

decl isAbsApp : FoldApp KBool =
  ΛA:*. ΛB:*. λf:Bool. λx:Bool. false;

decl isAbsTAbs : FoldTAbs KBool =
  ΛA:*. λp:IsAll A. λs:StripAll A. λu:UnderAll A. λf:All Id KBool A. true;
```

```
decl isAbsTApp : FoldTApp KBool =
  ΛA:*. ΛB:*. λp:IsAll A. λi:Inst A B. λf:Bool. false;

decl isAbsFold : FoldFold KBool =
  ΛF : (* → *) → * → *. ΛA:*. λn : Bool. false;

decl isAbsUnfold : FoldUnfold KBool =
  ΛF : (* → *) → * → *. ΛA:*. λn : Bool. false;

decl isAbs : (∀A:*. Exp A → Bool) =
  foldExp KBool isAbsAbs isAbsApp isAbsTAbs isAbsTApp isAbsFold isAbsUnfold;
```