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 $F_{	ext{exp}}^i$, the first $\lambda$-calculus that supports a polymorphically-typed self-evaluator. Our calculus extends $F_{\omega}$ 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 $\text{ifself}$; examples include Haskell [26], JavaScript [17], Python [32], Ruby [44], Scheme [3], and Standard ML [33]. The use of $\text{ifself}$ 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, $(\text{String} \rightarrow \text{String})$, where the input string represents a program and where the output string represents the result. However, this approach ignores that the source program type checks, and gives no guarantee that the interpreter preserves the type of its input. How can we do better type checking of self-interpreters? First, suppose we have a better representation scheme $\text{quote}(\cdot)$ and a type function $\text{Exp}$ such that if $e : T$, then $\text{quote}(e) : \text{Exp} T$. This enables us to consider two polymorphic types of self-interpreters:

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

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

The functionality of a self-recognizer $\text{unquote}$ is to recover a program from its representation, while the functionality of a self-evaluator $\text{eval}$ is to evaluate the represented program and produce a representation of the result. The representation between a self-recognizer and a self-evaluator is illustrated in Figure 1. The relation between evaluations 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 $\text{unquote}$ to $e$ 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 $\lambda$-calculus $F_{\omega}$. In previous work we presented self-recognizers with type $(1)$ for System U [7], a typed $\lambda$-calculus with decidable type checking, and for $F_{\omega}$ [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 $\lambda$-calculus?

Our result: Yes, we present three self-evaluators for a typed $\lambda$-calculus with decidable type checking. Our calculus, $F_{\text{exp}}^i$, extends $F_{\omega}$ with recursive types and intensional type functions.

Our starting point is an evaluator for simply-typed $\lambda$-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 $F^u_\omega$, which is designed to support these constructs.

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

To tie the knot for a language with type equality, we need to consider two questions. First, how expressive must a theory of type equality be in order to implement a typed evaluator for a particular object language? Second, what meta-language features are needed to represent and evaluate a particular theory of type equality? In Section 2 we show that to evaluate STLC, type equality between arrow types should be decomposable. In particular, if we know $(A \rightarrow B) = (S \rightarrow T)$, then we also know $A = S$ and $B = T$. What then is needed to represent and evaluate decomposable type equalities? Haskell implements type equality using built-in type equality coercions [36]. These support decomposition, but have complex typing rules and evaluation semantics that make representation and evaluation difficult. On the other hand, Leibniz equality proofs [5, 28, 39] can be encoded in $\lambda$-terms typeable in pure $F^u_\omega$. This means that representing and evaluating Leibniz equality proofs is no harder than representing and evaluating $F^u_\omega$. However, Leibniz equality proofs are not decomposable in $F^u_\omega$. 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 $F^u_\omega$ 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 $\beta$-normal form. The first only reduces closed terms, while the others may reduce under abstractions. We also implement a self-recognizer $\text{unquote}$ 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 $F^u_\omega$ and our meta-programs, as well as an appendix containing proofs of the theorems stated in this paper.

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 $F^u_\omega$. In Section 4 we first implement type equality proofs for simple types in $F^u_\omega$ 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 $F^u_\omega$ 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 $\lambda$-Calculus (STLC) terms in Haskell using GADTs. The representation is Higher-Order Abstract Syntax (HOAS), which means that STLC variables are represented as Haskell variables that range over representations, and we use Haskell functions to bind STLC variables. In the Abs constructor, the function type $(\text{Exp} \ t \rightarrow \text{Exp} \ t)$ corresponds to a STLC term of type $\text{Exp} \ t \rightarrow \text{Exp} \ t$ that includes a free variable of type $\text{Exp} \ t$.

Also in Figure 2 is a meta-circular evaluator with type (2). That type guarantees that $\text{eval}$ 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 $\beta$-reduction (function application) to implement STLC $\beta$-reduction.

The evaluator $\text{eval}$ implements weak head-normal evaluation. This means that it reduces the left-most $\beta$-redex, but does not evaluate under $\lambda$-abstractions or in the argument position of applications. If $e = \text{Abs} \ f$, then $e$ is already in weak head-normal form, and $\text{eval} \ e = e$. If $e = \text{App} \ e_1 \ e_2$, we first recursively evaluate $e_1$, letting $e_1'$ be value of $e_1$. If $e_1'$ is an abstraction $\text{Abs} \ f$, then $\text{App} \ e_1' \ e_2$ is a redex. We reduce it by applying $f$ to $e_2$, and then we recursively evaluate the result. If $e_1'$ is not an abstraction, then we return $\text{App} \ e_1' \ e_2$.

We now consider how Haskell type checks $\text{eval}$. First, the type annotation on $\text{eval}$ determines that $\text{App} \ e_1 \ e_2$ has type $\text{Exp} \ t$. According to the type of $\text{App}$, $e_1$ has type $\text{Exp} \ t$ and $e_2$ has type $\text{Exp} \ t$, for some type $t$. Since $\text{eval}$ 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

Figure 3: STLC using ADTs and equality coercions

code

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
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 existential 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 reimplments 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 GADTs 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
c coercions manually. We can implement the functions in Figure 4
using Haskell, as we show in the appendix, or we can implement
them in Fω, 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

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 Fω. 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
hypostases 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) t1, 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). Similary,
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 encod-
ing [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 (\Abs fApp -> fApp refl f)

app :: (Exp (t1 -> t2) -> Exp t1 -> Exp t2)
app el e2 = Exp (\Abs el2 ->
  let el1 = eval el in
  matchExp el1
  (\eq f ->
   let eqL = eqApp (sym (arrL eq))
          eqR = eqApp (arrR eq)
          f' = coerce eqR . f . coerce eqL
   in f' e2)
  (\_ _ -> app el1 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 to 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 (\_ \_ \_ -> _), recursive types, and type equality proofs. In the next section we present F^\omega_\mu, which supports each of these features. It extends F_\omega with iso-recursive \mu 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, which are similar to Figure 6. Then we go beyond STLC and implement our self-representation and self-evaluator for F^\omega_\mu.

3. System F^\omega_\mu

System F^\omega_\mu is defined in Figure 7. It extends F_\omega with iso-recursive \mu types and a type operator Typecase that is used to define intensional type functions. The kinds are the same as in F_\omega. The kind + classifies base types (the types of terms), and arrow kinds classify type level functions. The types are those of F_\omega, plus \mu and Typecase. The rules of type formation are those of F_\omega, plus axioms for \mu and Typecase. The terms are those of F_\omega, plus fold and unfold that respectively contract or expand a recursive type. The rules of term formation are those of F_\omega, 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_\omega, plus the three reduction rules for Typecase. The semantics is full F_\omega \beta-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, \texttt{Bool} range over types. We will often use f for higher-kindred 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 \mu 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 \ast \to \ast (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 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 \( \{ \} \vdash e : T \), then e is a normal form, or there exists an e’ such that \( \{ \} \vdash e’ : T \) and e \( \vdash \) e’.

Theorem 3.2. Type checking is decidable.

3.2 Syntactic Sugar and Abbreviations

System F^\omega_\mu 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 \forall T : *( T \to T ) \to 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.
(kinds) K ::= * | K₁ → K₂
(types) T ::= X | T₁ → T₂ | ∀X:K.T | T₁ T₂ | μ T₁ T₂ | μ | Typecase
(terms) e ::= x | e₁ e₂ | λX:T.e | μ T₁ T₂ | fold T₁ T₂ e | unfold T₁ T₂ e
(environments) Γ ::= ⟨⟩ | Γ, (x:T)
(normal form terms) v ::= n | (λx:T.v) | (λX:K.v) | fold T₁ T₂ v
(neutral terms) n ::= x | n v | n T | unfold T₁ T₂ n

Grammar

\[
\begin{array}{l}
(X:K) \in \Gamma \\
\Gamma \vdash X : K \\
Γ \vdash T₁ : * \quad Γ \vdash T₂ : *
\end{array}
\]

\[
\begin{array}{l}
Γ \vdash T₁ \rightarrow T₂ : * \\
 Γ, (X:K) \vdash T : * \\
Γ \vdash T₁ \rightarrow T₂ : * \\
Γ \vdash T :
\end{array}
\]

\[
\begin{array}{l}
\Gamma \vdash \mu : ((\rightarrow \rightarrow) \rightarrow \star) \\
\Gamma \vdash Typecase : (\star \rightarrow \star \rightarrow \star) \\
\quad \rightarrow (\star \rightarrow \star) \\
\quad \rightarrow ((\rightarrow \star) \rightarrow \star) \\
\quad \rightarrow \star \rightarrow \star \\
\end{array}
\]

Type Formation

\[
\begin{array}{l}
T \\ T₁ \\ T₂ \\ T₁' \\ T₂' \\
\end{array}
\]

Term Formation

\[
\begin{array}{l}
(\lambda X:K.T) \equiv (\lambda X:K.T') \\
\end{array}
\]

Type Equivalence

Figure 7: Definition of F^fix
We use $S \times T$ for pair types, which can be easily encoded in System $F^\mu$. Intuitively, $\times$ is an infix type function of kind $\mu \to \to *$. We use $(x, y)$ to construct the pair of $x$ and $y$. $\text{fst}$ and $\text{snd}$ project the first and second component from a pair, respectively.

### 3.3 Intensional Type Functions

Our Typecase operator allows us to define type functions that depend on the top-level structure of a base type. It is parameterized by four case functions, one for arrow types, two for quantified types, and one for recursive types. When applied to an arrow type or a recursive type, Typecase decomposes the input type and applies the corresponding case function to the components. For example, when applied to an arrow type $T_1 \to T_2$, Typecase applies the case function for arrows to $T_1$ and $T_2$. When applied to a recursive type $\mu F \ T$, 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 $F^\mu$. Previous work on ITA for quantified types [37] would decompose a quantified type $\forall X : \cdot. T$ into the kind $\cdot$ and a type function of kind $\cdot \to *$. The components would then be passed as arguments to a function case for quantified types. This approach requires kind polymorphism in types, which is outside of $F^\mu$. 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 $F = \lambda X : \cdot. T$ where $ArrL$ and $ArrR$ are the case functions for arrow types and recursive types, respectively. $Out$ and $In$ are the case functions for quantified types, with $Out$ being applied outside the type, and $In$ inside the type. Then $F (\forall X : \cdot. T) \equiv Out (\forall X : \cdot. T)$. Note that to avoid variable capture, we require that $X$ not occur free in $T$ (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 $\bot = (\forall T : \cdot. T)$, which we use to indicate an error. The type $\bot$ is only inhabited by non-normalizing terms.

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

\[
\text{ArrL } T \equiv \begin{cases} 
T_1 & \text{if } T \equiv T_1 \to T_2 \\
\bot & \text{otherwise} 
\end{cases}
\]

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

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

### 3.4 Equality Proofs for Simple Types

Implementation of type equality proofs in $F^\mu$.

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

\[
Unfold T \equiv \begin{cases} 
F (\mu F) A & \text{if } T \equiv \mu F A \\
\bot & \text{otherwise} 
\end{cases}
\]

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 $F^\mu$

In Section 4.1 we implement decomposable type equality proofs in $F^\mu$ 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 $F^\mu$ self-representation and self-evaluators.

#### 4.1 Equality Proofs for Simple Types

Figure 9 shows the $F^\mu$ 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 $F$-style languages. The type $Eq A B$ is defined as $\forall F : \cdot \to * \to *. F A \to F B$. 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
A with B and resulting in the type E B A. Transitivity is encoded by 
\( trans \), which uses function composition to combine two coercions. 
A proof of type E 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 \( E \), subset of \( F \). The addition of intensional type functions 
allows \( F \) to decompose Leibniz equality proofs. The key is that 
eqApp is stronger in \( F \) than in \( E \), because the type function F 
can be intensional. In particular, arr and arr are defined using eqApp 
with the intensional type functions \( E \) and \( E \), respectively.

Figure 10 shows the STLC representation and evaluator in \( F \),. It 
uses a Mogensen-Scott encoding similar to the one in Figure 6, with 
a few notable differences. The type Exp is a stratified version of Exp. 
In particular, it uses a A-abstraction to unite the recursive knot. Exp 
is defined as \( E \), which re-ties the knot. Now Exp T and Exp T 
are isomorphic, with unfold Exp T converting from Exp T to Exp T, 
and fold Exp T converting from 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 Exp 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.

\[
\text{decl ExpF : } (* \to *) \to * \to * = \\
\lambda \text{Exp : } * \to *, \lambda T : * \to *, \forall R : *.
\]
\[
\text{fold ExpF (A B)} = \\
(\lambda F. \text{eqApp. fAbs A B (refl (A B) f))};
\]
\[
\text{matchExp : } (* \to *) \to * \to * = \\
(\lambda T1 T2. \text{eq f e}) = \\
(\lambda T1 T2. \text{let e1' : Exp (S T) = eval (S T) e1 in} \\
matchExp (S T) e1'; \\
\text{let e2 : Exp A \to Exp B.} = \\
\text{let eq : Exp S (Exp S) (Exp E) A =} \\
\text{eqApp S A Exp (sym A S (arr A A S B S T eq)) in} \\
\text{let eqR : Exp B T Exp (arrR A A S B S T eq) in} \\
\text{let f : Exp S \to Exp T = \lambda x : Exp S.} = \\
\text{let x' : Exp A = coerc (Exp S) (Exp A) eqL x in} \\
\text{coerce (Exp B) (Exp T) eqR (f x')} = \\
\text{eval (T f e2) =} \\
(\lambda T2. \text{let e3 e4. app S T e1' e2)};
\]

Figure 10: Encoding and evaluation of STLC in \( F \),

In this section, we move beyond STLC in preparation for our 
self-representation of \( F \). We will focus on the question: How can we 
establish that an unknown type is a quantified 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 \( \to \) T2) 
T. This will work for any arrow type because T1 and T2 must have kind 
\( * \) in order for T1 \( \to \) T2 to kind check. The case for recursive types is 
similar. In \( F \), a recursive type \( F \) \( \text{kindcheck} \) only if \( \text{is recursive} \) \( \text{kind} \) + 1 \( \to \) \( * \) and has kind \( + \). Therefore, we can establish that some type T is a recursive type by abstracting over \( F \) and a proof of 

\[
\text{tca1} E T = \lambda \text{arr. \text{arr. } } \lambda \text{mu. refl } (\text{out } (\forall X:K. T)) \\
\text{isAll} E T = (\text{tca1} E T, \text{isAll} E T)
\]

Figure 11: IsAll proofs.

4.2 Beyond Simple Types

In this section, we move beyond STLC in preparation for our 
self-representation of \( F \). We will focus on the question: How can we 
establish that an unknown type is a quantified 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 \( \to \) T2) 
T. This will work for any arrow type because T1 and T2 must have kind 
\( * \) in order for T1 \( \to \) T2 to kind check. The case for recursive types is 
similar. In \( F \), a recursive type \( F \) \( \text{kind} \) only if \( \text{recursive} \) \( \text{kind} \) + 1 \( \to \) \( * \) and has kind \( + \). Therefore, we can establish that some type T is a recursive type by abstracting over \( F \) and a proof of 

\[
\text{tca1} E T = \lambda \text{arr. \text{arr. } } \lambda \text{mu. refl } (\text{out } (\forall X:K. T)) \\
\text{isAll} E T = (\text{tca1} E T, \text{isAll} E T)
\]
on a GADT. For example, suppose we added integers to our Haskell representation of STLC. When matching on a representation of type \(\text{Exp Int}\), the \(\text{Abs}\) case would provide a proof that \(\text{Int}\) is equal to an arrow type \(\text{t1} \rightarrow \text{t2}\), 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 \(\perp\):

\[
\begin{align*}
\text{eqArrMu} & : \forall A B F T. \text{Eq} \ (A \rightarrow B) \ (\mu F T) \rightarrow \perp \\
\text{arrIsAll} & : \forall A B. \text{IsAll} \ (A \rightarrow B) \rightarrow \perp \\
\text{muIsAll} & : \forall T. \text{IsAll} \ (\mu F T) \rightarrow \perp
\end{align*}
\]

There are three kinds of contradictory equality proofs in \(\text{F}^\mu\): 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 \(\text{F}^\mu\) 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 \(\text{PExp}\) is parametric in \(V\), which determines the type of free variables in a representation. Intuitively, \(\text{PExp}\) can be understood as the type of representations that may contain free variables. The type \(\text{Exp}\) quantifies over \(V\), which ensures that the representation is closed. Our quotient assumes that the designated variable \(V\) is globally fresh.

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

We represent variables meta-circularly, that is, by other variables. In particular, a variable of type \(T\) is represented by a variable of the same name with type \(\text{PExp} V T\). The cases for quoting \(\lambda\)-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 quotient generates IsAll proofs itself, since they depend on the kind of the type (which cannot be passed as an argument to the constructors in \(\text{F}^\mu\)). The quotient also generates utility functions \(\text{stripAll}\), \(\text{underAll}\), and \(\text{insts}_{k,T,S}\) that are useful for 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 \(\text{F}^\mu\) in previous work [8]. The purpose of the extensional representation is to represent polymorphic terms in languages like \(\text{F}^\mu\) and \(\text{F}^\mu\) that do not include kind polymorphism.

The function \(\text{stripAll}\) has the type \(\forall X : K. \forall T. \text{stripAll} \ (\forall Y : K. T)\). As long as \(\forall Y : K. T\) is well-typed. For any type \(A\) in which \(X\) does not occur free, \(\text{stripAll}\) can map \(\text{All Id} A\) to \(\text{All}\) \((A : K. \forall T. T)\) \equiv (\forall Y : K. T) \equiv (\forall X : K. T)\). 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 \(\text{F}^\mu\) are inhabited to define \(\text{stripAll}\). It uses the kind inhabitant \(\text{T}_K\) for the instantiation. For each kind \(K\), \(\text{T}_K\) is a closed type of kind \(K\).

The function \(\text{underAll}\) has the type \(\forall X : K. T. \text{underAll} \ (\forall Y : K. T)\). It can apply a function under the quantifier of a type \(\text{All Id} F T\) to produce a result of type \(\text{All Id} F 2\) \(\equiv (\forall X : K. T)\). In particular, our evaluators use \(\text{underAll}\) to make recursive calls on the body of a type abstraction. The representation of a type abstraction \((\forall X : K. e)\) of type \((\forall X : K. T)\) contains the term \((\forall X : K. q)\) here, \(q\) is the representation of \(e\), in which the type variable \(X\) can occur free. The type of \((\forall X : K. q)\) is \(\text{All Id} [\text{PExp} V T] \equiv (\forall X : K. T)\).

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

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

The combination of IsAll proofs and the utility functions \(\text{stripAll}\), \(\text{underAll}\), and \(\text{insts}_{k,T,S}\) allows us to represent higher-kinded polymorphism without kind polymorphism. Notice that in the types oftabs and \(\text{tapp}\) (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 \(t\) \(\vdash e : T\), then \(\langle t \rangle \vdash q : \text{Exp} T\).

The proof is by induction on the derivation of the typing judgment \(\langle t \rangle \vdash e : T\). 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 \(t\) \(\vdash e : T\), then \(\pi\) 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 \(\beta\)-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 \(\text{F}^\mu\) are \(\lambda\) and \(\Lambda\) abstractions, and fold expressions. There is no evaluation under abstractions or fold expressions, and function arguments are not evaluated before \(\beta\)-reduction.

The function eval evaluates closed representations, which have \text{Exp} types. The main evaluator is evalV, which operates on \text{PExp} types. If the input is a variable, a \(\lambda\) or \(\Lambda\) 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 \(\text{matchAbs}\). This is a specialized version of \text{matchExp} that takes a only one case function, for \(\lambda\)-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 \(e_1\) has type \text{PExp} \(V B\). If evalV evaluates to a type abstraction, we


\[
T_* = (\forall X: \ast). X \\
T_{K1 \to K2} = \lambda X:K1.T_{K2}
\]

Kind Inhabitants

\[
decl \: \text{Id} : \ast \to \ast = \lambda A: \ast. A;
\]
\[
decl \: \text{UnderAll} : \ast \to \ast = \\
\lambda T: \ast. \forall A: \ast. T A \to A;
\]
\[
decl \: \text{StripAll} : \ast \to \ast = \\
\lambda T: \ast. \forall A: \ast. \forall Id (\lambda A: \ast. A) T \to A;
\]
\[
decl \: \text{Inst} : \ast \to \ast \to \ast = \\
\lambda A: \ast. \forall B: \ast. PExp V (B \to A) \to PExp V A;
\]

underAll_{x, k, T} =

\[\lambda F1, \lambda F2, \lambda f : (\forall A: \ast, F1 A \to F2 A), \lambda e : (\forall X:K. F1 T). \Delta X:K. f T (e X)\]

\[
\text{stripAll}_{x, k, T} = \lambda A. \lambda e : (\forall X:K.A). e T_x
\]

\[
\text{inst}_{x, k, T} = \lambda F. Af : (\forall X:K.F T). f S
\]

Operators on quantified types.

\[
decl \: \text{PExpF : } (\ast \to \ast) \to (\ast \to \ast) \to \ast \to \ast = \\
\lambda V: \ast. \ast : \ast. PExpV: \ast \to \ast. \lambda V: \ast. \forall R: \ast. V
\]

\[
(V A \to R) \to \\
(\forall T. \exists (S \to T) A \to (PExp V S \to PExp V T) \to R) \to \\
(\forall B. PExp V (B \to A) \to PExp V B \to R) \to \\
(IsAll A \to StripAll A \to UnderAll A \to \\
All Id (PExp V A) \to R) \to \\
(\forall B. IsAll B \to Inst B A \to PExp V B \to R) \to \\
(\forall F B. Eq (F (\mu F) B) A \to PExp V (F (\mu F) B) \to R) \to \\
(\forall F B. Eq (F (\mu F) B) A \to PExp V (\mu F B) \to R) \to \\
R
\]

\[
decl \: \text{PExp : } (\ast \to \ast) \to (\ast \to \ast) \to \ast \to \ast = \lambda V: \ast. \ast : \ast. PExpV: \ast \to \ast. \mu (PExpF V)
\]

\[
decl \: \text{Exp : } \ast \to \ast = \lambda A: \ast. \forall V: \ast. PExp V A;
\]

Definitions of PExp and Exp

\[
(x : T) \in \Gamma \\
\Gamma \vdash x : T \to x
\]

\[
\Gamma \vdash T_1 : \ast \\
\Gamma, (x : T_1) \vdash e : T_2 \to q
\]

\[
\Gamma \vdash (\lambda x: T_1.e) : T_1 \to T_2 \to \text{abs} V T_1 T_2 (\lambda x: \text{PExp F V T}_1). q
\]

\[
\Gamma \vdash e_1 : T_2 \to T \to q_1 \\
\Gamma \vdash e_2 : T_2 \to q_2
\]

\[
\Gamma \vdash e_1 e_2 : T \to \text{app} V T_1 T_2 q_1 q_2
\]

\[
\text{inst}_{x, k, T} = p \\
\text{stripAll}_{x, k} = s
\]

\[
\Gamma \vdash (\lambda x:K.e) : (\forall X:K.T) \to \text{tabs} V (\forall X:K.T) p s u (\lambda x:K.q)
\]

Quotation and pre-quotation

\[
\text{Figure 12: Self-representation of } F^\mu_\ast.
\]
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 reduct, 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, \((∀V:∗→. ∥A:∗. PExp V A → PExp V A) = ∆V:∗ →. ∥A:∗. λε:PExp V A.\)

matchExp V A e (PExp V A)
(constVar V A (PExp V A) e)
(constAbs V A (PExp V A) e)
(\(\DeltaA:∗. λf: PExp V (B → A). λx : PExp V B.\)
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 A (B → A) (PExp V A) f1 def
(\(\DeltaA:∗. λA:∗. λeq : Eq (B1 → A1) (B → A).\)
let eqL : Eq B1 B = sym B1 B (arrl B1 A B eq) in
let eqR : Eq A1 A = arrR B1 A1 B A eq in
let f' : PExp V B → PExp V A =
λeqR (PExp V) o f o eqL (PExp V)
in evalV V A (f’ x));
(matchTabs V A (PExp V A) e)
(\(\DeltaB : *. λξ : IsAll B. λι : Inst B A. λe1 : PExp V B.\)
let e2 : PExp V B = evalV V B e1 in
let def : PExp V A = tapp V B A p e2 in
matchTabs V B (PExp V A) e2 def
(\(\DeltaP : IsAll B. λi : StripAll B. λu : UnderAll B.\)
λe3 : All Id (PExp V B) B. evalV V A (p (PExp V e3)));
(matchFold V A (PExp V A) e)
(\(\DeltaF : (∗ → *) → → * → *. \ΔB : *.\)
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) (unfold V F B e2) in
matchFold V (μ F B) (PExp V A) e2 def
(\(\DeltaF1 : (∗ → *) → → *. \ΔB1 : *.\)
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 (eq unfold V F1 B1 F eql) eq
in evalV V A (eq2 (PExp V e3)));

decl eval : (∀A:∗. Exp A → Exp A) =
λA:∗. λε:Exp A. (∆V:∗→. evalV V A (e V));
foldExp : 
VR : * → *
(VA:+, VB:+, (R A → R B) → R (A → B)) →
(VA:+, VB:+, R (A → B) → R A → R B) →
(VA:+, IsAll A → StripAll A → UnderAll A →
All Id R A → R A) →
(VA:+, VB:+, IsAll A → Inst A B → R A → R B) →
(VF:(+→→)→→→.VA:+. R (F (μ F) A) → R (μ F A)) →
(VF:(+→→)→→→.VA:+. R (μ F A) → R (F (μ F A)) →
VA:+. 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 NB Ε 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_0\mu. We type check and test our evaluators and benchmark meta-programs using an implementation of F_0\mu 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_0, including a self-interpreter and a continuation-passing-style transformation. We reimplemented all of their meta-programs for F_0\mu. 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 VA:+. 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 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 ∀aAll A component of an IsAll A proof to convert All 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_0\mu is shown in Figure 16. T1 \cong 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) \cong Ct (CPS A → CPS B)
CPS (∀X.K.T) \cong Ct (∀X.K.K. CPS T)
CPS (μ F A) \cong 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 TeAll A component of the IsAll A proof is useful. Since we know A is a quantified type, the Typecase cases for arrow types and recursive types are irrelevant. The function eqCPSAll uses TeAll A to prove CPS1F CPS1 A and All Id CPS A are equal.

dcl eqCPSAll : (VA:+, IsAll A →
Eq (CPS1F CPS1 A) (All Id CPS A))
= (μA:+. λp : IsAll A.
fst p (μA1:+.μA2:+. CPS A1 → CPS A2)
Id CPS
(λF:(→→→→→→). μA:+. 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:

λu:UnderAll A. λf:All Id (λT:+.Nat) A.
suc (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 \cong (∀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 is size instrumental in programming size on representations of polymorphic terms.

Implementation. We have implemented System F_0\mu 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 NB \Ε similar to Figure 14, except that it operates on untyped first-order abstract syntax based on DeBrujin 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.
8. Related Work

Typed self-representation. Pfennig 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 \( F_\omega \), and of \( F_\omega \) in \( \mu F \), which extends \( F_\omega \) 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 \( \lambda \)-calculus \( F_\omega \) that has undecidable type checking. They implemented a self-recognizer, but not a self-evaluator. Jay and Palsberg [22] presented a typed self-representation for a combinator calculus that also has undecidable type checking. Their representation supports a self-recognizer and a self-evaluator, but not with the types described in Section 1. In their representation scheme, terms have the same type as their representations, and both their interpreters have the type \( \forall T . T \rightarrow T \). In previous work we presented self-representations for System U [7], the first for a language with decidable type checking, and for \( F_\omega \) [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 \( F_\omega \), 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-erase 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 \( \lambda^Q \) [37], which supports fully-reflexive ITA – analysis of all types of kind \( * \), including quantified and recursive types.

The most notable difference between \( F_\omega \) and previous languages with ITA is that \( F_\omega \) 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 \( \lambda^Q \). Unlike our Typecase operator, the type-level ITA operator in \( \lambda^Q \) 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 \( F_\omega \). 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,
Type equality. Type equality has been used to encode GADTs [11, 23, 36, 46], for generic programming [10, 47], 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 $\mathbb{F}_\omega^\omega$, a typed \(\lambda\)-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 \(\beta\)-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 Benders, Iris Cong, Christian Kahlauge, Oleg Kiselevyov, 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 $\mathbb{F}_\omega^\omega$ and our operations.


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 → (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 typings). Suppose Γ ⊢ T : K. Then:

1. If T=K, then (X:K) ∈ Γ.
2. If T=μT1 T2, then K=μ and Γ ⊢ T1 : * and Γ ⊢ T2 : *.
3. If T=(∀X:K1.T), then K=∀ and Γ1(X:K1) ⊢ T : *.
4. If T=(λX:K1.T′), then K=λ and Γ1(X:K1) ⊢ T′ : K2.
5. If T=(μ T1 T2), then Γ ⊢ T1 : K2 → K and Γ ⊢ T2 : K2.
6. If T=μ T1 T2, then K=μ and Γ ⊢ T1 : * → * → * and Γ ⊢ T2 : *.
7. If T=(Typecase T1 T2 T3 T4 T5), then K=μ and Γ ⊢ T1 : * → * → * and Γ ⊢ T2 : * → * and Γ ⊢ T3 : * → * and Γ ⊢ T4 : (* → *) → * → * and Γ ⊢ T5 : *.

Proof. Straightforward.

Lemma B.2 (Inversion of equalities). Suppose Γ ⊢ e : T. Then Γ ⊢ T : * and:

1. If e = x, then (x:T′) ∈ Γ and T′ ⊢ T.
2. If e = λx:T1.e1, then T ⊢ T1 → T2 and Γ ⊢ e1 : T2.
3. If e = (e1 e2), then Γ ⊢ e1 : T2 → T and Γ ⊢ e2 : T2.
4. If e = (λx:K1.e1), then T ⊢ (∀x:K.T′) and Γ ⊢ e1 : T′.
5. If e = e1 T2, then T ⊢ (μ T1 T2) and Γ ⊢ e1 : (μ T1 T2).
6. If e = fold T1 T2 e′, then T ⊢ (μ T1 T2) and Γ ⊢ e′ : T1 (μ T1 T2).
7. If e = unfold T1 T2 e′, then T ⊢ (μ T1 T2) and Γ ⊢ e′ : μ T1 T2.

Proof. By induction on the derivation Γ ⊢ e : T.

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

\[ \Gamma \vdash e : T_1 \quad T_1 \equiv T_2 \quad \Gamma \vdash e' : T_2 \]

By induction, the result holds for Γ ⊢ e : T1. Since T1 ⊢ T2, the result holds for T also.

Lemma B.3 (Canonical β-normal forms). Suppose Γ ⊢ v : T and v is a β-normal form. Then either v is neutral, or:

1. T ⊢ T1 → T2 and v = (λx:T1.e).
2. T ⊢ (∀x:K.T) and v = (λx:K.e).
3. T ⊢ μ T1 T2 and v = fold T1 T2 e.

Proof. By induction on the derivation Γ ⊢ e : T.

If T = χ, then v is neutral.

If v = (λx:T1.e), then by Lemma B.2, T ⊢ T1 → T2, so T ⊢ T1 → T2 as required.

If v = e1 e2, then v must be neutral as required.

If v = (λx:K.e), then by Lemma B.2, T = (∀x:K.T), so T ⊢ (∀x:K.T) as required.

If v = fold T1 T2 e, then by Lemma B.2, T = (μ T1 T2), so T ⊢ (μ T1 T2) as required.

If v = unfold T1 T2 e, then v must be neutral as required.

Lemma B.4 (Progress). If Γ ⊢ e : T, then either e is a β-normal or there exists a term e′ such that e → e′.

Proof. By induction on the typing derivation Γ ⊢ e : 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 e = (λx:T1.e1). By Lemma B.2, we have that T ⊢ T1 → T2 and Γ ⊢ T1 e1 : T2. By induction, either e1 is a β-normal form, or there exists an e1′ such that e1 → e1′. In the former case, (λx:T1.e1) is also a β-normal form as required. In the latter case, (λx:T1.e1 → (λx:T1.e1′)) 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 e = (e1 e2). By Lemma B.2, we have that T ⊢ e1 : T2 → T and T ⊢ e2 : T2. By induction, e1 is either a β-normal form or there exists an e1′ such that e1 → e1′. Similarly, e2 is either a β-normal form or there exists an e2′ such that e2 → e2′. If e1 → e1′, then e1 e2 → e1′ e2. If e1 → e1′, then e1 e2 → e1′ e2. The remaining case is when e1 and e2 are both β-normal. Since T ⊢ e1 : T2 → T, by Lemma B.3 either e1 is neutral or e1 = (λx:T2.e1′). If e1 is neutral then e1 e2 is β-normal as required. If e1 = (λx:T2.e1′), then e = (e1 e2) = (λx:T2.e1′) → e1 [x := 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 eighth rule (type conversion) is by straightforward induction.
Lemma B.5 (Weakening of kindings).

1. If \( \Gamma \vdash T_1 : K \) and \( x \notin \text{dom}(\Gamma) \) and \( \Gamma \vdash T_2 : \star \), then \( \Gamma, (x : T_2) \vdash T_1 : K \).
2. If \( \Gamma \vdash T : K \) and \( x \notin \text{dom}(\Gamma) \), then \( \Gamma, (x : T_1) \vdash T : K \).

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

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

\[
\begin{align*}
\Gamma & \approx \Gamma_1 \\
\Gamma_1 & \approx \Gamma_2 \\
\Gamma_2 & \approx \Gamma_3 \\
\Gamma_1 & \approx \Gamma_2 \\
\Gamma_2 & \approx \Gamma_3 \\
\end{align*}
\]

Lemma B.6 (Preservation of kinds under permutation of context). If \( \Gamma_1 \vdash T : K \) and \( \Gamma_1 \approx \Gamma_2 \), then \( \Gamma_2 \vdash T : K \).

Proof. By straightforward induction on the kinding derivation.

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

Proof. By straightforward induction on the typing derivation.

Lemma B.8 (Preservation of kinds under type substitution). If \( \Gamma, x : K_1 \vdash T_1 : K_2 \) and \( \Gamma \vdash T_2 : K_2 \), then \( \Gamma \vdash T_1 [x : T_2] : K_1 \).

Proof. By straightforward induction on kinding derivations.

Lemma B.9 (Substitution of types on typings). If \( \Gamma, x : K \vdash e : T_1 \) and \( \Gamma \vdash T_2 : K \), then \( \Gamma \vdash e[x : T_2] : T_1 \).

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

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

Proof. By straightforward induction on typing derivations.

Lemma B.11 (Preservation of type equivalence under type substitution). If \( \Gamma \equiv T_1, T_2 \), then \( \Gamma \equiv T_1 [x : T] \equiv T_2 [x : T] \).

Proof. By straightforward induction on equivalence derivations.

Lemma B.12 (Type Preservation). If \( \Gamma \vdash e : T \) and \( e \rightarrow e' \), then \( \Gamma \vdash e' : T \).

Proof. By induction on the derivation of \( \Gamma \vdash e : T \).

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

Suppose \( e \rightarrow e' \) is by the second rule. Then \( e = \lambda x : K \, e_1 \, T_2 \) and \( e' = e_1[x := e_2] \). By Lemma B.2, we have that \( \Gamma \vdash \langle \lambda x : K, e_1 \rangle \vdash (\forall x : K, T_1) \) and \( \Gamma \equiv T_1 [x := T_1] \). By Lemma B.2, we again have that \( \Gamma \equiv (T_1, T_2) \vdash e_1 : T_1 \). By Lemma B.9, we have that \( \Gamma \vdash e_1[x := T_2] : T_1 [x := T_2] \) as required.

The remaining cases are by straightforward induction.

Theorem 3.1. [Type Safety]

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


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 \( \alpha \)-conversion.

\[
\begin{array}{c|c|c}
T_1 \rightarrow T_1' & (T_1 \rightarrow T_2) & (T_2 \rightarrow T_2') \\
(T_1' \rightarrow T_2') & (T_1 \rightarrow T_2') & (T_1 \rightarrow T_2') \\
(\forall x : K, T) \rightarrow (\forall x : K, T') & (\forall x : K, T) \rightarrow (\forall x : K, T') & (\forall x : K, T) \rightarrow (\forall x : K, T') \\
(T_1 \rightarrow T_2) & (T_1 \rightarrow T_2) & (T_1 \rightarrow T_2) \\
(\lambda x : K, T_1) \rightarrow (\lambda x : K, T_2) & (\lambda x : K, T_1) \rightarrow (\lambda x : K, T_2) & (\lambda x : K, T_1) \rightarrow (\lambda x : K, T_2) \\
\end{array}
\]

We use \( \rightarrow^* \) to denote the reflexive transitive closure of \( \rightarrow \). Also, we use \( T_1 \rightarrow T_2 \) to denote that \( T_1 \) and \( T_2 \) are syntactically equal types, up to renaming (i.e. they are \( \alpha \)-equivalent).

Lemma B.13 (Preservation of kinds under type reduction). If \( \Gamma \vdash T_1 \rightarrow T_2 \), then \( \Gamma \vdash T : K \).

Proof. By induction on the derivation of \( T_1 \rightarrow T_2 \).

We consider only the cases for \( \beta \)-reduction and the three Typecase eliminations. The others are straightforward by induction.

Suppose the equivalence is by \( \beta \)-reduction. Then \( T_1 \rightarrow (\lambda x : K_1, A) \) and \( T_2 \rightarrow A[x := B] \). By Lemma B.1, we have that \( \Gamma \vdash (x : K_1) \vdash A : K \) and \( \Gamma \vdash B : K_1 \). By Lemma B.8, we have that \( \Gamma \vdash A[x := B] : K \) as required.

Suppose the equivalence is by the first Typecase reduction rule. Then \( K \rightarrow A \) and \( T_1 \rightarrow \text{Typecase} F_1 \). By Lemma B.1, we have that \( \Gamma \vdash F_1 : * \rightarrow * \rightarrow \ast \) and \( \Gamma \vdash A : * \), and \( \Gamma \vdash B : * \). Therefore, \( \Gamma \vdash F_1 \rightarrow A \rightarrow B \) as required.

The remaining 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 $F^\mu$ consist of the simply-typed $\lambda$-calculus, extended with constructors for arrow, quantified, and recursive types, and Typecase.

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

Proof. Straightforward.

Definition B.3 (Reducibility).
1. If $\Gamma \vdash T : S$ and $T$ is SN, then $T \in RED_\alpha$.
2. If $\Gamma \vdash T : K_1 \rightarrow K_2$ and for all $T_1 \in RED_{\alpha_1}$, $T_2 \in RED_{\alpha_2}$, then $T \in RED_{K_1 \rightarrow K_2}$.
3. (a) If $(T_1 \rightarrow T_2) \in RED_\alpha$, then $T_1 \in RED_\alpha$ and $T_2 \in RED_\alpha$.
   (b) If $(\forall x : K.T) \in RED_\alpha$, then $T \in RED_\alpha$.
   (c) If $(\mu F) \in RED_{\rightarrow \alpha}$, then $F \in RED_{(\rightarrow \alpha)\rightarrow \alpha}$.
   (d) If $(\mu F.A) \in RED_\alpha$, then $F \in RED_{\rightarrow \alpha}$ and $A \in RED_\alpha$.

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:
1. (CR1) If $T \in RED_\alpha$, then $T$ is SN.
2. (CR2) If $T \in RED_\alpha$ and $T \rightarrow^* T'$, then $T' \in RED_\alpha$.
3. (CR3) If $T$ is neutral and for all $T'$, $T \rightarrow^* T'$ implies $T' \in RED_\alpha$, then $T \in RED_\alpha$.

A consequence of (CR3) will be that if $\Gamma \vdash T : K$ and $T$ neutral and normal, then $T \in RED_\alpha$. This is called (CR 4) by GTL [19].

Lemma B.15. For all kinds $K$, the conditions of reducibility hold for $RED_\alpha$.

Proof. By induction on $K$, following the same argument as GTL [19].

Lemma B.16 (Reducibility of $\rightarrow$). If $T_1, T_2 \in RED_\alpha$, then $(T_1 \rightarrow T_2) \in RED_\alpha$.

Proof. It is clear that if $T_1$ and $T_2$ are SN, then so is $(T_1 \rightarrow T_2)$. That satisfies requirement 1 of reducibility. Requirement 3a is by assumption.

Lemma B.17 (Reducibility of $\forall$). If $T \in RED_\alpha$, then $(\forall x : K.T) \in RED_\alpha$.

Proof. Similar to Lemma B.16.

Lemma B.18 (Reducibility of $\mu$). $\mu F.A \in RED_{((\rightarrow \alpha)\rightarrow \alpha)\rightarrow \alpha}$. $\nu(F) + \nu(A) = 0$.

Proof. It suffices to show that for all $F$ and $A$, if $F \in RED_{((\rightarrow \alpha)\rightarrow \alpha)\rightarrow \alpha}$, then $F \in RED_\alpha$.

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

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

If $\nu(A) = 0$, then $F$ and $A$ are normal, and $\mu F.A$ is neutral and normal. Therefore, $\mu F.A \in RED_\alpha$.

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

Suppose $\mu F.A \rightarrow \mu F'.A$ and $F \rightarrow^* F'$. Then by (CR 2) $F'$ is reducible, and $\nu(F') < \nu(F)$. By induction, $\mu F'.A$ is reducible.

Suppose $\mu F.A \rightarrow \mu F'.A'$ and $A \rightarrow^* A'$. Then $A'$ is reducible and $\nu(F') < \nu(F)$. By induction, $\mu F'.A$ is reducible.

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

Lemma B.19 (Reducibility of Typecase). Typecase $C \in RED_{((\rightarrow \alpha)\rightarrow \alpha)\rightarrow \alpha}$.

Proof. It suffices to show that for all $F_1 \in RED_{\rightarrow \alpha}$, $F_2 \in RED_{\rightarrow \alpha}$, $F_3 \in RED_{\rightarrow \alpha}$, $F_4 \in RED_{((\rightarrow \alpha)\rightarrow \alpha)\rightarrow \alpha}$, and $S \in RED_\alpha$, it is true that Typecase $F_1 F_2 F_3 F_4 S \in RED_\alpha$.

We proceed by induction on $\nu(F_1) + \nu(F_2) + \nu(F_3) + \nu(F_4) + \nu(S)$.

1. Suppose $F_1 \in RED_{\rightarrow \alpha}$, $F_2 \in RED_{\rightarrow \alpha}$, $F_3 \in RED_{\rightarrow \alpha}$, $F_4 \in RED_{((\rightarrow \alpha)\rightarrow \alpha)\rightarrow \alpha}$, and $S \in RED_\alpha$.

Let $T = $ Typecase $F_1 F_2 F_3 F_4 S$. Consider the possible reduction steps from $T$:

$T \rightarrow$ Typecase $F_1' F_2 F_3 F_4 S$ and $F_1 \rightarrow F_1'$. By (CR 2) $F_1'$ is reducible and $\nu(F_1') < \nu(F_1)$, so by induction Typecase $F_1' F_2 F_3 F_4 S \in RED_\alpha$. The cases for steps in one of $F_2, F_3, F_4, S$ are similar.

2. $T \rightarrow F_1 S_1 S_2$ and $S_1 \rightarrow S_2$. By definition 3(a) of reducibility, $S_1, S_2 \in RED_\alpha$, so $F_1 S_1 S_2 \in RED_\alpha$.

3. $T \rightarrow F_2 (\forall x : K.F_3 S_1)$ and $S = (\forall x : K.S_1)$. By definition 3(b) of reducibility, $S_1 \in RED_\alpha$, so $F_3 S_1$ is reducible, so by Lemma B.17 (\forall x : K.F_3 S_1) \in RED_\alpha, so $F_2 (\forall x : K.F_3 S_1) \in RED_\alpha$.

4. $T \rightarrow F_4 S_1 S_2$ and $S = \mu S_1 S_2$. By definition 3(d) of reducibility, $S_1 \in RED_{((\rightarrow \alpha)\rightarrow \alpha)\rightarrow \alpha}$, and $S_2 \in RED_\alpha$, so $F_4 S_1 S_2 \in RED_\alpha$.

In all cases, the result of stepping $T$ is contained in $RED_\alpha$, so by (CR 3) we have that $T \in RED_\alpha$.

Lemma B.20. If $\Gamma \vdash T : K$, and $\sigma$ is a reducible substitution for $\Gamma$, then $T \sigma \in RED_\alpha$.

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

Suppose $\Gamma \vdash T : K$ is by the rule for variables. Since $\sigma$ is a reducible substitution, $T \sigma$ is equal to some type in $RED_\alpha$.

Suppose $\Gamma \vdash T : K$ is by the rule for arrow types. Then $T = (T_1 \rightarrow T_2)$, and $K = \ast$ and $\Gamma : T_1 \vdash \ast$ and $\Gamma : T_2 \vdash \ast$. By induction, $T_1 \sigma, T_2 \sigma \in RED_\alpha$, and so are $T_1$ and $T_2$. Therefore $T \sigma$ is SN, and the requirements 1 and 3(a) of reducibility are satisfied, so $T \sigma \in RED_\alpha$.

Suppose $\Gamma \vdash T : K$ is by the rule for quantified types. Then $T = (\forall x : K.T')$ and $K = \ast$ and $\Gamma, x : K \vdash \ast$. Without loss of generality (by renaming), assume that $X$ does not occur in $T$. Since the type variable $X$ is neutral and normal, $X \in RED_\alpha$ and so $\sigma[X := X_0]$ is a reducible substitution for $\Gamma, X : K$. By induction, $T' \sigma[X := X_0] = T \sigma \in RED_\alpha$, so by Lemma B.17 (\forall x : K. T' \sigma) = $T \sigma \in RED_\alpha$.

Suppose $\Gamma \vdash T : K$ is by the rule for $\lambda$-abstraction. Then $T = (\lambda x : K_1.T')$ and $K_2 = K_1 \rightarrow K_2$, and $K_1, x : K_2 \vdash T'$. Without loss of generality (by renaming), assume that $X$ does not occur in $T$. Since the type variable $X$ is neutral and normal, $X \in RED_\alpha$ and so $\sigma[X := X_0]$ is a reducible substitution for $\Gamma, X : K$. By induction, $T \sigma[X := X_0] = T \sigma \in RED_\alpha$. Therefore, $(\lambda x : K_1.T') \sigma \in RED_{\rightarrow \alpha}$ as required.

The case for type application is by straightforward induction. The case for $\mu$ is by Lemma B.18.

The case for Typecase is by Lemma B.19.

Lemma B.21 (SN of type reduction). If $\Gamma \vdash T : K$, then $T$ 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 $T \rightarrow T_1$ and $T \rightarrow T_2$, then there exists a type $T'$ such that $T_1 \rightarrow^* T'$ and $T_2 \rightarrow^* T'$.

Proof. Standard.

Lemma B.23. If $T_1 \equiv T_2$ then there exists a type $T$ such that $T_1 \rightarrow^+ T$ and $T_2 \rightarrow^+ T$.

Proof. By induction on the derivation of $T_1 \equiv T_2$.

If $T_1 \equiv T_2$ is by reflexivity. Then $T_1 = T_2$. The result follows by Lemma B.21.

If $T_1 \equiv T_2$ is by symmetry, then the result follows from the induction hypothesis.

If $T_1 \equiv T_2$ is by transitivity, then there exists a type $T_3$ such that $T_1 \equiv T_3$ and $T_3 \equiv T_2$. By two uses of induction, there exist types $T$ and $T'$ such that $T_1 \rightarrow^* T$, $T \rightarrow^* T'$, $T_3 \rightarrow^* T'$, and $T_2 \rightarrow^* T'$. By Lemma B.22, $T \rightarrow^* T$ and $T_3 \rightarrow^* T'$ imply that there exists a $T''$ such that $T \rightarrow^+ T''$ and $T' \rightarrow^+ T''$. Therefore $T_1 \rightarrow^+ T''$ and $T_2 \rightarrow^+ T''$ as required.

If $T_1 \equiv T_2$ is by the congruence rule for arrows, we have that $T_1 = (A_1 \rightarrow B_1), T_2 = (A_2 \rightarrow B_2)$, and $A_1 \equiv A_2$ and $B_1 \equiv B_2$. By induction, there exist normal form types $A$ and $B$ such that $A_1 \rightarrow^* A$, $A_2 \rightarrow^* A$, $B_1 \rightarrow^* B$, and $B_2 \rightarrow^* B$. Let $T = (A \rightarrow B)$. Construct $T_1 \rightarrow^* T$ and $T_2 \rightarrow^* T$ by first reducing $A_1$ and $A_2$ to $A$ and then reducing $B_1$ and $B_2$ to $B$.

The congruence rules for $\mu$ 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 $T \rightarrow^+ N_1$ and $T \rightarrow^+ N_2$, where $N_1, N_2$ are normal forms, then $N_1 = N_2$.

Proof. By Lemma B.22, there exists a type $T'$ such that $N_1 \rightarrow^* T'$ and $N_2 \rightarrow^* T'$. But $N_1$ and $N_2$ are normal forms, so it must be the case that $N_1 = N_2$ and $N_2 = T'$. So $N_1 = N_2$ as required.

We say “$T$ has a normal form” to mean that there exists a normal form $N$ such that $T \rightarrow^* N$.

Lemma B.25 (Types of 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 $N_1$ and $N_2$. Then by Lemma B.24, $N_1 = N_2$.

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 $T \rightarrow^+ nf(T)$.

B.5 Type equivalence and type checking are decidable

Lemma B.26. If $T \rightarrow^* T'$, then $T \equiv T'$.

Proof. Straightforward.

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

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

Lemma B.28. If $\Gamma \vdash N_1 : K$ and $\Gamma \vdash N_2 : K$ and $N_1$ and $N_2$ are normal forms, then $N_1 \equiv N_2$ if and only if $N_1 = N_2$.

Proof. ($\Leftarrow$) By Lemma B.23, there exists a type $T$ such that $N_1 \rightarrow^* T$ and $N_2 \rightarrow^* T$. But since $N_1$ and $N_2$ are normal forms, $N_1 = T$ and $N_2 = T$, so $N_1 = N_2$ as required.

Lemma B.29. If $\Gamma \vdash T : K$ and $\Gamma \vdash T_1 : K$ and $T_1 \equiv T_2$ is decidable.

Proof. By Lemma B.27, $T_1 \equiv nf(T_1)$ and $T_2 \equiv nf(T_2)$. Therefore, $T_1 \equiv T_2$ if and only if $nf(T_1) \equiv nf(T_2)$. By Lemma B.28, $nf(T_1) \equiv nf(T_2)$ if and only if $nf(T_1) = nf(T_2)$.

Therefore, we can decide $T_1 \equiv T_2$ 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).

\[
\begin{align*}
\Gamma, x : \mathcal{K} & \mapsto \Gamma, x : \mathcal{K} \\
\Gamma, x : \mathcal{K} & \mapsto \Gamma, x : \mathcal{K} \\
\Gamma, x : \mathcal{K} & \mapsto \Gamma, x : \mathcal{K} \\
\end{align*}
\]

Lemma C.1. If $\Gamma \vdash T : K$, then $\Gamma \vdash T : K$.

Proof. Straightforward, since $\Gamma$ 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 $\Gamma \vdash e : T$.


Lemma C.3. If $\Gamma \vdash (\mathcal{K} : \mathcal{T}) : *$, then

1. $\Gamma \vdash tcAllK, T : tcAll (\mathcal{K} : \mathcal{T})$
2. $\Gamma \vdash unAllK, T : unAll (\mathcal{K} : \mathcal{T})$
3. $\Gamma \vdash isAllK, T : isAll (\mathcal{K} : \mathcal{T})$

Proof. Suppose $\Gamma \vdash (\mathcal{K} : \mathcal{T}) : *$. Then $\Gamma, x : \mathcal{K} \vdash \Gamma, T : *$.

1) $tcAllK, T = tcAllK, \Gamma, x : \mathcal{K} \vdash \Gamma, x : \mathcal{K} \rightarrow \Gamma, x : \mathcal{K}$
2) $unAllK, T = unAllK, \Gamma, x : \mathcal{K} \vdash \Gamma, x : \mathcal{K}$
3) $isAllK, T = isAllK, \Gamma, x : \mathcal{K} \vdash \Gamma, x : \mathcal{K}$

Lemma C.4. If $\Gamma \vdash (\mathcal{K} : \mathcal{T}) : *$ and $\Gamma \vdash S : \mathcal{K}$, then

1. $\Gamma \vdash underAllK, S : underAll (\mathcal{K} : \mathcal{T})$
2. $\Gamma \vdash stripAllK, S : stripAllK (\mathcal{K} : \mathcal{T})$
3. $\Gamma \vdash instK, T, S : instK, T (\mathcal{K} : \mathcal{T}) (\mathcal{K} : \mathcal{T})$

Proof. First, note that $underAllK, S, stripAllK$, and $instK, T, S$ contain no free term variables. Therefore, by Lemma C.2 it is sufficient to show

1. $\Gamma \vdash underAllK, S : underAll (\mathcal{K} : \mathcal{T})$
2. $\Gamma \vdash stripAllK, S : stripAllK (\mathcal{K} : \mathcal{T})$
3. $\Gamma \vdash instK, T, S : instK, T (\mathcal{K} : \mathcal{T}) (\mathcal{K} : \mathcal{T})$
For 1, it is easily checked that $\Gamma \vdash \text{underAll}_{x : K} : (\forall V. F x \rightarrow \ast, \forall V. F y \rightarrow \ast). (\forall V. A x \rightarrow F A) \rightarrow \forall (X : K, F \rightarrow F_2)$. The result follows from the type equivalences $\text{All Id F} \equiv (\forall (X : K, F \rightarrow F) \equiv (\forall (X : K, F \rightarrow F_2))$ and $\text{All Id F} \equiv (\forall (X : K, F \rightarrow F_2)$.

The cases for 2 and 3 are similar.

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

**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 the rule for variables. Then $e = x$ and $(x : T) \in \Gamma$ and $\Gamma \vdash x : T \triangleright x$. By the definition of $\vdash$, $(x : T) \triangleright x$ is encoded as $\lambda x : T. x$.

Suppose $\Gamma \vdash e : T$ is the rule for $\lambda$-abstractions. Then $e = (\lambda x : T. e_1)$ and $\Gamma, x : T \vdash e_1 : T_1$ and $T = T_1 \rightarrow T_2$. Also, $\Gamma, x : T \vdash e_1 : T_2 \triangleright q_1$ and $\Gamma \vdash (\lambda x : T. e_1) \triangleright \text{abs} V T_1 T_2 (\lambda x : T. e_1) q_1$. By induction, $q_1$ is unique and $\Gamma, x : T \vdash q_1 : \text{Exp} V T_2$. But $\Gamma, x : T \vdash (\lambda x : T. e_1) \triangleright \text{Exp} V T_1$, so $\Gamma \vdash (\lambda x : T. e_1) \triangleright \text{Exp} V T_2$. Therefore, $q = \text{abs} V T_1 T_2 (\lambda x : T. e_1) q_1$ is unique and by the type of abs, it is easily checked that $\Gamma \vdash \text{abs} V T_1 T_2 (\lambda x : T. e_1) q_1 : \text{Exp} V (T_1 \rightarrow T_2)$.

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

Suppose $\Gamma \vdash e : T$ is the rule for type abstractions. Then $e = (\Lambda X : K. e_1)$ and $\Gamma, X : K \vdash e_1 : T_1$ and $T = (\forall X. K. T_1)$. Also, $\Gamma, X : K \vdash e_1 : T_1 \triangleright q_1$ and $\Gamma \vdash (\Lambda X : K. e_1) \triangleright \text{tabs} V (\forall X. K. T_1)$ $\text{p} s u (\Lambda X. K. q_1)$, where $p = \text{isAll}_{x : K, T}, s = \text{stripAll}_{x : K} = s$, and $u = \text{underAll}_{x : K, T}$. By induction, $q_1$ is unique and $\Gamma, X : K \vdash q_1 : \text{Exp} V T_1$. Since $\Gamma, X : K \vdash (\Lambda X : K. q_1) : (\forall X. K. \text{Exp} V T_1)$, it follows from $\text{All Id} \vdash (\text{Exp} V (\forall X. K. T_1)) = (\forall X. K. \text{Exp} V T_1)$ that $\Gamma \vdash (\Lambda X : K. q_1) : (\text{All Id} \vdash (\text{Exp} V (\forall X. K. T_1)))$. By Theorem C.3, $\Gamma \vdash p : \text{IsAll} (\forall X. K.)$. But $p$ is not contained in any free term variables, so by Lemma C.2 we have that $\Gamma \vdash p : \text{IsAll} (\forall X. K.)$. By Theorem C.4, we have that $\Gamma \vdash s : \text{StripAll} (\forall X. K.)$ and $\Gamma \vdash u : \text{underAll} (\forall X. K.)$. Therefore, $q = \text{tabs} V (\forall X. K. T_1) p s u (\Lambda X. K. q_1)$ is unique and by the type of tabs, we have that $\Gamma \vdash \text{tabs} V (\forall X. K. T_1) p s u (\Lambda X. K. q_1) : \text{Exp} V (\forall X. K. T_1)$ as required.

The case for type applications is similar.

Suppose $\Gamma \vdash e : T$ is the rule for type conversion. The $\Gamma \vdash e : S$ and $S \equiv T$. By the induction hypothesis, there exists a unique $q$ such that $\Gamma \vdash e : S \triangleright q$ and $\Gamma \vdash q : \text{Exp} V S$. Since $\text{Exp} V S \equiv \text{Exp} V T$, we have that $\Gamma \vdash q : \text{Exp}_V T$ as required.

**Theorem 5.1.** If $\{\} \vdash e : T$, then $\{\} \vdash \pi : \text{Exp}_V T$.

**Proof.** Follows straightforwardly from Theorem C.5.

**Lemma C.6 (SN of refl1).** For any type $T$, refl $T$ is SN.

**Proof.** Straightforward.

**Lemma C.7 (SN of isAll_{x : K,T}).** If $\Gamma \vdash (\forall X. K. T) : \ast$, then $\text{isAll}_{x : K, T}$ is SN.

**Proof.** By Lemma C.6, tcAll_{x : K,T} and unAll_{x : K,T} are SN. The pair tcAll_{x : K,T} and unAll_{x : K,T} is encoded as $(\lambda R. \ast. \lambda f : \text{tcAll} (\forall X. K. T) \rightarrow \text{unAll} (\forall X. K. T) \rightarrow R. f \text{tcAll}_{x : K, T} \text{unAll}_{x : K, T})$, which is SN since tcAll_{x : K,T} and unAll_{x : K,T} are.

**Lemma C.8 (SN of inst_{x : K,T,5}).** If $\Gamma \vdash (\forall X. K. T) : \ast$ and $\Gamma \vdash S : K$, then inst_{x : K,T,5} is SN.

**Proof.** Immediate, the definition of inst_{x : K,T,5} is a normal form.

**Lemma C.9 (SN of stripAll_{x : K}).** For any kind $K$, stripAll_{x : K} is SN.

**Proof.** Immediate, the definition of stripAll_{x : K} is a normal form.

**Lemma C.10 (SN of underAll_{x : K,T}).** If $\Gamma \vdash (\forall X. K. T) : \ast$, then underAll_{x : K,T} is SN.

**Proof.** Immediate, the definition of underAll_{x : K,T} is a normal form.

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

1. For any $V, A, e$, if $e$ is SN and $\forall A e$ is well-typed, then $\forall V A e$ 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 $V, A, B, e_1, e_2$, if $e_1$ and $e_2$ are SN and app $V A B e_1 e_2$ is well-typed, then app $V A B e_1 e_2$ 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 $p, i, e$ 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 fold $V F A e$ is well-typed, then fold $V F A e$ is SN.
7. For any $V, F, A, e$, if $e$ is SN and unfold $V F A e$ is well-typed, then unfold $V F A e$ is SN.

**Proof.** By Lemma C.6, the type equality proofs created by abs, fold and unfold are SN.

The result holds since each case reduces in a few steps to a term of the form $f$old $\lambda x. \lambda v. A x. \lambda a. \lambda b. \lambda app. \lambda tabs. \lambda tapp. \lambda fold. \lambda unfold. e$, where $e$ is an application with a head position, and only SN term arguments.

**Lemma C.12.** If $\Gamma \vdash e : T \triangleright q$, then $q$ is strongly normalizing.

**Proof.** By straightforward induction on the derivation of $\Gamma \vdash e : T \triangleright q$, using Lemmas C.7, C.8, C.9, C.10, and C.11.

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

**Proof.** We have that $\pi = \lambda V. \ast \rightarrow \ast. q$ and $\{\} \vdash e : T \triangleright q$. By Lemma C.12, $q$ is SN. Therefore $\{\} e$ is also.
D. Code Listings

D.1 Prelude

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

```plaintext
decl Id : * → * = λA:*. A;

-- Pairs
decl Pair : * → * → * =
  λA:*. λB:*. λC:*. (A → B → C) → C;
decl pair : ∀A:*. ∀B:*. A → B → Pair A B =
  λA:*. λB:*. λa:A. λb:B. λC:*. (A → B → C) → 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 =
  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 =
\lambda n : Nat.
unfold NatF Bottom n Nat
-- n = 0
  zero
-- n > 0
  (\lambda m : Nat. m);

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

decl rec times : Nat → Nat → Nat =
\lambda m : Nat. \lambda n : Nat.
unfold NatF Bottom m Nat
  zero
  (\lambda pm : Nat. plus (times pm n) n);

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

decl rec fact : Nat → Nat =
\lambda n : 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 : (* → *) → (* → *) → * → *
  \lambda Out : * → *. \lambda In : * → *.
Typecase
  (\lambda A : *. \lambda B : *. Bottom)
  Out
  In
(λF:(* → *) → * → *. λA:* →. Bottom);

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

dcl ArrL : * → *
  Typecase
  (λa:* →. λb:* →. a)
  (λa:* →. Bottom)
  (λF:(* → *) → * → *. λa:* →. Bottom);

dcl ArrR : * → *
  Typecase
  (λa:* →. λb:* →. b)
  (λa:* →. Bottom)
  (λF:(* → *) → * → *. λa:* →. Bottom);

D.3 Type Equality

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

dcl refl : (∀A:* →. Eq A A) = λA:* →. λF:* → *. λx : F A. x;

dcl sym : (∀A:* →. ∀B:* →. Eq A B) =
  λA:* →. λB:* →. λeq : Eq A B.
  let p : Eq A A = refl A in
  Eq (λT:* →. Eq T A) p;

dcl 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);

dcl 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 B) = refl (F A) in
  Eq (λT:* →. Eq (F A) (F T)) p;

dcl coerce : (∀A:* →. ∀B:* →. Eq A B → A → B) =
  λA:* →. λB:* →. λeq : Eq A B. eq Id;

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

dcl UnAll : * → *
  λArr : * → * → *. ∀Out : * → *. Eq (All Out Id A) (Out A);

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

dcl IsAll : (∀A:* →. TcAll A → UnAll A → IsAll A) =
  λA:* →. pair (TcAll A) (UnAll A);

dcl tcAll : (∀A:* →. IsAll A → TcAll A) = λA:* →. fst (TcAll A) (UnAll A);

dcl unAll : (∀A:* →. IsAll A → UnAll A) = λA:* →. snd (TcAll A) (UnAll A);

  Eq (A1 → A2) (B1 → B2) →
  Eq A1 B1) =
\[\Lambda A1: \ast. \Lambda A2: \ast. \Lambda B1: \ast. \Lambda B2: \ast. \text{eqApp} (A1 \rightarrow A2) (B1 \rightarrow B2) \text{ArrL};\]

decl \text{arrR} : (\forall A1: \ast. \forall A2: \ast. \forall B1: \ast. \forall B2: \ast. \\
\text{Eq} (A1 \rightarrow A2) (B1 \rightarrow B2) \\
\text{Eq} A2 B2) = \Lambda A1: \ast. \Lambda A2: \ast. \Lambda B1: \ast. \Lambda B2: \ast. \text{eqApp} (A1 \rightarrow A2) (B1 \rightarrow B2) \text{ArrR};\]

decl \text{coerce} : (\forall A: \ast. \forall B: \ast. \text{Eq} A B \rightarrow A \rightarrow B) = \\
\Lambda A: \ast. \Lambda B: \ast. \lambda p: \text{Eq} A B. \text{eq Id};

decl \text{eqUnfold} : (\forall F: \ast \rightarrow \ast. \forall F1: \ast \rightarrow \ast. \forall F2: \ast \rightarrow \ast. \\
\forall p: \text{IsAll} (\ast \rightarrow \ast. \ast) \\
\text{Eq} (F1 (\mu F1) A1) (F2 (\mu F2) A2) = \\
\lambda F1 : (\ast \rightarrow \ast. \ast). \lambda F2 : (\ast \rightarrow \ast. \ast). \lambda A1: \ast. \\
\text{eqApp} (F1 (\mu F1) A1) (F2 (\mu F2) A2) \text{Unfold};

\text{-- Contradictions}

decl \text{eqArrMu} : (\forall A1: \ast. \forall A2: \ast. \forall F: \ast \rightarrow \ast. \\
\forall p: \text{IsAll} (\ast \rightarrow \ast. \ast) \\
\text{Eq} (A1 \rightarrow A2) (\mu F B) = \\
\lambda A: \ast. \lambda B: \ast. \lambda A1: \ast. \lambda A2: \ast. \\
\lambda F: (\ast \rightarrow \ast. \ast). \lambda p: \text{IsAll} (\ast \rightarrow \ast. \ast). \\
\text{let id} : (\forall A: \ast. A \rightarrow A) = \Delta A: \ast. \lambda x: A. x \text{ in} \\
\text{let id1} : \text{Typecase} (\lambda x: \ast. \lambda y: \ast. (\forall A: \ast. A \rightarrow A)) (\lambda x: \ast. \lambda y: \ast) (\lambda F: (\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast. \lambda B: \ast. \ast) (A1 \rightarrow A2) = \text{id in} \\
\text{let id2} : \text{Typecase} (\lambda x: \ast. \lambda y: \ast. (\forall A: \ast. A \rightarrow A)) (\lambda x: \ast. \lambda y: \ast) (\lambda F: (\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast. \lambda B: \ast. \ast) (\mu F B) = \\
\text{eq} (\lambda x: \ast. \lambda y: \ast. (\forall A: \ast. A \rightarrow A)) (\lambda x: \ast. \lambda y: \ast) (\lambda F: (\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast. \lambda B: \ast. \ast) T) \\
\text{id1} \\
\text{in} \text{id2};

\text{decl} \text{arrIsAll} : (\forall A: \ast. \forall B: \ast. \text{IsAll} (A \rightarrow B) \rightarrow \text{Bottom}) = \\
\lambda A: \ast. \lambda B: \ast. \lambda p: \text{IsAll} (A \rightarrow B). \\
\text{let id} : (\forall A: \ast. A \rightarrow A) = \Delta A: \ast. \lambda x: A. x \text{ in} \\
\text{let id1} : \text{Typecase} (\lambda x: \ast. \lambda y: \ast. (\forall A: \ast. A \rightarrow A)) \\
(\lambda x: \ast. \text{Bottom}) (\lambda x: \ast. \text{Bottom}) \\
(\lambda F: (\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast. \lambda B: \ast. \ast) (A \rightarrow B) = \text{id in} \\
\text{let eq} : \text{Eq} (\text{Typecase} (\lambda x: \ast. \lambda y: \ast. (\forall A: \ast. A \rightarrow A)) \\
(\lambda x: \ast. \text{Bottom}) (\lambda x: \ast. \text{Bottom}) \\
(\lambda F: (\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast. \lambda B: \ast. \ast) (A \rightarrow B)) \\
(\text{All} (\lambda x: \ast. \text{Bottom}) (\lambda x: \ast. \text{Bottom}) (A \rightarrow B)) = \\
\text{tcAll} (A \rightarrow B) p \\
(\lambda x: \ast. \lambda y: \ast. (\forall A: \ast. A \rightarrow A)) \\
(\lambda x: \ast. \text{Bottom}) (\lambda x: \ast. \text{Bottom}) \\
(\lambda F: (\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast. \lambda B: \ast. \ast) (A \rightarrow B) = \text{id in} \\
\text{let eq} : \text{Eq} (\text{Typecase} (\lambda x: \ast. \lambda y: \ast. \text{Bottom}) (\lambda x: \ast. \text{Bottom}) (\lambda x: \ast. \text{Bottom}) \\
(\lambda F: (\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast. \lambda B: \ast. (\forall A: \ast. A \rightarrow A)) \\
(\mu F A) = \text{id in} \\
\text{let eq} : \text{Eq} (\text{Typecase} (\lambda x: \ast. \lambda y: \ast. \text{Bottom}) (\lambda x: \ast. \text{Bottom}) (\lambda x: \ast. \text{Bottom}) \\
(\lambda F: (\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast. \lambda B: \ast. (\forall A: \ast. A \rightarrow A)) \\
(\mu F A) = \text{id in} \\
\text{let eq} : \text{Eq} (\text{Typecase} (\lambda x: \ast. \lambda y: \ast. \text{Bottom}) (\lambda x: \ast. \text{Bottom}) (\lambda x: \ast. \text{Bottom}) \\
(\lambda F: (\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast. \lambda B: \ast. (\forall A: \ast. A \rightarrow A)) \\
(\mu F A) = \text{id in}
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:*.  
-- 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) 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) B) → R;
decl UnfoldF : (* → *) → * → * → * = λV:* → *.  λA:*.  λR:*
∀F : (* → *) → * → * . ∀B : *.
\[
\text{Eq} \ (F \mu F) A \to \text{PExp} V (\mu F B) \to R;
\]

\textbf{decl var} : \ ((VV : * \to *). \forall A: *. V A \to \text{PExp} V A) =
\[\Lambda V:* \to *. \Lambda A: *. \lambda x: V A.\]

\text{fold} (\text{PExp} V) A (\n \lambda R:*.
\lambda \text{var}: \text{VarF} V A R.
\lambda \text{abs}: \text{AbsF} V A R.
\lambda \text{app}: \text{AppF} V A R.
\lambda \text{tabs}: \text{TAbsF} V A R.
\lambda \text{tapp}: \text{TAppF} V A R.
\lambda \text{fld}: \text{FoldF} V A R.
\lambda \text{unfld}: \text{UnfoldF} V A R.
\text{var} x);\]

\textbf{decl abs} : \ ((VV : * \to *). \forall A: *. \forall B: *).
\[\text{PExp} V (A \to B) \to \text{PExp} V (A \to B)) =\]
\[\lambda V:* \to *. \lambda A: *. \lambda B: *. \lambda f: \text{PExp} V A \to \text{PExp} V B.\]

\text{fold} (\text{PExp} V) (A \to B) (\n \lambda R:*.
\lambda \text{var}: \text{VarF} V (A \to B) R.
\lambda \text{abs}: \text{AbsF} V (A \to B) R.
\lambda \text{app}: \text{AppF} V (A \to B) R.
\lambda \text{tabs}: \text{TAbsF} V (A \to B) R.
\lambda \text{tapp}: \text{TAppF} V (A \to B) R.
\lambda \text{fld}: \text{FoldF} V (A \to B) R.
\lambda \text{unfld}: \text{UnfoldF} V (A \to B) R.
\text{abs} A B \ (\text{refl} (A \to B)) f);\]

\textbf{decl app} : \ ((VV : * \to *). \forall A: *. \forall B: *).
\[\text{PExp} V (A \to B) \to \text{PExp} V A \to \text{PExp} V B) =\]
\[\lambda V:* \to *. \lambda A: *. \lambda f: \text{PExp} V (A \to B). \lambda x: \text{PExp} V A.\]

\text{fold} (\text{PExp} V) B (\n \lambda R:*.
\lambda \text{var}: \text{VarF} V B R.
\lambda \text{abs}: \text{AbsF} V B R.
\lambda \text{app}: \text{AppF} V B R.
\lambda \text{tabs}: \text{TAbsF} V B R.
\lambda \text{tapp}: \text{TAppF} V B R.
\lambda \text{fld}: \text{FoldF} V B R.
\lambda \text{unfld}: \text{UnfoldF} V B R.
\text{app} A f x);\]

\textbf{decl tabs} : \ ((VV : * \to *). \forall A: *).
\[\text{IsAll} A \to \text{StripAll} A \to \text{UnderAll} A \to \text{All Id (PExp V) A} \to \text{PExp V A}) =\]
\[\lambda V:* \to *. \lambda A: *. \lambda p: \text{IsAll} A. \lambda s: \text{StripAll} A. \lambda u: \text{UnderAll} A. \lambda e: \text{All Id (PExp V) A}.\]

\text{fold} (\text{PExp} V) A (\n \lambda R:*.
\lambda \text{var}: \text{VarF} V A R.
\lambda \text{abs}: \text{AbsF} V A R.
\lambda \text{app}: \text{AppF} V A R.
\lambda \text{tabs}: \text{TAbsF} V A R.
\lambda \text{tapp}: \text{TAppF} V A R.
\lambda \text{fld}: \text{FoldF} V A R.
\lambda \text{unfld}: \text{UnfoldF} V A R.
\nu \nu : (\forall V : * \to *, \forall A : *, \forall B : *.
IsAll A \to Inst A B \to PExp V A \to PExp V B) =
\Delta V : * \to *, \Delta A : *, \Delta B : *.
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);

\nu \nu : (\forall V : * \to *, \forall F : (* \to *) \to * \to *, \forall A : *.
PExp V (F (μ F) A) \to PExp V (μ F A)) =
\Delta V : * \to *, \Delta F : (* \to *) \to * \to *, \Delta A : *.
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);

\nu \nu : (\forall V : * \to *, \forall F : (* \to *) \to * \to *, \forall A : *.
PExp V (μ F A) \to PExp V (F (μ F) A)) =
\Delta V : * \to *, \Delta F : (* \to *) \to * \to *, \Delta A : *.
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 : (\forall V : * \to *, \forall A : *, \forall R : *.
R \to VarF V A R) =
\Delta V : * \to *, \Delta A : *, \Delta R : *, \lambda r : R. \lambda x : V A. r;

decl constAbs : (\forall V : * \to *, \forall A : *, \forall R : *.
R \to AbsF V A R) =
\Delta V : * \to *, \Delta A : *, \Delta R : *, \lambda r : R.
\Delta A1 : *, \Delta A2 : *, \lambda eq : Eq (A1 \to A2) A. 
\lambda f : PExp V A1 \to PExp V A2. r;

decl constApp : (\forall V : * \to *, \forall A : *, \forall R : *.
R \to AppF V A R) =
\Delta V : * \to *, \Delta A : *, \Delta R : *, \lambda r : R.
\( \lambda B \, : \, \lambda e_1 : \text{PExp} \, V \, (B \, \rightarrow \, A) \, . \, \lambda e_2 : \text{PExp} \, V \, B \, . \, r \);  

\text{decl} \text{constTAbs} \, : \, (\forall V \, : \, * \, . \, \forall A \, : \, \forall R \, : \, * \, . \, \lambda p : \text{IsAll} \, A \, . \, \lambda s : \text{StripAll} \, A \, . \, \lambda u : \text{UnderAll} \, A \, . \, \lambda e : \text{All Id} \, (\text{PExp} \, V) \, A \, . \, r \);  

\text{decl} \text{constTApp} \, : \, (\forall V \, : \, * \, . \, \forall A \, : \, \forall R \, : \, * \, . \, \lambda p : \text{IsAll} \, B \, . \, \lambda \text{inst} : (\forall F \, : \, * \, . \, \text{All Id} \, F \, B \, \rightarrow \, F \, A) \, . \, \lambda f : \text{PExp} \, V \, B \, . \, r \);  

\text{decl} \text{constFold} \, : \, (\forall V \, : \, * \, . \, \forall A \, : \, \forall R \, : \, * \, . \, \lambda eqFold : \text{Eq} \, (\mu F \, B) \, A \, . \, \lambda e : \text{PExp} \, V \, (\mu F \, B) \, . \, r \);  

\text{decl} \text{matchVar} \, : \, (\forall V \, : \, * \, . \, \forall A \, : \, \forall R \, : \, * \, . \, \text{PExp} \, V \, A \, \rightarrow \, R \, \rightarrow \, \text{VarF} \, V \, A \, \rightarrow \, R) \, . \, \lambda \text{eq} : \text{Eq} \, (F \, (\mu F) \, B) \, A \, . \, \lambda e : \text{PExp} \, V \, (\mu F \, B) \, . \, r \);  

\text{decl} \text{matchAbs} \, : \, (\forall V \, : \, * \, . \, \forall A \, : \, \forall R \, : \, * \, . \, \text{PExp} \, V \, A \, \rightarrow \, R \, \rightarrow \, \text{AbsF} \, V \, A \, \rightarrow \, R) \, . \, \lambda \text{eq} : \text{Eq} \, (F \, (\mu F) \, B) \, A \, . \, \lambda e : \text{PExp} \, V \, (\mu F \, B) \, . \, r \);  

\text{decl} \text{matchApp} \, : \, (\forall V \, : \, * \, . \, \forall A \, : \, \forall R \, : \, * \, . \, \text{PExp} \, V \, A \, \rightarrow \, R \, \rightarrow \, \text{AppF} \, V \, A \, \rightarrow \, R) \, . \, \lambda \text{eq} : \text{Eq} \, (F \, (\mu F) \, B) \, A \, . \, \lambda e : \text{PExp} \, V \, (\mu F \, B) \, . \, r \);
decl matchTAbs : (\forall V : * \to *. \forall A : *. \forall R : *.
PExp V A \to R \to TAbsF V A R \to R) =
\Lambda V : * \to *. \Lambda A : *. \Lambda R : *.
\lambda e : PExp V A. \lambda default : R. \lambda whenTAbs : 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 : (\forall V : * \to *. \forall A : *. \forall R : *.
PExp V A \to R \to TAppF V A R \to R) =
\Lambda V : * \to *. \Lambda A : *. \Lambda R : *.
\lambda e : PExp V A. \lambda default : R. \lambda whenTApp : 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)
whenTApp
  (constTApp V A R default)
  (constFold V A R default)
  (constUnfold V A R default);

decl matchFold : (\forall V : * \to *. \forall A : *. \forall R : *.
PExp V A \to R \to FoldF V A R \to R) =
\Lambda V : * \to *. \Lambda A : *. \Lambda R : *.
\lambda e : PExp V A. \lambda default : R. \lambda whenFold : 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)
whenFold
  (constTApp V A R default)
  (constFold V A R default)
  (constUnfold V A R default);

decl matchUnfold : (\forall V : * \to *. \forall A : *. \forall R : *.
PExp V A \to R \to UnfoldF V A R \to R) =
\Lambda V : * \to *. \Lambda A : *. \Lambda R : *.
\lambda e : PExp V A. \lambda default : R. \lambda whenUnfold : 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)
whenUnfold
  (constTApp V A R default)
  (constFold V A R default)
  (constUnfold V A R default);

decl matchExp :
(\forall V : * \to *. \forall A : *. \forall R : *.
VarF V A R \to AbsF V A R \to AppF V A R \to
TAbsF V A R \to TAppF V A R \to
FoldF V A R \to UnfoldF V A R \to R) =
\Lambda V : * \to *. \Lambda A : *. \Lambda e : PExp V A.
unfold (PExpF V) A e e;

D.6 Weak Head Normal Form Evaluator

dec rec evalV : (\forall V : * \to *. \forall A : *. PExp V A \to PExp V A) =
\Lambda V : * \to *. \Lambda A : *. \lambda e : PExp V A.
matchExp V A e (PExp V A) e
  (constVar V A (PExp V A) e)
(constAbs V A (PExp V A) e)
(ΔB : *.*. λf : PExp V (B → A). λx : PExp V B.
  let f1 : PExp V B → 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
  let eqL : Eq B B1 = sym B1 B (arrR 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)
(ΔB : *. λp : IsAll B. λi : Inst B A. λe3 : All Id (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
(Δp : 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)
(ΔF : (* → *) → (* → *).
  ΔB : *.*. λeq : Eq (F (μ F) B) A.
  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
(ΔF1 : (* → *) → (* → *). ΔB1 : *.
  λeq1 : Eq (μ F1 B1) (μ F B).
  λeq3 : 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) =
  ΔΛ : *. λe : Exp A. ΔV : * → *. 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.

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

dec stepAbs : (∀V : * → *.
  (∀A : *. PExp (PExp V) A → PExp V A) →
  ∀A : *. AbsF (PExp V) A (PExp V A)) =
  λeq : Eq (A1 → A2) A.
  eq (PExp V)
  (abs V A1 A2 (λx : PExp V A1.
    step A2 (f (var (PExp V) A1 x)))));

dec stepApp : (∀V : * → *.
  (∀A : *. PExp (PExp V) A → PExp V A) →
  ∀A : *. AppF (PExp V) A (PExp V A)) =
  Δf : PExp (PExp V) B → 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
(ΔB1 : *. ΔA1 : *.).
\[ \begin{align*}
\lambda \text{eq} & : \text{Eq } (B_1 \to A_1) (B_1 \to A_1). \\
\lambda \text{f} & : \text{PExp } (\text{PExp } V) B_1 \to \text{PExp } (\text{PExp } V) A_1. \\
\text{let } & \text{eqB} \equiv \text{Eq } B_1 B = \text{arrL } B_1 A_1 B A \text{ eq in} \\
\text{let } & \text{eqA} \equiv \text{Eq } A_1 A = \text{arrR } B_1 A_1 B A \text{ eq in} \\
\text{let } & x_1 : \text{PExp } (\text{PExp } V) B_1 = \text{sym } B_1 B \text{ eqB } (\text{PExp } (\text{PExp } V)) x \text{ in} \\
\text{eqA } (\text{PExp } V) (\text{outExp } V A_1 (f \ x_1)).
\end{align*} \]

\[ \begin{align*}
\text{decl } & \text{stepTAbs} : (\forall V : * \to *.
\text{∀A : *}. \text{TAbsF } (\text{PExp } V) A (\text{PExp } V A)) = \\
\text{λV : * \to *. λstep : (∀A : *}. \text{PExp } (\text{PExp } V) A \to \text{PExp } V A). \\
\text{λA : *}. \lambda \rho : \text{IsAll } A. \lambda s : \text{StripAll } A. \\
\text{λu : UnderAll } A. \lambda e : \text{All } \text{Id } (\text{PExp } (\text{PExp } V)) A. \\
\text{tabs } V A p s u u (\text{PExp } V) (\text{step } e);
\end{align*} \]

\[ \begin{align*}
\text{decl } & \text{stepTApp} : (\forall V : * \to *.
\text{∀A : *}. \text{TAppF } (\text{PExp } V) A (\text{PExp } V A)) = \\
\text{λV : * \to *. λstep : (∀A : *}. \text{PExp } (\text{PExp } V) A \to \text{PExp } V A). \\
\text{λA : *}. \lambda B : *.
\lambda p : \text{IsAll } B. \lambda i : \text{Inst } B A. \\
\lambda e : \text{PExp } (\text{PExp } V) B. \\
\text{let } & \text{default } : \text{PExp } V A = \text{tapp } V B A \rho i (\text{step } B e) \text{ in} \\
\text{matchTAbs } (\text{PExp } V) B (\text{PExp } V A) e \text{ default} \\
(\lambda \rho : \text{IsAll } B. \lambda s : \text{StripAll } B. \lambda u : \text{UnderAll } B. \lambda e : \text{All } \text{Id } (\text{PExp } (\text{PExp } V)) B. \\
\text{outExp } V A (i (\text{PExp } (\text{PExp } V)) e));
\end{align*} \]

\[ \begin{align*}
\text{decl } & \text{stepFold} : (\forall V : * \to *.
\text{∀A : *}. \text{FoldF } (\text{PExp } V) A (\text{PExp } V A)) = \\
\text{λV : * \to *. λstep : (∀A : *}. \text{PExp } (\text{PExp } V) A \to \text{PExp } V A). \\
\text{λA : *}. \lambda F : (* \to *). \lambda B : *.
\lambda eq : \text{Eq } (\mu F B) A. \lambda e : \text{PExp } (\text{PExp } V) (F (\mu F) B). \\
\text{eq } (\text{PExp } V) (\text{fld } V F B (\text{step } F (\mu F) B) e);
\end{align*} \]

\[ \begin{align*}
\text{decl } & \text{stepUnfold} : (\forall V : * \to *.
\text{∀A : *}. \text{UnfoldF } (\text{PExp } V) A (\text{PExp } V A)) = \\
\text{λV : * \to *. λstep : (∀A : *}. \text{PExp } (\text{PExp } V) A \to \text{PExp } V A). \\
\text{λA : *}. \lambda F : (* \to *). \lambda B : *.
\lambda eq : \text{Eq } (\text{PExp } (\text{PExp } V) (\mu F B)) A. \lambda e : \text{PExp } (\text{PExp } V) (F (\mu F) B). \\
\text{let } & \text{default } : \text{PExp } V F (F (\mu F) B) = \text{unfld } V F B (\text{step } F (\mu F) B) e \text{ in} \\
\text{eq } (\text{PExp } V) (\text{matchFold } (\text{PExp } V) (\mu F B) (\text{PExp } V (F (\mu F) B)) e \text{ default} \\
(\lambda F1 : (* \to *) \to * \to *). \lambda B1 : *.
\lambda eq1 : \text{Eq } (\mu F1 B1) (\mu F B). \\
\lambda e : \text{PExp } (\text{PExp } V) (F1 (\mu F1) B1). \\
\text{eqUnfold } F1 B1 F B eq1 (\text{PExp } V) (\text{outExp } V (F1 (\mu F1) B1) e));
\end{align*} \]

\[ \begin{align*}
\text{decl rec } & \text{stepV} : (\forall V : * \to *. \forall A : *}. \text{PExp } (\text{PExp } V) A \to \text{PExp } V A) = \\
\text{λV : * \to *. λA : *}. \lambda e : \text{PExp } (\text{PExp } V) A. \\
\text{unfold } (\text{PExpF } (\text{PExp } V) A) e \text{ in} \\
(\text{PExp } V A) \quad \text{-- result type} \\
(\lambda x : \text{PExp } V A. x) \\
(\text{stepAbs } V (\text{stepV } V) A) (\text{stepApp } V (\text{stepV } V) A) \\
(\text{stepTBabs } V (\text{stepV } V) A) (\text{stepTApp } V (\text{stepV } V) A) \\
(\text{stepFold } V (\text{stepV } V) A) (\text{stepUnfold } V (\text{stepV } V) A); \\
\text{decl } & \text{step : (∀A : *}. \text{Exp } A \to \text{Exp } A) = \\
\text{λA : *}. \lambda e : \text{Exp } A. \\
\text{let } & \text{nf : Bool } = \text{nf } A e \text{ in} \\
\text{nf } (\text{Exp } A) e (\text{stepNorm } A (\text{step } e));
\end{align*} \]
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 : (* → *) → {* → *} → * → * =
          ∀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 PNfExp : (* → *) → * → * = λV : * → *. μ (PNeExpF V);

decl PNeExp : (* → *) → * → * = λV : * → *. μ (PNfExp FNe);

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

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

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

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

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

decl NfTApp : (* → *) → * → * → * =
          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
(ΛA:*. 
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 R. 
λabs : NfAbs V A R. 
λtabs : NfTabs V A R. 
λfld : NfFold V A R. 
abs A B (refl (A → B)) f);

decl mkNfTAbs : (∀V:* → *. ∀A:* → * All Id (PNfExp V) A → PNfExp V A) =
∀V:* → *. ∀A:*. 
IsAll A → StripAll A → UnderAll A → 
All Id (PNfExp V) A → PNfExp V A) =
∀V:* → *. ∀A:*. 
IsAll A → StripAll A → UnderAll A. 
λe : All Id (PNfExp V) A. 
fold (PNfExpF (PNeExp V)) A
(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
(ΛA : *. 
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) =
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);

  PNeExp V (μ F A) → PNeExp V (F (μ F) A)) = \\
  PNeExp V (μ F A) → PNeExp V (F (μ F) A)) = \\
  λe : PNeExp V (μ F A).
  fold (PNeExpF V) (B A) (ΛR : *.
    λvar : NeVar V (B A) R.
    λapp : NeApp V (B A) R.
    λtapp : NeTApp V (B A) R.
    λunfld : NeUnfold V (B A) R.
    unfld F A (refl (F (μ F) A)) e);

  PNeExp V (μ F A) → PNeExp V (F (μ F) A)) = \\
  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) → \\
  λe : PNeExp V (μ F A).
  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 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 : *.
  Eq (μ F B) A → \\
  Sem V (F (μ F) B) → \\
  R;

decl nbeNe : (∀V : * → *. ∀A : * → *. PNeExp V A → Sem V A) =
\[ \Lambda V: * \rightarrow *. \quad \Delta A: *. \quad \lambda e: \text{PNeExp V A.} \]
fold (SemF V) A (\lambda R: *).
\lambda ne: SemNe V A R.
\lambda arr: SemArr V A R.
\lambda all: SemAll V A R.
\lambda mu: SemMu V A R.
ne e);

\text{decl semAbs : (} \forall V: * \rightarrow *, \forall A: * , \forall B: *, (Sem V A \rightarrow Sem V B) \rightarrow Sem V (A \rightarrow B) =
\Lambda V: * \rightarrow *, \Lambda A: *. \quad \lambda f: Sem V A \rightarrow Sem V B.
fold (SemF V) (A \rightarrow B) (\lambda R: *) .
\lambda ne: SemNe V (A \rightarrow B) R.
\lambda arr: SemArr V (A \rightarrow B) R.
\lambda all: SemAll V (A \rightarrow B) R.
\lambda mu: SemMu V (A \rightarrow B) R.
arr A B (refl (A \rightarrow B)) f);

\text{decl semTAbs : (} \forall V: * \rightarrow *, \forall A: * .
IsAll A \rightarrow StripAll A \rightarrow UnderAll A \rightarrow All Id (Sem V) A \rightarrow Sem V A) =
\Lambda V: * \rightarrow *, \Lambda A: *
\lambda p: IsAll A. \lambda s: StripAll A. \lambda u: UnderAll A. \lambda e: All Id (Sem V) A.
fold (SemF V) (F (\mu F) A) (\lambda R: *) .
\lambda ne: SemNe V (\mu F) A R.
\lambda arr: SemArr V (\mu F) A R.
\lambda all: SemAll V (\mu F) A R.
\lambda mu: SemMu V (\mu F) A R.
all p s u e);

\text{decl semFold : (} \forall V: * \rightarrow *, \forall F: (* \rightarrow *) \rightarrow * \rightarrow *, \forall A: * .
Sem V (F (\mu F) A) \rightarrow Sem V (\mu F A) =
\Lambda V: * \rightarrow *, \Lambda F: (* \rightarrow *) \rightarrow * \rightarrow *, \Lambda A: *
\lambda e: Sem V (F (\mu F) A).
fold (SemF V) (F (\mu F) A) (\lambda R: *) .
\lambda ne: SemNe V (\mu F) A R.
\lambda arr: SemArr V (\mu F) A R.
\lambda all: SemAll V (\mu F) A R.
\lambda mu: SemMu V (\mu F) A R.
mu F A (refl (\mu F) A) e);

\text{decl reifyArr : (} \forall V: * \rightarrow *, \forall A: *
(\forall A1:* . Sem V A \rightarrow \text{PNfExp V A}) \rightarrow
\forall A: *, \forall A1: *, \forall A2: *.
Eq (A1 \rightarrow A2) A \rightarrow
(Sem V A1 \rightarrow Sem V A2) \rightarrow
\text{PNfExp V A) =
\Lambda V: * \rightarrow *, \lambda reify: (\forall A: *. Sem V A \rightarrow \text{PNfExp V A}).
\Lambda A: *, \Lambda A1:* . \Lambda A2:* . \lambda e: Eq (A1 \rightarrow A2) A. \lambda f: Sem V A1 \rightarrow Sem V A2.
eq (\text{PNfExp V})
(mkNfAbs V A1 A2 (\lambda x: \text{PNfExp V A1. reify A2 (f (nbeNe V A1 x)))));

\text{decl reifyAll : (} \forall V: * \rightarrow *, \forall A: *
(\forall A1:* . Sem V A \rightarrow \text{PNfExp V A}) \rightarrow
\forall A: *, \forall A1: *, \forall A2: *.
Eq (A1 \rightarrow A2) A \rightarrow
(Sem V A1 \rightarrow Sem V A2) \rightarrow
\text{PNfExp V A) =
\Lambda V: * \rightarrow *, \lambda reify: (\forall A: *. Sem V A \rightarrow \text{PNfExp V A}). \lambda A: *
\lambda p: IsAll A. \lambda s: StripAll A. \lambda u: UnderAll A. \lambda e: All Id (Sem V) A.
mkNfAbs V A1 A2 (\lambda x: \text{PNfExp V A1. reify A2 (f (nbeNe V A1 x)))));

\text{decl reifyMu : (} \forall V: * \rightarrow *, \forall A: *
(\forall A1:* . Sem V A \rightarrow \text{PNfExp V A}) \rightarrow
\forall A: *, \forall F: (* \rightarrow *) \rightarrow * \rightarrow *, \forall B: *.
Eq (F (\mu F) B) A \rightarrow Sem V (F (\mu F) B) \rightarrow \text{PNfExp V A) =
\Lambda V: * \rightarrow *, \lambda reify: (\forall A: *. Sem V A \rightarrow \text{PNfExp V A}).
\Lambda A: *, \Lambda F: (* \rightarrow *) \rightarrow * \rightarrow *, \Lambda B: *,
\[ \lambda e : \text{Eq } (\mu F B) \text{ A. } \lambda e : \text{Sem V } (F (\mu F) B). \]

let el : PNFExp V (\mu F B) = mkNFApply V (\mu F) F B (\text{reify } (F (\mu F) B) \text{ e} ) in

eq (PNFExp V) el;

decl rec reify : (\forall V : * \to *. \forall A :*. \text{Sem } V A \to \text{PNFExp } V A) =

\lambda V : * \to *., \lambda A :*. . \lambda e : \text{Sem V A.}

unfold (SemF V) A e

\begin{align*}
(\text{PNFExp } V) \text{ A} &\quad -- \text{ out} \\
(\text{mkNFApply } V) \text{ F B} &\quad -- \text{ ne} \\
(\text{reifyArr } V) \text{ (reify V) A} &\quad \text{ (reifyMu } V) \text{ (reify V) A);}
\end{align*}

decl semApp : (\forall V : * \to *. \forall B : *. \forall A : *.

\text{Sem } V (B \to A) \to \text{Sem V B} \to \text{Sem V A} = 

\lambda V : * \to *., \lambda A :*. . \lambda f: \text{Sem V } (B \to A). \lambda x: \text{Sem V B.}

unfold (SemF V) (B \to A) f (Sem V A)

-- ne

\begin{align*}
(\lambda f : \text{PNFExp } V (B \to A). \text{ nbeNe } V A (\text{mkNFApp } V B A f (\text{reify V B x}))) &\quad -- \text{ arr} \\
(\lambda A1 :*. . \lambda A2 :*. . \lambda eq: \text{Eq } (A1 \to A2) (B \to A). \\
\lambda f: \text{Sem V } A1 \to \text{Sem V } A2. \\
\text{let eqL } : \text{Eq } B A1 = \text{sym } A1 B (\text{arrL } A1 A2 B A eq) \text{ in} \\
\text{let eqV } : \text{Eq } A2 A = \text{arrR } A1 A2 B A eq \text{ in} \\
eq V (\text{Sem V} (f (\text{eqL } (\text{Sem V} x)))) &\quad -- \text{ all} \\
(\lambda p : \text{IsAll } (B \to A). \lambda s : \text{StripAll } (B \to A). \lambda u : \text{UnderAll } (B \to A). \\
\lambda e : \text{All Id } (\text{Sem V} (B \to A)). \text{ arrIsAll } B A p (\text{Sem V } A) &\quad -- \text{ mu} \\
(\lambda F : (* \to *) \to *., \lambda T :*. . \lambda p1 : \text{IsAll } (\mu F T) (B \to A). \\
\lambda eq : \text{Eq } (\mu F T) (\text{Sym } \mu F T) (B \to A) \text{ eq } (\text{Sem V } A)) &\quad \text{ (reifyMu } F T) (B \to A) \text{ eq } (\text{Sem V } A));
\end{align*}

decl semTApp : (\forall V : * \to *. \forall A : *. \forall B : *.

\text{Inst } A B \to \text{Sem V A} \to \text{Sem V B} = 

\lambda V : * \to *., \lambda A :*. . \lambda B :*. . \lambda f: \text{IsAll A. } \lambda i: \text{Inst } A B. \lambda e: \text{Sem V A.}

unfold (SemF V) (B \to A) f (\text{Sem V } B)

-- ne

\begin{align*}
(\lambda f : \text{PNFExp } V A. \text{ nbeNe } V B (\text{mkNFApp } V A B p i f)) &\quad -- \text{ arr} \\
(\lambda A1 :*. . \lambda A2 :*. . \lambda eq : \text{Eq } (A1 \to A2) A. \lambda f : \text{Sem V } A1 \to \text{Sem V } A2. \\
\text{let pl1 } : \text{IsAll } (A1 \to A2) = \text{sym } (A1 \to A2) A \text{ eq } (\text{IsAll } p \text{ in} \\
\text{arrIsAll } A1 A2 p1 (\text{Sem V } B)) &\quad -- \text{ all} \\
(\lambda p : \text{IsAll A. } \lambda s : \text{StripAll A. } \lambda u : \text{UnderAll A}. \\
\lambda e : \text{All Id } (\text{Sem V} ) A. i (\text{Sem V} e) &\quad -- \text{ mu} \\
(\lambda F : (* \to *) \to *., \lambda T :*. . \lambda eq : \text{Eq } (\mu F T) A. \\
\lambda e : \text{Sem V } (F (\mu F) T). \\
\text{let pl1 } : \text{IsAll } (\mu F T) = \text{sym } (\mu F T) A \text{ eq } (\text{IsAll } p \text{ in} \\
\text{muIsAll } F T pl1 (\text{Sem V } B));
\end{align*}

decl semUnfold : (\forall V : * \to *. \forall F : (* \to *) \to * \to *., \forall A :*.

\text{Sem V } (\mu F A) \to \text{Sem V } (F (\mu F) A) = 

\lambda V : * \to *., \lambda F : (* \to *) \to * \to *., \lambda A :*. 

\lambda e : \text{Sem V } (\mu F A).

unfold (SemF V) (\mu F A) e (\text{Sem V } (F (\mu F) A))

-- ne

\begin{align*}
(\lambda x : \text{PNFExp } V (\mu F A). \text{ nbeNe } V (F (\mu F) A) (\text{mkNFApp } V F A x)) &\quad -- \text{ arr} \\
(\lambda A1 :*. . \lambda A2 :*. . \lambda eq : \text{Eq } (A1 \to A2) (\mu F A). \lambda f : \text{Sem V } A1 \to \text{Sem V } A2. \\
\end{align*}
let bot : (∀C:*). C = eqArrMu A1 A2 F A eq in
bot (Sem V (F (μ F) A))
- all
(λAll : 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
(∃F1 : (* → *) → * → *). ΔA1:*.
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. ΔV:* → *. 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)
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
(λA : *. λf : PNeExp (PExp V) (B → A)). λx : PNfExp (PExp V) B.
app V B A (neToExp (B → A) f) (nftoExp B x)
-- tapp
tapp V B A p i (neToExp B f))
-- unfold
(λf : (* → *) → * → *.. ΔB : *).
λeq : Eq (F (μ F) B) A.
λe : PNeExp (PExp V) (μ F B).
let e1 : PExp V (F (μ F) B) = unfold V F B (neToExp (μ F B) e) in
eq (PExp V) e1);

decl nftoExpVar : (∀V : * → *).
(∀A : *. PNfExp (PExp V) A → PExp V A) →
ΔA : *.
neToExp V nftoExp A;

decl nftoExpAbs : (∀V : * → *).
(∀A : *. PNfExp (PExp V) A → PExp V A) →
∀A : *. NFAbs (PExp V) A (PExp V A))
ΔA:*.. ΔA1:*.. ΔA2:*.. λeq:Eq (A1 → A2) A.
eq (PEExp 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)));
\[\Lambda A : \ast. \lambda r : \text{IsAll} A. \lambda s : \text{StripAll} A. \lambda u : \text{UnderAll} A.\]
\[\lambda e : \text{All Id} (\text{PNfExp} (\text{PExp} V)) A.\]
\[\text{tabs} V A p s u (u (\text{PNfExp} (\text{PExp} V)) (\text{PExp} V) \text{nfToExp} e);\]

\[\text{decl} \text{nfToExpFold} : (V V : \ast \rightarrow \ast. (V A : \ast. \text{PNfExp} (\text{PExp} V) A \rightarrow \text{PExp} V A) A \rightarrow \text{PExp} V A) \rightarrow \text{PExp} V A) \text{nfToExp} (V A : \ast. \text{PNfExp} (\text{PExp} V) A \rightarrow \text{PExp} V A).\]
\[\Delta \Lambda : \ast. \Delta F : (\ast \rightarrow \ast) \rightarrow \ast. \Delta B : \ast.\]
\[\lambda e q : \text{Eq} (\mu F B) A. \lambda e : \text{PNfExp} (\text{PExp} V) (F (\mu F) B).\]
\[\text{let} e 1 : \text{PExp} V (\mu F B) = \text{fld} V F B (\text{nfToExp} (F (\mu F) B) e) \text{ in}\]
\[\text{eq} (\text{PExp} V) e 1;\]

\[\text{decl} \text{rec} \text{nfToExp} : (V V : \ast \rightarrow \ast. (V A : \ast. \text{PNfExp} (\text{PExp} V) A \rightarrow \text{PExp} V A) = \Delta V : \ast \rightarrow \ast. \Delta \Lambda : \ast. \lambda e : \text{PNfExp} (\text{PExp} V) A.\]
\[\text{unfold} (\text{PNfExp} (\text{PNeExp} (\text{PExp} V))) A e\]
\[\text{(PExp V A)}\]
\[\text{(nfToExpVar} V (\text{nfToExp} V) A) (\text{nfToExpAbs} V (\text{nfToExp} V) A)\]
\[\text{(nfToExpFold} V (\text{nfToExp} V) A);\]

\[\text{decl} \text{unNf} : (\forall T : \ast. \text{NfExp} T \rightarrow \text{Exp} T) = \Delta T : \ast. \lambda e : \text{NfExp} T. \lambda \Delta V : \ast \rightarrow \ast.\]
\[\text{nfToExp} V T (e (\text{PExp} V));\]

\[\text{decl} \text{norm} : (\forall A : \ast. \text{Exp} A \rightarrow \text{Exp} A) = \Delta A : \ast. \lambda e : \text{Exp} A. (\text{unNf} A (\text{nbe} A e));\]

\[\textbf{D.9 POPL'16 Meta-Programs}\]

\[\textbf{D.9.1 foldExp}\]

\[\text{decl} \text{foldAbs} : (V V : \ast \rightarrow \ast. (V A : \ast. \text{PExp} V A \rightarrow V A) A \rightarrow \Delta \Lambda : \ast. \lambda \Delta F : (\ast \rightarrow \ast) \rightarrow \ast. \Delta B : \ast.\]
\[\lambda v : (V A : \ast. \text{PExp} V A \rightarrow V A).\]
\[\text{AbsF} V A (V A) =\]
\[\Delta V : \ast \rightarrow \ast. \Delta \Lambda : \ast. \lambda \Delta F : (\ast \rightarrow \ast) \rightarrow \ast. \Delta B : \ast.\]
\[\lambda s : (V A : \ast. \text{PExp} V A \rightarrow V A).\]
\[\text{abs : (V A : \ast. \text{PExp} V A \rightarrow V A).}\]
\[\lambda e : (V A : \ast. \text{PExp} V A \rightarrow V A).\]
\[\text{eq} V (\text{abs} A 1 A 2 (\lambda x : V A 1).\]
\[\lambda \text{foldExp} A 2 (f (\text{var} V A 1 x)));\]

\[\text{decl} \text{foldApp} : (V V : \ast \rightarrow \ast. (V A : \ast. \text{PExp} V A \rightarrow V A) A \rightarrow (V A : \ast. \text{PExp} V A \rightarrow V A).\]
\[\lambda \text{AppF} V A (V A);\]
\[\Delta V : \ast \rightarrow \ast. \Delta \Lambda : \ast. \lambda \Delta F : (\ast \rightarrow \ast) \rightarrow \ast. \Delta B : \ast.\]
\[\lambda v : (V A : \ast. \text{PExp} V A \rightarrow V A).\]
\[\lambda \lambda v : (V A : \ast. \text{PExp} V A \rightarrow V A).\]
\[\lambda p : (V A : \ast. \text{PExp} V A \rightarrow V A).\]
\[\text{app B A (foldExp} B (A) e 1 (\text{foldExp} A B e 2);\]

\[\text{decl} \text{foldTAbs} : (V V : \ast \rightarrow \ast. (V A : \ast. \text{PExp} V A \rightarrow V A) A \rightarrow (V A : \ast. \text{IsAll} A \rightarrow \text{StripAll} A \rightarrow \text{UnderAll} A \rightarrow (\text{All Id} V A) \rightarrow V A) A \rightarrow \Delta \Lambda : \ast. \lambda \Delta F : (\ast \rightarrow \ast) \rightarrow \ast. \Delta B : \ast.\]
\[\lambda \lambda v : (V A : \ast. \text{IsAll} A \rightarrow \text{StripAll} A \rightarrow \text{UnderAll} A \rightarrow (\text{All Id V A) \rightarrow V A).\]
\[\lambda p : (V A : \ast. \text{IsAll} A \rightarrow \text{StripAll} A \rightarrow \text{UnderAll} A \rightarrow (\text{All Id V A) \rightarrow V A).\]
\[\lambda r : (V A : \ast. \text{IsAll} A \rightarrow \text{StripAll} A \rightarrow \text{UnderAll} A \rightarrow (\text{All Id V A) \rightarrow V A).\]
\[\text{tabs} A p s u (u (\text{PExp} V) V \text{foldExp} e);\]

\[\text{decl} \text{foldTApp} : (V V : \ast \rightarrow \ast. (V A : \ast. \text{PExp} V A \rightarrow V A) A \rightarrow (V A : \ast. \text{IsAll} A \rightarrow \text{Inst} A B \rightarrow \text{V A} \rightarrow \text{V B) A \rightarrow \Delta \Lambda : \ast. \lambda \Delta F : (\ast \rightarrow \ast) \rightarrow \ast. \Delta B : \ast.\]
\[\lambda \lambda v : (V A : \ast. \text{IsAll} A \rightarrow \text{Inst} A B \rightarrow \text{V A} \rightarrow \text{V B).\]
\[\Delta B : \ast. \lambda p : (V A : \ast. \text{IsAll} A \rightarrow \text{Inst} A B \rightarrow \text{V A} \rightarrow \text{V B).\]
\[\lambda \lambda v : (V A : \ast. \text{IsAll} A \rightarrow \text{Inst} A B \rightarrow \text{V A} \rightarrow \text{V B).\]
\[\text{tapp B A p i (foldExp} B e);\]

\[\text{decl} \text{foldFold} : (V V : \ast \rightarrow \ast. (V A : \ast. \text{PExp} V A \rightarrow V A) A \rightarrow (V F : (\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast. (V A : \ast. (V (\mu F) A) \rightarrow V (\mu F) A) \rightarrow \text{FoldF} V A (V A) =\)
\[ \Lambda V : * \rightarrow *. \quad \Lambda A : * . \quad \lambda \text{foldExp} : (\forall A : *. \text{PExp} V A \rightarrow V A). \]

\[ \lambda \text{fld} : (\forall F : (* \rightarrow *) \rightarrow * \rightarrow * . \quad \forall A : * . \quad \text{V} (F (\mu F) A) \rightarrow V (\mu F A)). \]

\[ \Lambda F : (* \rightarrow *) \rightarrow * \rightarrow * . \quad \Lambda B : * . \quad \lambda \text{eqFold} : (\forall \mu F : A . \quad \lambda e : \text{PExp} V (F (\mu F) B). \quad \text{eqFold} V (\text{fld} F B (\text{foldExp} (F (\mu F) B) e)). \]

\[ \text{decl} \quad \text{foldUnfold} : (\forall V : * \rightarrow *. \quad \forall A : * . \quad (\forall A : * . \quad \text{PExp} V A \rightarrow V A) \rightarrow \text{UnfoldF} V A (\text{V} A)). \]

\[ \lambda V : * \rightarrow * . \quad \forall A : * . \quad \forall B : * . \quad \text{eq} : \text{Eq} (F (\mu F) B) A. \quad \lambda e : \text{PExp} V (\mu F B). \quad \text{eq} V (\text{unfld} F B (\text{foldExp} (\mu F B) e)); \]

\[ \text{decl} \quad \text{FoldAbs} : (* \rightarrow *) \rightarrow * = \lambda V : * \rightarrow * . \quad \forall A : * . \quad \forall B : * . \quad (V A \rightarrow V B) \rightarrow V (A \rightarrow B); \]

\[ \text{decl} \quad \text{FoldApp} : (* \rightarrow *) \rightarrow * = \lambda V : * \rightarrow * . \quad \forall A : * . \quad \forall B : * . \quad V (A \rightarrow B) \rightarrow V A \rightarrow V B; \]

\[ \text{decl} \quad \text{FoldTabs} : (* \rightarrow *) \rightarrow * = \lambda V : * \rightarrow * . \quad \forall A : * . \quad \text{IsAll} A \rightarrow \text{StripAll} A \rightarrow \text{UnderAll} A \rightarrow (\text{All Id} V A) \rightarrow V A; \]

\[ \text{decl} \quad \text{FoldTApp} : (* \rightarrow *) \rightarrow * = \lambda V : * \rightarrow * . \quad \forall A : * . \quad \forall B : * . \quad \text{IsAll} A \rightarrow \text{Inst} A B \rightarrow V A \rightarrow V B; \]

\[ \text{decl} \quad \text{FoldFold} : (* \rightarrow *) \rightarrow * = \lambda V : * \rightarrow * . \quad \forall F : (* \rightarrow *) \rightarrow * \rightarrow *. \quad \forall A : *. \quad V (F (\mu F) A) \rightarrow V (\mu F A); \]

\[ \text{decl} \quad \text{rec} \quad \text{foldExpV} : (\forall V : * \rightarrow * . \quad \text{FoldAbs} V \rightarrow \text{FoldApp} V \rightarrow \text{FoldTabs} V \rightarrow \text{FoldTApp} V \rightarrow \text{FoldFold} V \rightarrow \text{FoldUnfold} V \rightarrow \forall A : *. \quad \text{PExp} V A \rightarrow V A) = \]

\[ \lambda V : * \rightarrow *, \quad \lambda \text{abs} : \text{FoldAbs} V A. \quad \lambda \text{app} : \text{FoldApp} V A. \quad \lambda \text{tabs} : \text{FoldTabs} V A. \quad \lambda \text{tapp} : \text{FoldTApp} V A. \quad \lambda \text{fld} : \text{FoldFold} V A. \quad \lambda \text{unfld} : \text{FoldUnfold} V A. \quad \text{let} \quad \text{foldExp} : (\forall A : * . \quad \text{PExp} V A \rightarrow V A) = \text{foldExpV} V A \rightarrow \text{abs} \rightarrow \text{app} \rightarrow \text{tabs} \rightarrow \text{tapp} \rightarrow \text{fld} \rightarrow \text{unfld}. \]

\[ \text{let} \quad \text{foldExp} : (\forall V : * \rightarrow * . \quad \text{FoldAbs} V \rightarrow \text{FoldApp} V \rightarrow \text{FoldTabs} V \rightarrow \text{FoldTApp} V \rightarrow \text{FoldFold} V \rightarrow \text{FoldUnfold} V \rightarrow \forall A : *. \quad \text{Exp} A \rightarrow V A) = \text{foldExpV} V A \rightarrow \text{abs} \rightarrow \text{app} \rightarrow \text{tabs} \rightarrow \text{tapp} \rightarrow \text{fld} \rightarrow \text{unfld}; \]

\[ \text{decl} \quad \text{foldExp} : (\forall V : * \rightarrow * . \quad \text{FoldAbs} V \rightarrow \text{FoldApp} V \rightarrow \text{FoldTabs} V \rightarrow \text{FoldTApp} V \rightarrow \text{FoldFold} V \rightarrow \text{FoldUnfold} V \rightarrow \forall A : *. \quad \text{Exp} A \rightarrow V A) = \]

\[ \lambda V : * \rightarrow *, \quad \lambda \text{abs} : \text{FoldAbs} V A. \quad \lambda \text{app} : \text{FoldApp} V A. \quad \lambda \text{tabs} : \text{FoldTabs} V A. \quad \lambda \text{tapp} : \text{FoldTApp} V A. \quad \lambda \text{fld} : \text{FoldFold} V A. \quad \lambda \text{unfld} : \text{FoldUnfold} V A. \]

\[ \lambda A : * . \quad \lambda e : \text{Exp} A. \quad \text{foldExpV} V A \rightarrow \text{abs} \rightarrow \text{app} \rightarrow \text{tabs} \rightarrow \text{tapp} \rightarrow \text{fld} \rightarrow \text{unfld} A \rightarrow (e V); \]
D.9.2 unquote

\[ \text{decl unquoteAbs : FoldAbs Id} = \Delta A:\ast. \Delta B:\ast. \lambda f : A \to B. f; \]
\[ \text{decl unquoteApp : FoldApp Id} = \Delta A:\ast. \Delta B:\ast. \lambda f : A \to B. f; \]
\[ \text{decl unquoteTabs : FoldTabs Id} = \]
\[ \Delta A:\ast. \lambda p : \text{IsAll A}. \lambda s : \text{StripAll A}. \lambda u : \text{UnderAll A}. \lambda e : \text{All Id Id A}. \]
\[ \text{let eq : Eq (All Id Id A) A = } \text{unAll A p Id} \text{ in eq Id e; } \]
\[ \text{decl unquoteTApp : FoldTApp Id} = \]
\[ \Delta A:\ast. \Delta B:\ast. \lambda p : \text{IsAll A}. \lambda i : \text{Inst A B}. \lambda x : A. \]
\[ \text{let eq : Eq A (All Id Id A) = } \text{sym (All Id Id A) A (unAll A p Id) in i Id (eq Id x); } \]
\[ \text{decl unquoteFold : FoldFold Id} = \]
\[ F: (\ast \to \ast) \to \ast \to \ast. \Delta A:\ast. \lambda x : F (\mu F) A. \text{fold F A x; } \]
\[ \text{decl unquoteUnfold : FoldUnfold Id} = \]
\[ F: (\ast \to \ast) \to \ast \to \ast. \Delta A:\ast. \lambda x : \mu F A. \text{unfold F A x; } \]
\[ \text{decl unquote : } (\forall A: \ast. \text{Exp A} \to A) = \]
\[ \text{foldExp Id unquoteAbs unquoteApp unquoteTabs unquoteTApp unquoteFold unquoteUnfold; } \]

D.9.3 cps

\[ \text{decl Ct : } * \to = \lambda A:\ast. \forall B: \ast. (A \to B) \to B; \]
\[ \text{decl ct : } (\forall A: \ast. A \to \text{Ct A}) = \]
\[ \lambda A:\ast. \lambda x : A. \lambda B:\ast. \lambda f : A \to B. f x; \]
\[ \text{decl CPS1F : } (\ast \to *) \to * \to * = \lambda CPS1:* \to *. \lambda A:\ast. \]
\[ \text{Typecase} \]
\[ (\lambda x:\ast. \lambda y:*). \text{Ct } (\text{CPS1 } x) \to \text{Ct } (\text{CPS1 } y)) \]
\[ \text{Id } (\lambda x:\ast. \lambda y:*). \text{Ct } (\text{CPS1 } x)) \]
\[ (\lambda F : (\ast \to *) \to * \to *. \lambda B : *. \text{Ct } (\text{CPS1 } (F (\mu F) B))) \]
\[ A; \]
\[ \text{decl CPS1 : } * \to = \mu \text{ CPS1F; } \]
\[ \text{decl CPS : } * \to = \lambda A:\ast. \text{Ct } (\text{CPS1 } A); \]
\[ \text{decl cpsAbs : FoldAbs CPS } = \]
\[ \Delta A:\ast. \Delta B:\ast. \lambda f : CPS A \to CPS B. \]
\[ \Delta V:* . \lambda k : CPS1 (A \to B) \to V. \]
\[ k \text{ (fold CPS1F } (A \to B) f); \]
\[ \text{decl cpsApp : FoldApp CPS } = \]
\[ \Delta A:* . \Delta B:* . \lambda e1 : CPS (A \to B). \lambda e2 : CPS A. \]
\[ \Delta V:* . \lambda k : CPS1 B \to V. \]
\[ e1 V (\lambda V:* . \lambda k : CPS1 (A \to B). \text{unfold CPS1F } (A \to B) f e2 V k); \]
\[ \text{decl eqCPSAll : } (\forall A : * . \text{IsAll A} \to \text{Eq } (\text{CPS1F CPS1 A}) \text{ (All Id CPS A)}) = \]
\[ \Delta A:* . \lambda p : \text{IsAll A}. \]
\[ \text{tcAll A p } (\lambda x:* . \lambda y:* . \text{CPS X } \to \text{CPS Y}) \text{ Id CPS} \]
\[ (\lambda F : (\ast \to *) \to * \to *. \lambda B:* . \text{CPS } (F (\mu F) B)); \]
\[ \text{decl cpsTabs : FoldTabs CPS } = \]
\[ \Delta A:* . \lambda p : \text{IsAll A}. \lambda s : \text{StripAll A}. \lambda u : \text{UnderAll A}. \lambda e : \text{All Id CPS A}. \]
\[ \text{let e1 : CPS1 CPS1 A } = \]
\[ \text{sym (CPS1 CPS1 A) (All Id CPS A) (eqCPSAll A p) Id e} \]
\[ \text{in} \]
\[ \text{let e2 : CPS1 A } = \text{fold CPS1F A e1 in} \]
\[ \Delta V:* . \lambda k : CPS1 A \to V. k e2; \]
D.10 Normal form checker

In previous work [8], we implemented a normal form checker as a fold. While this is possible for F_\text{FD} too, it would only work on closed representations (with types of the form \text{Exp} T). This normal form checker is not implemented as a fold, which allows it to check open representations (of type \text{PExp} V T for any V, T). We use this capability in step 1 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 : (\forall V:* \to *. \forall A:*. \text{VarF} V A \text{ Bools}) =
\Lambda A:*. \Lambda V:* . \lambda x : V A. \text{bools} \text{ true true};

decl nfNeAbs : (\forall V:* \to *. \forall A:*. (\forall A:* . \text{PExp} V A \to \text{Bools}) \to \text{AbsF} V A \text{ Bools}) =
\Lambda A:* . \Lambda V:* . \lambda nfNe : (\forall A:* . \text{PExp} V A \to \text{Bools}).
\Lambda A1:* . A2:* . \lambda eq : Eq (A1 \to A2) A . \lambda f : \text{PExp} V A1 \to \text{PExp} V A2.
let x : \text{PExp} V A1 = \text{var} V A1 (\text{bottom} (V A1)) \text{ in}
\text{bools} (\text{fstBools} (\text{nfNeA} (A2 \to A) f x)) \text{ false};

decl nfNeApp : (\forall V:* \to *. \forall A:* . (\forall A:* . \text{PExp} V A \to \text{Bools}) \to \text{AppF} V A \text{ Bools}) =
\Lambda A:* . \Lambda V:* . \lambda nfNe : (\forall A:* . \text{PExp} V A \to \text{Bools}).
\Lambda B:* . \lambda f : \text{PExp} V B (B \to A) . \lambda x : \text{PExp} V B.
let f_nfNe : \text{Bools} = nfNe (B \to A) f \text{ in }
let x_nfNe : \text{Bools} = nfNe B x \text{ in }
let ne : \text{Bools} = \text{and} (\text{sndBools} f_nfNe) (\text{fstBools} x_nfNe) \text{ in }
\text{bools ne ne};

decl nfNeTabs : (\forall V:* \to *. \forall A:* . (\forall A:* . \text{PExp} V A \to \text{Bools}) \to \text{TabF} V A \text{ Bools}) =
\Lambda V:* . \Lambda A:* . \lambda nfNe : (\forall A:* . \text{PExp} V A \to \text{Bools}).
\lambda p : \text{IsAll A} . \lambda s : \text{ StripAll A} . \lambda u : \text{UnderAll A} . \lambda e : \text{All Id} (\text{PExp} V) A.
let e1 : \text{All Id} (\lambda A:* . \text{Bools}) A = u (\text{PExp} V) (\lambda A:* . \text{Bools}) \text{ nfNe e in }
\text{bools} (\text{fstBools} bs) \text{ false};

decl nfNeTapp : (\forall V:* \to *. \forall A:* . (\forall A:* . \text{PExp} V A \to \text{Bools}) \to \text{TAppF} V A \text{ Bools}) =
\Lambda V:* . \Lambda A:* . \lambda nfNe : (\forall A:* . \text{PExp} V A \to \text{Bools}).
\Lambda B:* . \lambda p : \text{IsAll B} . \lambda i : \text{Inst B A} . \lambda e : \text{PExp} V B.
let ne : \text{Bools} = \text{sndBools} (\text{nfNe B e}) \text{ in }
\text{bools ne ne};
decl nfNeFold : (\forall V : * \to *. \forall A : * . (PExp V A \to Bools) \to FoldF V A Bools) =
\Lambda V : * \to *. \Lambda A : *. AnfNe : (\forall A : *. PExp V A \to Bools).
\Lambda F : (* \to *) \to * \to *. \Lambda B : *. \lambda eqFold : Eq (\mu F B) A. \lambda e : PExp V (F (\mu F B)).
bools (fstBools (nfNe (F (\mu F B) e)) false;

decl nfNeUnfold : (\forall V : * \to *. \forall A : *. (PExp V A \to Bools) \to UnfoldF V A Bools) =
\Lambda V : * \to *. \Lambda A : *. \lambda nfNe : (\forall A : *. PExp V A \to Bools).
\Lambda F : (* \to *) \to * \to *. \Lambda B : *. \lambda eq: Eq (F (\mu F B) A). \lambda e : PExp V (F (\mu F B)).
let ne : Bool = sndBools (nfNe (\mu F B) e) in
bools ne ne;

decl rec nfNe : (\forall V : * \to *. \forall A : *. PExp V A \to Bools) =
\Lambda V : * \to *. \Lambda A : *. \lambda e : 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)) (nfNeFold V A (nfNe V)) (nfNeUnfold V A (nfNe V));

decl nfV : (\forall V : * \to *. \forall A : *. PExp V A \to Bools) =
\Lambda V : * \to *. \Lambda A : *. \lambda e : PExp V A.
fstBools (nfNe V A e);

decl nf : (\forall A : *. Exp A \to Bool) = \Lambda A : *. \lambda e : Exp A. nfV Id A (e Id);

D.11 size

decl KNat : * \to * = \lambda A : *. Nat;

decl sizeAbs : FoldAbs KNat =
\Lambda A : *. \Lambda B : *. \lambda f : Nat \to Nat. succ (f (succ zero));

decl sizeApp : FoldApp KNat =
\Lambda A : *. \Lambda B : *. \lambda f : Nat. \lambda x : Nat. succ (plus f x);

decl sizeTAbs : FoldTAbs KNat =
\Lambda A : *. \lambda p : IsAll A. \lambda s : StripAll A. \lambda u : UnderAll A. \lambda f : All Id KNat A.
succ (s Nat f);

decl sizeTApp : FoldTApp KNat =
\Lambda A : *. \Lambda B : *. \lambda p : IsAll A. \lambda i : Inst A B. \lambda f : Nat. succ f;

decl sizeFold : FoldFold KNat =
\Lambda F : (* \to *) \to * \to *. \Lambda A : *. \lambda n : Nat. succ n;

decl sizeUnfold : FoldUnfold KNat =
\Lambda F : (* \to *) \to * \to *. \Lambda A : *. \lambda n : Nat. succ n;

decl size : (\forall A : *. Exp A \to Nat) =
foldExp KNat sizeAbs sizeTAbs sizeTApp sizeFold sizeUnfold;

D.12 isAbs

decl KBool : * \to * = \lambda A : *. Bool;

decl isAbsAbs : FoldAbs KBool =
\Lambda A : *. \Lambda B : *. \lambda f : Bool \to Bool. true;

decl isAbsApp : FoldApp KBool =
\Lambda A : *. \Lambda B : *. \lambda f : Bool. \lambda : Bool. false;

decl isAbsTAbs : FoldTAbs KBool =
\Lambda A : *. \lambda p : IsAll A. \lambda s : StripAll A. \lambda u : UnderAll A. \lambda f : All Id KBool A. true;
decl isAbsTApp : FoldTApp KBool =
  \A:*. \B:*. \p:IsAll A. \i:Inst A B. \f:Bool. false;

decl isAbsFold : FoldFold KBool =
  \F : (* → *) → * → *. \A:*. \n : Bool. false;

decl isAbsUnfold : FoldUnfold KBool =
  \F : (* → *) → * → *. \A:*. \n : Bool. false;

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