Formalization of Pure Type Systems

Last modified: March 29, 2015 at 7:15pm

1. Definition

(i) A *pure type system* (*PTS*) is a triple tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where

  (a) $\mathcal{S}$ is a set of *sorts*;

  (b) $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of *axioms*;

  (c) $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of *rules*.

Following standard practice, we use (s_1, s_2) to denote rules of the form (s_1, s_2, s_2).

(ii) *Raw expressions* A and *raw environments* Γ are defined by

$$A ::= x \mid s \mid AA \mid \lambda x : A.\ A \mid \Pi x : A.\ A$$
$$\Gamma ::= \varnothing \mid \Gamma, x : A$$

where we use s, t, u, etc., to range over sorts, x, y, z, etc., to range over variables, and A, B, C, a, b, c, etc., to range over expressions.

(iii) Π and λ are used to bind variables. Let $\mathrm{FV}(A)$ denote free variable set of A. Let $A[x := B]$ denote the substitution of x in A with B. Standard notational conventions are applied here. Besides we also let $A \to B$ be an abbreviation for $(\Pi_{\_} : A.\ B)$.

(iv) The relation $\to_\beta$ is the smallest binary relation on raw expressions satisfying

$$(\lambda x : A.\ M)N \to_\beta M[x := N]$$

which can be used to define the notation $\twoheadrightarrow_\beta$ and $=_\beta$ by convention.

(v) Type assignment rules for $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ are given in Table 3. Particularly, the rule *(Conv)* is needed to make everything work.

2. Examples of PTSs

(i) Here we present the formal definition of a type system called *the calculus of construction (λC)*, where

  (a) $\mathcal{S} = \{\star, \square\}$

  (b) $\mathcal{A} = \{(\star, \square)\}$

  (c) $\mathcal{R} = \{(\star, \star), (\star, \square), (\square, \star), (\square, \square)\}$

and the typing relation is shown in Table 1.

(Ax)	$\dfrac{}{\vdash \star : \Box}$	
(Var)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$	$x \notin \mathrm{dom}(\Gamma)$
(Weak)	$\dfrac{\Gamma \vdash b : B \qquad \Gamma \vdash A : s}{\Gamma, x : A \vdash b : B}$	$x \notin \mathrm{dom}(\Gamma)$
(App)	$\dfrac{\Gamma \vdash f : (\Pi x : A.\ B) \qquad \Gamma \vdash a : A}{\Gamma \vdash f a : B[x := a]}$	
(Lam)	$\dfrac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.\ B) : t}{\Gamma \vdash (\lambda x : A.\ b) : (\Pi x : A.\ B)}$	$t \in \{\star, \Box\}$
(Pi)	$\dfrac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : t}{\Gamma \vdash (\Pi x : A.\ B) : t}$	$(s, t) \in \mathcal{R}$
(Conv)	$\dfrac{\Gamma \vdash a : A \qquad \Gamma \vdash B : s \qquad A =_\beta B}{\Gamma \vdash a : B}$	

Table 1. Typing rules for λC

(ii) An extension of $\lambda\omega$ that supports "polymorphic identity function on types", where

(a) $\mathcal{S} = \{\star, \Box, \Box'\}$

(b) $\mathcal{A} = \{(\star, \Box), (\Box, \Box')\}$

(c) $\mathcal{R} = \{(\star, \star), (\Box, \star), (\Box, \Box), (\Box', \Box')\}$

in which we can have $\vdash (\lambda\kappa : \Box.\ \lambda\alpha : \kappa.\ \alpha) : (\Pi\kappa : \Box.\ \kappa \to \kappa)$, justified as follows:

$$\dfrac{\dfrac{\dfrac{\mathcal{B}}{\kappa : \Box, \alpha : \kappa \vdash \alpha : \kappa}\,\mathit{Var} \qquad \mathcal{A}}{\kappa : \Box \vdash (\lambda\alpha : \kappa.\ \alpha) : (\Pi\alpha : \kappa.\ \kappa)}\,\mathit{Lam} \qquad \dfrac{\dfrac{}{\vdash \Box : \Box'}\,\mathit{Ax} \qquad \mathcal{A}}{\vdash (\Pi\kappa : \Box.\ \Pi\alpha : \kappa.\ \kappa) : \Box}\,\mathit{Pi}}{\vdash (\lambda\kappa : \Box.\ \lambda\alpha : \kappa.\ \alpha) : (\Pi\kappa : \Box.\ \Pi\alpha : \kappa.\ \kappa)}\,\mathit{Lam}$$

$$\mathcal{A} = \quad \dfrac{\mathcal{B} \qquad \dfrac{\mathcal{B} \qquad \mathcal{B}}{\kappa : \Box, \alpha : \kappa \vdash \kappa : \Box}\,\mathit{Weak}}{\kappa : \Box \vdash (\Pi\alpha : \kappa.\ \kappa) : \Box}\,\mathit{Pi}$$

$$\mathcal{B} = \quad \dfrac{\dfrac{}{\vdash \Box : \Box'}\,\mathit{Ax}}{\kappa : \Box \vdash \kappa : \Box}\,\mathit{Var}$$

3. Extending PTSs

3.1 Recursive types

3.1.1 Definition

We extend Calculus of Constructions (λC, see Section 2) with recursive types, namely λC_μ. The raw expressions are extended as follows:

$$\begin{array}{rcl} A & ::= & x \mid \star \mid \square \\ & \mid & AA \mid \lambda x : A.A \mid \Pi x : A.A \\ & \mid & \mu x.A \mid \mathsf{fold}[A]\,A \mid \mathsf{unfold}[A]\,A \\ & \mid & \mathsf{beta}\,A \end{array}$$

We introduce a new reduction rule for unfold and fold:

$$\mathsf{unfold}[A]\,(\mathsf{fold}[B]\,a) \to a$$

The extended typing rules are shown in Table 2. Compared with λC, the original *Conv* rule is replaced by the new *Beta* rule where the latter only performs one step of β-reduction.

(Ax)	$\dfrac{}{\vdash \star : \square}$	
(Var)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$	$x \notin \mathrm{dom}(\Gamma)$
(Weak)	$\dfrac{\Gamma \vdash b : B \qquad \Gamma \vdash A : s}{\Gamma, x : A \vdash b : B}$	$x \notin \mathrm{dom}(\Gamma)$
(App)	$\dfrac{\Gamma \vdash f : (\Pi x : A.\, B) \qquad \Gamma \vdash a : A}{\Gamma \vdash f a : B[x := a]}$	
(Lam)	$\dfrac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.\, B) : t}{\Gamma \vdash (\lambda x : A.\, b) : (\Pi x : A.\, B)}$	$t \in \{\star, \square\}$
(Pi)	$\dfrac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : t}{\Gamma \vdash (\Pi x : A.\, B) : t}$	$(s, t) \in \mathcal{R}$
(Mu)	$\dfrac{\Gamma, x : s \vdash A : s}{\Gamma \vdash (\mu x.A) : s}$	
(Fold)	$\dfrac{\Gamma \vdash a : (A[x := \mu x.A]) \qquad \Gamma \vdash \mu x.A : s}{\Gamma \vdash (\mathsf{fold}[\mu x.A]\,a) : \mu x.A}$	
(Unfold)	$\dfrac{\Gamma \vdash a : \mu x.A \qquad \Gamma \vdash A[x := \mu x.A] : s}{\Gamma \vdash (\mathsf{unfold}[\mu x.A]\,a) : A[x := \mu x.A]}$	
(Beta)	$\dfrac{\Gamma \vdash a : A \qquad \Gamma \vdash B : s \qquad A \to_\beta B}{\Gamma \vdash (\mathsf{beta}\,a) : B}$	

Table 2. Typing rules for λC_μ

3.1.2 Examples of typable terms

By convention, we can abbreviate a product $\Pi x : A.B$ to $A \to B$ when $x \notin \mathrm{FV}(B)$.

- A polymorphic fixed-point constructor fix : $(\Pi\alpha : \star.(\alpha \to \alpha) \to \alpha)$ can be defined as follows:

$$\begin{aligned}\mathsf{fix} =& \lambda\alpha : \star.\lambda f : \alpha \to \alpha.\\ &\quad (\lambda x : (\mu\sigma.\sigma \to \alpha).f((\mathsf{unfold}[\mu\sigma.\sigma \to \alpha]\, x)x))\\ &\quad (\mathsf{fold}[\mu\sigma.\sigma \to \alpha]\,(\lambda x : (\mu\sigma.\sigma \to \alpha).f((\mathsf{unfold}[\mu\sigma.\sigma \to \alpha]\, x)x)))\end{aligned}$$

- Using fix, we can build recursive functions. For example, given a "hungry" type $H = \mu\sigma.\alpha \to \sigma$, the "hungry" function h where

$$h = \lambda\alpha : \star.\mathsf{fix}\,(\alpha \to H)\,(\lambda f : \alpha \to H.\lambda x : \alpha.\mathsf{fold}[H]\, f)$$

can take arbitrary number of arguments.

3.2 Encoding of Datatypes

3.2.1 Examples of Simple Datatypes

- We can encode the type of natural numbers as follow:

$$\mathsf{Nat} = \mu X.\, \Pi(a : \star).\, a \to (X \to a) \to a$$

then we can define zero and suc as follows:

$$\begin{aligned}\mathsf{zero} &: \mathsf{Nat}\\ \mathsf{zero} &= \mathsf{fold}[\mathsf{Nat}]\,(\lambda(a : \star)(z : a)(f : \mathsf{Nat} \to a).\, z)\\ \mathsf{suc} &: \mathsf{Nat} \to \mathsf{Nat}\\ \mathsf{suc} &= \lambda(n : \mathsf{Nat}).\, \mathsf{fold}[\mathsf{Nat}]\,(\lambda(a : \star)(z : a)(f : \mathsf{Nat} \to a).\, f\, n)\end{aligned}$$

Using fix, we can define a recursive function plus as follow:

$$\begin{aligned}\mathsf{plus} &: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\ \mathsf{plus} &= \mathsf{fix}\,(\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat})\,(\lambda(p : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat})(n : \mathsf{Nat})(m : \mathsf{Nat}).\\ &\qquad (\mathsf{unfold}[\mathsf{Nat}]\, n)\,\mathsf{Nat}\, m\,(\lambda(n' : \mathsf{Nat}).\, \mathsf{suc}\,(p\, n'\, m)))\end{aligned}$$

- We can encode the type of lists of a certain type:

$$\mathsf{List} = \mu X.\, \Pi(a : \star).\, a \to (\Pi(b : \star).\, b \to X \to a) \to a$$

then we can define nil and cons as follows:

$$\begin{aligned}\mathsf{nil} &: \mathsf{List}\\ \mathsf{nil} &= \mathsf{fold}[\mathsf{List}]\,(\lambda(a : \star)(z : a)(f : \Pi(b : \star).\, b \to \mathsf{List} \to a).\, z)\\ \mathsf{cons} &: \Pi(b : \star).\, b \to \mathsf{List} \to \mathsf{List}\\ \mathsf{cons} &= \lambda(b : \star)(x : b)(xs : \mathsf{List}).\\ &\qquad \mathsf{fold}[\mathsf{List}]\,(\lambda(a : \star)(z : a)(f : \Pi(b : \star).\, b \to \mathsf{List} \to a).\, f\, b\, x\, xs)\end{aligned}$$

Using fix, we can define a recursive function length as follow:

$$\begin{aligned}
&\mathsf{length} : \mathsf{List} \to \mathsf{Nat} \\
&\mathsf{length} = \mathsf{fix}\,(\mathsf{List} \to \mathsf{Nat})\,(\lambda(l : \mathsf{List} \to \mathsf{Nat})(xs : \mathsf{List}). \\
&\qquad\qquad (\mathsf{unfold}[\mathsf{List}]\, xs)\, \mathsf{Nat}\, \mathsf{zero}\, (\lambda(b : \star)(y : b)(ys : \mathsf{List}).\, \mathsf{suc}\, (l\, ys)))
\end{aligned}$$

3.2.2 Elaboration of Datatypes

We can extend λC_μ with *first-order* datatypes [1]:

$$\mathbf{data}\quad D = K_1\, T_1^1(D) \dots T_{\mathsf{ar}(1)}^1(D) \mid \cdots \mid K_n\, T_1^n(D) \dots T_{\mathsf{ar}(n)}^n(D)$$

where each of the $T_i^j(X)$ is either X or a type expression that does not contain X. This defines an algebraic datatype D with n constructors. Each constructor K_i has arity $\mathsf{ar}(i)$, which can be zero.

We adopt the following convention: we write $T^1(X)$ for $T_1^1(X) \dots T_{\mathsf{ar}(1)}^1(X)$ etc. So each data constructor has the following types:

$$\begin{array}{lcl}
K_1 & : & T^1(D) \to D \\
 & & \dots \\
K_n & : & T^n(D) \to D
\end{array}$$

Next we show how datatypes can be translated to our system with recursive types.

Given a datatype D, with constructors $K_1, \dots, K_n$, the encoding of D in our system is given by:

$$D ::= \mu\beta.\, \Pi(\alpha : \star).\, (T^1(\beta) \to \alpha) \to \cdots \to (T^n(\beta) \to \alpha) \to \alpha$$

The constructors are encoded by:

$$\begin{aligned}
K_i ::= {} & \lambda(x_1 : T_1^i(D)) \dots (x_{\mathsf{ar}(i)} : T_{\mathsf{ar}(i)}^i(D)). \\
& \mathsf{fold}[D]\, (\lambda(\alpha : \star)(c_1 : T^1(D) \to \alpha) \dots (c_n : T^n(D) \to \alpha).\, c_i\, x_1 \dots x_{\mathsf{ar}(i)})
\end{aligned}$$

3.2.3 Elaboration of Case Analysis

The set of expressions A of λC_μ extended with case analysis is defined by

$$\begin{array}{lcl}
A & ::= & x \mid \star \mid \Box \\
 & | & A A \mid \lambda x : A.A \mid \Pi x : A.A \\
 & | & \mu x.A \mid \mathsf{fold}[A]\, A \mid \mathsf{unfold}[A]\, A \\
 & | & \mathsf{beta}\, A \\
 & | & \mathsf{case}\, A\, \mathsf{of}\, \{x\, x_1\, x_2 \cdots \Rightarrow A; \dots\}
\end{array}$$

Suppose we have

$$\begin{aligned}
&\mathsf{case}\, x\, \mathsf{of}\, \{ \\
&\quad K_1\, x_1 \ldots x_{\mathsf{ar}(1)} \Rightarrow r_1 \\
&\quad \ldots \\
&\quad K_n\, x_1 \ldots x_{\mathsf{ar}(n)} \Rightarrow r_n \\
&\quad \}
\end{aligned}$$

where $x : D$ and $r_1, \ldots, r_n : T$ (T is some known type).

This can be translated to our system as follows:

$$\begin{aligned}
(\mathsf{unfold}[D]\, x)\, T\, &(\lambda(x_1 : T_1^1(D)) \ldots (x_{\mathsf{ar}(1)} : T_{\mathsf{ar}(1)}^1(D)).\, r_1) \\
&\ldots \\
&(\lambda(x_1 : T_1^n(D)) \ldots (x_{\mathsf{ar}(n)} : T_{\mathsf{ar}(n)}^n(D)).\, r_n)
\end{aligned}$$

3.3 Proof of soundness

Lemma 3.3.1 (λC_μ Substitutions)
Assume we have

$$\Gamma, x : A \vdash B : C \tag{1}$$
$$\Gamma \vdash D : A, \tag{2}$$

then

$$\Gamma[x := D] \vdash B[x := D] : C[x := D].$$

Proof. This is trivial by induction on the typing derivation of (1). We only discuss two cases for example. Let E^* denote $E[x := D]$. Consider following cases

- The last applied rule to obtain (1) is *Var*. There are 2 sub-cases:
  1. It is derived by
     $$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A},$$
     then we have $(B : C) \equiv (x : A)$. And $\Gamma \vdash (x : A)^* \equiv (D : A)$ which holds by (2).
  2. It is derived by
     $$\frac{\Gamma, x : A \vdash E : s}{\Gamma, x : A, y : E \vdash y : E},$$
     then we need to show $\Gamma^*, y : E^* \vdash y : E^*$. And it directly follows the induction hypothesis, i.e. $\Gamma^* \vdash E^* : s$.
- The last applied rule to obtain (1) is *App*, i.e.
  $$\frac{\Gamma, x : A \vdash B_1 : (\Pi y : C_1.\, C_2) \qquad \Gamma, x : A \vdash B_2 : C_1}{\Gamma, x : A \vdash (B_1 B_2) : C_2[y := B_2]}.$$
  By the induction hypothesis, we can obtain $\Gamma^* \vdash B_1^* : (\Pi y : C_1^*.C_2^*)$ and $\Gamma^* \vdash B_2^* : C_1^*$. Thus, $\Gamma^* \vdash (B_1^* B_2^*) : (C_2^*[y := B_2^*])$, i.e. $\Gamma^* \vdash (B_1 B_2)^* : (C_2[y := B_2])^*$.

$\square$

Theorem 3.3.2 (λC_μ Subject Reduction)
If $\Gamma \vdash A : B$ *and* $A \twoheadrightarrow_\beta A'$ *then* $\Gamma \vdash A' : B$.

Proof. Let $\mathcal{D}$ be the derivation of $\Gamma \vdash A : B$. The proof is by induction on the derivation of $A \twoheadrightarrow_\beta A'$.

case *App*: $(\lambda x : A.M)N \to_\beta M[x := N]$.

Derivation $\mathcal{D}$ has the following form

$$\dfrac{\dfrac{\Gamma, x : A \vdash M : A'}{\Gamma \vdash (\lambda x : A.M) : (\Pi x : A.A')}\,\textit{Lam} \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x : A.M)N : A'}\,\textit{App}$$

Thus, by Lemma 3.3.1 we can obtain $\Gamma \vdash M[x := N] : A'$.

case *Lam*: $\dfrac{M \twoheadrightarrow_\beta M'}{\lambda x : A.M \twoheadrightarrow_\beta \lambda x : A.M'}$.

Derivation $\mathcal{D}$ has the following form

$$\dfrac{\Gamma, x : A \vdash M : A'}{\Gamma \vdash (\lambda x : A.M) : (\Pi x : A.A')}\,\textit{Lam}$$

By the induction hypothesis we have $\Gamma, x : A \vdash M' : A'$. Hence,

$$\dfrac{\Gamma, x : A \vdash M' : A'}{\Gamma \vdash (\lambda x : A.M') : (\Pi x : A.A')}\,\textit{Lam}$$

case *App (Left)*: $\dfrac{M \twoheadrightarrow_\beta M'}{MN \twoheadrightarrow_\beta M'N}$.

Derivation $\mathcal{D}$ has the following form

$$\dfrac{\Gamma \vdash M : (\Pi x : A.A') \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : A'}\,\textit{App}$$

By the induction hypothesis we have $\Gamma \vdash M' : (\Pi x : A.A')$. Hence,

$$\dfrac{\Gamma \vdash M' : (\Pi x : A.A') \qquad \Gamma \vdash N : A}{\Gamma \vdash M'N : A'}\,\textit{App}$$

case *App (Right)*: $\dfrac{M \twoheadrightarrow_\beta M'}{vM \twoheadrightarrow_\beta vM'}$.

Derivation $\mathcal{D}$ has the following form

$$\dfrac{\Gamma \vdash v : (\Pi x : A.A') \qquad \Gamma \vdash M : A}{\Gamma \vdash vM : A'}\,\textit{App}$$

By the induction hypothesis we have $\Gamma \vdash M' : A$. Hence,

$$\dfrac{\Gamma \vdash v : (\Pi x : A.A') \qquad \Gamma \vdash M' : A}{\Gamma \vdash vM' : A'}\,\textit{App}$$

case *Fold*: $\dfrac{M \twoheadrightarrow_\beta M'}{\mathsf{fold}[N]\,M \twoheadrightarrow_\beta \mathsf{fold}[N]\,M'}$, where $N = \mu x.A$.

Derivation $\mathcal{D}$ has the following form

$$\dfrac{\Gamma \vdash M : (A[x := \mu x.A])}{\Gamma \vdash (\mathsf{fold}[\mu x.A]\,M) : \mu x.A}\,\textit{Fold}$$

By the induction hypothesis we have $\Gamma \vdash M' : (A[x := \mu x.A])$. Hence,

$$\dfrac{\Gamma \vdash M' : (A[x := \mu x.A])}{\Gamma \vdash (\mathsf{fold}[\mu x.A]\,M') : \mu x.A}\,\textit{Fold}$$

case *Unfold*: $\dfrac{M \twoheadrightarrow_\beta M'}{\mathsf{unfold}[N]\, M \twoheadrightarrow_\beta \mathsf{unfold}[N]\, M'}$, where $N = \mu x.A$.

Derivation $\mathcal{D}$ has the following form

$$\dfrac{\Gamma \vdash M : \mu x.A}{\Gamma \vdash (\mathsf{unfold}[\mu x.A]\, M) : A[x := \mu x.A]}\ \textit{Unfold}$$

By the induction hypothesis we have $\Gamma \vdash M' : \mu x.A$. Hence,

$$\dfrac{\Gamma \vdash M' : \mu x.A}{\Gamma \vdash (\mathsf{unfold}[\mu x.A]\, M') : A[x := \mu x.A]}\ \textit{Unfold}$$

case *Unfold-Fold*: $\mathsf{unfold}[N]\,(\mathsf{fold}[N]\, M) \to M$, where $N = \mu x.A$.

Derivation $\mathcal{D}$ has the following form

$$\dfrac{\dfrac{\Gamma \vdash M : (A[x := \mu x.A])}{\Gamma \vdash (\mathsf{fold}[N]\, M) : \mu x.A}\ \textit{Fold}}{\Gamma \vdash \mathsf{unfold}[N]\,(\mathsf{fold}[N]\, M) : (A[x := \mu x.A])}\ \textit{Unfold}$$

which immediately proves the statement.

case *Beta*: $\dfrac{M \twoheadrightarrow_\beta M'}{\mathsf{beta}\, M \twoheadrightarrow_\beta \mathsf{beta}\, M'}$.

Derivation $\mathcal{D}$ has the following form

$$\dfrac{\Gamma \vdash M : A \qquad A \rightarrow_\beta B}{\Gamma \vdash (\mathsf{beta}\, M) : B}\ \textit{Beta}$$

By the induction hypothesis we have $\Gamma \vdash M' : A$. Hence,

$$\dfrac{\Gamma \vdash M' : A \qquad A \rightarrow_\beta B}{\Gamma \vdash (\mathsf{beta}\, M') : B}\ \textit{Beta}$$

□

Theorem 3.3.3 (λC_μ Progress)
If $\cdot \vdash A : B$ then either A is a value v or $A \twoheadrightarrow_\beta A'$.

Proof. Note that expressions with following forms can be *values* if not able to be reduced any more.

$$\begin{array}{rcl} v & ::= & x \mid \star \mid \lambda x : A.A \mid \mathsf{beta}\, A \\ & \mid & \mathsf{fold}[A]\, A \mid \mathsf{unfold}[A]\, A \end{array}$$

We can give the proof by induction on the derivation of $\cdot \vdash A : B$ as follows

case *Var*: $\dfrac{}{\cdot, x : A \vdash x : A}$.

The proof is given by contraction. If x is not a value and there exists no x' such that $x \twoheadrightarrow_\beta x'$, then x is in normal form, which belongs to one of the value forms listed above. This contradicts that x is not a value. Thus, the original statement holds.

case *Weak*: $\dfrac{\cdot \vdash b : B}{\cdot, x : A \vdash b : B}$.

The result is trivial by induction hypothesis.

case *App*: $\dfrac{\cdot \vdash M : (\Pi x : A.B) \qquad \cdot \vdash N : A}{\cdot \vdash MN : B}$.

By induction hypothesis on $\cdot \vdash M : (\Pi x : A.B)$, there are two possible cases.

1. $M = v$ is a value. Hence $v = \lambda x : A.M'$ where $\cdot \vdash M' : B$. Then $MN = vN = (\lambda x : A.M')N = M'[x := N]$. By the substitution lemma, $\cdot \vdash (M'[x := N]) : B$ which is just $\cdot \vdash MN : B$.
2. $M \twoheadrightarrow_\beta M'$. The result is obvious by the operational semantic $\dfrac{M \twoheadrightarrow_\beta M'}{MN \twoheadrightarrow_\beta M'N}$ *App -Left* .

case *Lam*: $\dfrac{\ldots}{\cdot \vdash (\lambda x : A.M) : (\Pi x : A.B)}$.

The result is trivial if let $v = \lambda x : A.M$.

case *Fold*: $\dfrac{\ldots}{\cdot \vdash (\mathsf{fold}[\mu x.A]\, M) : \mu x.A}$.

The result is trivial if let $v = \mathsf{fold}[\mu x.A]\, M$.

case *Unfold*: $\dfrac{\ldots}{\cdot \vdash (\mathsf{unfold}[\mu x.A]\, M) : A[x := \mu x.A]}$.

The result is trivial if let $v = \mathsf{unfold}[\mu x.A]\, M$.

case *Beta*: $\dfrac{\ldots}{\cdot \vdash (\mathsf{beta}\, M) : B}$.

The result is trivial if let $v = \mathsf{beta}\, M$.

□

References

[1] Herman Geuvers. The church-scott representation of inductive and coinductive data. Types, 2014.

[2] Simon Peyton Jones and Erik Meijer. Henk: a typed intermediate language. *TIC*, 97, 1997.

[3] J-W Roorda and JT Jeuring. Pure type systems for functional programming. 2007.

[4] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier, 2006.

A. Appendix

(Ax)	$\dfrac{}{\vdash s : t}$	$(s,t) \in \mathcal{A}$
(Var)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$	$x \notin \mathrm{dom}(\Gamma)$
(Weak)	$\dfrac{\Gamma \vdash b : B \qquad \Gamma \vdash A : s}{\Gamma, x : A \vdash b : B}$	$x \notin \mathrm{dom}(\Gamma)$
(App)	$\dfrac{\Gamma \vdash f : (\Pi x : A.\ B) \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B[x := a]}$	
(Lam)	$\dfrac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.\ B) : t}{\Gamma \vdash (\lambda x : A.\ b) : (\Pi x : A.\ B)}$	
(Pi)	$\dfrac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : t}{\Gamma \vdash (\Pi x : A.\ B) : u}$	$(s,t,u) \in \mathcal{R}$
(Conv)	$\dfrac{\Gamma \vdash a : A \qquad \Gamma \vdash B : s \qquad A =_\beta B}{\Gamma \vdash a : B}$	

Table 3. Typing rules for a PTS