Formalization of Pure Type Systems

Last modified: March 13, 2015 at 4:53pm

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*.

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

$$\begin{aligned} A &::= x \mid s \mid AA \mid \lambda x : A.\ A \mid \Pi x : A.\ A \\ \Gamma &::= \varnothing \mid \Gamma, x : A \end{aligned}$$

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 1. 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 2.

(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 1. Typing rules for a PTS

(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)	$\dfrac{\Gamma \vdash f : (\Pi x : A.\, B) \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B[x := a]}$	
(Lam)	$\dfrac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.\, B) : t}{\Gamma \vdash (\lambda x : A.\, b) : (\Pi x : A.\, B)}$	$t \in \{\star, \Box\}$
(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 2. 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:

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

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

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

3. Extending PTSs

This section investigates how to extend PTSs to have algebraic datatypes, case expressions, etc.

3.1 Algebraic Datatypes

An algebraic datatype has the form:

$$T\ u_1 \dots u_k = K_1\ \alpha_{11} \dots \alpha_{1k_1} \mid \cdots \mid K_n\ \alpha_{n1} \dots \alpha_{nk_n}$$

where T denotes a new *type constructor* with zero or more constituent *data constructors* $K_1, \dots, K_n$. We call $u_1, \dots, u_k$ the arguments of the type constructor T, and $t_{j1}, \dots, t_{jk_j}$ the arguments of the $K_j (1 \leqslant j \leqslant n)$ data constructor, whose types are $\alpha_{j1}, \dots, \alpha_{jk_j}$ respectively. Each u_i is a variable of *sort type*, each α_{jk} is an expression of *kind type* (i.e., $\alpha_{jk} : \star$), which may contain T and $u_1, \dots, u_k$ as free variables.

We use the following notation: $\vec{\mathbf{u}} = [u_1, \dots, u_k]$, $\vec{\mathbf{t}}_j = [t_{j1}, \dots t_{jk_j}]$, etc. If $\vec{\mathbf{a}} = [a_1, \dots, a_n]$ and $\vec{\mathbf{A}} = [A_1, \dots, A_n]$, then $\Pi\vec{\mathbf{a}} : \vec{\mathbf{A}}.\ B$ denotes $\Pi a_1 : A_1 \dots \Pi a_n : A_n.\ B$. Let $\tau_1, \dots, \tau_k$ be the types of $u_1, \dots, u_k$.

A PTS with ADTs is a tuple $(P, ADTS)$ where:

(i) P is a Pure Type System, let V, E be the sets of variables and expressions of P.

(ii) $ADTS$ is a set of ADTs, each consisting of $[T : T', K_1 : K'_1, \dots, K'_n]$ such that:

- $T, K_j \in V$ and $T', K'_j \in E$, for every $1 \leqslant j \leqslant n$
- $T' = \Pi\vec{\mathbf{u}} : \vec{\tau}.\ \star$
- $K'_j = \Pi\vec{\mathbf{u}} : \vec{\tau}.\ \Pi\vec{\mathbf{t}}_j : \vec{\alpha}_j.\ (T\ \vec{\mathbf{u}})$, for every $1 \leqslant j \leqslant n$
- $T : T' \vdash K_j : K'_j : \star$

Note that the use of the dependent product (Π) makes it possible to let the types of the data constructor arguments depend on other data constructor arguments.

(iii) (**Typability in a PTS with ADTs**) let $\Sigma = [c : ct \mid c : ct \leftarrow ADT, ADT \leftarrow ADTS]$, we say that $\Gamma \vdash_{(P,ADTS)} a : A$ if and only if $\Sigma \uplus \Gamma \vdash a : A$

3.1.1 An Example of a PTS with ADTs

Let $P = \lambda C$ and

$$\begin{aligned}
\Sigma_a &= [Int : \star, Zero : Int, Suc : Int \to Int,\\
&\qquad Bool : \star, True : Bool, False : Bool]\\
\Sigma_b &= [Vec : (\Pi n : Int.\, \Pi \alpha : \star.\, \star),\\
&\qquad Nil : (\Pi \alpha : \star.\, Vec\, Zero\, \alpha),\\
&\qquad Cons : (\Pi n : Int.\, \Pi \alpha : \star.\, \alpha \to Vec\, n\, \alpha \to Vec\, (Suc\, n)\, \alpha)]\\
ADTS &= [\Sigma_a, \Sigma_b]
\end{aligned}$$

then we can derive $\vdash_{(P,ADTS)}$ Cons Zero Bool True (Nil Bool) : Vec (Suc Zero) Bool

3.2 Case Expressions

3.2.1 Definition

The set of expressions E_c of a PTS extended with ADTs and case expressions is defined by

$$\begin{aligned}
A_c ::= x \mid s \mid A_c A_c \mid \lambda x : A_c.\, A_c \mid \Pi x : A_c.\, A_c\\
\mid case\ A_c\ of\ \{x_1\, x_2 \cdots \Rightarrow A_c; \dots\}
\end{aligned}$$

3.2.2 Evaluation Relation

Let $(P, ATDS)$ be a PTS extended with ADTs. The evaluation relation $\to_c$ is the smallest binary relation on A_c such that

$$\text{case } (K_j\ \vec{\mathbf{x}}) \text{ of } \{K_j\ \vec{\mathbf{u}}\ \vec{\mathbf{t}}_j \Rightarrow res_j\} \to_c (\lambda \vec{\mathbf{u}} : \vec{\tau},\ \vec{\mathbf{t}}_j : \vec{\alpha}_j.\, res_j)\ \vec{\mathbf{x}}$$

and closed under usual rules.

The evaluation relation $\to_{\beta c}$ is defined by

$$\to_{\beta c} = \to_\beta \cup \to_c$$

3.2.3 Typing Relation

The extended typing relation $\vdash_c$ is shown below

$$\text{(Case)}\quad \frac{\begin{array}{c}\Gamma \vdash_c e : T\ \vec{\mathbf{u}'}\\ \forall j. \Gamma \vdash_c K_j\ \vec{\mathbf{u}'} : \Pi \vec{\mathbf{t}}_j : \vec{\alpha}_j.\, (T\ \vec{\mathbf{u}'})\\ \forall j. \Gamma, \vec{\mathbf{t}}_j : \vec{\alpha}_j \vdash_c res_j[\vec{\mathbf{u}} := \vec{\mathbf{u}'}] : t\\ \Gamma \vdash_c t : s\end{array}}{\Gamma \vdash_c \text{case } e \text{ of } \{K_j\ \vec{\mathbf{u}}\ \vec{\mathbf{t}}_j \Rightarrow res_j\} : t}$$

$$\text{(Conv)}\quad \frac{\Gamma \vdash_c a : A \qquad \Gamma \vdash_c B : s \qquad A =_{\beta c} B}{\Gamma \vdash_c a : B}$$

the $(Conv)$ rule is extended with reduction of case expressions.

In the $(Case)$ rule, the first premise binds the actual type constructor arguments to $\vec{\mathbf{u}'}$. The second premise derives the types of the data constructor arguments $\vec{\mathbf{t}}_j$ and binds them to $\vec{\alpha}_j$, using the data constructor K_j and the actual type constructor arguments. The third premise checks whether the types of the right hand sides, instantiated to the actual type constructor arguments, are equal, and if so the result type is bound to t. Finally, the forth premise checks whether the derived type t is well formed.

3.3 Recursive types

3.3.1 Definition

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

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

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 3. 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)	$\dfrac{\Gamma \vdash f : (\Pi x : A.\, B) \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B[x := a]}$	
(Lam)	$\dfrac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.\, B) : t}{\Gamma \vdash (\lambda x : A.\, b) : (\Pi x : A.\, B)}$	$t \in \{\star, \square\}$
(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 3. Typing rules for λC_μ

3.3.2 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 $\mathsf{fix} : (\Pi\alpha : \star.(\alpha \to \alpha) \to \alpha)$ 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.

References

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

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

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