Calculus of Constructions with Recursive Types

Last modified: April 14, 2015 at 12:18am

1. Calculus of Constructions

Our language is based on the *Calculus of Constructions*, a special case of the *Pure Type System*. We give the definition as follows:

(i) A *Calculus of Constructions* (λC) is a triple tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where

  (a) $\mathcal{S} = \{\star, \Box\}$ is a set of *sorts*;

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

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

(ii) *Raw expressions* A and *raw environments* Γ are defined in Figure 1.

$$
\begin{array}{rcll}
A & ::= & x & \text{(variable)} \\
 & | & \star & \text{(star)} \\
 & | & \Box & \text{(square)} \\
 & | & A\ A & \text{(application)} \\
 & | & \lambda x : A.A & \text{(abstraction)} \\
 & | & \Pi x : A.A & \text{(product)} \\
\Gamma & ::= & \varnothing & \text{(empty)} \\
 & | & \Gamma, x : A & \text{(variable binding)}
\end{array}
$$

Figure 1. Syntax of λC

We use s, t to range over *sorts*, x, y, z to range over *variables*, and A, B, C, a, b, c 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. We use $A \to B$ as a syntactic sugar for $(\Pi\_ : A.\ B)$.

(iv) The β-reduction ($\to_\beta$) 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. Reduction rules are given in Figure 2. Highlighted premises and rules are only for call-by-value evaluation.

(v) Type assignment rules for $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ are given in Figure 3.

Values: v ::= $\lambda x : A.B$

(R-Beta) $$\frac{\boxed{N \in \textit{Value}}}{(\lambda x : A.M)N \longrightarrow M[x := N]}$$

(R-AppL) $$\frac{M \longrightarrow M'}{MN \longrightarrow M'N}$$

(R-AppR) $$\boxed{\frac{v \in \textit{Value} \qquad M \longrightarrow M'}{vM \longrightarrow vM'}}$$

Figure 2. Reduction rules for λC

(Ax) $$\frac{}{\varnothing \vdash \star : \square}$$

(Var) $$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \qquad x \notin \mathrm{dom}(\Gamma)$$

(Weak) $$\frac{\Gamma \vdash b : B \qquad \Gamma \vdash A : s}{\Gamma, x : A \vdash b : B} \qquad x \notin \mathrm{dom}(\Gamma)$$

(App) $$\frac{\Gamma \vdash f : (\Pi x : A.B) \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B[x := a]}$$

(Lam) $$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.B) : t}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)} \qquad t \in \{\star, \square\}$$

(Pi) $$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : t}{\Gamma \vdash (\Pi x : A.B) : t} \qquad (s, t) \in \mathcal{R}$$

(Conv) $$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash B : s \qquad A =_\beta B}{\Gamma \vdash a : B}$$

Figure 3. Typing rules for λC

2. Extend with recursive types

2.1 Core language

We extend Calculus of Constructions (λC) with recursive types, namely λC_μ. Differences with λC are highlighted. Figure 4 shows the extended syntax.

Since recursive types are introduced and due to the practical concern, we use the *call-by-name* reduction strategy, i.e. iteratively reducing the *left-most outer-most* redex. Figure 5 shows the dynamic semantics with no call-by-value specific premises or rules.

The extended typing rules are shown in Figure 6. Compared with λC, the original *Conv* rule is replaced by the new *Beta* rule where the latter only performs one step of reduction defined in Fig.5.

2.2 Soundness of core language

Lemma 2.2.1 (Substitutions)
Assume we have

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

$$\Gamma \vdash D : A, \tag{2}$$

$$
\begin{array}{llll}
A & ::= & x & \text{(variable)} \\
& | & \star & \text{(star)} \\
& | & \square & \text{(square)} \\
& | & A\ A & \text{(application)} \\
& | & \lambda x : A.A & \text{(abstraction)} \\
& | & \Pi x : A.A & \text{(product)} \\
& | & \mu x.A & \text{(recursive type)} \\
& | & \mathsf{fold}[\mu x.A]\ A & \text{(roll)} \\
& | & \mathsf{unfold}\ A & \text{(unroll)} \\
& | & \mathsf{beta}\ A & \text{(type reduction)} \\
\Gamma & ::= & \varnothing & \text{(empty)} \\
& | & \Gamma, x : A & \text{(variable binding)}
\end{array}
$$

Figure 4. Syntax of λC_μ

$$
\begin{array}{lllll}
\textbf{values:} & v & ::= & \lambda x : A.B & \text{(abstraction)} \\
& & | & \mu x.A & \text{(recursive type)} \\
& & | & \mathsf{fold}[\mu x.A]\ B & \text{(roll)} \\
& & | & \mathsf{beta}\ A & \text{(type reduction)}
\end{array}
$$

(R-Beta) $$\frac{}{(\lambda x : A.M)N \longrightarrow M[x := N]}$$

(R-AppL) $$\frac{M \longrightarrow M'}{MN \longrightarrow M'N}$$

(R-Unfold) $$\frac{M \longrightarrow M'}{\mathsf{unfold}\ M \longrightarrow \mathsf{unfold}\ M'}$$

(R-Unfold-Fold) $$\frac{}{\mathsf{unfold}\ (\mathsf{fold}[\mu x.A]\ M) \longrightarrow M}$$

Figure 5. Reduction rules for λC

then

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

Proof. This is trivial by induction on the typing derivation of (1) by typing rules in Fig.6. 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.
$$\frac{\Gamma, x : A \vdash B_1 : (\Pi y : C_1.\ C_2) \qquad \Gamma, x : A \vdash B_2 : C_1}{\Gamma, x : A \vdash (B_1 B_2) : C_2[y := B_2]}.$$

$$\text{(Ax)} \quad \frac{}{\varnothing \vdash \star : \Box}$$

$$\text{(Var)} \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad x \notin \mathrm{dom}(\Gamma)$$

$$\text{(Weak)} \quad \frac{\Gamma \vdash b : B \qquad \Gamma \vdash A : s}{\Gamma, x : A \vdash b : B} \quad x \notin \mathrm{dom}(\Gamma)$$

$$\text{(App)} \quad \frac{\Gamma \vdash f : (\Pi x : A.B) \qquad \Gamma \vdash a : A}{\Gamma \vdash f a : B[x := a]}$$

$$\text{(Lam)} \quad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash (\Pi x : A.B) : t}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)} \quad t \in \{\star, \Box\}$$

$$\text{(Pi)} \quad \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : t}{\Gamma \vdash (\Pi x : A.B) : t} \quad (s, t) \in \mathcal{R}$$

$$\text{(Mu)} \quad \frac{\Gamma, x : s \vdash A : s}{\Gamma \vdash (\mu x.A) : s}$$

$$\text{(Fold)} \quad \frac{\Gamma \vdash a : (A[x := \mu x.A]) \qquad \Gamma \vdash \mu x.A : s}{\Gamma \vdash (\mathsf{fold}[\mu x.A]\, a) : \mu x.A}$$

$$\text{(Unfold)} \quad \frac{\Gamma \vdash a : \mu x.A \qquad \Gamma \vdash A[x := \mu x.A] : s}{\Gamma \vdash (\mathsf{unfold}\, a) : A[x := \mu x.A]}$$

$$\text{(Beta)} \quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash B : s \qquad A \longrightarrow B}{\Gamma \vdash (\mathsf{beta}\, a) : B}$$

Figure 6. Typing rules for λC_μ

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. $\Gamma^* \vdash (B_1 B_2)^* : (C_2[y := B_2])^*$.

□

Theorem 2.2.2 (Subject Reduction)
If $\Gamma \vdash A : B$ *and* $A \longrightarrow A'$ *then* $\Gamma \vdash A' : B$.

Proof. Let $\mathcal{D}$ be the derivation of $\Gamma \vdash A : B$. The proof is by induction on dynamic semantics shown in Fig.5.

case *R-Beta*: $\dfrac{}{(\lambda x : A.M)N \longrightarrow M[x := N]}$.

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

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

Thus, by Lemma 2.2.1 we can obtain $\Gamma \vdash M[x := N] : A'$.

case *R-AppL*: $\dfrac{M \longrightarrow M'}{MN \longrightarrow M'N}$.

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

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

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

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

case *R-Unfold*: $\dfrac{M \longrightarrow M'}{\mathsf{unfold}\, M \longrightarrow \mathsf{unfold}\, M'}$.

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

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

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

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

case *R-Unfold-Fold*: $\overline{\mathsf{unfold}\, (\mathsf{fold}[\mu x.A]\, M) \longrightarrow M}$.

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

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

which immediately proves the statement.

□

Theorem 2.2.3 (Progress)
If $\cdot \vdash A : B$ then either A is a value v or there exists A' such that $A \longrightarrow A'$.

Proof. We can give the proof by induction on the derivation of $\cdot \vdash A : B$ by typing rules in Fig.6:

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

This case cannot be reached. Proof is by contradiction. If we have $\cdot \vdash x : A$ then x is assigned with type A from a context "$\cdot$" without A, which is not possible.

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

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. $M \longrightarrow M'$. The result is obvious by the operational semantic $\dfrac{M \longrightarrow M'}{MN \longrightarrow M'N}$ *R-AppL* .

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 *Pi*: $\dfrac{\cdot \vdash A : s \qquad \cdot, x : A \vdash B : t}{\cdot \vdash (\Pi x : A.B) : t}$.

This case cannot be reached. Proof is by contradiction. If we have $\cdot \vdash (\Pi x : A.B) : t$, then we can assign type t from a context "$\cdot$" that doesn't have t, which is not possible.

case *Mu*: $\dfrac{\ldots}{\cdot \vdash (\mu x.A) : s}$.

The result is trivial if let $v = \mu x.A$.

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*: $\dfrac{\cdot \vdash a : \mu x.A \qquad \cdot \vdash A[x := \mu x.A] : s}{\cdot \vdash (\mathsf{unfold}\, a) : A[x := \mu x.A]}$.

By induction hypothesis on $\cdot \vdash a : \mu x.A$, there are two possible cases.

1. $a = v$ is a value. Hence $a = \mathsf{fold}[\mu x.A]\, b$ where $\cdot \vdash b : (A[x := \mu x.A])$. Then by the *R-Unfold-Fold* rule, $\mathsf{unfold}\, a = \mathsf{unfold}\, (\mathsf{fold}[\mu x.A]\, b) = b$. Thus $\cdot \vdash (\mathsf{unfold}\, a) : A[x := \mu x.A]$.
2. $a \longrightarrow a'$. The result is obvious by the reduction rule $\dfrac{M \longrightarrow M'}{\mathsf{unfold}\, M \longrightarrow \mathsf{unfold}\, M'}$ *R-Unfold* .

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

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

□

2.3 Examples of typable terms

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

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

  Note that this is the so called call-by-name fixed point combinator. It is useless in a call-by-value setting, since the expression $\mathsf{fix}\, \alpha\, g$ diverges for any g.

- 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. Extend with data types

3.1 Encoding of data types

3.1.1 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:

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

- 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:

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

3.1.2 Elaboration of Datatypes

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

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

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 $T_1^1(X) \ldots T_{\mathsf{ar}(1)}^1(X)$ etc. So each data constructor has the following types:

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

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

Given a datatype D, with constructors $K_1, \ldots, K_n$, 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:

$$K_i ::= \lambda(x_1 : T^i_1(D)) \dots (x_{\mathsf{ar}(i)} : T^i_{\mathsf{ar}(i)}(D)).$$
$$\mathsf{fold}[D]\,(\lambda(\alpha : \star)(c_1 : T^1(D) \to \alpha) \dots (c_n : T^n(D) \to \alpha).\, c_i\, x_1 \dots x_{\mathsf{ar}(i)})$$

3.1.3 Elaboration of Case Analysis

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

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

Suppose we have

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

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

This can be translated to our system as follows:

$$\begin{array}{l} (\mathsf{unfold}\,x)\,T\,(\lambda(x_1 : T^1_1(D)) \dots (x_{\mathsf{ar}(1)} : T^1_{\mathsf{ar}(1)}(D)).\, r_1) \\ \qquad \dots \\ \qquad (\lambda(x_1 : T^n_1(D)) \dots (x_{\mathsf{ar}(n)} : T^n_{\mathsf{ar}(n)}(D)).\, r_n) \end{array}$$

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