

Calculus of Constructions with Recursive Types

Last modified: April 14, 2015 at 5:15pm

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) $\mathcal{S} = \{\star, \square\}$ is a set of *sorts*;
 - (b) $\mathcal{A} = \{(\star, \square)\} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of *axioms*;
 - (c) $\mathcal{R} = \{(\star, \star), (\star, \square), (\square, \star), (\square, \square)\} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of *rules*.
- (ii) *Raw expressions* A and *raw environments* Γ are defined in Figure 1.

A	$::=$	x	(variable)
		\star	(star)
		\square	(square)
		$A A$	(application)
		$\lambda x : A. A$	(abstraction)
		$\Pi x : A. A$	(product)
Γ	$::=$	\emptyset	(empty)
		$\Gamma, x : A$	(variable binding)

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 $FV(A)$ denote free variable set of A . Let $A[x := B]$ denote the substitution of x in A with B . We use $A \rightarrow B$ as a syntactic sugar for $(\Pi_+ : A. B)$.
- (iv) The β -reduction (\rightarrow_β) is the smallest binary relation on raw expressions satisfying

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

which can be used to define the notation \rightarrow_β 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: $v ::=$	$\lambda x : A.B \mid \Pi x : A.B$
(R-Beta)	$\frac{N \in \text{Value}}{(\lambda x : A.M)N \longrightarrow M[x := N]}$
(R-AppL)	$\frac{M \longrightarrow M'}{MN \longrightarrow M'N}$
(R-AppR)	$\frac{v \in \text{Value} \quad M \longrightarrow M'}{vM \longrightarrow vM'}$

Figure 2. Reduction rules for λC

(Ax)	$\frac{}{\emptyset \vdash \star : \square}$	
(Var)	$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$	$x \notin \text{dom}(\Gamma)$
(Weak)	$\frac{\Gamma \vdash b : B \quad \Gamma \vdash A : s}{\Gamma, x : A \vdash b : B}$	$x \notin \text{dom}(\Gamma)$
(App)	$\frac{\Gamma \vdash f : (\Pi x : A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash fa : B[x := a]}$	
(Lam)	$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A.B) : t}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)}$	$t \in \{\star, \square\}$
(Pi)	$\frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : t}{\Gamma \vdash (\Pi x : A.B) : t}$	$(s, t) \in \mathcal{R}$
(Conv)	$\frac{\Gamma \vdash a : A \quad \Gamma \vdash B : s \quad A =_{\beta} B}{\Gamma \vdash a : B}$	

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.

$A ::=$	x	(variable)
	\star	(star)
	\square	(square)
	$A A$	(application)
	$\lambda x : A.A$	(abstraction)
	$\Pi x : A.A$	(product)
	$\mu x.A$	(recursive type)
	$\text{fold}[\mu x.A] A$	(roll)
	$\text{unfold } A$	(unroll)
	$\text{beta } A$	(type reduction)
$\Gamma ::=$	\emptyset	(empty)
	$\Gamma, x : A$	(variable binding)

Figure 4. Syntax of λC_{μ}

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.

values:	$v ::= \lambda x : A.B$	(abstraction)
	$ \Pi x : A.B$	(product)
	$ \text{fold}[\mu x.A] B$	(roll)
(R-AppLam)	$\frac{}{(\lambda x : A.M)N \longrightarrow M[x := N]}$	
(R-AppL)	$\frac{M \longrightarrow M'}{MN \longrightarrow M'N}$	
(R-Unfold)	$\frac{M \longrightarrow M'}{\text{unfold } M \longrightarrow \text{unfold } M'}$	
(R-Unfold-Fold)	$\frac{}{\text{unfold } (\text{fold}[\mu x.A] M) \longrightarrow M}$	
(R-Mu)	$\frac{}{\mu x.M \longrightarrow M[x := \mu x.M]}$	
(R-Beta)	$\frac{}{\text{beta } M \longrightarrow M}$	

Figure 5. Reduction rules for λC

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

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

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

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).

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

Figure 6. Typing rules for λC_μ

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.

$$\frac{\Gamma, x : A \vdash B_1 : (\Pi y : C_1. C_2) \quad \Gamma, x : A \vdash B_2 : C_1}{\Gamma, x : A \vdash (B_1 B_2) : C_2[y := B_2]}.$$

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

□

Theorem 2.2.2 (Subject Reduction)

If $\Gamma \vdash A : B$ and $A \longrightarrow A'$ then $\Gamma \vdash A' : B'$ for some B' such that either $B' \equiv B$ or $B' \longrightarrow B$.

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

Derivation \mathcal{D} has the following form

$$\frac{\frac{\Gamma, x : A \vdash M : A'}{\Gamma \vdash (\lambda x : A. M) : (\Pi x : A. A')} \text{Lam} \quad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x : A. M)N : A'} \text{App}$$

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

$$\text{case } \mathbf{R-AppL}: \frac{M \longrightarrow M'}{MN \longrightarrow M'N}.$$

Derivation \mathcal{D} has the following form

$$\frac{\Gamma \vdash M : (\Pi x : A.A') \quad \Gamma \vdash N : A}{\Gamma \vdash MN : A'} \text{App}$$

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

$$\frac{\Gamma \vdash M' : (\Pi x : A.A') \quad \Gamma \vdash N : A}{\Gamma \vdash M'N : A'} \text{App}$$

$$\text{case } \mathbf{R-Unfold}: \frac{M \longrightarrow M'}{\text{unfold } M \longrightarrow \text{unfold } M'}.$$

Derivation \mathcal{D} has the following form

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

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

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

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

Derivation \mathcal{D} has the following form

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

$$\text{case } \mathbf{R-Mu}: \frac{}{\mu x.M \longrightarrow M[x := \mu x.M]}.$$

Derivation \mathcal{D} has the following form

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

$$\text{Hence, by Lemma 2.2.1 we have } \frac{\Gamma, x : s \vdash M : s \quad \Gamma \vdash \mu x.M : s}{\Gamma \vdash (M[x := \mu x.M]) : s}.$$

$$\text{case } \mathbf{R-Beta}: \frac{}{\text{beta } M \longrightarrow M}.$$

Derivation \mathcal{D} has the following form

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s \quad A \longrightarrow B}{\Gamma \vdash (\text{beta } M) : B} \text{Beta}$$

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

$$\frac{\Gamma \vdash M' : A \quad \Gamma \vdash B : s \quad A \longrightarrow B}{\Gamma \vdash (\text{beta } M') : B} \text{Beta}$$

□

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:

$$\text{case } \mathbf{Var}: \frac{\cdot \vdash A : s}{\cdot, x : A \vdash x : A}.$$

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 “.” without A , which is not possible.

case Weak: $\frac{\cdot \vdash b : B \quad \cdot \vdash A : s}{\cdot, x : A \vdash b : B}$.

The result is trivial by induction hypothesis.

case App: $\frac{\cdot \vdash M : (\Pi x : A.B) \quad \cdot \vdash N : A}{\cdot \vdash MN : B}$.

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 $\frac{M \longrightarrow M'}{MN \longrightarrow M'N} R\text{-AppL}$.

case Lam: $\frac{\dots}{\cdot \vdash (\lambda x : A.M) : (\Pi x : A.B)}$.

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

case Pi: $\frac{\cdot \vdash A : s \quad \cdot, x : A \vdash B : t}{\cdot \vdash (\Pi x : A.B) : t}$.

The result is trivial if let $v = \Pi x : A.B$.

case Mu: $\frac{\dots}{\cdot \vdash (\mu x.A) : s}$.

The result is trivial since we always have such reduction $\mu x.A \longrightarrow A[x := \mu x.A]$.

case Fold: $\frac{\dots}{\cdot \vdash (\text{fold}[\mu x.A] M) : \mu x.A}$.

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

case Unfold: $\frac{\cdot \vdash a : \mu x.A \quad \cdot \vdash A[x := \mu x.A] : s}{\cdot \vdash (\text{unfold } a) : A[x := \mu x.A]}$.

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

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

case Beta: $\frac{\dots}{\cdot \vdash (\text{beta } a) : B}$.

The result is trivial since we always have such reduction $\text{beta } a \longrightarrow a$.

□

2.3 Examples of typable terms

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

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

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 $\text{fix } \alpha g$ diverges for any g .

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

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

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:

$$\text{Nat} = \mu X. \Pi(a : \star). a \rightarrow (X \rightarrow a) \rightarrow a$$

then we can define zero and suc as follows:

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

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

$$\begin{aligned} \text{plus} &: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\ \text{plus} &= \text{fix}(\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}) (\lambda(p : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat})(n : \text{Nat})(m : \text{Nat}). \\ &\quad (\text{unfold } n) \text{ Nat } m (\lambda(n' : \text{Nat}). \text{suc } (p n' m))) \end{aligned}$$

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

$$\text{List} = \mu X. \Pi(a : \star). a \rightarrow (\Pi(b : \star). b \rightarrow X \rightarrow a) \rightarrow a$$

then we can define nil and cons as follows:

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

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

$$\begin{aligned} \text{length} &: \text{List} \rightarrow \text{Nat} \\ \text{length} &= \text{fix} (\text{List} \rightarrow \text{Nat}) (\lambda(l : \text{List} \rightarrow \text{Nat})(xs : \text{List}). \\ &\quad (\text{unfold } xs) \text{Nat zero } (\lambda(b : \star)(y : b)(ys : \text{List}). \text{suc } (l \text{ } ys))) \end{aligned}$$

3.1.2 Elaboration of Datatypes

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

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

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 $\text{ar}(i)$, which can be zero.

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

$$\begin{aligned} K_1 &: T^1(D) \rightarrow D \\ &\quad \vdots \\ K_n &: T^n(D) \rightarrow D \end{aligned}$$

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:

$$D ::= \mu\beta. \Pi(\alpha : \star). (T^1(\beta) \rightarrow \alpha) \rightarrow \dots \rightarrow (T^n(\beta) \rightarrow \alpha) \rightarrow \alpha$$

The constructors are encoded by:

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

3.1.3 Elaboration of Case Analysis

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

$$\begin{aligned} A &::= x \mid \star \mid \square \\ &\mid AA \mid \lambda x : A. A \mid \Pi x : A. A \\ &\mid \mu x. A \mid \text{fold}[A] A \mid \text{unfold } A \\ &\mid \text{beta } A \\ &\mid \text{case } A \text{ of } \{x \ x_1 \ x_2 \dots \Rightarrow A; \dots\} \end{aligned}$$

Suppose we have

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

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

This can be translated to our system as follows:

$$\begin{aligned} &(\text{unfold } x) T (\lambda(x_1 : T_1^1(D)) \dots (x_{\text{ar}(1)} : T_{\text{ar}(1)}^1(D)). r_1) \\ &\dots \\ &(\lambda(x_1 : T_1^n(D)) \dots (x_{\text{ar}(n)} : T_{\text{ar}(n)}^n(D)). r_n) \end{aligned}$$

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