

# Calculus of Constructions with Recursive Types

*Last modified: April 14, 2015 at 7:57pm*

## 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* ( $\lambda C$ ) 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*  $\Gamma$  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)
$\Gamma$	$::=$	$\emptyset$	(empty)
		$\Gamma, x : A$	(variable binding)

**Figure 1.** Syntax of  $\lambda C$

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)  $\Pi$  and  $\lambda$  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  $(\Pi_1 : A. B)$ .
- (iv) The  $\beta$ -reduction ( $\rightarrow_\beta$ ) 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  $\rightarrow_\beta$  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.

<b>Values:</b> $v ::=$	$\lambda x : A.B \mid \Pi x : A.B$
(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  $\lambda C$

(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  $\lambda C$

## 2. Extend with recursive types

### 2.1 Core language

We extend Calculus of Constructions ( $\lambda C$ ) with recursive types, namely  $\lambda C_{\mu}$ . Differences with  $\lambda C$  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  $\lambda C_{\mu}$

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.

<b>values:</b>	$v ::= \lambda x : A. B$	(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  $\lambda C$

The extended typing rules are shown in Figure 6. Compared with  $\lambda C$ , 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

$$\Gamma, x : A \vdash B : C \quad (1)$$

$$\Gamma \vdash D : A, \quad (2)$$

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  $\lambda C_\mu$

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. Formal Elaboration of Datatypes and Case Analysis

#### 3.1 Extended Language

We extend  $\lambda C_\mu$  with simple datatypes and case analysis, namely  $\lambda C_{\mu c}$ . Differences with  $\lambda C_\mu$  are highlighted in Figure 7.

$pgm$	$::=$	$\overline{decl}; A$	(Declarations)
$decl$	$::=$	<b>data</b> $T$ <b>where</b> $\overline{K} : A \rightarrow \overline{T}$	(Datatype)
$u$	$::=$	$x \mid K$	(Variables and data constructors)
$A$	$::=$	$u$	(Term atoms)
		$\star$	(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)
		<b>let</b> $x : A = A$ <b>in</b> $A$	(Let binding)
		<b>case</b> $A$ <b>of</b> $\overline{p} \Rightarrow \overline{A}$	(Case analysis)
$p$	$::=$	$\overline{K} \ x : \overline{A}$	(Pattern)
$\Gamma$	$::=$	$\emptyset$	(Empty)
		$\Gamma, u : A$	(Variable binding)

**Figure 7.** Syntax of  $\lambda C_{\mu c}$

The extended typing rules are shown in Figure 8. To save space, we only show the new typing rules.

#### 3.2 Translation Overview

We use a type-directed translation. The typing relations have the form:

$$\Gamma \vdash e : A \rightsquigarrow \hat{e}$$

It states that  $\lambda C_\mu$  expression  $\hat{e}$  is the translation of  $\lambda C_{\mu c}$  expression  $e$ .

### 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.

$\boxed{\Gamma \vdash \text{pgm} : A}$	
(Pgm)	$\frac{\overline{\Gamma \vdash \text{decl} : \Gamma_d} \quad \Gamma = \Gamma_0, \overline{\Gamma_d} \quad \Gamma \vdash e : A}{\Gamma_0 \vdash \overline{\text{decl}}; e : A}$
$\boxed{\Gamma \vdash \text{decl} : \Gamma'}$	
(Data)	$\frac{\overline{\Gamma, T : \star \vdash A : \star}}{\Gamma \vdash (\mathbf{data} T \mathbf{where} \overline{K : A}) : (T : \star, \overline{K : A})}$
$\boxed{\Gamma \vdash e : A}$	
(Case)	$\frac{\Gamma \vdash e : T \quad \overline{\Gamma \vdash_p p \Rightarrow e : T \rightarrow B}}{\Gamma \vdash \mathbf{case} e \mathbf{of} \overline{p \Rightarrow e} : B}$
(Let)	$\frac{\Gamma \vdash e_1 : A \quad \Gamma, x : A \vdash e_2 : B}{\mathbf{let} x : A = e_1 \mathbf{in} e_2 : B}$
$\boxed{\Gamma \vdash_p p \Rightarrow e : T \rightarrow B}$	
(Alt)	$\frac{K : \overline{A} \rightarrow T \in \Gamma \quad \Gamma, x : \overline{A} \vdash e : B}{\Gamma \vdash_p \overline{K x : \overline{A} \Rightarrow e : T \rightarrow B}}$

**Figure 8.** Typing rules for  $\lambda C_{\mu}c$

- [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