

# Calculus of Constructions with Recursive Types

*Last modified: April 16, 2015 at 10:20am*

## 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  $(\Pi_ : A. B)$ .
- (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.

$$\begin{array}{l}
\mathbf{Values: } v ::= \lambda x : A. B \mid \Pi x : A. B \\
\\
\text{(R-Beta)} \quad \frac{N \in \text{Value}}{(\lambda x : A. M) N \longrightarrow M[x := N]} \\
\\
\text{(R-AppL)} \quad \frac{M \longrightarrow M'}{MN \longrightarrow M'N} \\
\\
\text{(R-AppR)} \quad \frac{v \in \text{Value} \quad M \longrightarrow M'}{vM \longrightarrow vM'}
\end{array}$$

**Figure 2.** Reduction rules for  $\lambda C$

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

**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.

<b>Terms</b>		
$E, T$	$::=$	$x$ (variable)
	$ $	$\star$ (star)
	$ $	$\square$ (square)
	$ $	$E E$ (application)
	$ $	$\lambda x : T. E$ (abstraction)
	$ $	$\Pi x : T. T$ (product)
	$ $	$\mu x. T$ (recursive type)
	$ $	$\text{fold}[\mu x. T] E$ (roll)
	$ $	$\text{unfold } E$ (unroll)
	$ $	$\text{beta } E$ (type reduction)
<b>Environments</b>		
$\Gamma$	$::=$	$\emptyset$ (empty)
	$ $	$\Gamma, x : T$ (variable binding)
<b>Syntactic sugar</b>		
$\text{let } x : T = E_1 \text{ in } E_2$	$::=$	$(\lambda x : T. E_2) E_1$

**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>		$v ::= \lambda x : T. E$ (abstraction)
	$ $	$\Pi x : T_1. T_2$ (product)
	$ $	$\text{fold}[\mu x. T] E$ (roll)
(R-AppLam)	$\frac{}{(\lambda x : T. E_1) E_2 \longrightarrow E_1[x := E_2]}$	
(R-AppL)	$\frac{E_1 \longrightarrow E'_1}{E_1 E_2 \longrightarrow E'_1 E_2}$	
(R-Unfold)	$\frac{E \longrightarrow E'}{\text{unfold } E \longrightarrow \text{unfold } E'}$	
(R-Unfold-Fold)	$\frac{}{\text{unfold } (\text{fold}[\mu x. T] E) \longrightarrow E}$	
(R-Mu)	$\frac{}{\mu x. T \longrightarrow T[x := \mu x. T]}$	
(R-Beta)	$\frac{}{\text{beta } E \longrightarrow E}$	

**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 Figure 5.

(Ax)	$\frac{}{\emptyset \vdash \star : \square}$	
(Var)	$\frac{\Gamma \vdash T : s}{\Gamma, x : T \vdash x : T}$	$x \notin \text{dom}(\Gamma)$
(Weak)	$\frac{\Gamma \vdash E : T_2 \quad \Gamma \vdash T_1 : s}{\Gamma, x : T_1 \vdash E : T_2}$	$x \notin \text{dom}(\Gamma)$
(App)	$\frac{\Gamma \vdash E_1 : (\Pi x : T_2. T_1) \quad \Gamma \vdash E_2 : T_2}{\Gamma \vdash E_1 E_2 : T_1[x := E_2]}$	
(Lam)	$\frac{\Gamma, x : T_1 \vdash E : T_2 \quad \Gamma \vdash (\Pi x : T_1. T_2) : t}{\Gamma \vdash (\lambda x : T_1. E) : (\Pi x : T_1. T_2)}$	$t \in \{\star, \square\}$
(Pi)	$\frac{\Gamma \vdash T_1 : s \quad \Gamma, x : T_1 \vdash T_2 : t}{\Gamma \vdash (\Pi x : T_1. T_2) : t}$	$(s, t) \in \mathcal{R}$
(Mu)	$\frac{\Gamma, x : s \vdash T : s}{\Gamma \vdash (\mu x. T) : s}$	
(Fold)	$\frac{\Gamma \vdash E : (T[x := \mu x. T]) \quad \Gamma \vdash \mu x. T : s}{\Gamma \vdash (\text{fold}[\mu x. T] E) : \mu x. T}$	
(Unfold)	$\frac{\Gamma \vdash E : \mu x. T \quad \Gamma \vdash T[x := \mu x. T] : s}{\Gamma \vdash (\text{unfold } E) : T[x := \mu x. T]}$	
(Beta)	$\frac{\Gamma \vdash E : T_1 \quad \Gamma \vdash T_2 : s \quad T_2 \longrightarrow T_1}{\Gamma \vdash (\text{beta } E) : T_2}$	

**Figure 6.** Typing rules for  $\lambda C_\mu$

## 2.2 Soundness of core language

We show the soundness of the core language by subject reduction and progress theorems.

### Theorem 2.1 (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.* See Appendix A. □

### Theorem 2.2 (Progress)

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

*Proof.* See Appendix A. □

## 2.3 Examples of typable terms

- Polymorphic identity function: if  $\Gamma \vdash e : \tau$ , we have  $(\lambda \alpha : \star. \lambda x : ((\lambda y : \star. y) \alpha). x) \tau \text{ (beta } e) \longrightarrow \text{beta } e \longrightarrow e$ .
- 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.

Declarations		
$pgm$	$::=$	$\overline{decl}; e$ (Declarations)
$decl$	$::=$	<b>data</b> $D = \overline{K} \overline{\tau}$ (Datatype)
Terms		
$u$	$::=$	$x \mid K$ (Variables and data constructors)
$e, \tau$	$::=$	$u$ (Term atoms)
	$\mid$	$\star$ (Star)
	$\mid$	$\square$ (Square)
	$\mid$	$e e$ (Application)
	$\mid$	$\lambda x : \tau. e$ (Abstraction)
	$\mid$	$\Pi x : \tau. \tau$ (Product)
	$\mid$	$\mu x. \tau$ (Recursive type)
	$\mid$	$\text{fold}[\mu x. \tau] e$ (Roll)
	$\mid$	$\text{unfold } e$ (Unroll)
	$\mid$	$\text{beta } e$ (Type reduction)
	$\mid$	<b>case</b> $e \text{ of } \overline{p} \Rightarrow \overline{e}$ (Case analysis)
$p$	$::=$	$K \overline{x} : \overline{\tau}$ (Pattern)
Environments		
$\Gamma$	$::=$	$\emptyset$ (Empty)
	$\mid$	$\Gamma, u : \tau$ (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:

$$\Gamma \vdash e : \tau \rightsquigarrow E$$

It states that  $\lambda C_\mu$  expression  $E$  is the translation of  $\lambda C_{\mu c}$  expression  $e$  of type  $\tau$ . Figure 9 shows the

	$\boxed{\Gamma \vdash \text{pgm} : \tau}$
(Pgm)	$\frac{\overline{\Gamma_0 \vdash \text{decl} : \Gamma_d} \quad \Gamma = \Gamma_0, \overline{\Gamma_d} \quad \Gamma \vdash e : \tau}{\Gamma_0 \vdash \overline{\text{decl}}; e : \tau}$
	$\boxed{\Gamma \vdash \text{decl} : \Gamma'}$
(Data)	$\frac{\overline{\Gamma, D : \star \vdash \bar{\tau} \rightarrow D : \star}}{\Gamma \vdash (\mathbf{data} D = \overline{K \bar{\tau}}) : (D : \star, \overline{K : \bar{\tau} \rightarrow D})}$
	$\boxed{\Gamma \vdash e : \tau}$
(Case)	$\frac{\Gamma \vdash e : D \quad \overline{\Gamma \vdash_p p \Rightarrow e : D \rightarrow \tau}}{\Gamma \vdash \mathbf{case} e \mathbf{of} \overline{p \Rightarrow \bar{e}} : \tau}$
	$\boxed{\Gamma \vdash_p p \Rightarrow e : D \rightarrow \tau}$
(Alt)	$\frac{K : \bar{\tau} \rightarrow D \in \Gamma \quad \Gamma, \overline{x : \bar{\tau}} \vdash e : \tau'}{\Gamma \vdash_p \overline{K x : \bar{\tau}} \Rightarrow e : D \rightarrow \tau'}$

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

translation rules, which are the typing rules of the previous section extended with the resulting expression  $E$ .

### 3.3 Examples of Simple Datatypes

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

$$\begin{aligned} \mathbf{data} \text{Nat} &= \text{zero} \mid \text{suc Nat} \\ \text{Nat} &::= \mu X. \Pi(a : \star). a \rightarrow (X \rightarrow a) \rightarrow a \end{aligned}$$

zero and suc are encoded as follows:

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

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 natural numbers:

$$\begin{aligned} \mathbf{data} \text{List} &= \text{nil} \mid \text{cons Nat List} \\ \text{List} &::= \mu X. \Pi(a : \star). a \rightarrow (\text{Nat} \rightarrow X \rightarrow a) \rightarrow a \end{aligned}$$

	$\boxed{\Gamma \vdash e : \tau \rightsquigarrow E}$	
(Ax)	$\frac{}{\emptyset \vdash \star : \square \rightsquigarrow \star}$	
(Var)	$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow x}$	
(App)	$\frac{\Gamma \vdash e_1 : (\Pi x : \tau_2. \tau_1) \rightsquigarrow E_1 \quad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow E_2}{\Gamma \vdash e_1 e_2 : \tau_1[x := e_2] \rightsquigarrow E_1 E_2}$	
(Lam)	$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow E \quad \Gamma \vdash (\Pi x : \tau_1. \tau_2) : t}{\Gamma \vdash (\lambda x : \tau_1. e) : (\Pi x : \tau_1. \tau_2) \rightsquigarrow \lambda x : \tau_1. E} \quad t \in \{\star, \square\}$	
(Pi)	$\frac{\Gamma \vdash \tau_1 : s \quad \Gamma, x : \tau_1 \vdash \tau_2 : t}{\Gamma \vdash (\Pi x : \tau_1. \tau_2) : t \rightsquigarrow \Pi x : \tau_1. \tau_2} \quad (s, t) \in \mathcal{R}$	
(Mu)	$\frac{\Gamma, x : s \vdash \tau : s}{\Gamma \vdash (\mu x. \tau) : s \rightsquigarrow \mu x. \tau}$	
(Fold)	$\frac{\Gamma \vdash e : (\tau[x := \mu x. \tau]) \rightsquigarrow E \quad \Gamma \vdash \mu x. \tau : s}{\Gamma \vdash (\text{fold}[\mu x. \tau] e) : \mu x. \tau \rightsquigarrow \text{fold}[\mu x. \tau] E}$	
(Unfold)	$\frac{\Gamma \vdash e : \mu x. \tau \rightsquigarrow E \quad \Gamma \vdash \tau[x := \mu x. \tau] : s}{\Gamma \vdash (\text{unfold } e) : \tau[x := \mu x. \tau] \rightsquigarrow \text{unfold } E}$	
(Beta)	$\frac{\Gamma \vdash e : \tau_1 \rightsquigarrow E \quad \Gamma \vdash \tau_2 : s \quad \tau_2 \longrightarrow \tau_1}{\Gamma \vdash (\text{beta } e) : \tau_2 \rightsquigarrow \text{beta } E}$	
(Case)	$\frac{\Gamma \vdash e : D \rightsquigarrow E \quad \overline{\Gamma \vdash_p p \Rightarrow e : D \rightarrow \tau \rightsquigarrow E_1}}{\Gamma \vdash \text{case } e \text{ of } \overline{p \Rightarrow e} : \tau \rightsquigarrow (\text{unfold } E) \tau \overline{E_1}}$	
	$\boxed{\Gamma \vdash_p p \Rightarrow e : D \rightarrow \tau \rightsquigarrow E}$	
(Alt)	$\frac{K : \bar{\tau} \rightarrow D \in \Gamma \quad \Gamma, \bar{x} : \bar{\tau} \vdash e : \tau' \rightsquigarrow E}{\Gamma \vdash_p K \bar{x} : \bar{\tau} \Rightarrow e : D \rightarrow \tau' \rightsquigarrow \lambda(\bar{x} : \bar{\tau}). E}$	
	$\boxed{\Gamma \vdash \text{decl} : \Gamma' \rightsquigarrow E}$	
(Data)	$\frac{\overline{\Gamma, D : \star \vdash \bar{\tau} \rightarrow D : \star}}{\Gamma \vdash (\text{data } D = \overline{K \bar{\tau}}) : (D : \star, \overline{K : \bar{\tau} \rightarrow D}) \rightsquigarrow E}$	
	$E ::= \text{let } D : \star = \mu\beta. \Pi\alpha : \star. \overline{(\tau[D := \beta] \rightarrow \alpha)} \rightarrow \alpha \text{ in}$ $\text{let } K_i^{i \in 1..n} : \bar{\tau}_i \rightarrow D = \lambda(x : \tau_i). \text{fold}[D] (\lambda(\alpha : \star) (\overline{c : \bar{\tau} \rightarrow \alpha}). c_i \bar{x}) \text{ in}$	
	$\boxed{\Gamma \vdash \text{pgm} : \tau \rightsquigarrow E}$	
(Pgm)	$\frac{\overline{\Gamma_0 \vdash \text{decl} : \Gamma_d \rightsquigarrow E_1} \quad \Gamma = \Gamma_0, \overline{\Gamma_d} \quad \Gamma \vdash e : \tau \rightsquigarrow E}{\Gamma_0 \vdash \text{decl}; e : A \rightsquigarrow \overline{E_1} \oplus E}$	

**Figure 9.** Type-directed translation from  $\lambda C_{\mu} c$  to  $\lambda C_{\mu}$

nil and cons are encoded as follows:

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

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

```
length : List → Nat
length = fix (List → Nat) (λ(l : List → Nat)(xs : List).
  (unfold xs) Nat zero (λ(y : Nat)(ys : List). suc (l ys)))
```

## 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

### Lemma A.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).

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 A.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 A.1 we can obtain  $\Gamma \vdash M[x := N] : A'$ .

**case 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,

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

**case 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,

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

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

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

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

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

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

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

Hence, by Lemma A.1 we have  $\frac{\Gamma, x : s \vdash M : s \quad \Gamma \vdash \mu x.M : s}{\Gamma \vdash (M[x := \mu x.M]) : s}$ .

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

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

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

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

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

□

**Theorem A.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:**  $\frac{\cdot \vdash A : s}{\cdot, x : A \vdash x : A}$ .

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  $\text{beta } a \longrightarrow a$ .

□