Calculus of Constructions with Recursive Types

Branch: Generalized fold and unfold

Last modified: April 21, 2015 at 5:37pm

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.

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

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.

$$
\textbf{Values}: v ::= \quad \lambda x : A.B \mid \Pi x : A.B
$$

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

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

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

Figure 2. Reduction rules for λC

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

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

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

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

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

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

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

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.

Terms			
E, T	$::=$	x	(variable)
	$\mid$	$\star$	(star)
	$\mid$	$\square$	(square)
	$\mid$	$E\ E$	(application)
	$\mid$	$\lambda x : T.E$	(abstraction)
	$\mid$	$\Pi x : T.T$	(product)
	$\mid$	$\mu x.T$	(recursive type)
	$\mid$	$\mathsf{fold}[T]\ E$	(generalized roll)
	$\mid$	$\mathsf{unfold}\ E$	(generalized unroll)
Environments			
Γ	$::=$	$\varnothing$	(empty)
	$\mid$	$\Gamma, x : T$	(variable binding)
Syntactic sugar			
let $x : T = E_1$ **in** E_2	$::=$	$(\lambda x : T.E_2)\ E_1$	

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	$::=$	$\lambda x : T.E$	(abstraction)
	$\mid$	$\Pi x : T_1.T_2$	(product)
	$\mid$	$\mathsf{fold}[T]\ E$	(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) $$\frac{E \longrightarrow E'}{\mathsf{unfold}\ E \longrightarrow \mathsf{unfold}\ E'}$$

(R-Unfold-Fold) $$\frac{}{\mathsf{unfold}\ (\mathsf{fold}[T]\ E) \longrightarrow E}$$

(R-Mu) $$\frac{}{\mu x.T \longrightarrow T[x := \mu x.T]}$$

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 Figure 5.

2.2 Soundness of core language

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

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

Figure 6. Typing rules for λC_μ

Theorem 2.1 (Subject Reduction)
If $\Gamma \vdash A : B$ *and* $A \longrightarrow A'$ *then* $\Gamma \vdash A' : B$.

Proof. See Appendix A. $\square$

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. $\square$

2.3 Examples of typable terms

- Polymorphic identity function: if $\Gamma \vdash e : \tau$, we have

  $$\mathsf{unfold}\,((\lambda\alpha : \star.\lambda x : ((\lambda y : \star.y)\ \alpha).x)\ \tau\ (\mathsf{fold}[(\lambda y : \star.y)\ \tau]\ e)) \longrightarrow \mathsf{unfold}\,(\mathsf{fold}[(\lambda y : \star.y)\ \tau]\ e) \longrightarrow e.$$

- 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. Formal Elaboration of Datatypes and Case Analysis

3.1 Extended Language

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

Declarations			
pgm	$::=$	$\overline{decl}; e$	(Declarations)
$decl$	$::=$	$\textbf{data}\, D : \overline{\kappa} \to \star\, \textbf{where}$	
		$\quad K : \Pi\overline{x : \kappa}.\Pi\overline{y : \iota}.\overline{\tau} \to D\,\overline{x}$	(Datatype)
Terms			
u	$::=$	$x \mid K$	(Variables and data constructors)
e, τ, κ, ι	$::=$	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)
	$\mid$	$\mathsf{fold}[\tau]\, e$	(Generalized roll)
	$\mid$	$\mathsf{unfold}\, e$	(Generalized unroll)
	$\mid$	$\textbf{case}\, e\, \textbf{of}\, \overline{p \Rightarrow e}$	(Case analysis)
p	$::=$	$K\,\overline{y : \kappa}\,\overline{x : \tau}$	(Pattern)
Environments			
Γ	$::=$	$\varnothing$	(Empty)
	$\mid$	$\Gamma, u : \tau$	(Variable binding)

Figure 7. Syntax of $\lambda C_{\mu c}$ (e for terms, τ for types, κ, ι for kinds)

An *algebraic data type* D is introduced as a top-level **data** declaration with its *data constructors*. The type of a data constructor K has the form:

$$K : \Pi\overline{x : \kappa}^n.\Pi\overline{y : \iota}.\overline{\tau} \to D\,\overline{x}^n$$

The first n quantified type variables $\overline{x}$ appear in the same order in the return type $D\,\overline{x}$, and $\overline{y}$ stands for existentially quantified type variables. There is a **case** expression to take apart values built with data constructors. The patterns of a case expression are flat (no nested patterns), and bind existential type variables.

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

$\boxed{\Gamma \vdash pgm : \tau}$

$$\text{(Pgm)} \qquad \frac{\overline{\Gamma_0 \vdash decl : \Gamma_d} \qquad \Gamma = \Gamma_0, \overline{\Gamma_d} \qquad \Gamma \vdash e : \tau}{\Gamma_0 \vdash \overline{decl}; e : \tau}$$

$\boxed{\Gamma \vdash decl : \Gamma'}$

$$\text{(Data)} \qquad \frac{\overline{\Gamma, D : \star \vdash \overline{\tau} \to D : \star}}{\Gamma \vdash (\mathbf{data}\, D = \overline{K\,\overline{\tau}}) : (D : \star, \overline{K : \overline{\tau} \to D})}$$

$\boxed{\Gamma \vdash e : \tau}$

$$\text{(Case)} \qquad \frac{\Gamma \vdash e : D \qquad \overline{\Gamma \vdash_p p \Rightarrow e : D \to \tau}}{\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, \overline{p \Rightarrow e} : \tau}$$

$\boxed{\Gamma \vdash_p p \Rightarrow e : D \to \tau}$

$$\text{(Alt)} \qquad \frac{K : \overline{\tau} \to D \in \Gamma \qquad \Gamma, \overline{x : \tau} \vdash e : \tau'}{\Gamma \vdash_p K\, \overline{x : \tau} \Rightarrow e : D \to \tau'}$$

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

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 λC_μ expression E is the translation of $\lambda C_{\mu c}$ expression e of type τ. Figure 9 shows the 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:

$$\mathbf{data}\, \mathsf{Nat} = \mathsf{zero} \mid \mathsf{suc}\, \mathsf{Nat}$$
$$\mathsf{Nat} ::= \mu X.\, \Pi(a : \star).\, a \to (X \to a) \to a$$

zero and suc are encoded as follows:

$$\mathsf{zero} ::= \mathsf{fold}[\mathsf{Nat}]\, (\lambda(a : \star)(z : a)(f : \mathsf{Nat} \to a).\, z)$$
$$\mathsf{suc} ::= \lambda(n : \mathsf{Nat}).\, \mathsf{fold}[\mathsf{Nat}]\, (\lambda(a : \star)(z : a)(f : \mathsf{Nat} \to a).\, f\, n)$$

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

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

$$\boxed{\Gamma \vdash e : \tau \rightsquigarrow E}$$

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

(Var) $$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow x}$$

(App) $$\frac{\Gamma \vdash e_1 : (\Pi x : \tau_2 . \tau_1) \rightsquigarrow E_1 \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow E_2}{\Gamma \vdash e_1 e_2 : \tau_1[x := e_2] \rightsquigarrow E_1 E_2}$$

(Lam) $$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow E \qquad \Gamma \vdash (\Pi x : \tau_1 . \tau_2) : t}{\Gamma \vdash (\lambda x : \tau_1 . e) : (\Pi x : \tau_1 . \tau_2) \rightsquigarrow \lambda x : \tau_1 . E} \qquad t \in \{\star, \Box\}$$

(Pi) $$\frac{\Gamma \vdash \tau_1 : s \qquad \Gamma, x : \tau_1 \vdash \tau_2 : t}{\Gamma \vdash (\Pi x : \tau_1 . \tau_2) : t \rightsquigarrow \Pi x : \tau_1 . \tau_2} \qquad (s, t) \in \mathcal{R}$$

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

(Fold) $$\frac{\Gamma \vdash e : \tau_2 \rightsquigarrow E \qquad \Gamma \vdash \tau_1 : s \qquad \tau_1 \longrightarrow \tau_2}{\Gamma \vdash (\mathsf{fold}[\tau_1]\, e) : \tau_1 \rightsquigarrow \mathsf{fold}[\tau_1]\, E}$$

(Unfold) $$\frac{\Gamma \vdash e : \tau_1 \rightsquigarrow E \qquad \Gamma \vdash \tau_2 : s \qquad \tau_1 \longrightarrow \tau_2}{\Gamma \vdash (\mathsf{unfold}\, e) : \tau_2 \rightsquigarrow \mathsf{unfold}\, E}$$

(Case) $$\frac{\Gamma \vdash e : D \rightsquigarrow E \qquad \overline{\Gamma \vdash_p p \Rightarrow e : D \to \tau \rightsquigarrow E_1}}{\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, \overline{p \Rightarrow e} : \tau \rightsquigarrow (\mathsf{unfold}\, E)\, \tau\, \overline{E_1}}$$

$$\boxed{\Gamma \vdash_p p \Rightarrow e : D \to \tau \rightsquigarrow E}$$

(Alt) $$\frac{K : \overline{\tau} \to D \in \Gamma \qquad \Gamma, \overline{x : \tau} \vdash e : \tau' \rightsquigarrow E}{\Gamma \vdash_p K\, \overline{x : \tau} \Rightarrow e : D \to \tau' \rightsquigarrow \lambda \overline{(x : \tau)} . E}$$

$$\boxed{\Gamma \vdash decl : \Gamma' \rightsquigarrow E}$$

(Data) $$\frac{\overline{\Gamma, D : \star \vdash \overline{\tau} \to D : \star}}{\Gamma \vdash (\mathbf{data}\, D = \overline{K\, \overline{\tau}}) : (D : \star, \overline{K : \overline{\tau} \to D}) \rightsquigarrow E}$$

$$\begin{array}{lll} E & ::= & \mathbf{let}\, D : \star = \mu\beta . \Pi\alpha : \star . (\overline{\overline{\tau[D := \beta]} \to \alpha}) \to \alpha\, \mathbf{in} \\ & & \mathbf{let}\, K_i^{i \in 1..n} : \overline{\tau_i} \to D = \lambda \overline{(x : \tau_i)} . \\ & & \quad \mathsf{fold}[D]\, (\lambda(\alpha : \star) \overline{(c : \overline{\tau} \to \alpha)} . c_i\, \overline{x})\, \mathbf{in} \end{array}$$

$$\boxed{\Gamma \vdash pgm : \tau \rightsquigarrow E}$$

(Pgm) $$\frac{\overline{\Gamma_0 \vdash decl : \Gamma_d \rightsquigarrow E_1} \qquad \Gamma = \Gamma_0, \overline{\Gamma_d} \qquad \Gamma \vdash e : \tau \rightsquigarrow E}{\Gamma_0 \vdash \overline{decl}; e : A \rightsquigarrow \overline{E_1} \oplus E}$$

Figure 9. Type-directed translation from $\lambda C_\mu c$ to λC_μ

- We can encode the type of lists of natural numbers:

$$\mathbf{data}\, \mathsf{List} = \mathsf{nil} \mid \mathsf{cons}\, \mathsf{Nat}\, \mathsf{List}$$

$$\mathsf{List} ::= \mu X . \Pi(a : \star) . a \to (\mathsf{Nat} \to X \to a) \to a$$

nil and cons are encoded as follows:

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

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

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

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

Lemma A.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).
  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.
$$\frac{\Gamma, x : A \vdash B_1 : (\Pi y : C_1.\, C_2) \qquad \Gamma, x : A \vdash B_2 : C_1}{\Gamma, x : A \vdash (B_1 B_2) : C_2[y := B_2]}\,.$$
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*: $\dfrac{}{(\lambda x : A.M)N \longrightarrow M[x := N]}\,.$

Derivation $\mathcal{D}$ has the following form
$$\dfrac{\dfrac{\Gamma, x : A \vdash M : A'}{\Gamma \vdash (\lambda x : A.M) : (\Pi x : A.A')}\,\textit{Lam} \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x : A.M)N : A'}\,\textit{App}$$
Thus, by Lemma A.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
$$\dfrac{\Gamma \vdash M : (\Pi x : A.A') \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : A'}\,\textit{App}$$
By the induction hypothesis we have $\Gamma \vdash M' : (\Pi x : A.A')$. Hence,
$$\dfrac{\Gamma \vdash M' : (\Pi x : A.A') \qquad \Gamma \vdash N : A}{\Gamma \vdash M'N : A'}\,\textit{App}$$

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

Derivation $\mathcal{D}$ has the following form
$$\dfrac{\Gamma \vdash M : \mu x.A}{\Gamma \vdash (\mathsf{unfold}\, M) : A[x := \mu x.A]}\,\textit{Unfold}$$
By the induction hypothesis we have $\Gamma \vdash M' : \mu x.A$. Hence,
$$\dfrac{\Gamma \vdash M' : \mu x.A}{\Gamma \vdash (\mathsf{unfold}\, M') : A[x := \mu x.A]}\,\textit{Unfold}$$

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

Derivation $\mathcal{D}$ has the following form
$$\dfrac{\dfrac{\Gamma \vdash M : (A[x := \mu x.A])}{\Gamma \vdash (\mathsf{fold}[\mu x.A]\, M) : \mu x.A}\,\textit{Fold}}{\Gamma \vdash \mathsf{unfold}\,(\mathsf{fold}[\mu x.A]\, M) : (A[x := \mu x.A])}\,\textit{Unfold}$$

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

Derivation $\mathcal{D}$ has the following form
$$\dfrac{\Gamma, x : s \vdash M : s}{\Gamma \vdash (\mu x.M) : s}\,\textit{Mu}$$

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

$\square$

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

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

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

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

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

case *Fold*: $\dfrac{\ldots}{\cdot \vdash (\mathsf{fold}[\mu x.A]\, M) : \mu x.A}$.

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

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

$\square$