Calculus of Constructions with Recursive Types

Last modified: April 16, 2015 at 12:49am

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 ::= $\lambda x : A.B \mid \Pi x : A.B$

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

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

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

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)
	$\mid$	$\mathsf{fold}[\mu x.T]\ E$	(roll)
	$\mid$	$\mathsf{unfold}\ E$	(unroll)
	$\mid$	$\mathsf{beta}\ E$	(type reduction)
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)
		$\mid$	$\mathsf{fold}[\mu x.T]\ E$	(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)
$$\frac{E \longrightarrow E'}{\mathsf{unfold}\ E \longrightarrow \mathsf{unfold}\ E'}$$

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

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

(R-Beta)
$$\frac{}{\mathsf{beta}\ E \longrightarrow E}$$

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.

$$
\begin{array}{lcl}
\text{(Ax)} & \dfrac{}{\varnothing \vdash \star : \square} & \\[2ex]
\text{(Var)} & \dfrac{\Gamma \vdash T : s}{\Gamma, x : T \vdash x : T} & x \notin \mathrm{dom}(\Gamma) \\[2ex]
\text{(Weak)} & \dfrac{\Gamma \vdash E : T_2 \qquad \Gamma \vdash T_1 : s}{\Gamma, x : T_1 \vdash E : T_2} & x \notin \mathrm{dom}(\Gamma) \\[2ex]
\text{(App)} & \dfrac{\Gamma \vdash E_1 : (\Pi x : T_2.T_1) \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash E_1 E_2 : T_1[x := E_2]} & \\[2ex]
\text{(Lam)} & \dfrac{\Gamma, x : T_1 \vdash E : T_2 \qquad \Gamma \vdash (\Pi x : T_1.T_2) : t}{\Gamma \vdash (\lambda x : T_1.E) : (\Pi x : T_1.T_2)} & t \in \{\star, \square\} \\[2ex]
\text{(Pi)} & \dfrac{\Gamma \vdash T_1 : s \qquad \Gamma, x : T_1 \vdash T_2 : t}{\Gamma \vdash (\Pi x : T_1.T_2) : t} & (s, t) \in \mathcal{R} \\[2ex]
\text{(Mu)} & \dfrac{\Gamma, x : s \vdash T : s}{\Gamma \vdash (\mu x.T) : s} & \\[2ex]
\text{(Fold)} & \dfrac{\Gamma \vdash E : (T[x := \mu x.T]) \qquad \Gamma \vdash \mu x.T : s}{\Gamma \vdash (\mathsf{fold}[\mu x.T]\, E) : \mu x.T} & \\[2ex]
\text{(Unfold)} & \dfrac{\Gamma \vdash E : \mu x.T \qquad \Gamma \vdash T[x := \mu x.T] : s}{\Gamma \vdash (\mathsf{unfold}\, E) : T[x := \mu x.T]} & \\[2ex]
\text{(Beta)} & \dfrac{\Gamma \vdash E : T_1 \qquad \Gamma \vdash T_2 : s \qquad T_2 \longrightarrow T_1}{\Gamma \vdash (\mathsf{beta}\, E) : T_2} &
\end{array}
$$

Figure 6. Typing rules for λC_μ

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

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

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

Thus, by Lemma 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

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

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

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

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

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

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

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

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

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

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

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

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

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

case *R-Beta*: $\overline{\textsf{beta}\, M \longrightarrow M}$.

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

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad B \longrightarrow A}{\Gamma \vdash (\textsf{beta}\, M) : B}\ \textit{Beta}$$

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

$$\frac{\Gamma \vdash M' : A \qquad \Gamma \vdash B : s \qquad B \longrightarrow A}{\Gamma \vdash (\textsf{beta}\, M') : B}\ \textit{Beta}$$

□

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:

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

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

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

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

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

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

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

□

2.3 Examples of typable terms

- Polymorphic identity function: if $\Gamma \vdash e : \tau$, we have $(\lambda\alpha : \star.\lambda x : ((\lambda y : \star.y)\alpha).x)\ \tau\ (\mathsf{beta}\, e) = e$.
- A polymorphic fixed-point constructor $\mathsf{fix} : (\Pi\alpha : \star.(\alpha \to \alpha) \to \alpha)$ can be defined as follows:

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

  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)
$decl$	$::=$	$\mathbf{data}\, D = \overline{K\, \overline{\tau}}$	(Datatype)

Terms

u	$::=$	$x \mid K$	(Variables and data constructors)
e, τ	$::=$	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)
	$\mid$	$\mathsf{fold}[\mu x.\tau]\, e$	(Roll)
	$\mid$	$\mathsf{unfold}\, e$	(Unroll)
	$\mid$	$\mathsf{beta}\, e$	(Type reduction)
	$\mid$	$\mathbf{case}\, e\, \mathbf{of}\, \overline{p \Rightarrow e}$	(Case analysis)
p	$::=$	$K\, \overline{x : \tau}$	(Pattern)

Environments

Γ	$::=$	$\varnothing$	(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.

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

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

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

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

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

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

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

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

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:

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

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:

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

  zero and suc are encoded as follows:

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

  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 natural numbers:

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

  nil and cons are encoded as follows:

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

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

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

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

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

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

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

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

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

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

$$\text{(Fold)} \qquad \frac{\Gamma \vdash e : (\tau[x := \mu x . \tau]) \rightsquigarrow E \qquad \Gamma \vdash \mu x . \tau : s}{\Gamma \vdash (\mathsf{fold}[\mu x . \tau]\, e) : \mu x . \tau \rightsquigarrow \mathsf{fold}[\mu x . \tau]\, E}$$

$$\text{(Unfold)} \qquad \frac{\Gamma \vdash e : \mu x . \tau \rightsquigarrow E \qquad \Gamma \vdash \tau[x := \mu x . \tau] : s}{\Gamma \vdash (\mathsf{unfold}\, e) : \tau[x := \mu x . \tau] \rightsquigarrow \mathsf{unfold}\, E}$$

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

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

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

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

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

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

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

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

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

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

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