Calculus of Constructions with Recursive Types

Last modified: April 15, 2015 at 10:00am

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.

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

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.

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

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

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

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

Figure 2. Reduction rules for λC

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

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

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

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

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

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

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

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.

$$\begin{array}{rcll}
A & ::= & x & \text{(variable)} \\
  & | & \star & \text{(star)} \\
  & | & \square & \text{(square)} \\
  & | & A\ A & \text{(application)} \\
  & | & \lambda x : A.A & \text{(abstraction)} \\
  & | & \Pi x : A.A & \text{(product)} \\
  & | & \mu x.A & \text{(recursive type)} \\
  & | & \mathsf{fold}[\mu x.A]\ A & \text{(roll)} \\
  & | & \mathsf{unfold}\ A & \text{(unroll)} \\
  & | & \mathsf{beta}\ A & \text{(type reduction)} \\
\Gamma & ::= & \varnothing & \text{(empty)} \\
  & | & \Gamma, x : A & \text{(variable binding)}
\end{array}$$

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.

$$\begin{array}{llll} \textbf{values:} \quad v & ::= & \lambda x : A.B & \text{(abstraction)} \\ & | & \Pi x : A.B & \text{(product)} \\ & | & \boxed{\mathsf{fold}[\mu x.A]\, B} & \text{(roll)} \end{array}$$

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

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

$$\text{(R-Unfold)} \quad \boxed{\frac{M \longrightarrow M'}{\mathsf{unfold}\, M \longrightarrow \mathsf{unfold}\, M'}}$$

$$\text{(R-Unfold-Fold)} \quad \boxed{\frac{}{\mathsf{unfold}\, (\mathsf{fold}[\mu x.A]\, M) \longrightarrow M}}$$

$$\text{(R-Mu)} \quad \boxed{\frac{}{\mu x.M \longrightarrow M[x := \mu x.M]}}$$

$$\text{(R-Beta)} \quad \boxed{\frac{}{\mathsf{beta}\, M \longrightarrow M}}$$

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

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.

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

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

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

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

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

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 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*: $\dfrac{}{\mathsf{beta}\, M \longrightarrow M}$.

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

$$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \longrightarrow B}{\Gamma \vdash (\mathsf{beta}\, M) : B}\;\textit{Beta}$$

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

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

□

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

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 $\mathsf{beta}\, a \longrightarrow a$.

□

2.3 Examples of typable terms

- 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.

pgm	$::=$	$\overline{decl}; A$	(Declarations)
$decl$	$::=$	$\mathbf{data}\,T = \overline{K\,\overline{A}}$	(Datatype)
u	$::=$	$x \mid K$	(Variables and data constructors)
A	$::=$	u	(Term atoms)
	$\mid$	$\star$	(Star)
	$\mid$	$\square$	(Square)
	$\mid$	$A\,A$	(Application)
	$\mid$	$\lambda x : A.A$	(Abstraction)
	$\mid$	$\Pi x : A.A$	(Product)
	$\mid$	$\mu x.A$	(Recursive type)
	$\mid$	$\mathsf{fold}[\mu x.A]\,A$	(Roll)
	$\mid$	$\mathsf{unfold}\,A$	(Unroll)
	$\mid$	$\mathsf{beta}\,A$	(Type reduction)
	$\mid$	$\mathbf{let}\,x = A\,\mathbf{in}\,A$	(Let binding)
	$\mid$	$\mathbf{case}\,A\,\mathbf{of}\,\overline{p \Rightarrow A}$	(Case analysis)
p	$::=$	$K\,\overline{x : A}$	(Pattern)
Γ	$::=$	$\varnothing$	(Empty)
	$\mid$	$\Gamma, u : A$	(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 λC_μ expression $\hat{e}$ is the translation of $\lambda C_{\mu c}$ expression e. Figure 9 shows the translation rules, which are the typing rules of the previous section extended with the resulting $\hat{e}$ expression.

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

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

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

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

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

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

(Let) $$\frac{\Gamma \vdash e_1 : A \qquad \Gamma, x : A \vdash e_2 : B}{\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : B}$$

$\boxed{\Gamma \vdash_p p \Rightarrow e : T \to B}$

(Alt) $$\frac{K : \overline{A} \to T \in \Gamma \qquad \Gamma, \overline{x : A} \vdash e : B}{\Gamma \vdash_p K\, \overline{x : A} \Rightarrow e : T \to B}$$

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

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:

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

- 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:

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

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

$\boxed{\Gamma \vdash e : A \leadsto \hat{e}}$

(Ax) $$\frac{}{\varnothing \vdash \star : \square \leadsto \star}$$

(Var) $$\frac{x : A \in \Gamma}{\Gamma \vdash x : A \leadsto x}$$

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

(Lam) $$\frac{\Gamma, x : A \vdash b : B \leadsto \hat{b} \qquad \Gamma \vdash (\Pi x : A.B) : t}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B) \leadsto \lambda x : A.\hat{b}} \qquad t \in \{\star, \square\}$$

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

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

(Fold) $$\frac{\Gamma \vdash a : (A[x := \mu x.A]) \leadsto \hat{a} \qquad \Gamma \vdash \mu x.A : s}{\Gamma \vdash (\mathsf{fold}[\mu x.A]\, a) : \mu x.A \leadsto \mathsf{fold}[\mu x.A]\, \hat{a}}$$

(Unfold) $$\frac{\Gamma \vdash a : \mu x.A \leadsto \hat{a} \qquad \Gamma \vdash A[x := \mu x.A] : s}{\Gamma \vdash (\mathsf{unfold}\, a) : A[x := \mu x.A] \leadsto \mathsf{unfold}\, \hat{a}}$$

(Beta) $$\frac{\Gamma \vdash a : A \leadsto \hat{a} \qquad \Gamma \vdash B : s \qquad A \longrightarrow B}{\Gamma \vdash (\mathsf{beta}\, a) : B \leadsto \mathsf{beta}\, \hat{a}}$$

(Let) $$\frac{\Gamma \vdash e_1 : A \leadsto \hat{e_1} \qquad \Gamma, x : A \vdash e_2 : B \leadsto \hat{e_2}}{\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : B \leadsto (\lambda x : A.\hat{e_2})\, \hat{e_1}}$$

(Case) $$\frac{\Gamma \vdash e : T \leadsto \hat{e} \qquad \overline{\Gamma \vdash_p p \Rightarrow e : T \to B \leadsto E_1}}{\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, \overline{p \Rightarrow e} : B \leadsto (\mathsf{unfold}\, \hat{e})\, B\, \overline{E_1}}$$

$\boxed{\Gamma \vdash_p p \Rightarrow e : T \to B \leadsto \hat{e}}$

(Alt) $$\frac{K : \overline{A} \to T \in \Gamma \qquad \Gamma, \overline{x : A} \vdash e : B \leadsto \hat{e}}{\Gamma \vdash_p K\, \overline{x : A} \Rightarrow e : T \to B \leadsto \lambda\overline{(x : A)}.\hat{e}}$$

$\boxed{\Gamma \vdash decl : \Gamma' \leadsto \hat{e}}$

(Data) $$\frac{\overline{\Gamma, T : \star \vdash \overline{A} \to T : \star}}{\Gamma \vdash (\mathbf{data}\, T = \overline{K\, \overline{A}}) : (T : \star, \overline{K : \overline{A} \to T}) \leadsto E}$$

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

$\boxed{\Gamma \vdash pgm : A \leadsto \hat{e}}$

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

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