Lambda calculus, part II

January 2018

Recap

$\lambda^{\rightarrow}$

Types

$\texttt{A,B} ::= \mathcal{B} \mid \texttt{A} \rightarrow \texttt{B}$

Terms

$\texttt{L,M} ::= \texttt{x} \mid \lambda\texttt{x:A.M} \mid \texttt{L M}$
$\langle\texttt{L,M}\rangle \mid \textbf{fst}\ \texttt{M} \mid \textbf{snd}\ \texttt{M}$
$\textbf{inl}\ \texttt{M} \mid \textbf{inr}\ \texttt{M}$
$\textbf{case}\ \texttt{L}\ \textbf{of}\ \texttt{x.M|y.N}$

System F

Types

$\texttt{A,B} ::= \ldots \mid \alpha$
$\forall\alpha :: K.A$
$\exists\alpha :: K.A$

Terms

$\texttt{L,M} ::= \ldots \mid \Lambda\alpha :: K\texttt{.M} \mid \texttt{L [A]}$
$\textbf{pack}\ \texttt{B,M}\ \textbf{as}\ \exists\alpha :: K\texttt{.A}$
$\textbf{open}\ \texttt{M}\ \textbf{as}\ \alpha\texttt{,x}\ \textbf{in}\ \texttt{M'}$

Recap (core System F)

Types

$$\texttt{A,B} ::= \texttt{A} \rightarrow \texttt{B} \mid \alpha \mid \forall\alpha :: K.A$$

Terms

$$\texttt{L,M} ::= \texttt{x} \mid \lambda\texttt{x:A.M} \mid \texttt{L M} \mid \Lambda\alpha :: K\texttt{.M} \mid \texttt{L [A]}$$

A kind for **binary type operators**

$* \Rightarrow * \Rightarrow *$

A binary type operator

$\lambda\alpha{::}*.\lambda\beta{::}*.\alpha + \beta$

A kind for **higher-order type operators**

$(* \Rightarrow *) \Rightarrow * \Rightarrow *$

A higher-order type operator

$\lambda\phi{::}* \Rightarrow *.\lambda\alpha{::}*.\phi\ (\phi\ \alpha)$

Kind rules for System Fω

$$\frac{K_1 \text{ is a kind} \qquad K_2 \text{ is a kind}}{K_1 \Rightarrow K_2 \text{ is a kind}} \Rightarrow\text{-kind}$$

Kinding rules for System Fω

$$\frac{\Gamma, \alpha{::}K_1 \vdash A :: K_2}{\Gamma \vdash \lambda\alpha{::}K_1.A :: K_1 \Rightarrow K_2} \Rightarrow\text{-intro} \qquad \frac{\Gamma \vdash A :: K_1 \Rightarrow K_2 \quad \Gamma \vdash B :: K_1}{\Gamma \vdash A\ B :: K_2} \Rightarrow\text{-elim}$$

A **sum** data type:

```
type ('a, 'b) sum =
  Inl : 'a -> ('a, 'b) sum
| Inr : 'b -> ('a, 'b) sum
```

A **destructor** for sums:

```
val case :
 ('a,'b) sum -> ('a -> 'c) -> ('b -> 'c) -> 'c

let case s l r =
  match s with
    Inl x -> l x
  | Inr y -> r y
```

Encoding data types in System Fω: sums

We can finally **define** sums within the language.
As for $\mathbb{N}$ sums are represented as a binary polymorphic function:

$$\texttt{Sum} = \lambda\alpha.\lambda\beta.\forall\gamma.(\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$$

The **inl** and **inr** constructors are represented as functions:

$$\begin{aligned}
\textbf{inl} &= \Lambda\alpha.\Lambda\beta.\lambda \texttt{v}{:}\alpha.\Lambda\gamma. \\
&\qquad \lambda \texttt{l}{:}\alpha \to \gamma.\lambda \texttt{r}{:}\beta \to \gamma.\texttt{l v} \\
\textbf{inr} &= \Lambda\alpha.\Lambda\beta.\lambda \texttt{v}{:}\beta.\Lambda\gamma. \\
&\qquad \lambda \texttt{l}{:}\alpha \to \gamma.\lambda \texttt{r}{:}\beta \to \gamma.\texttt{r v}
\end{aligned}$$

The **foldSum** function behaves like **case**:

$$\begin{aligned}
&\texttt{foldSum} = \\
&\quad \Lambda\alpha.\Lambda\beta.\lambda \texttt{c}{:}\forall\gamma.(\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma.\texttt{c}
\end{aligned}$$

Encoding data types: sums (continued)

Of course, we can package the definition of **Sum** as an existential:

$$
\begin{array}{l}
\textbf{pack}\ \ \lambda\alpha.\lambda\beta.\forall\gamma.(\alpha\rightarrow\gamma)\rightarrow(\beta\rightarrow\gamma)\rightarrow\gamma\texttt{,}\\
\quad\quad \Lambda\alpha\texttt{.}\Lambda\beta\texttt{.}\lambda\texttt{v:}\alpha\texttt{.}\Lambda\gamma\texttt{.}\lambda\texttt{l:}\alpha\rightarrow\gamma\texttt{.}\lambda\texttt{r:}\beta\rightarrow\gamma\texttt{.l v}\\
\quad\quad\ \ \Lambda\alpha\texttt{.}\Lambda\beta\texttt{.}\lambda\texttt{v:}\beta\texttt{.}\Lambda\gamma\texttt{.}\lambda\texttt{l:}\alpha\rightarrow\gamma\texttt{.}\lambda\texttt{r:}\beta\rightarrow\gamma\texttt{.r v}\\
\quad\quad\quad \Lambda\alpha\texttt{.}\Lambda\beta\texttt{.}\lambda\texttt{c:}\forall\gamma.(\alpha\rightarrow\gamma)\rightarrow(\beta\rightarrow\gamma)\rightarrow\gamma\texttt{.c}\\
\quad \textbf{as}\ \ \exists\phi{::}*\Rightarrow*\Rightarrow*\texttt{.}\\
\quad\quad\quad \forall\alpha.\forall\beta.\alpha\rightarrow\phi\ \alpha\ \beta\\
\quad\quad \times\ \ \forall\alpha.\forall\beta.\beta\rightarrow\phi\ \alpha\ \beta\\
\quad\quad \times\ \ \forall\alpha.\forall\beta.\phi\ \alpha\ \beta\rightarrow\forall\gamma.(\alpha\rightarrow\gamma)\rightarrow(\beta\rightarrow\gamma)\rightarrow\gamma
\end{array}
$$

(However, the pack notation becomes unwieldy as our definitions grow.)

A **list** data type:

```
type 'a list =
    Nil : 'a list
  | Cons : 'a * 'a list -> 'a list
```

A **destructor** for lists:

```
val foldList :
 'a list -> 'b -> ('a -> 'b -> 'b) -> 'b

let rec foldList l n c =
    match l with
      Nil -> n
    | Cons (x, xs) -> c x (foldList xs n c)
```

Encoding data types in System F: lists

We can define **parameterised recursive types** such as lists in System Fω.
As for $\mathbb{N}$ lists are represented as a binary polymorphic function:

$$\texttt{List} = \lambda\alpha.\forall\phi{::}{*}\Rightarrow{*}.\phi\ \alpha \to (\alpha \to \phi\ \alpha \to \phi\ \alpha) \to \phi\ \alpha$$

The **nil** and **cons** constructors are represented as functions:

$$\texttt{nil} = \Lambda\alpha.\Lambda\phi{::}{*}\Rightarrow{*}.\lambda\texttt{n}{:}\phi\ \alpha.\lambda\texttt{c}{:}\alpha \to \phi\ \alpha \to \phi\ \alpha.\texttt{n}$$

$$\begin{aligned}\texttt{cons} = \Lambda\alpha.\lambda\texttt{x}{:}\alpha.\lambda\texttt{xs}{:}\texttt{List}\ \alpha.\\ \Lambda\phi{::}{*}\Rightarrow{*}.\lambda\texttt{n}{:}\phi\ \alpha.\lambda\texttt{c}{:}\alpha \to \phi\ \alpha \to \phi\ \alpha.\\ \texttt{c x (xs [}\phi\texttt{] n c)}\end{aligned}$$

The destructor corresponds to the `foldList` function:

$$\begin{aligned}\texttt{foldList} = \Lambda\alpha.\Lambda\beta.\lambda\texttt{c}{:}\alpha \to \beta \to \beta.\lambda\texttt{n}{:}\beta.\\ \lambda\texttt{l}{:}\texttt{List}\ \alpha.\texttt{l [}\lambda\gamma.\beta\texttt{] n c}\end{aligned}$$

Encoding data types: lists (continued)

We defined **add** for $\mathbb{N}$, and we can define **append** for lists:

```
append = Λα.
           λl:List α.
             λr:List α.
               foldList [α] [List α] l
                 r (cons [α])
```

A **regular** type:

```
type 'a tree =
  Empty : 'a tree
| Tree  : 'a tree * 'a * 'a tree -> 'a tree
```

A **non-regular** type:

```
type 'a perfect =
   ZeroP : 'a -> 'a perfect
 | SuccP : ('a * 'a) perfect -> 'a perfect
```

Encoding data types in System Fω: nested types

We can represent non-regular types like **perfect** in System Fω:

$$
\begin{aligned}
\texttt{Perfect} = {} & \lambda\alpha.\forall\phi{::}* \Rightarrow *. \\
& (\forall\alpha.\alpha \to \phi\ \alpha) \to \\
& (\forall\alpha.\phi\ (\alpha \times \alpha) \to \phi\ \alpha) \to \\
& \quad \phi\ \alpha
\end{aligned}
$$

This time the arguments to **zeroP** and **succP** are themselves polymorphic:

$$
\begin{aligned}
\texttt{zeroP} = {} & \Lambda\alpha\,.\,\lambda\texttt{x}{:}\alpha\,.\,\Lambda\phi{::}* \Rightarrow *. \\
& \lambda\texttt{z}{:}\forall\alpha.\alpha \to \phi\ \alpha\,.\,\lambda\texttt{s}{:}\forall\alpha.\phi\ (\alpha \times \alpha) \to \phi\ \alpha\,. \\
& \quad \texttt{z}\ [\alpha]\ \texttt{x}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{succP} = {} & \Lambda\alpha\,.\,\lambda\texttt{p}{:}\texttt{Perfect}\ (\alpha \times \alpha)\,.\,\Lambda\phi{::}* \Rightarrow *. \\
& \lambda\texttt{z}{:}\forall\alpha.\alpha \to \phi\ \alpha\,.\,\lambda\texttt{s}{:}\forall\beta.\phi\ (\beta \times \beta) \to \phi\ \beta\,. \\
& \quad \texttt{s}\ [\alpha]\ (\texttt{p}\ [\phi]\ \texttt{z}\ \texttt{s})
\end{aligned}
$$

Encoding data types in System Fω: Leibniz equality

Recall Leibniz's equality:

consider objects equal if they behave identically in any context

In System Fω:

$$\texttt{Eql} = \lambda\alpha.\lambda\beta.\forall\phi{::}* \Rightarrow *.\phi\ \alpha \rightarrow \phi\ \beta$$

Why might we want proofs of type equality?

Safe cast operations

```
val cast : ('a, 'b) eq -> 'a -> 'b
```

Flexible abstraction

```
module M : sig
  type t
  type s
  val unlock : secret:string -> (t, s) eq option
  (* ... *)
```

Constraints on the structure of values

```
val combine: ('n,'m) eq -> 'n tree -> 'm tree ->
 'm suc tree
```

Encoding data types in System Fω: Leibniz equality (cont.)

$$\texttt{Eql} = \lambda\alpha.\lambda\beta.\forall\phi{::}* \Rightarrow *.\phi\ \alpha \to \phi\ \beta$$

Equality is **reflexive** ($A \equiv A$):

$$\texttt{refl} : \forall\alpha.\texttt{Eql}\ \alpha\ \alpha$$
$$\texttt{refl} = \Lambda\alpha.\Lambda\phi{::}* \Rightarrow *.\lambda \texttt{x}{:}\phi\ \alpha.\texttt{x}$$

and **symmetric** ($A \equiv B \to B \equiv A$):

$$\texttt{symm} : \forall\alpha.\forall\beta.\texttt{Eql}\ \alpha\ \beta \to \texttt{Eql}\ \beta\ \alpha$$
$$\texttt{symm} = \Lambda\alpha.\Lambda\beta.$$
$$\lambda\texttt{e}{:}(\forall\phi{::}* \Rightarrow *.\phi\ \alpha\ \to\ \phi\ \beta).\texttt{e}\ [\lambda\gamma.\texttt{Eql}\ \gamma\ \alpha]\ (\texttt{refl}\ [\alpha])$$

and **transitive** ($(A \equiv B)\ \wedge\ (B \equiv C)\ \to\ (A \equiv C)$):

$$\texttt{trans} : \forall\alpha.\forall\beta.\forall\gamma.\texttt{Eql}\ \alpha\ \beta \to \texttt{Eql}\ \beta\ \gamma \to \texttt{Eql}\ \alpha\ \gamma$$
$$\texttt{trans} = \Lambda\alpha.\Lambda\beta.\Lambda\gamma.$$
$$\lambda\texttt{ab}{:}\texttt{Eql}\ \alpha\ \beta.\lambda\texttt{bc}{:}\texttt{Eql}\ \beta\ \gamma.\texttt{bc}\ [\texttt{Eql}\ \alpha]\ \texttt{ab}$$

Encoding existentials in System Fω

Encoding existentials in System Fω

(See exercise 1)

Terms and types from types and terms

	term parameters	**type parameters**
building terms	$\lambda x{:}A.M$ $A \to B$	$\Lambda\alpha{::}K.M$ $\forall\alpha{::}K.A$
building types		$\lambda\alpha{::}K.A$ $K_1 \Rightarrow K_2$

Terms and types from types and terms

	term parameters	**type parameters**
building terms	$\lambda x{:}A.M$	$\Lambda\alpha{::}K.M$
	$A \to B$	$\forall\alpha{::}K.A$
building types	$\lambda x{:}A.M$	$\lambda\alpha{::}K.A$
	$\Pi x{:}A.B$	$K_1 \Rightarrow K_2$

λC syntax (pseudo-terms)

$*$	(type of types)
Π`x:M.P`	(product type)
`x`	(variables)
λ`x:M.N`	(abstraction)
`M N`	(application)

Environment formation and product formation rules in λC

Judgements

$\Gamma \vdash \Delta$ (context Δ is valid in Δ)

$\Gamma \vdash M : P$ (term M has type P in context Δ)

Environment formation

$$\frac{}{\vdash *}\ \Gamma\text{-}\star \qquad \frac{\Gamma \vdash \Delta}{\Gamma, x : \Delta \vdash *}\ \Gamma\text{-}\Delta \qquad \frac{\Gamma \vdash M : *}{\Gamma, x{:}M \vdash *}\ \Gamma\text{-M}$$

Product formation

$$\frac{\Gamma, x{:}M \vdash \Delta}{\Gamma \vdash x{:}M, \Delta}\ \Pi\text{-}\Delta \qquad \frac{\Gamma, x{:}M \vdash N : *}{\Gamma \vdash \Pi x{:}M.N : *}\ \Pi\text{-}*$$

Telescopic contexts

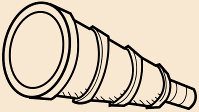
$\lambda^{\rightarrow}$: context **order** is **irrelevant** (no dependencies between bindings)

System F, System Fω: **term** bindings depend on **type** variables
e.g. $\Lambda\alpha.\lambda \mathtt{x}{:}\alpha.\mathtt{x}$ produces this environment:

$$\alpha :: *,\ \mathtt{x} : \alpha \vdash \mathtt{x} : \alpha$$

λ**C**: **term and type** bindings depend on **term and type** variables
e.g. $\lambda\alpha{:}*.\lambda \mathtt{P}{:}\alpha \rightarrow *.\lambda \mathtt{x}{:}\alpha.\lambda \mathtt{y}{:}\mathtt{P}\ \mathtt{x}.\mathtt{y}$ produces this environment:

$$\alpha : *,\ P : \alpha \rightarrow *,\ \mathtt{x} : \alpha,\ \mathtt{y} : \mathtt{P}\ \mathtt{x} \vdash \mathtt{y} : \mathtt{P}\ \mathtt{x}$$



Typing in λC

$$\frac{\Gamma, x : M, \Delta \vdash *}{\Gamma, x : M, \Delta \vdash x : M} \text{ tvar}$$

$$\frac{\Gamma, x{:}M \vdash N : P}{\Gamma \vdash \lambda x{:}M.N : \Pi x{:}M.P} \ \Pi\text{-intro}$$

$$\frac{\Gamma \vdash M : \Pi x{:}P.Q \qquad \Gamma \vdash N : P}{\Gamma \vdash M\ N : Q[x := N]} \ \Pi\text{-elim}$$

Typing in λC

$$\frac{\Gamma, x : M, \Delta \vdash *}{\Gamma, x : M, \Delta \vdash x : M}\ \textsf{tvar}$$

$$\frac{\Gamma, x{:}M \vdash N : P}{\Gamma \vdash \lambda x{:}M.N : \Pi x{:}M.P}\ \Pi\textsf{-intro}$$

bound variables appear in types

$$\frac{\Gamma \vdash M : \Pi x{:}P.Q \qquad \Gamma \vdash N : P}{\Gamma \vdash M\ N : Q[x := N]}\ \Pi\textsf{-elim}$$

arguments substituted into types

The generality of Π: $\rightarrow$ and $\forall$ in λC

Π subsumes $\rightarrow$ and $\forall$. Example: the identity function:

In System F	$\Lambda\alpha::*.\lambda\mathtt{x}:\alpha.\mathtt{x}$	has type	$\forall\alpha.\alpha\rightarrow\alpha$
In λC	$\lambda\alpha:*.\lambda\mathtt{x}:\alpha.\mathtt{x}$	has type	$\Pi\alpha:*.\Pi\mathtt{x}:\alpha.\alpha$

Type abbreviations

$\forall\alpha.\mathtt{B}$	abbreviates	$\Pi\alpha:*.\mathtt{B}$	
$\mathtt{A}\rightarrow\mathtt{B}$	abbreviates	$\Pi\mathtt{x}:A.\mathtt{B}$	(if $\mathtt{x}\notin\mathtt{fv(B)}$)

The generality of Π: type operators in λC

Π subsumes System Fω's λ Example: abstracting $\rightarrow$:

In System Fω	$\lambda\alpha.\lambda\beta.\alpha \rightarrow \beta$	has kind	$* \Rightarrow * \Rightarrow *$
In λC	$\lambda\alpha.\lambda\beta.\Pi x{:}\alpha.\beta$	has type	$\Pi\alpha{:}*.\Pi\beta{:}*.*$

Type abbreviations

$* \rightarrow * \rightarrow *$	abbreviates	$\Pi\alpha{:}*.\Pi\beta{:}*.*$
$\forall\alpha.\forall\beta.*$	abbreviates	$\Pi\alpha{:}*.\Pi\beta{:}*.*$

Equality between terms

$$\texttt{Eql} = \lambda\alpha.\lambda\texttt{x}{:}\alpha.\lambda\texttt{y}{:}\alpha.\Pi\texttt{P}{:}\alpha \to *.\texttt{P x} \to \texttt{P y}$$

Equality **proofs** have the same structure as in System Fω:

$$\texttt{refl} : \forall\alpha.\Pi\texttt{x}{:}\alpha.\texttt{Eql}\ \alpha\ \texttt{x x}$$
$$\texttt{refl} = \lambda\alpha.\lambda\texttt{x}{:}\alpha.\lambda\texttt{P}{:}\alpha \to *.\lambda\texttt{p}{:}\texttt{P x}.\texttt{p}$$

Why might we want proofs of term equality?

Term equality can represent facts about the **behaviour** of programs.

In general, types that mention terms act as **propositions** about programs:

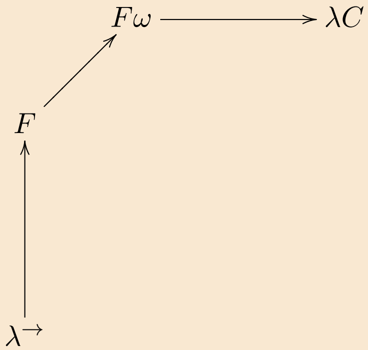
$$\texttt{compare} : \Pi\texttt{m}:\mathbb{N}.\Pi\texttt{n}:\mathbb{N}.$$
$$\texttt{Lt m n} \vee \texttt{Eq m n} \vee \texttt{Gt m n}$$

$$\texttt{append} : \forall\alpha.\Pi\texttt{n}:\mathbb{N}.\Pi\texttt{m}:\mathbb{N}.$$
$$\texttt{Seq m } \alpha \rightarrow \texttt{Seq n } \alpha \rightarrow \texttt{Seq (m+n) } \alpha$$

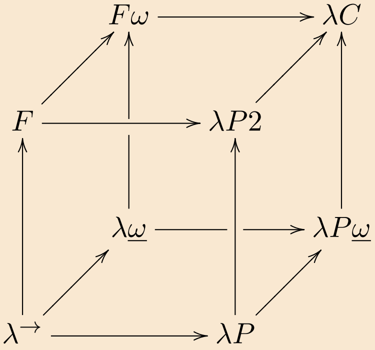
$$\texttt{sr} : \Pi\texttt{e}:\texttt{Expr}.\Pi\texttt{e'}:\texttt{Expr}.\Pi\texttt{t}:\texttt{Type}.$$
$$\texttt{HasType e t} \rightarrow \texttt{ReducesTo e e'} \rightarrow \texttt{HasType e' t}$$

(NB: to prove some of these propositions λC must be extended with support for induction — i.e. we need the Calculus of *Inductive* Constructions)

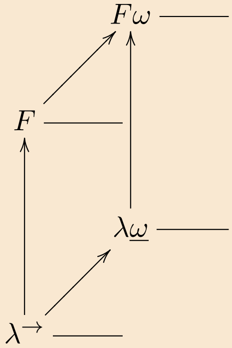
The roadmap again



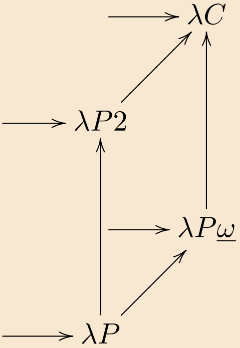
The lambda cube



Programming on the left face of the cube



Functional programming



Dependently-typed programming