Last time:

Simply typed lambda calculus

$A \to B$ $\quad \lambda x{:}A.M$ $\quad M\ N$

... with products

$A \times B$ $\quad \langle M, N \rangle$ $\quad$ **fst** M $\quad$ **snd** M

... and sums

$A + B$ $\quad$ **inl** M $\quad$ **inr** M $\quad$ **case** L **of** x.M | y.N

Polymorphic lambda calculus

$\forall\alpha{::}K.A$ $\quad \Lambda\alpha{::}K.M$ $\quad M\ [A]$

... with existentials

$\exists\alpha{::}K.A$ $\quad$ **pack** B,M **as** $\exists\alpha{::}K.A$ $\quad$ **open** L **as** α,x **in** M

Typing rules for existentials

$$\frac{\Gamma \vdash M : A[\alpha := B] \qquad \Gamma \vdash \exists\alpha{::}K.A :: *}{\Gamma \vdash \mathsf{pack}\ B, M\ \mathsf{as}\ \exists\alpha{::}K.A : \exists\alpha{::}K.A}\ \exists\text{-intro}$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \exists\alpha{::}K.A \\ \Gamma, \alpha{::}K, x : A \vdash M' : B\end{array}}{\Gamma \vdash \mathsf{open}\ M\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ M' : B}\ \exists\text{-elim}$$

Unit in OCaml

```
type u = Unit
```

Encoding data types in System F: unit

The **unit** type has **one inhabitant**.

We can **represent** it as the type of the **identity function**.

$$\mathsf{Unit} = \forall\alpha{::}*.\,\alpha\to\alpha$$

The unit value is the single inhabitant:

$$\mathsf{unit} = \Lambda\alpha.\,\lambda a{:}\alpha.\,a$$

We can package the type and value as an **existential**:

$$\textbf{pack}\ (\forall\alpha{::}*.\,\alpha\to\alpha,\ \Lambda\alpha.\,\lambda a{:}\alpha.\,a)\ \textbf{as}\ \exists U{::}*.\,u$$

We'll write 1 for the unit type and $\langle\rangle$ for its inhabitant.

Booleans in OCaml

A boolean data type:

```
type bool = False | True
```

A destructor for bool:

```
val _if_ : bool -> 'a -> 'a -> 'a

let _if_ b _then_ _else_ =
  match b with
    False -> _else_
  | True -> _then_
```

Encoding data types in System F: booleans

The **boolean** type has two inhabitants: **false** and **true**.

We can **represent** it using sums and unit.

```
Bool = 1+1
```

The constructors are represented as injections:

```
false = inl [1] ⟨⟩
true  = inr [1] ⟨⟩
```

The destructor (**if**) is implemented using **case**:

```
λb:Bool.
   Λα::*.
      λr:α.
         λs:α.case b of x.s | y.r
```

Encoding data types in System F: booleans

We can package the definition of booleans as an existential:

```
pack (1+1,
        ⟨inr [1] ⟨⟩,
        ⟨inl [1] ⟨⟩,
         λb:Bool.
            Λα::*.
               λr:α.
                  λs:α.
                     case b of x.s | y.r⟩⟩)
    as ∃β::*.
        β ×
        β ×
        (β → ∀α::*.α → α.α)
```

Natural numbers in OCaml

A `nat` data type

```
type nat =
    Zero : nat
  | Succ : nat -> nat
```

A destructor for `nat`:

```
val foldNat : nat -> 'a -> ('a -> 'a) -> 'a

let rec foldNat n z s =
    match n with
      Zero -> z
    | Succ n -> s (foldNat n z s)
```

Encoding data types in System F: natural numbers

The type of **natural numbers** is inhabited by **Z**, **SZ**, **SSZ**, …
We can represent it using a polymorphic function of two parameters:

$$\mathbb{N} = \forall\alpha::*.\,\alpha \to (\alpha \to \alpha) \to \alpha$$

The **Z** and **S** constructors are represented as functions:

$$\begin{aligned}
\mathsf{z} &: \mathbb{N} \\
\mathsf{z} &= \Lambda\alpha::*.\,\lambda \mathsf{z}:\alpha.\,\lambda \mathsf{s}:\alpha \to \alpha.\,\mathsf{z} \\
\mathsf{s} &: \mathbb{N} \to \mathbb{N} \\
\mathsf{s} &= \lambda \mathsf{n}:\forall\alpha::*.\,\alpha \to (\alpha \to \alpha) \to \alpha. \\
&\qquad \Lambda\alpha::*.\,\lambda \mathsf{z}:\alpha.\,\lambda \mathsf{s}:\alpha \to \alpha.\,\mathsf{s}\ (\mathsf{n}\ [\alpha]\ \mathsf{z}\ \mathsf{s}),
\end{aligned}$$

The fold$\mathbb{N}$ destructor allows us to analyse natural numbers:

$$\begin{aligned}
\mathsf{fold}\mathbb{N} &: \mathbb{N} \to \forall\alpha.\,\alpha \to (\alpha \to \alpha) \to \alpha \\
\mathsf{fold}\mathbb{N} &= \lambda \mathsf{n}:\forall\alpha::*.\,\alpha \to (\alpha \to \alpha) \to \alpha.\,\mathsf{n}
\end{aligned}$$

Encoding data types: natural numbers (continued)

$$\mathsf{fold}\mathbb{N} \; : \; \mathbb{N} \rightarrow \forall\alpha.\alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha$$

For example, we can use fold$\mathbb{N}$ to write a function to test for zero:

$$\lambda \mathsf{n}:\mathbb{N}.\,\mathsf{fold}\mathbb{N}\ \mathsf{n}\ [\mathsf{Bool}]\ \ \mathsf{true}\ \ (\lambda \mathsf{b}:\mathsf{Bool}.\,\mathsf{false})$$

Or we could instantiate the type parameter with $\mathbb{N}$ and write an addition function:

$$\lambda \mathsf{m}:\mathbb{N}.\,\lambda \mathsf{n}:\mathbb{N}.\,\mathsf{fold}\mathbb{N}\ \mathsf{m}\ [\mathbb{N}]\ \ \mathsf{n}\ \ \mathsf{succ}$$

Encoding data types: natural numbers (concluded)

Of course, we can package the definition of $\mathbb{N}$ as an existential:

$$
\begin{aligned}
&\textbf{pack}\ \ (\forall\alpha::*.\,\alpha\to(\alpha\to\alpha)\to\alpha\,,\\
&\qquad\langle\Lambda\alpha::*.\,\lambda\mathsf{z}:\alpha.\,\lambda\mathsf{s}:\alpha\to\alpha.\,\mathsf{z}\,,\\
&\qquad\langle\lambda\mathsf{n}:\forall\alpha::*.\,\alpha\to(\alpha\to\alpha)\to\alpha\,.\\
&\qquad\quad\Lambda\alpha::*.\,\lambda\mathsf{z}:\alpha.\,\lambda\mathsf{s}:\alpha\to\alpha.\,\mathsf{s}\ \ (\mathsf{n}\ \ [\alpha]\ \ \mathsf{z}\ \ \mathsf{s})\,,\\
&\qquad\langle\lambda\mathsf{n}:\forall\alpha::*.\,\alpha\to(\alpha\to\alpha)\to\alpha\,.\,\mathsf{n}\rangle\rangle\rangle)\\
&\quad\textbf{as}\ \ \exists\mathbb{N}::*.\\
&\qquad\ \mathbb{N}\ \ \times\\
&\qquad\ (\mathbb{N}\to\mathbb{N})\ \ \times\\
&\qquad\ (\mathbb{N}\to\forall\alpha.\,\alpha\to(\alpha\to\alpha)\to\alpha)
\end{aligned}
$$

System Fω

(polymorphism + type abstraction)

System Fω by example

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

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

Sums in OCaml

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


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:

$$\mathsf{Sum} = \lambda\alpha::*.\lambda\beta::*.\forall\gamma::*.(\alpha\to\gamma)\to(\beta\to\gamma)\to\gamma$$

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

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

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

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

Encoding data types: sums (continued)

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

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

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

Lists in OCaml

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:

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

The destructor corresponds to the foldList function:

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

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])
```

Nested types in OCaml

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:

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

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

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

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

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:

$$\mathsf{Eq} \;=\; \lambda\alpha::*.\,\lambda\beta::*.\,\forall\phi::*\Rightarrow*.\,\phi\,\alpha\to\phi\,\beta$$

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

$$\mathsf{refl} \;=\; \Lambda\alpha::*.\,\Lambda\phi::*\Rightarrow*.\,\lambda x:\phi\,\alpha.\,x$$

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

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

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

$$\mathsf{trans} \;=\; \Lambda\alpha::*.\,\Lambda\beta::*.\,\Lambda\gamma::*.$$
$$\lambda ab:\mathsf{Eq}\;\alpha\;\beta.\,\lambda bc:\mathsf{Eq}\;\beta\;\gamma.\,bc\;[\mathsf{Eq}\;\alpha]\;ab$$

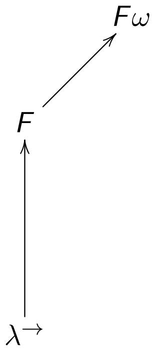
Abstract what where?

	abstract terms	**abstract types**
build terms	$A \to B$ $\lambda x : A.M$	$\forall\alpha{::}K.A$ $\Lambda\alpha{::}K.M$
build types		$K_1 \Rightarrow K_2$ $\lambda\alpha{::}K.A$

Abstract what where?

	abstract terms	**abstract types**
build terms	$A \to B$ $\lambda x : A.M$	$\forall \alpha{::}K.A$ $\Lambda \alpha{::}K.M$
build types	$\Pi x : A.K$ $\Pi x : A.B$	$K_1 \Rightarrow K_2$ $\lambda \alpha{::}K.A$

The roadmap again



The lambda cube