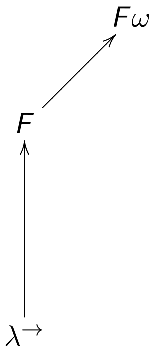

$F\omega$
F
$\lambda^{\rightarrow}$

Last time:

Simply typed lambda calculus

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

... with products

$A \times B$ $\langle M,\ N \rangle$ $\mathtt{fst}\ M$ $\mathtt{snd}\ M$

... and sums

$A + B$ $\mathtt{inl}\ [B]\ M$ $\mathtt{inr}\ [A]\ M$ $\mathtt{case}\ L\ \mathtt{of}\ x.M\ |\ y.N$

Polymorphic lambda calculus

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

... with existentials

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

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**.

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

The unit value is the single inhabitant:

$\texttt{Unit} = \Lambda\alpha{::}{*}.\lambda \texttt{a}{:}\alpha.\texttt{a}$

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

$$\begin{aligned}&\texttt{pack}\ (\forall\alpha{::}{*}.\alpha \to \alpha,\\ &\qquad \Lambda\alpha{::}{*}.\lambda \texttt{a}{:}\alpha.\texttt{a})\\ &\quad \texttt{as}\ \exists U :: {*}.u\end{aligned}$$

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:

$$
\begin{array}{l}
\texttt{pack (1+1,} \\
\qquad \langle \texttt{inr [1]} \; \langle\rangle, \\
\qquad \langle \texttt{inl [1]} \; \langle\rangle, \\
\qquad\; \lambda \texttt{b}:\mathit{Bool}. \\
\qquad\qquad \Lambda\alpha::*. \\
\qquad\qquad\quad \lambda \texttt{r}:\alpha. \\
\qquad\qquad\qquad \lambda \texttt{s}:\alpha. \\
\qquad\qquad\qquad\quad \texttt{case b of x.s | y.r}\rangle\rangle\texttt{)} \\
\quad \texttt{as } \exists\beta::*. \\
\qquad\; \beta \; \times \\
\qquad\; \beta \; \times \\
\qquad\; (\beta \to \forall\alpha::*.\alpha \to \alpha \to \alpha)
\end{array}
$$

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:

```
z : ℕ
z = Λα::∗.λz:α.λs:α → α.z
s : ℕ → ℕ
s = λn:∀α::∗.α → (α → α) → α.
      Λα::∗.λz:α.λs:α → α.s (n [α] z s),
```

The foldℕ destructor allows us to analyse natural numbers:

```
foldℕ : ℕ → ∀α::∗.α → (α → α) → α
foldℕ = λn:∀α::∗.α → (α → α) → α.n
```

Encoding data types: natural numbers (continued)

$\texttt{fold}\mathbb{N} : \mathbb{N} \to \forall\alpha{::}{*}.\alpha \to (\alpha \to \alpha) \to \alpha$

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

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

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

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

Encoding data types: natural numbers (concluded)

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

$$
\begin{array}{l}
\texttt{pack}\ (\forall\alpha{::}*.\alpha \to (\alpha \to \alpha) \to \alpha\,\texttt{,}\\
\quad\langle\Lambda\alpha{::}*\texttt{.}\lambda\texttt{z}\,\texttt{:}\,\alpha\,\texttt{.}\lambda\texttt{s}\,\texttt{:}\,\alpha \to \alpha\,\texttt{.z,}\\
\quad\langle\lambda\texttt{n}\,\texttt{:}\,\forall\alpha{::}*.\alpha \to (\alpha \to \alpha) \to \alpha\,\texttt{.}\\
\quad\quad\Lambda\alpha{::}*\texttt{.}\lambda\texttt{z}\,\texttt{:}\,\alpha\,\texttt{.}\lambda\texttt{s}\,\texttt{:}\,\alpha \to \alpha\,\texttt{.s (n [}\alpha\texttt{] z s),}\\
\quad\langle\lambda\texttt{n}\,\texttt{:}\,\forall\alpha{::}*.\alpha \to (\alpha \to \alpha) \to \alpha\,\texttt{.n}\rangle\rangle\rangle\texttt{)}\\
\quad\texttt{as}\ \exists\mathbb{N}{::}*.\\
\quad\quad\mathbb{N}\ \times\\
\quad\quad(\mathbb{N} \to \mathbb{N})\ \times\\
\quad\quad(\mathbb{N} \to \forall\alpha{::}*.\alpha \to (\alpha \to \alpha) \to \alpha)
\end{array}
$$

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

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

$$\frac{\Gamma \vdash A :: K_1 \Rightarrow K_2 \qquad \Gamma \vdash B :: K_1}{\Gamma \vdash A\ B :: K_2} \Rightarrow\text{-elim}$$

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:

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

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

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

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:

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

(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:

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

Encoding data types: lists (continued)

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

$$\begin{array}{l}\texttt{append} = \Lambda\alpha{::}*. \\ \quad\quad \lambda\texttt{l}:\texttt{List}\ \alpha.\lambda\texttt{r}:\texttt{List}\ \alpha. \\ \quad\quad\quad \texttt{foldList}\ [\alpha]\ [\texttt{List}\ \alpha] \\ \quad\quad\quad\quad \texttt{l}\ \texttt{r}\ (\texttt{cons}\ [\alpha])\end{array}$$

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:

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

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:

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

Encoding data types in System Fω: Leibniz equality (continued)

```
Eq = λα::*.λβ::*.∀φ::*⇒*.φ α → φ β
```

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

```
refl : ∀α::*.Eql α α
refl = Λα::*.Λφ::*⇒*.λx:φ α.x
```

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

```
symm : ∀α::*.∀β::*.Eql α β → Eql β α
symm = Λα::*.Λβ::*.
 λe:(∀φ::*⇒*.φ α  →  φ β).e [λγ::*.Eq γ α] (refl [α])
```

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

```
trans : ∀α::*.∀β::*.∀γ::*.Eql α β → Eql β γ → Eql α γ
trans = Λα::*.Λβ::*.Λγ::*.
  λab:Eq α β.λbc:Eq β γ.bc  [Eq α] ab
```

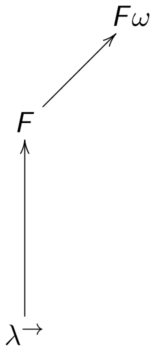
Terms and types from types and terms

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

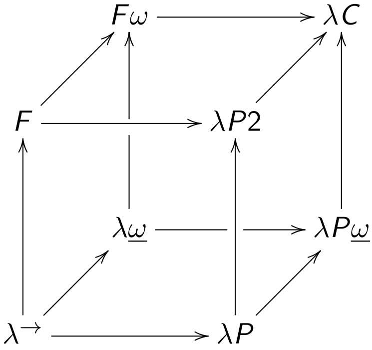
Terms and types from types and terms

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

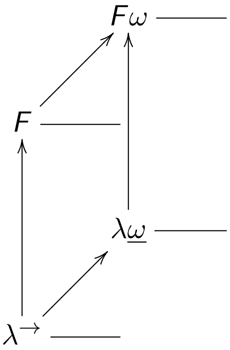
The roadmap again



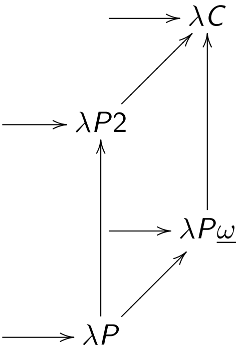
The lambda cube



Programming on the left face of the cube



Functional programming



Dependently-typed programming