$$
\begin{tikzcd}
 & F\omega \\
F \arrow[ur] & \\
\lambda{\to} \arrow[u] &
\end{tikzcd}
$$

Last time:

Simply typed lambda calculus

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

… with products

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

… and sums

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

Polymorphic lambda calculus

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

… with existentials

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

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**.

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

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**:

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

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.

$\texttt{Bool} = 1 + 1$

The constructors are represented as injections:

$\texttt{false = inl [}1\texttt{]}\ \langle\rangle$
$\texttt{true = inr [}1\texttt{]}\ \langle\rangle$

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

$\lambda\texttt{b:Bool.}$
$\quad\Lambda\alpha{::}{*}.$
$\quad\quad\lambda\texttt{r:}\alpha.$
$\quad\quad\quad\lambda\texttt{s:}\alpha\texttt{.case b of x.s | y.r}$

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:

z : $\mathbb{N}$
z = $\Lambda\alpha{::}*.\lambda$z:$\alpha$.$\lambda$s:$\alpha \to \alpha$.z
s : $\mathbb{N} \to \mathbb{N}$
s = λn:$\forall\alpha{::}*.\alpha \to (\alpha \to \alpha) \to \alpha$.
    $\Lambda\alpha{::}*.\lambda$z:$\alpha$.$\lambda$s:$\alpha \to \alpha$.s (n [$\alpha$] z s)

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

fold$\mathbb{N}$: $\mathbb{N} \to \forall\alpha{::}*.\alpha \to (\alpha \to \alpha) \to \alpha$
fold$\mathbb{N}$ = λn:$\forall\alpha{::}*.\alpha \to (\alpha \to \alpha) \to \alpha$.n

Encoding data types: natural numbers (continued)

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

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

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

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

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

Encoding data types: natural numbers (concluded)

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

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

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

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

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:

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

(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:

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

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:

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

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

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

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

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:

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

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

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

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

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

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

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

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

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

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	$A \to B$ $\lambda x : A.M$	$\forall \alpha :: K.A$ $\Lambda \alpha :: K.M$
building types	$\Pi x : A.K$ $\Pi x : A.B$	$K_1 \Rightarrow K_2$ $\lambda \alpha :: K.A$

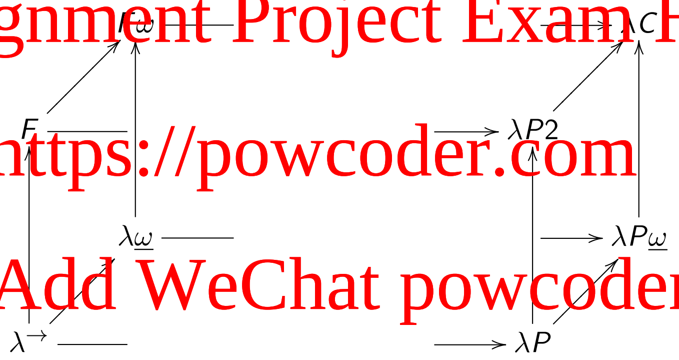
The roadmap again

$$\begin{tikzcd} & F\omega \\ F \arrow[ur] & \\ \lambda^{\rightarrow} \arrow[u] & \end{tikzcd}$$

The lambda cube

$$\begin{tikzcd}
 & F\omega \arrow[rr] & & \lambda C \\
F \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda^{\rightarrow} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Programming on the left face of the cube



Functional programming

Dependently-typed programming