Lambda calculus, part I

January 2018

Motivation & background

System Fω

Function composition in OCaml:

```
fun f g x -> f (g x)
```

Function composition in System Fω:

$$
\begin{array}{l}
\Lambda\alpha{::}*.\\
\quad\Lambda\beta{::}*.\\
\quad\quad\Lambda\gamma{::}*.\\
\quad\quad\quad\lambda\mathtt{f}{:}\alpha\to\beta.\\
\quad\quad\quad\quad\lambda\mathtt{g}{:}\gamma\to\alpha.\\
\quad\quad\quad\quad\quad\lambda\mathtt{x}{:}\gamma.\mathtt{f\ (g\ x)}
\end{array}
$$

What's the point of System F, System Fω, &c.?

Frameworks for understanding language features and programming patterns:

- the elaboration language for **type inference** (lecture 2)
- the proof system for reasoning with **propositional & predicate logic** (lecture 3)
- the background for **parametricity properties** (lectures 4-5)
- the elaboration language for **modules** (lectures 4-5)
- the core calculus for **indexed data** (lectures 8-9)
- an elaboration language for **implicits** (lecture 10)
- a foundation for **multi-stage programming** (lectures 14-15)

Roadmap

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

Inference rules

$$\frac{\begin{array}{c}\text{premise 1}\\ \text{premise 2}\\ \cdots\\ \text{premise N}\end{array}}{\text{conclusion}}\ \text{rule name}$$

Inference rules

$$\frac{\begin{array}{c}\text{premise 1}\\ \text{premise 2}\\ \cdots\\ \text{premise N}\end{array}}{\text{conclusion}}\ \text{rule name}$$

$$\frac{\begin{array}{c}\text{all } M \text{ are } P\\ \text{all } S \text{ are } M\end{array}}{\text{all } S \text{ are } P}\ \text{modus barbara}$$

Inference rules

$$\frac{\begin{array}{c}\text{premise 1}\\ \text{premise 2}\\ \cdots\\ \text{premise N}\end{array}}{\text{conclusion}}\ \text{rule name}$$

$$\frac{\begin{array}{c}\text{all } M \text{ are } P\\ \text{all } S \text{ are } M\end{array}}{\text{all } S \text{ are } P}\ \text{modus barbara}$$

$$\frac{\begin{array}{c}\text{all programs are buggy}\\ \text{all functional programs are programs}\end{array}}{\text{all functional programs are buggy}}\ \text{modus barbara}$$

Typing rules

$$\frac{\begin{array}{c}\Gamma \vdash M : A \to B \\ \Gamma \vdash N : A\end{array}}{\Gamma \vdash M\ N : B} \to\text{-elim}$$

Terms, types, kinds

Kinds: K, K_1, K_2, . . .

K is a kind

Environments: Γ

Γ is an environment

Types: A, B, C, . . .

$\Gamma \vdash A :: K$

Terms: L, M, N, . . .

$\Gamma \vdash M : A$

Simply-typed lambda calculus by example

In $\lambda^{\rightarrow}$:

```
λx:A.x

λf:B → C.
   λg:A → B.
      λx:A.f (g x)
```

In OCaml:

```
fun x -> x


fun f g x -> f (g x)
```

Kinds in $\lambda^{\rightarrow}$

$$\frac{}{\ast \text{ is a kind}}\ \ast\text{-kind}$$

Kinding rules (type formation) in $\lambda^{\rightarrow}$

$$\frac{}{\Gamma \vdash \mathcal{B} :: *}\ \textsf{kind-}\mathcal{B}$$

$$\frac{\Gamma \vdash A :: * \qquad \Gamma \vdash B :: *}{\Gamma \vdash A \rightarrow B :: *}\ \textsf{kind-}\rightarrow$$

A kinding derivation

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathcal{B} :: *}\,\text{kind-}\mathcal{B} \qquad \dfrac{}{\Gamma \vdash \mathcal{B} :: *}\,\text{kind-}\mathcal{B}}{\Gamma \vdash \mathcal{B} \to \mathcal{B} :: *}\,\text{kind-}\to \qquad \dfrac{}{\Gamma \vdash \mathcal{B} :: *}\,\text{kind-}\mathcal{B}}{\Gamma \vdash (\mathcal{B} \to \mathcal{B}) \to \mathcal{B} :: *}\,\text{kind-}\to$$

Environment formation rules

$$\frac{}{\cdot \text{ is an environment}}\;\Gamma\text{-}\cdot$$

$$\frac{\Gamma \text{ is an environment} \qquad \Gamma \vdash A :: *}{\Gamma, x{:}A \text{ is an environment}}\;\Gamma\text{-}{:}$$

Typing rules (term formation) in $\lambda^{\rightarrow}$

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A}\ \textsf{tvar}$$

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda \mathsf{x}{:}A.M : A \to B}\ \to\textsf{-intro}$$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ \to\textsf{-elim}$$

A typing derivation for the identity function

$$\dfrac{\dfrac{x{:}A \in (\cdot,\, x{:}A)}{\cdot,\, x{:}A \vdash x : A}\;\textsf{tvar}}{\cdot \vdash \lambda \mathsf{x}{:}A.x : A \to A}\;\to\text{-}\textsf{intro}$$

Products by example

In $\lambda^{\to}$ with products:	**In OCaml**:
$\lambda p{:}(A \to B) \times A.$ $\quad \mathbf{fst}\ p\ (\mathbf{snd}\ p)$	`fun (f,p) -> f p`
$\lambda x{:}A.\langle x, x\rangle$	`fun x -> (x, x)`
$\lambda f{:}A \to C.$ $\quad \lambda g{:}B \to C.$ $\quad\quad \lambda p{:}A \times B.$ $\quad\quad\quad \langle f\ (\mathbf{fst}\ p),$ $\quad\quad\quad\ \ g\ (\mathbf{snd}\ p)\rangle$	`fun f g (x,y) -> (f x, g y)`
$\lambda p.A \times B.\langle \mathbf{snd}\ p, \mathbf{fst}\ p\rangle$	`fun (x,y) -> (y,x)`

Kinding and typing rules for products

$$\frac{\Gamma \vdash A :: * \qquad \Gamma \vdash B :: *}{\Gamma \vdash A \times B :: *}\ \textsf{kind-}\times$$

$$\frac{\begin{array}{c}\Gamma \vdash M : A \\ \Gamma \vdash N : B\end{array}}{\Gamma \vdash \langle M, N \rangle : A \times B}\ \times\textsf{-intro}$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathbf{fst}\ M : A}\ \times\textsf{-elim-1}$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathbf{snd}\ M : B}\ \times\textsf{-elim-2}$$

Sums by example

In $\lambda^{\rightarrow}$ with sums:

```
λf:A → C.
  λg:B → C.
    λs:A + B.
      case s of
       x.f x
     | y.g y
```

```
λs:A + B.
  case s of
    x.inr [B] x
  | y.inl [A] y
```

In OCaml:

```
fun f g s ->
  match s with
    Inl x -> f x
  | Inr y -> g y
```

```
function
  Inl x -> Inr x
| Inr y -> Inl y
```

Kinding and typing rules for sums

$$\frac{\Gamma \vdash A :: * \qquad \Gamma \vdash B :: *}{\Gamma \vdash A + B :: *}\ \textsf{kind-}+$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \textsf{inl}\ [B]\ M : A + B}\ +\textsf{-intro-1}$$

$$\frac{\Gamma \vdash N : B}{\Gamma \vdash \textsf{inr}\ [A]\ N : A + B}\ +\textsf{-intro-2}$$

$$\frac{\begin{array}{c}\Gamma \vdash L : A + B \\ \Gamma, x{:}A \vdash M : C \\ \Gamma, y{:}B \vdash N : C\end{array}}{\Gamma \vdash \textsf{case}\ L\ \textsf{of}\ x.M \mid y.N : C}\ +\textsf{-elim}$$

System F by example

$\Lambda\alpha{::}{*}.\lambda \mathtt{x}{:}\alpha.\mathtt{x}$

$$
\begin{array}{l}
\Lambda\alpha{::}{*}.\\
\quad\Lambda\beta{::}{*}.\\
\quad\quad\Lambda\gamma{::}{*}.\\
\quad\quad\quad\lambda \mathtt{f}{:}\beta\to\gamma.\\
\quad\quad\quad\quad\lambda \mathtt{g}{:}\alpha\to\beta.\\
\quad\quad\quad\quad\quad\lambda \mathtt{x}{:}\alpha.\mathtt{f\ (g\ x)}
\end{array}
$$

$\Lambda\alpha{::}{*}.\Lambda\beta{::}{*}.\lambda \mathtt{p}{:}(\alpha\to\beta)\times\alpha.\,\mathbf{fst}\ \mathtt{p}\ (\mathbf{snd}\ \mathtt{p})$

New kinding rules for System F

$$\frac{\Gamma, \alpha{::}K \vdash A :: *}{\Gamma \vdash \forall \alpha{::}K.A :: *}\ \textsf{kind-}\forall$$

$$\frac{\alpha{::}K \in \Gamma}{\Gamma \vdash \alpha :: K}\ \textsf{tyvar}$$

New environment rule for System F

$$\frac{\Gamma \text{ is an environment} \qquad K \text{ is a kind}}{\Gamma, \alpha{::}K \text{ is an environment}}\ \Gamma\text{-::}$$

New typing rules for System F

$$\frac{\Gamma, \alpha{::}K \vdash M : A}{\Gamma \vdash \Lambda\alpha{::}K.M : \forall\alpha{::}K.A}\ \forall\text{-intro}$$

$$\frac{\Gamma \vdash M : \forall\alpha{::}K.A \qquad \Gamma \vdash B :: K}{\Gamma \vdash M\ [B] : A[\alpha{::}{=}B]}\ \forall\text{-elim}$$

What's the point of existentials?

- ∀ and ∃ in logic are closely connected to polymorphism and existentials in type theory
- As in logic, ∀ and ∃ for types are closely related to each other
- Module types can be viewed as a kind of existential type
- OCaml's variant types now support existential variables

Existential intuition

Existentials
correspond to
abstract types

Kinding rules for existentials

$$\frac{\Gamma, \alpha{::}K \vdash A :: *}{\Gamma \vdash \exists\alpha{::}K.A :: *}\ \textsf{kind-}\exists$$

Typing rules for existentials

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

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

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**.

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

The unit value is the single inhabitant:

`Unit` = $\Lambda\alpha{::}*.\,\lambda\mathtt{a}{:}\alpha.\,\mathtt{a}$

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

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

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:

```
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)
```

ℕ 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: $\mathbb{N}$

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

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

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

$$\begin{array}{lll}
\texttt{z} & \texttt{:} & \mathbb{N} \\
\texttt{z} & \texttt{=} & \Lambda\alpha{::}*.\lambda\texttt{z}{:}\alpha.\lambda\texttt{s}{:}\alpha \to \alpha.\texttt{z} \\
\texttt{s} & \texttt{:} & \mathbb{N} \to \mathbb{N} \\
\texttt{s} & \texttt{=} & \lambda\texttt{n}{:}\forall\alpha{::}*.\alpha \to (\alpha \to \alpha) \to \alpha. \\
 & & \quad\Lambda\alpha{::}*.\lambda\texttt{z}{:}\alpha.\lambda\texttt{s}{:}\alpha \to \alpha.\texttt{s (n [}\alpha\texttt{] z s),}
\end{array}$$

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

$$\begin{array}{lll}
\texttt{fold}\mathbb{N} & \texttt{:} & \mathbb{N} \to \forall\alpha{::}*.\alpha \to (\alpha \to \alpha) \to \alpha \\
\texttt{fold}\mathbb{N} & \texttt{=} & \lambda\texttt{n}{:}\forall\alpha{::}*.\alpha \to (\alpha \to \alpha) \to \alpha.\texttt{n}
\end{array}$$

Encoding data types in System F: $\mathbb{N}$ (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 in System F: $\mathbb{N}$ (concluded)

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

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

Next time

$$\Gamma \vdash M : \ ?$$