System F

Polymorphic Functions

Functions that operate "in the same way" on different data types

Example:

$$\text{Length}_A : \text{List}_A \longrightarrow \text{Nat}$$

the definition of length doesn't depend on A

$$\text{length}_A\ \text{nil} = 0$$
$$\text{Length}_A\ (a :: \ell) = (\text{length}_A\ \ell) + 1$$

Using the elimination rule for lists:

$$\text{length}_A = \text{recList}_A\ (\lambda a.\lambda \ell.\,\text{succ})\ \text{zero}$$

For every type A, we must define a separate length function

But they're all defined in the same way

We would like to say that a single length function has many types:

$$\text{length} \in \forall T\,\text{type}.\,\text{List}_T \to \text{Nat}$$

System F allows us to make
type-level products,
abstractions,
applications

The type of length in system F

$$\text{Length} : \Pi X. \text{List}_X \to \text{Nat}$$

second order product
length can be applied
to any type X

X is a type variable

Second order application:

$$\text{Length Bool} : \text{List}_{\text{Bool}} \to \text{Nat}$$

The definition of length
uses second-order abstraction

$$\text{length} = \Lambda X.\ \text{recList}_X \ \cdots$$

abstraction of
type variable X

Abstraction and application
at type level work
similarly to object level:

$$(\Lambda X.t)\, T \leadsto t[X := T]$$

Church Numerals Revisited

$$\overline{2} = \lambda f.\lambda x.\, f\,(f\,x)$$

In $\lambda\!\to$ we gave numerals the type $(o \to o) \to o \to o$

But this was limited:

We can only do iteration of functions on o.

No exponentiation.

We could try to give $\overline{2}$ a higher type:

$$\overline{2} : (T \to T) \to T \to T$$

For exponentiation we need numerals at different levels

Solution in system F:

Numerals have polymorphic types:

$$\overline{2} : \underbrace{\Pi X.\,(X \to X) \to X \to X}_{\text{Nat}}$$

in system F

The definition of $\overline{2}$ must

- abstract over X
- use X in the types of f, x

$$\bar{2} = \Lambda X.\, \lambda f : \underbrace{X \to X}.\, \lambda x : \underbrace{X}.\, f\,(f\,x)$$

the types of the abstracted first-order variables may depend on the abstracted second-order variable

The definition of exponential in the untyped λ-calculus can be typed in system F

$$\mathrm{exp} = \lambda m.\, \lambda n.\, \Lambda X.\, \underbrace{n\,(X \to X)}\, \underbrace{(m\,X)}$$

We instantiate the two numerals with different types

$$n, m : Nat = \Pi X.\, (X \to X) \to X \to X$$

$$m\,X : \underbrace{(X \to X) \to X \to X}$$

$$n\,(X \to X) : (\overbrace{(X \to X) \to (X \to X)}) \to (X \to X) \to (X \to X)$$

They can be applied one to the other

$$n\,(X\to X)\,(m\,X) : (X\to X)\to X\to X$$

$$\Lambda X.\; n\,(X\to X)\,(m\,X)$$

$$: \underbrace{\Pi X.\,(X\to X)\to X\to X}_{\substack{\shortparallel \\ \mathrm{Nat}}}$$

We can now do iteration of functions on any type

But can we do recursion? (elimination rule of Nat in system T)

We can use the trick with pairing that we used in the untyped λ-calculus for factorial and predecessor

For this we need pairing in system F

Cartesian Product:

$$A\times B = \Pi X.\,(A\to B\to X)\to X$$

$$\langle a,b\rangle = \Lambda X.\,\lambda g.\; g\,a\,b$$

$$\mathrm{fst} = \lambda p.\; p\,A\,(\lambda x.\,\lambda y.\,x)$$

$$\mathrm{snd} = \lambda p.\; p\,B\,(\lambda x.\,\lambda y.\,y)$$

Lists in System F

Same idea as Nat

Lists are iterators

$$\mathrm{List}_A = \Pi X. (A \to X \to X) \to X \to X$$

$$\mathrm{nil} = \Lambda X. \lambda f. \lambda x. x$$

$$a :: \ell = \Lambda X. \lambda f. \lambda x. f a (\ell X f x)$$

A list is an iterator that:

- iterates a function f with an argument $a : A$ for the current list element
- with a starting value x

The polymorphic length function

$$\mathrm{length} : \Pi Y. \mathrm{List}_Y \longrightarrow \mathrm{Nat}$$

$$\mathrm{length} = \Lambda Y. \lambda \ell. \; \ell \; \mathrm{Nat} \; (\lambda a. \mathrm{succ}) \; \mathrm{zero}$$

Exercise:

What is the system F definition of the type of binary trees?