Universal Types

Motivation

In the simply typed lambda-calculus, we often have to write several versions of the same code, differing only in type annotations.

```
doubleNat = λf:Nat→Nat. λx:Nat. f (f x)
doubleRcd = λf:{l:Bool}→{l:Bool}. λx:{l:Bool}. f (f x)
doubleFun = λf:(Nat→Nat)→(Nat→Nat). λx:Nat→Nat. f (f x)
```

Bad! Violates a basic principle of software engineering:

> Write each piece of functionality once

Motivation

In the simply typed lambda-calculus, we often have to write several versions of the same code, differing only in type annotations.

```
doubleNat = λf:Nat→Nat. λx:Nat. f (f x)
doubleRcd = λf:{l:Bool}→{l:Bool}. λx:{l:Bool}. f (f x)
doubleFun = λf:(Nat→Nat)→(Nat→Nat). λx:Nat→Nat. f (f x)
```

Bad! Violates a basic principle of software engineering:

> Write each piece of functionality once... and parameterize it on the details that vary from one instance to another.

Motivation

In the simply typed lambda-calculus, we often have to write several versions of the same code, differing only in type annotations.

```
doubleNat = λf:Nat→Nat. λx:Nat. f (f x)
doubleRcd = λf:{l:Bool}→{l:Bool}. λx:{l:Bool}. f (f x)
doubleFun = λf:(Nat→Nat)→(Nat→Nat). λx:Nat→Nat. f (f x)
```

Bad! Violates a basic principle of software engineering:

> Write each piece of functionality once... and parameterize it on the details that vary from one instance to another.

Here, the details that vary are the types!

Idea

We'd like to be able to take a piece of code and "abstract out" some type annotations.

We've already got a mechanism for doing this with terms: λ-abstraction. So let's just re-use the notation.

Abstraction:

```
double = λX. λf:X→X. λx:X. f (f x)
```

Application:

```
double [Nat]
double [Bool]
```

Computation:

```
double [Nat] ⟶ λf:Nat→Nat. λx:Nat. f (f x)
```

(N.b.: Type application is commonly written `t [T]`, though `t T` would be more consistent.)

Idea

What is the *type* of a term like

λX. λf:X$\rightarrow$X. λx:X. f (f x) ?

This term is a function that, when applied to a type X, yields a term of type (X$\rightarrow$X)$\rightarrow$X$\rightarrow$X.

Idea

What is the *type* of a term like

$$\lambda\mathtt{X.}\ \lambda\mathtt{f{:}X{\rightarrow}X.}\ \lambda\mathtt{x{:}X.}\ \mathtt{f\ (f\ x)}\ ?$$

This term is a function that, when applied to a type $\mathtt{X}$, yields a term of type $\mathtt{(X{\rightarrow}X){\rightarrow}X{\rightarrow}X}$.

I.e., for all types $\mathtt{X}$, it yields a result of type $\mathtt{(X{\rightarrow}X){\rightarrow}X{\rightarrow}X}$.

Idea

What is the *type* of a term like

λ`X. `λ`f:X`$\rightarrow$`X. `λ`x:X. f (f x)` ?

This term is a function that, when applied to a type `X`, yields a term of type `(X`$\rightarrow$`X)`$\rightarrow$`X`$\rightarrow$`X`.

I.e., for all types `X`, it yields a result of type `(X`$\rightarrow$`X)`$\rightarrow$`X`$\rightarrow$`X`.

We'll write it like this: $\forall$`X. (X`$\rightarrow$`X)`$\rightarrow$`X`$\rightarrow$`X`

System F

System F (aka "the polymorphic lambda-calculus") formalizes this idea by extending the simply typed lambda-calculus with type abstraction and type application.

`t ::=`		*terms*
	`x`	*variable*
	λ`x:T.t`	*abstraction*
	`t t`	*application*
	λ`X.t`	*type abstraction*
	`t [T]`	*type application*

System F

System F (aka "the polymorphic lambda-calculus") formalizes this idea by extending the simply typed lambda-calculus with type abstraction and type application.

`t ::=`		*terms*
	`x`	*variable*
	λ`x:T.t`	*abstraction*
	`t t`	*application*
	λ`X.t`	*type abstraction*
	`t [T]`	*type application*
`v ::=`		*values*
	λ`x:T.t`	*abstraction value*
	λ`X.t`	*type abstraction value*

System F: new evaluation rules

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ [\mathtt{T}_2] \longrightarrow \mathtt{t}_1'\ [\mathtt{T}_2]} \qquad \text{(E-TApp)}$$

$$(\lambda \mathtt{X}.\mathtt{t}_{12})\ [\mathtt{T}_2] \longrightarrow [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{t}_{12} \qquad \text{(E-TappTabs)}$$

System F: Types

To talk about the types of "terms abstracted on types," we need to introduce a new form of types:

$\mathtt{T}$ ::=		*types*
	$\mathtt{X}$	*type variable*
	$\mathtt{T}\longrightarrow\mathtt{T}$	*type of functions*
	$\forall\mathtt{X.T}$	*universal type*

System F: Typing Rules

$$\frac{\mathtt{x}:\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T_1} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \lambda \mathtt{x}:\mathtt{T_1}.\mathtt{t_2} : \mathtt{T_1} \rightarrow \mathtt{T_2}} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_{11}} \rightarrow \mathtt{T_{12}} \qquad \Gamma \vdash \mathtt{t_2} : \mathtt{T_{11}}}{\Gamma \vdash \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_{12}}} \qquad \text{(T-App)}$$

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \lambda \mathtt{X}.\mathtt{t_2} : \forall \mathtt{X}.\mathtt{T_2}} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \forall \mathtt{X}.\mathtt{T_{12}}}{\Gamma \vdash \mathtt{t_1}\ [\mathtt{T_2}] : [\mathtt{X} \mapsto \mathtt{T_2}]\mathtt{T_{12}}} \qquad \text{(T-TApp)}$$

History

Interestingly, System F was invented independently and almost simultaneously by a computer scientist (John Reynolds) and a logician (Jean-Yves Girard).

Their results look very different at first sight — one is presented as a tiny programming language, the other as a variety of second-order logic.

The similarity (indeed, isomorphism!) between them is an example of the *Curry-Howard Correspondence*.

Examples

Lists

```
cons : ∀X. X → List X → List X
head : ∀X. List X → X
tail : ∀X. List X → List X
nil : ∀X. List X
isnil : ∀X. List X → Bool

map =
  λX. λY.
    λf: X→Y.
      (fix (λm: (List X) → (List Y).
                 λl: List X.
                   if isnil [X] l
                     then nil [Y]
                     else cons [Y] (f (head [X] l))
                                   (m (tail [X] l))));

l = cons [Nat] 4 (cons [Nat] 3 (cons [Nat] 2 (nil [Nat])));

head [Nat] (map [Nat] [Nat] (λx:Nat. succ x) l);
```

Church Booleans

```
CBool = ∀X.X→X→X;

tru = λX. λt:X. λf:X. t;
fls = λX. λt:X. λf:X. f;

not = λb:CBool. λX. λt:X. λf:X. b [X] f t;
```

Church Numerals

```
CNat = ∀X. (X→X) → X → X;

c0  =  λX. λs:X→X. λz:X. z;
c1   =  λX. λs:X→X. λz:X. s z;
c2   =  λX. λs:X→X. λz:X. s (s z);

csucc = λn:CNat. λX. λs:X→X. λz:X. s (n [X] s z);

cplus = λm:CNat. λn:CNat. m [CNat] csucc n;
```

Properties of System F

Preservation and Progress: unchanged.

(Proofs similar to what we've seen.)

Strong normalization: every well-typed program halts. (Proof is challenging!)

Type reconstruction: undecidable (major open problem from 1972 until 1994, when Joe Wells solved it).

Parametricity

Observation: Polymorphic functions cannot do very much with their arguments.

- The type $\forall \mathtt{X.}\ \mathtt{X} \to \mathtt{X} \to \mathtt{X}$ has exactly two members (up to observational equivalence).
- $\forall \mathtt{X.}\ \mathtt{X} \to \mathtt{X}$ has one.
- etc.

The concept of parametricity gives rise to some useful "free theorems..."