System F

System F

System F, also called the **polymorphic λ-calculus** (Girard, 1972; Reynolds, 1974) adds the idea of type quantification to the simply-typed lambda calculus.

Quantifiers can occur anywhere in a type (not just at the front/top).

In ML-style let-polymorphism, we typed the identity function as $\forall X.X \rightarrow X$.

When we apply the identity function to a value, we substitute the type of that value for X in the body of the for-all.

This resembles applying a λ-abstraction to a value.

System F introduces type-level lambdas.

$id = \lambda X.\ \lambda x : X.\ x$

If we want to apply this to a value, we must first apply it to the type of that value.

$id = \lambda X.\ \lambda x : X.\ x$
$id[Nat] = \lambda x : Nat.\ x$
$id[Nat]\ 3 = (\lambda x : Nat.\ x)\ 3 = 3$

New syntax:

```
t ::= ...
      λX.t
      t [T]
v ::= ...
      λX.t
T ::= ...
      X
      ∀X.T
```


Recall the rule for function application:
$(\lambda x : T.t)v \to [x \mapsto v]t$

Type application is similar:
$(\lambda X.t)[T] \to [X \mapsto T]t$

The type inference rules for polymorphic abstraction and application resemble $\forall$-introduction and elimination in logic.

Type variables:

$$\frac{X : T \in \Gamma}{\Gamma \vdash X : T}$$

Type abstraction:

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash (\lambda X.t : \forall X.T)}$$

What does Γ, X mean?

Γ, X simply means that X is a fresh variable.

We will assume α-renaming of type variables to avoid name clashes.

Type application:

$$\frac{\Gamma \vdash t : \forall X.\ T}{\Gamma \vdash t[S] : [X \mapsto S]\ T}$$

A type judgment for the identity function:

$$\cfrac{\cfrac{\cfrac{x : X \in X, x : X}{X, x : X \vdash x : t}\ \text{T-VAR}}{X \vdash (\lambda x : X.x) : (X \to X)}\ \text{T-ABS}}{\vdash (\lambda X.\lambda x : X.x) : \forall X.X \to X}\ \text{T-TABS}$$

A type judgment for an application of the identity function:

$$\cfrac{\cfrac{\vdash (\lambda X.\lambda x : X.x) : \forall X.(X \to X)}{\vdash (\lambda X.\lambda x : X.x)[\mathrm{Int}] : \mathrm{Int} \to \mathrm{Int}}\ \text{T-TAPP} \qquad \cfrac{}{\vdash 3 : \mathrm{Int}}}{\vdash ((\lambda X.\lambda x : X.x)[\mathrm{Int}]\ 3) : \mathrm{Int}}\ \text{T-APP}$$

`id id` is not typable in the simply-typed λ-calculus, or by using let-polymorphism.

But we can create a typable version in System F.

All we need to do is instantiate `id` with its own type.

$((\lambda X.\lambda x : X.x)\ [\forall X.X \to X])$
$\quad (\lambda X.\lambda x : X.x)$

This version of `id id` has type $\forall X.X \to X$, as expected from the way we typed `id`.

We could not type $\lambda x.x\ x$ in the simply-typed lambda calculus.

But $(\lambda y.y)(\lambda y.y)$ is the result of one reduction step on $(\lambda x.x\ x)(\lambda y.y)$.

So if we annotate x with the type of `id`, and use the same instantiation idea, it should typecheck.

$\lambda x : \forall X.X \to X\ .\ x\ [\forall X.X \to X]\ x$ is the resulting term.

Exercise: show this typechecks and has the type $(\forall X.X \to X) \to (\forall X.X \to X)$

But this is not the only version possible.

We can create versions with these types:

$\forall W.(\forall X.X \to X) \to (W \to W)$

$\forall W.(\forall X.X \to X) \to W$

Can we create a version of
$(\lambda x.x\ x)(\lambda x.x\ x)$
that typechecks?

As it turns out, no.

Theorems about System F

Typechecking is straightforward structural recursion.

Progress and preservation theorems are exercises in Pierce.

Girard showed that System F is strongly normalizing.

Despite this, it is much more powerful than the simply-typed lambda calculus, as we will see on the next few slides.

Unfortunately, type inference is undecidable for System F (Wells, 1994).

Much recent work has involved decidable fragments.

Programming in System F

System F allows us to type many (not all) of the terms we used to simulate more expressive models of computation in the lambda calculus.

Booleans (review from M01)

true $= \lambda x.\lambda y.x$
false $= \lambda x.\lambda y.y$

if b **then** t **else** f

$$= b\ t\ f$$
$$= \lambda b.\lambda t.\lambda f.\ b\ t\ f$$

A Boolean value is thus a function consuming two arguments of the same type (the two clauses of the `if`) and producing that type (one of them).

CBool $= \forall X\ .\ X \to X \to X$

$$\mathbf{true} = \lambda X\,.\,\lambda x : X\,.\,\lambda y : X\,.\,x$$
$$\mathbf{false} = \lambda X\,.\,\lambda x : X\,.\,\lambda y : X\,.\,y$$

$$\mathbf{if}\ b\ \mathbf{then}\ t\ \mathbf{else}\ f$$
$$= \lambda b.\lambda X.\lambda t : X.\lambda f : X.b[X]\ t\ f$$

Natural numbers (review from M01)

$$\begin{aligned}
\mathbf{c_0} &= \lambda s.\lambda z.\ z\\
\mathbf{c_1} &= \lambda s.\lambda z.\ s\ z\\
\mathbf{c_2} &= \lambda s.\lambda z.\ s\ (s\ z)\\
&\dots\\
\mathbf{succ} &= \lambda n.\lambda s.\lambda z.\ s\ (n\ s\ z)
\end{aligned}$$

A natural number is thus a function that consumes a successor function and a zero.

$$\mathbf{Nat} = \forall X\,.\,(X \to X) \to X \to X$$

$$\begin{aligned}
\mathbf{c_0} &= \lambda X.\lambda s : X \to X.\lambda z : X.z\\
\mathbf{c_1} &= \lambda X.\lambda s : X \to X.\lambda z : X.s\ z\\
\mathbf{c_2} &= \lambda X.\lambda s : X \to X.\lambda z : X.s\ (s\ z)\\
&\dots\\
\mathbf{succ} &= \lambda n : \mathbf{Nat}.\lambda X.\\
&\qquad \lambda s : X \to X.\lambda z : X.s\ (n[X]\ s\ z)
\end{aligned}$$

Tuples with first component of type T_1 and second of type T_2 have the type $T_1 \times T_2 = \forall X \,.\, (T_1 \to T_2 \to X) \to X$. The pairing function is:

$$\langle e_1, e_2 \rangle = \lambda X \,.\, \lambda f : t_1 \to t_2 \to X \,.\, f\ e_1\ e_2$$

$$\mathbf{fst} = \lambda p : t_1 \times t_2 . p[t_1](\lambda x : t_1 . \lambda y : t_2 . x)$$

$$\mathbf{snd} = \lambda p : t_1 \times t_2 . p[t_2](\lambda x : t_1 . \lambda y : t_2 . y)$$

We can implement general recursion schemes.

Suppose we want

$$g\ 0 = c$$
$$g\ (n+1) = h\ (g\ n)$$

where $g : \mathbf{Nat} \to t$, $c : t$, $h : t \to t$.

Then $g\ n = h^n\ c$, so
$\mathbf{g} = \lambda n : \mathbf{Nat} \,.\, n[t]\ h\ c$.

For example,

$$\textbf{add}\ m\ 0 = m$$
$$\textbf{add}\ m\ (n+1) = \textbf{succ}\ (\textbf{add}\ m\ n)$$

So
$\textbf{add}\ m = \lambda n : \textbf{Nat}\ .\ n[\textbf{Nat}]\ \textbf{succ}\ m.$

Using pair constructors and deconstructors, we can extend this scheme to compute any primitive recursive function.

It's even possible to compute Ackermann's function (so we can go beyond the primitive recursive functions).

To represent lists, we use
an ingenious idea:
we represent a list by
its own `foldr` function.

Recall that `foldr` consumes a combine function and a base value.

Suppose that the list it is applied to has values of type X, and the base value has type R.

The type of `foldr` is then
$(X \to R \to R) \to R \to R$.

The "list of type X" type:

$$\textbf{List}[\textbf{X}] = \forall R\,.\,(X \to R \to R) \to R \to R$$

empty $=$
$\lambda X\,.\,\lambda R\,.\,\lambda c : X \to R \to R\,.\,\lambda b : R\,.\,b$

cons $= \lambda X\,.\,\lambda hd : X\,.\,\lambda tl : List[X]\,.$
$\lambda R :\ \lambda c : X \to R \to R. \lambda b : R.$
$c\ hd\ (tl[R]\ c\ b)$

Since `foldr` is universal for structural recursion on lists, we can express many computations on lists (for example, polymorphic insertion sort).

This brief look at programming in System F demonstrates its expressivity, which is why it is a desirable target for extending the type systems of programming languages.

It is also a key component in much theoretical work.

The core intermediate language of GHC is based on System F.

(But we can't do type inference in System F.)

GHC Core is explicitly typed.

It includes literals, `let`, `case`, term abstractions, type abstractions.

Although Core can express all of System F, translated Haskell code does not use all of System F's expressivity.

System F is useful here for slight extensions to let-polymorphism, and to facilitate optimization.

GHC lets us dump intermediate formats so we can look at the Core produced.

This presentation cleans up both Haskell source and resulting Core.

```
id x = x

map f [] = []
map f (x:xs)
 = (f x) : (map f xs)
```



```
ghc -c -ddump-tc map.hs

TYPE SIGNATURES
  id :: forall t. t -> t
  map :: forall t a.
         (t -> a) -> [t] -> [a]
```



```
Binds:
map f [] = GHC.Types.[]
map f (x : xs)
 = GHC.Types.: (f x) (map f xs)
```



```
ghc -c -ddump-simpl map.hs

id :: forall t. t -> t

id = \ (@ t)
       (x :: t) ->
         x_aeH
```

```
map :: forall t a.
       (t -> a) -> [t] -> [a]

map =
  \ (@ t)
    (@ a)
    (f :: t -> a)
    (xs :: [t]) ->
    case xs of _ {
      [] -> GHC.Types.[] @ a;
      : x xs ->
        GHC.Types.:
          @ a
          (f x)
          (map @ t @ a f xs)}
```


Can System F deal with type operators?

$\lambda x.t$ maps terms to terms.

$\lambda X.t$ maps types to terms.

System F has nothing like $\lambda X.T$, which maps types to types.

We move beyond System F to introduce this capability (Pierce, Chapters 29 and 30).

Earlier, we used kinds to describe what a type operator could be applied to.

Recall that $*$ was the kind of a proper type such as $\mathtt{Nat}$ or $\mathtt{Bool} \to \mathtt{Nat}$.

Type operators had kinds such as $* \Rightarrow *$.

Kinds show up in GHC Core.

```
class Monad m where
 return :: a -> m a
 bind :: m a -> (a -> m b) -> m b
```



```
return ::
 forall (m :: * -> *).
     Monad m =>
     forall a. a -> m a
return =
  \ (@ (m :: * -> *))
    (B1 :: Monad m) ->
    case B1 of _ {
      D:Monad B2 _ -> B2 }
```


In general, we treat kinds like the simply-typed lambda calculus with one base type (namely $*$).

The “type of a type” is a kind.

We add grammatical rules for kinds.

$$\begin{aligned} K ::= &\ * \\ &\ K \Rightarrow K \end{aligned}$$

We annotate the type variable in operator and type abstractions and quantified types with a kind, and allow operator application.

```
t ::= λX::K.t
T ::= λX::K.T
      ∀X::K.T
      T T
```


The type annotations on variables for term abstractions $\lambda x : T.t$ are used to make judgments $\Gamma \vdash t : T$ about the type T of the term t.

Similarly, the kind annotations on type variables for type abstractions $\lambda X :: K.t$ are used to make judgments $\Gamma \vdash T :: K$ about the kind K of the type T.

Earlier, we added type variables to contexts, but just so we could avoid name clashes.

Now, the type variables are annotated with their kind, just as ordinary variables in contexts are annotated with their type.

Type variables:

$$\frac{x :: K \in \Gamma}{\Gamma \vdash x :: K}$$

Type abstraction:

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash (\lambda X :: K_1.T_2) :: (K_1 \Rightarrow K_2)}$$

Type application:

$$\frac{\Gamma \vdash T_1 :: K_1 \Rightarrow K_2 \qquad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash T_1\ T_2 :: K_2}$$

Base types and arrow types have kind $*$.

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \rightarrow T_2\ ::\ *}$$

Quantified types have kind $*$.

$$\frac{\Gamma, X :: K \vdash T :: *}{\Gamma \vdash \forall X :: K.T\ ::\ *}$$

Types can now be expressions built from kinded type variables, type abstractions, and operator application.

We might be tempted to add an evaluation rule to deal with type expressions.

But type judgments take place before any evaluation has been done.

Types do not play a role in term evaluation.

We add a notion of **equivalence** for types, with notation $S \equiv T$.

Two types will be equivalent if they can be reduced to the same value.

The inference rules for type equivalence come in three flavours: equality properties, structural, and reduction.

Equality properties:
reflexive, symmetric, transitive.

$$T \equiv T \qquad \frac{T \equiv S}{S \equiv T}$$

$$\frac{S \equiv U \qquad U \equiv T}{S \equiv T}$$

Structural rules ensure that type expressions using equivalent types are themselves equivalent.

Equivalence of type abstractions:

$$\frac{S \equiv T}{\lambda X :: K.S \equiv \lambda X :: K.T}$$

Equivalence of quantified types:

$$\frac{S \equiv T}{\forall X :: K.S \equiv \forall X :: K.T}$$

Equivalence of operator applications:

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2}$$

Reduction rule:

$$(\lambda X :: K.T_1)\ T_2 \equiv [X \mapsto T_2]T_1$$

Term abstractions have arguments of kind $*$.

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t \ : \ T_1 \to T_2}$$

A type of kind $*$ can be replaced by an equivalent type.

$$\frac{\Gamma \vdash t : S \qquad S \equiv T \qquad \Gamma \vdash T :: *}{\Gamma \vdash t : T}$$

We have completed the description of System F_ω (pronounced "F-omega").

Why ω?

ω is the first infinite ordinal number.

We can think of it as the set of natural numbers.

System F_ω can be viewed as the limit of Systems F_1, F_2, and so on.

We can define a hierarchy of kinds.

$\mathcal{K}_0 = \phi$
$\mathcal{K}_{i+1} = \{*\} \cup \{J \Rightarrow K \mid J \in K_i, K \in K_{i+1}\}$

Note $\mathcal{K}_1 = \{*\}$, and $\mathcal{K}_2$ adds only kinds with $*$ to the left of an arrow.

System F_1 only has types of kind $*$, no quantified types, no type abstraction.

This is the simply-typed lambda calculus.

System F_{i+1} allows abstraction over types with kinds in $\mathcal{K}_i$.

Thus these abstractions have kinds in $\mathcal{K}_{i+1}$.

So we also allow quantification over types with kinds in $\mathcal{K}_{i+1}$.

System F_2 is System F, because it allows type abstraction over types with kind $*$.

System F_3 suffices to describe all programming language features in Pierce.

Theorems about System F_ω

Typechecking is not so straightforward, because the rule about replacing equivalent types is not structural.

Instead, types are normalized by the typechecker (as was our first idea!)

Pierce sketches details of typechecking, and covers progress and preservation theorems more carefully.

System F_ω is also strongly normalizing (also proved by Girard).

What is the next step?

Pierce, Advanced Topics in Types and Programming Languages.

Pierce, Software Foundations (to be used in CS 798 F14 by yrs truly).

$\lambda x.t$ maps terms to terms.

$\lambda X.t$ maps types to terms.

$\lambda X.T$ maps types to types.

New: $\Pi x.T$ maps terms to types.

Why is this useful?

Consider implementing vectors.

```
data Vec0   = Vec0
data Vec1 a = Vec1 a
data Vec2 a = Vec2 a a
data Vec3 a = Vec3 a a a
```

It would be nice to generalize.

`Vec :: Nat → *`

`Vec` is a type family.

`Vec 3` is isomorphic to `Vec3` on the previous slide.

Consider a function to create and initialize a vector of length `n` with a value `v` of type `T`.

`init :: Πn:Nat.T → Vec n`

Here is a more interesting example.

`first :: Πn:Nat.Vec(n+1)→T`

`first` cannot be applied to a value of type `Vec 0`.

Types can be used to describe and enforce properties of values and functions (for example, that the elements of a vector are sorted).

On a typechecking level, this erases the distinction between types, kinds, and terms.

For example, everything is now a term, and we now have abstractions over kinds.

On a programming level, this means that the programmer must write programs whose typechecking amounts to verifying a proof of the properties that the types encode.

Implementations of dependently-typed languages (Coq, Agda, Idris) often have a “proof assistant” feel to them, in the source code and/or the IDE.

The goal is to improve the capabilities and guarantees of static type checking while preserving the ability to erase types before run time.

That’s it for now.