

System F

1 / 102

# System F

2 / 102

System F, also called the **polymorphic  $\lambda$ -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).

3 / 102

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.

4 / 102

This resembles applying a  $\lambda$ -abstraction to a value.

System F introduces type-level lambdas.

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

5/102

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

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

$$id[Nat] = \lambda x : Nat. x$$

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

6/102

New syntax:

$$\begin{aligned} t ::= & \dots \\ & \lambda X. t \\ & t [T] \end{aligned}$$
$$\begin{aligned} v ::= & \dots \\ & \lambda X. t \end{aligned}$$
$$\begin{aligned} T ::= & \dots \\ & X \\ & \forall X. T \end{aligned}$$

7/102

Recall the rule for function application:

$$(\lambda x : T. t) v \rightarrow [x \mapsto v] t$$

Type application is similar:

$$(\lambda X. t) [T] \rightarrow [X \mapsto T] t$$

8/102

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

9/102

Type variables:

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

10/102

Type abstraction:

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

What does  $\Gamma, X$  mean?

11 / 102

$\Gamma, X$  simply means that  $X$  is a fresh variable.

We will assume  $\alpha$ -renaming of type variables to avoid name clashes.

12 / 102

Type application:

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

13/102

A type judgment for the identity function:

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

14/102

A type judgment for an application of the identity function:

$$\begin{array}{c}
 \displaystyle \frac{\vdash (\lambda X. \lambda x : X. x) : \forall X. (X \rightarrow X)}{\vdash (\lambda X. \lambda x : X. x)[\text{Int}] : \text{Int} \rightarrow \text{Int}} \text{ T-TAPP} \\
 \displaystyle \frac{\displaystyle \frac{}{\vdash 3 : \text{Int}}}{\vdash ((\lambda X. \lambda x : X. x)[\text{Int}] 3) : \text{Int}} \text{ T-APP}
 \end{array}$$

15/102

id id is not typable in the simply-typed  $\lambda$ -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.

16/102



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

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

17 / 102

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.

18 / 102

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

Exercise: show this typechecks and has the type

$$(\forall X. X \rightarrow X) \rightarrow (\forall X. X \rightarrow X)$$

19/102

But this is not the only version possible.

We can create versions with these types:

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

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

20/102

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

As it turns out, no.

21 / 102

# Theorems about System F

22 / 102

Typechecking is straightforward structural recursion.

Progress and preservation theorems are exercises in Pierce.

Girard showed that System F is strongly normalizing.

23 / 102

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.

24 / 102

# Programming in System F

25 / 102

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.

26 / 102

## Booleans (review from M01)

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

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

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

=  $b\ t\ f$

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

27 / 102

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** =  $\forall X . X \rightarrow X \rightarrow X$

28 / 102

**true** =  $\lambda X . \lambda x : X . \lambda y : X . x$

**false** =  $\lambda X . \lambda x : X . \lambda y : X . y$

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

=  $\lambda b . \lambda X . \lambda t : X . \lambda f : X . b[X] \ t \ f$

29/102

**Natural numbers** (review from M01)

**c**<sub>0</sub> =  $\lambda s . \lambda z . z$

**c**<sub>1</sub> =  $\lambda s . \lambda z . s \ z$

**c**<sub>2</sub> =  $\lambda s . \lambda z . s \ (s \ z)$

...

**succ** =  $\lambda n . \lambda s . \lambda z . s \ (n \ s \ z)$

30/102

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

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

31 / 102

$$\mathbf{c}_0 = \lambda X . \lambda s : X \rightarrow X . \lambda z : X . z$$

$$\mathbf{c}_1 = \lambda X . \lambda s : X \rightarrow X . \lambda z : X . s \ z$$

$$\mathbf{c}_2 = \lambda X . \lambda s : X \rightarrow X . \lambda z : X . s \ (s \ z)$$

...

$$\mathbf{succ} = \lambda n : \mathbf{Nat} . \lambda X .$$

$$\lambda s : X \rightarrow X . \lambda z : X . s \ (n[X] \ s \ z)$$

32 / 102



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

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

33/102

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

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

34/102

We can implement general recursion schemes.

35 / 102

Suppose we want

$$\begin{aligned} g\ 0 &= c \\ g\ (n + 1) &= h\ (g\ n) \end{aligned}$$

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

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

36 / 102

For example,

$$\mathbf{add} \ m \ 0 = m$$

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

So

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

37 / 102

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).

38 / 102

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

39 / 102

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  
 $(X \rightarrow R \rightarrow R) \rightarrow R \rightarrow R$ .

40 / 102

The “list of type  $X$ ” type:

**List**[ $X$ ] =

$$\forall R . (X \rightarrow R \rightarrow R) \rightarrow R \rightarrow R$$

41 / 102

**empty** =

$$\lambda X . \lambda R . \lambda c : X \rightarrow R \rightarrow R . \lambda b : R . b$$

**cons** =  $\lambda X . \lambda hd : X . \lambda tl : List[X] .$

$$\lambda R : \lambda c : X \rightarrow R \rightarrow R . \lambda b : R .$$

$$c \text{ } hd \text{ } (tl[R] \text{ } c \text{ } b)$$

42 / 102

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

43 / 102

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.

44 / 102

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.)

45 / 102

GHC Core is explicitly typed.

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

46 / 102

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.

47 / 102

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

This presentation cleans up both Haskell source and resulting Core.

48 / 102



```
id x = x
```

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

49/102

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

TYPE SIGNATURES

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

50/102

Binds:

```
map f [] = GHC.Types.[]
```

```
map f (x : xs)
```

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

51/102

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

```
id :: forall t. t -> t
```

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

52/102

```
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)}
```

53/102

Can System F deal with type operators?

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

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

54/102

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).

55 / 102

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 `Nat` or `Bool  $\rightarrow$  Nat`.

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

56 / 102

Kinds show up in GHC Core.

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

57/102

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

58/102

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

The “type of a type” is a kind.

59 / 102

We add grammatical rules for kinds.

$$\begin{array}{l} K ::= * \\ \quad K \Rightarrow K \end{array}$$

60 / 102

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

$$\begin{aligned} t &::= \lambda X :: K. t \\ T &::= \lambda X :: K. T \\ &\quad \forall X :: K. T \\ &\quad T \ T \end{aligned}$$

61 / 102

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  $T$ .

62 / 102

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.

63 / 102

Type variables:

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

64 / 102



Type abstraction:

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

65/102

Type application:

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

66/102

Base types and arrow types have kind  $*$ .

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

67 / 102

Quantified types have kind  $*$ .

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

68 / 102

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.

69 / 102

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

Types do not play a role in term evaluation.

70 / 102

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.

71 / 102

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

72 / 102

Equality properties:  
reflexive, symmetric, transitive.

$$\begin{array}{c} T \equiv T \qquad \frac{T \equiv S}{S \equiv T} \\[1em] \frac{S \equiv U \quad U \equiv T}{S \equiv T} \end{array}$$

73 / 102

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

74 / 102

Equivalence of type abstractions:

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

75 / 102

Equivalence of quantified types:

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

76 / 102

Equivalence of operator applications:

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

77/102

Reduction rule:

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

78/102

Term abstractions have arguments of kind  $*$ .

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

79/102

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

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

80/102



We have completed the description of System  $F_\omega$  (pronounced “F-omega”).

Why  $\omega$ ?

81 / 102

$\omega$  is the first infinite ordinal number.

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

System  $F_\omega$  can be viewed as the limit of Systems  $F_1$ ,  $F_2$ , and so on.

82 / 102

We can define a hierarchy of kinds.

$$\mathcal{K}_0 = \phi$$

$$\mathcal{K}_{i+1} = \{*\} \cup \{J \Rightarrow K \mid J \in \mathcal{K}_i, K \in \mathcal{K}_{i+1}\}$$

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

83 / 102

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

This is the simply-typed lambda calculus.

84 / 102

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  $\mathcal{K}_{i+1}$ .

85 / 102

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.

86 / 102

# Theorems about System $F_\omega$

87 / 102

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!)

88 / 102

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

System  $F_\omega$  is also strongly normalizing (also proved by Girard).

89 / 102

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).

90 / 102

$\lambda x.t$  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.

91 / 102

Why is this useful?

92 / 102

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.

93 / 102

$\text{Vec} :: \text{Nat} \rightarrow *$

$\text{Vec}$  is a type family.

$\text{Vec } 3$  is isomorphic to  $\text{Vec3}$  on the previous slide.

94 / 102

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

$$\text{init} :: \prod n:\text{Nat}. T \rightarrow \text{Vec } n$$

95 / 102

Here is a more interesting example.

$$\text{first} :: \prod n:\text{Nat}. \text{Vec } (n+1) \rightarrow T$$

$\text{first}$  cannot be applied to a value of type  $\text{Vec } 0$ .

96 / 102



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

97 / 102

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.

98 / 102

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.

99 / 102

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

100 / 102

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

101 / 102

That's it for now.

102 / 102