# Type Systems

Lecture 9 Dec. 15th, 2004
Sebastian Maneth

http://lampwww.epfl.ch/teaching/typeSystems/2004

## Today Parametric Polymorphism

1. Recall Let-Polymorphism
2. System F
3. Properties of System F
4. System F-sub
5. Properties of F-sub

## 1. Recall Let-Polymorphism

In simply-typed lambda-calculus, we can leave out ALL type annotations:

→ insert new type variables
→ do type reconstruction (using unification)

In this way, changing the let-rule, we obtain

**Let-Polymorphism**

→ Simple form of polymorphism

→ Introduced by [ Milner 1978 ] in ML

→ also known as Damas-Milner polymorphism

→ in ML, basis of powerful *generic libraries*
(e.g., lists, arrays, trees, hash tables, …)

## 1. Recall Let-Polymorphism

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash [x \to t_1]t_2 : T_2}{\Gamma \vdash \texttt{let } x{=}t_1 \texttt{ in } t_2 : T_2}$$

```
let double = λx.λy. x(x(y)) in
{
  let a = double (λx:int. x+2) 2 in {
  let b = double (λx:bool. x) false in {..}
  }
}
```

CAN be typed now!! Because the new let rule creates two copies of double, and the rule for abstraction assigns a *different* type variable to each one.

## 1. Recall Let-Polymorphism

Limits of Let-Polymorphism?

→ Only let-bound variables can be used polymorphically!
→ NOT lambda-bound variables

Ex.:
```
let f = λg.  … g(1) … g(true) …
 in { f(λx.x) }
```

is not typable: when typechecking the def. of f, g has type X (fresh)
Which is then constrained by X = int → Y and X = bool → Z.

Functions cannot take polymorphic functions as parameters.

(= no polymorphic arguments!)

## 2. System F

Aka polymorphic lambda-calculus or second-order lambda-calculus.

→ do lambda-abstraction over type variables,
define functions over types

Invented by

→ Girard (1972) motivated by logics

→ Reynolds (1974) motivated by programming.

## 2. System F

Aka polymorphic lambda-calculus or second-order lambda-calculus.

→ Add (universal) quantification over TYPEs!
→ Straightforward extension of simply typed lambda-calculus by
two new constructs:

Type Abstraction: `λX. t`
Type Application: `t [T]`

For example, the polymorphic identity function

```
id = λX. λx:X. x
```

## 2. System F

Aka polymorphic lambda-calculus or second-order lambda-calculus.

→ Add (universal) quantification over TYPEs!
→ Straightforward extension of simply typed lambda-calculus by
two new constructs:

Type Abstraction: `λX. t`
Type Application: `t [T]`

For example, the polymorphic identity function

```
id = λX. λx:X. x
```



can be applied to `Nat` by writing `id [Nat]`. The result is

```
[X/Nat](λx:X. x)  =  λx:Nat.x
```

## 2. System F

What is the type of

`id = λX. λx:X. x`

→ If applied to a type T, `id` yields function of type T→T.

## 2. System F

What is the type of

`id = λX. λx:X. x`

→ If applied to a type T, `id` yields function of type T→T.

Therefore denote its type by: ∀X.X→X

Typing rules for type abstraction and type application:

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \qquad \frac{\Gamma \vdash t_1 : \forall X.T_{12}}{\Gamma \vdash t_1[T_2] : [X/T_2]T_{12}}$$

## 2. System F

Evaluation, like simply typed lambda (3 rules), plus two new rules:

$$\frac{t_1 \to t_2'}{t_1[T_2] \to t_1'[T_2]} \qquad (\lambda X.t_{12})[T_2] \to [X/T_2]t_{12}$$

Values (in the pure system) are

→ λx:T.t abstraction value
→ λX.t type abstraction value

Contexts contain x:T term variable binding
and X type variable binding

## 2. System F

**Examples.**

Polymorphic identity function: `id = λX.λx:X. x`

Apply it:

```
  id [Nat] 5
=  (λX.λx:X. x) [Nat] 5
→ [X/Nat](λx:X. x) 5
→ (λx:Nat. x) 5
→ [x/5](x)
→ 5
```

As we saw, the type of `id` is ∀X. X → X.

Can you find a function different from `id`, with the SAME TYPE??

## 2. System F

**Examples.**

Polymorphic doubling function:

```
double = λX. λf:X→X. λa:X. f (f a)
```

```
double [Nat] (λx:Nat. succ(succ(x))) 3
→7
```

```
quadruple = λX. double [X→X] (double [X])
```

What's the type of quadruple?

## 2. System F

**Examples.**

In simply typed lambda-calculus

```
omega = (λx.x x) (λx. x x)
```

canNOT be typed!

Neither can the self-application fragment `(λx.x x)`
In System-F we CAN type it:

```
self  =  λx: (∀X. X→X). x [∀X. X→X] x
self: (∀X.X→X) → (∀X.X→X)
```

## 2. System F

**Examples.**

In simply typed lambda-calculus

```
omega = (λx.x x) (λx. x x)
```

canNOT be typed!

Neither can the self-application fragment `(λx.x x)`
In System-F we CAN type it:

```
self  =  λx: (∀X. X→X). x [∀X. X→X] x
self: (∀X.X→X) → (∀X.X→X)
```

Apply `self` to some function; e.g., to `(λY.λy:Y. y)`

Polym. Fu's are "first class citizen"

## 2. System F

Main advantage of polymorphism:

→ many things need not be built into the language, but can be moved into **libraries**

Example. Lists.

Before:

For a type T: List T describes finite-length lists of elements from T.

New syntactic forms:

```
nil[T]
cons[T]  t1 t2
isnil[T] t
head[T]  t
tail[T]  t
```

## 2. System F

Main advantage of polymorphism:

→ many things need not be built into the language, but can be moved into **libraries**

Example. Lists.

Now:

`List X` describes finite-length lists of elements of type X.

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

## 2. System F

Now we can build a library of polymorphic operations on lists.
For example, a polymorphic `map` function.

```
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))))
```

What is the type of map?

## 2. System F

Now we can build a library of polymorphic operations on lists.
For example, a polymorphic `map` function.

```
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))))
```

What is the type of map?

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

head[Nat](map[Nat][Nat] (λx:Nat. succ x) l)
→5
```

## 3. Properties of System F

System F is *impredicative*:

→ Polymorphic types are universally quantified over the universe of ALL types. This includes polymorphic types themselves!

→ Polymorphic types are "1st class" citizens in the world of types

E.g. `(λf: (∀X.X→X). f) id`

ML (let) – polymorphims is *predicative*:

→ Polymorphic types are 2nd class. Arguments do not have polymorphic types! (prenex polymorphism)

E.g. `(fn f => fn x => f x) id 3`

## 3. Properties of System F

System F is *impredicative*:

→ Polymorphic types are universally quantified over the universe of ALL types. This includes polymorphic types themselves!

→ Polymorphic types are "1st class" citizens in the world of types

E.g. `(λf: (∀X.X→X). f) id`

→Type variables range only over quantifier-free types (*monotypes*)
→Quantified types (*polytypes*, or *type schemes*) not allowed on left of arrow

ML (let) – polymorphims is *predicative*:

→ Polymorphic types are 2nd class. Arguments do not have polymorphic types! prenex polymorphism

E.g. `(fn f => fn x => f x) id 3`

## 3. Properties of System F

**Parametricity**
Evaluation of polymorphic applications does not depend on the type that is supplied!

→ There is exactly one function of type ∀X.X→X (namely, the identity)

→There are exactly two functions of type ∀X.X→X→X which have different behavior.
Namely, `λX.λa:X.λb:X. a`
and `λX.λa:X.λb:X. b`

These do not (and cannot) alter their behavior depending on X!

## 3. Properties of System F

**Parametricity**
Evaluation of polymorphic applications does not depend on the type that is supplied!

Nevertheless, we defined a *type-passing semantics*:

$$(\lambda X.t_{12})[T_2] \rightarrow [X/T_2]t_{12}$$

Why do this,
if eval. does not depend on it?

→ type-erasure semantics:

After the typechecking phase, all types are erased!

## Erasure and Type Reconstruction

```
erase(x)       = x
erase(λx:T.t)  = λx.erase(t)
erase(t t')    = erase(t) erase(t')
erase(λX.t)    = erase(t)
erase(t [T])   = erase(t)
```

**Theorem**. (Wells, 1994): Let u be a closed term. It is undecidable, whether there is a well-typed system-F-term t with erase(t)=u.

→ Type Reconstruction for system F is not possible!

## Erasure and Type Reconstruction

Even if we leave intact all typing annotations, except the arguments to type applications:

```
perase(x)      = x
perase(λx:T.t) = λx:T.perase(t)
perase(t t')   = perase(t) perase(t')
perase(λX.t)   = λX.perase(t)
perase(t [T])  = perase(t) []
```

Then Type Reconstruction is still not possible!

**Theorem**. (Boehm, 1989): Let u be a closed term. It is undecidable, whether there is a well-typed system-F-term t with perase(t)=u.

## Erasure and Evaluation

```
erase(x)       = x
erase(λx:T.t)  = λx.erase(t)
erase(t t')    = erase(t) erase(t')
erase(λX.t)    = erase(t)
erase(t [T])   = erase(t')
```

We claimed that if t is well-typed, then t and erase(t) evaluate to the same.

Is this also true in the presence of side effects??

What about

```
let f = (λX.error) in 0
```

## Erasure and Evaluation

```
erase(x)       = x
erase(λx:T.t)  = λx.erase(t)
erase(t t')    = erase(t) erase(t')
erase(λX.t)    = λ_.erase(t)
erase(t [T])   = erase(t') dummyv
```

We claimed that if t is well-typed, then t and erase(t) evaluate to the same.

Is this also true in the presence of side effects??

NO! --- but can be fixed easily.

## 3. Properties of System F

**Uniqueness**
Every well-typed System-F-term has exactly one type.

**Preservation**
If $t \vdash t:T$ and $t \rightarrow t'$, then $\Gamma \vdash t':T$.

**Progress**
If t is a closed and well-typed term, then either
- t is a value, or
- t → t' for some term t'.

Proofs: straightforward induction on the structure of terms.

## 3. Properties of System F

**Normalization**
Every well-typed System-F-term `t` is normalizing, i.e.,

$\exists$`t'`: `t` $\rightarrow^*$ `t'` $\not\rightarrow$

Proof: very hard (Girard's PhD thesis, 1972)
→ later simplified to about 5 pages

Surprising: normalization holds even though MANY things can be coded in System F!

---

Can the (erased) term `(λx.x x)(λx.x x)` be typed in System F?


## 3. Properties of System F

**Normalization**
Every well-typed System-F-term `t` is normalizing, i.e.,

$\exists$`t'`: `t` $\rightarrow^*$ `t'` $\not\rightarrow$

Proof: very hard (Girard's PhD thesis, 1972)
→ later simplified to about 5 pages

Surprising: normalization holds even though MANY things can be coded in System F!

---

Can the (erased) term `(λx.x x)(λx.x x)` be typed in System F?


→ This can even be proved directly! Do EXERCISE 23.6.3 in TAPL!

## 3. Properties of System F

When is (partial) type reconstruction possible??

→ First-class existential types (e.g., using ML's `datatype` mechanism)

→ Add to that universal quantifiers which may appear in annotations of function arguments

In the presence of subtyping:

→ Local type inference

## 4. System F-Sub

Want to combine subtyping and polymorphism.
How?

`f = λx:{a:Nat}. x`        `: {a:Nat} → {a:Nat}`

`f {a=0}`
`→ {a=0}: {a:Nat}`        (works in any system)

`f {a=0, b=true}`
`→ {a=0, b=true} : {a:Nat}`        (using the subsumption rule)

result type has no b field!

`(f {a=0, b=true}).b` is ill-typed.
→ we cannot access the b field anymore!!

## 4. System F-Sub

Use polymorphic identity `fpoly` instead of `f`:

```
f = λx:{a:Nat}. x : {a:Nat} → {a:Nat}
fpoly = λX. λx:X. x  : (∀X.X→X)
```

## 4. System F-Sub

Use polymorphic identity `fpoly` instead of `f`:

```
f = λx:{a:Nat}. x : {a:Nat} → {a:Nat}
fpoly = λX. λx:X. x  : (∀X.X→X)

fpoly [{a:Nat, b:Bool}] {a=0, b=true}
→ {a=0, b=true} : {a:Nat, b:Bool}

    HURRA!
```

## 4. System F-Sub

```
f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)}

Has type  {a:Nat} → {orig:{a:Nat}, asucc:Nat}

fpoly = λX. λx:X. x
f2poly = λX. λx:X. {orig=x, asucc=succ(x.a)};
```

## 4. System F-Sub

```
f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)}

Has type  {a:Nat} → {orig:{a:Nat}, asucc:Nat}

fpoly = λX. λx:X. x
f2poly = λX. λx:X. {orig=x, asucc=succ(x.a)};
                   Type Error: Expected Record Type
```

## 4. System F-Sub

f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)}

Has type {a:Nat} → {orig:{a:Nat}, asucc:Nat}

fpoly = λX. λx:X. x

f2poly = λX. λx:X. {orig=x, asucc=succ(x.a)};

Type Error: Expected Record Type

f2 should take ANY record type, which has at least the field a: Nat.

## 4. System F-Sub

f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)}

Has type {a:Nat} → {orig:{a:Nat}, asucc:Nat}

fpoly = λX. λx:X. x

f2poly = λX. λx:X. {orig=x, asucc=succ(x.a)};

Type Error: Expected Record Type

f2 should take ANY record type, which has at least the field a: Nat.

= any subtype of {a:Nat}

X<:{a:Nat}

## 4. System F-Sub

f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)}

Has type {a:Nat} → {orig:{a:Nat}, asucc:Nat}

fpoly = λX. λx:X. x

f2poly = λ●. λx:X. {orig=x, asucc=succ(x.a)};

Type Error: Expected Record Type

f2 should take ANY record type, which has at least the field a: Nat.

= any subtype of {a:Nat}

X<:{a:Nat}

## 4. System F-Sub

**Bounded Quantification**

f2poly = λX<:{a:Nat}. λx:. {orig=x, asucc=succ(x.a)};

System $F_{<:}$

type abstraction: λX<:T

quantified type: ∀X<:T.T

In contexts we now have Γ, x:T, X<:T

Evaluation (nothing changes): $(\lambda X{<:}T.t_{12})[T_2] \rightarrow [X/T_2]t_{12}$

Typing rules for type abstraction and type application:

$$\frac{\Gamma, X{<:}T \vdash t_2 : T_2}{\Gamma \vdash \lambda X{<:}T.t_2 : \forall X{<:}T.T_2} \qquad \frac{\Gamma \vdash t_1 : \forall X{<:}T_{11}.T_{12} \quad \Gamma \vdash T_2{<:}T_{11}}{\Gamma \vdash t_1[T_2] : [X/T_2]T_{12}}$$

## 4. System F-Sub

**Bounded Quantification**

```
f2poly = λX<:{a:Nat}. λx:. {orig=x, asucc=succ(x.a)};
```

System $F_{<:}$

type abstraction: λX<:T
quantified type: ∀X<:T.T
In contexts we now have Γ, x:T, X<:T

Evaluation (nothing changes): $(\lambda X{<:}T.t_{12})[T_2] \rightarrow [X/T_2]t_{12}$

Typing rules for type abstraction and type application:

subtyping

$$\frac{\Gamma, X{<:}T \vdash t_2 : T_2}{\Gamma \vdash \lambda X{<:}T.t_2 : \forall X{<:}T.T_2} \qquad \frac{\Gamma \vdash t_1 : \forall X{<:}T_{11}.T_{12} \quad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1[T_2] : [X/T_2]T_{12}}$$

## 4. System F-Sub

Unbounded Quantification: ∀X.T := ∀X<:Top.T

Subtyping Quantified Types:

$$\frac{\Gamma, X{<:}U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X{<:}U_1.S_2 <: \forall X{<:}U_1.T_2}$$

"the *kernel* rule"

→ "kernel F-sub"

## 4. System F-Sub

Scoping:

Γ1 = X<:Top, y:X→Nat

Γ2 = y:X→Nat, X<:Top

Γ3 = X<:{a:Nat,b:X}

Γ4 = X<:{a:Nat,b:Y}, Y<:{c:Bool,d:X}

Which of these contexts are not well-scoped?

## 4. System F-Sub

Scoping:

Γ1 = X<:Top, y:X→Nat

Γ2 = y:X→Nat, X<:Top

Γ3 = X<:{a:Nat,b:X}

Γ4 = X<:{a:Nat,b:Y}, Y<:{c:Bool,d:X}

Which of these contexts are not well-scoped?

## 4. System F-Sub

Scoping:

Γ1 = X<:Top, y:X→Nat

Γ2 = y:X→Nat, X<:Top

Γ3 = X<:{a:Nat,b:X}

Γ4 = X<:{a:Nat,b:Y}, Y<:{c:Bool,d:X}

Which of these contexts are not well-scoped?

λX<:{a:Nat, b:X}

X<:{a:Nat, b:{a:Nat, b:Top}}

→ "F-bounded Quantification"

## 4. System F-Sub

F-Bounded Quantification:

→ used in GJ design

→ more complex than F-sub,

And "it only becomes really interesting when recursive types are also included ..

.. No non-recursive type can satisfy X<:{a:Nat,b:X}"

## 4. System F-Sub

→ In kernel F-sub, two quantified types can only be compared if their upper bounds are identical.

Similar to restricting the arrow rule

$$\frac{S_2 <: T_2}{U \to S_2 <: U \to T_2}$$

$$\frac{\Gamma, X<:U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X<:U_1.S_2 <: \forall X<:U_1.T_2}$$

## 4. System F-Sub

→ In kernel F-sub, two quantified types can only be compared if their upper bounds are identical.

Similar to restricting the arrow rule

$$\frac{S_2 <: T_2}{U \to S_2 <: U \to T_2}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X<:T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X<:S_1.S_2 <: \forall X<:T_1.T_2}$$

→ "full F-sub"

## 4. System F-Sub

Which types are related by the subtype relation

of full F-sub, but NOT in kernel F-sub???

$$\frac{\Gamma, X{<:}U_1 \vdash S_2{<:}T_2}{\Gamma \vdash \forall X{<:}U_1.S_2 \;<:\; \forall X{<:}U_1.T_2}$$

$$\frac{\Gamma \vdash T_1{<:}S_1 \qquad \Gamma, X{<:}T_1 \vdash S_2{<:}T_2}{\Gamma \vdash \forall X{<:}S_1.S_2 \;<:\; \forall X{<:}T_1.T_2}$$

## 4. System F-Sub

Which types are related by the subtype relation

of full F-sub, but NOT in kernel F-sub???

$$\frac{\Gamma, X{<:}U_1 \vdash S_2{<:}T_2}{\Gamma \vdash \forall X{<:}U_1.S_2 \;<:\; \forall X{<:}U_1.T_2}$$

$$\frac{\Gamma \vdash T_1{<:}S_1 \qquad \Gamma, X{<:}T_1 \vdash S_2{<:}T_2}{\Gamma \vdash \forall X{<:}S_1.S_2 \;<:\; \forall X{<:}T_1.T_2}$$

→ Are there any USEFUL ones???

## 5. Properties of F-Sub

**Preservation**
If $t \vdash t{:}T$ and $t \to t'$, then $\Gamma \vdash t'{:}T$.

**Progress**
If t is a closed and well-typed term, then either
- t is a value, or
- t → t' for some term t'.

Proofs: Induction on the structure of terms.

→Use canonical forms lemma:
If v is closed value of type $T_1 \to T_2$, then $v = \lambda x{:}S_1.t_2$

If v is closed value of type $\forall X{<:}T_1.T_2$, then $v = \lambda X{<:}T_1.t_2$.

## 5. Properties of F-Sub

**Theorem.**
Typing and Subtyping in kernel F-sub is **decidable**.

**Theorem.**
Subtyping in full F-sub is **undecidable**.

Next time: (1) prove these theorems.

(2) look at FGJ = FJ+generics.