Universal Types

Advanced Compiler Construction and Program Analysis

Lecture 12

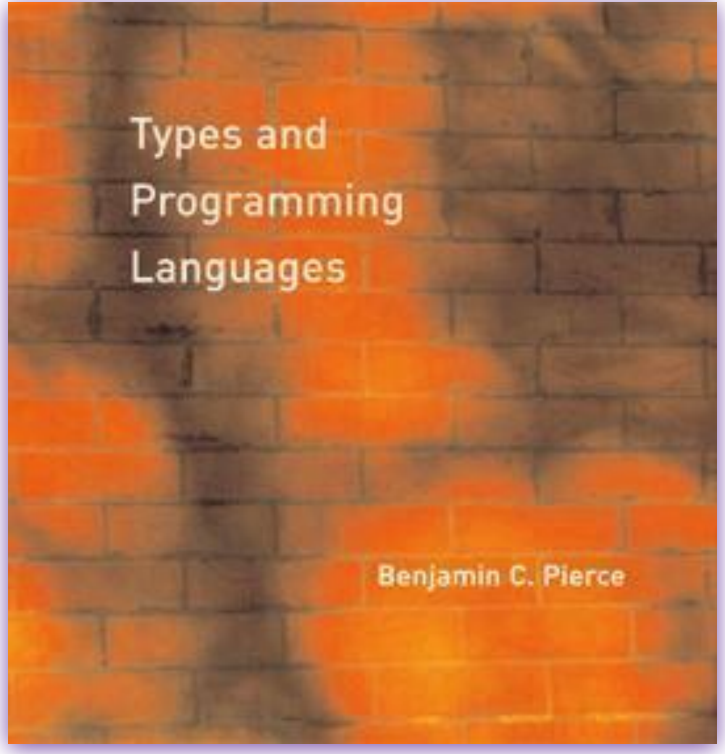
Innopolis University, Spring 2022

The topics of this lecture are covered in detail in...

Benjamin C. Pierce.

Types and Programming Languages

MIT Press 2002



Let-polymorphism

```
let twice = λf. λa. f(f(a)) in
let a = twice (λx:Nat. succ (succ x)) 2 in
let b = twice (λx:Bool. x) false in
if b then a else 0
```

Let-polymorphism

```
let twice = λf. λa. f(f(a)) in
let a = twice (λx:Nat. succ (succ x)) 2 in
let b = twice (λx:Bool. x) false in
if b then a else 0
```

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \qquad \Gamma \vdash [x \mapsto t_1]t_2 : T_2 \mid C_2}{\Gamma \vdash \textbf{let } x{=}t_1 \textbf{ in } t_2 : T_2 \mid C_1 \cup C_2}$$

Polymorphism

1. Parametric polymorphism
    a. “Generics”
    b. Impredicative polymorphism
2. Ad-hoc polymorphism
    a. Overloading
    b. Multi-method dispatch
    c. Intensional polymorphism
    d. Reflection
3. Subtype polymorphism

Simply Typed λ-calculus: syntax

`t ::=`	***terms***
`x`	*variable*
`λx:T.t`	*abstraction*
`t t`	*application*

`T ::=`	***types***
`T → T`	*function type*

`v ::=`	***values***
`λx:T.t`	*abstraction*

`Γ ::=`	***context***
`∅`	*empty context*
`Γ,x:T`	*variable*

System F: syntax

`t ::= ...`	***terms***
`x`	*variable*
`λx:T.t`	*abstraction*
`t t`	*application*
`λX.t`	*type abstraction*
`t [T]`	*type application*

`T ::=`	***types***
`X`	*type variable*
`T → T`	*function type*
`∀X.T`	*universal type*

`v ::= ...`	***values***
`λx:T.t`	*abstraction*
`λX.t`	*type abstraction*

`Γ ::=`	***context***
`∅`	*empty context*
`Γ,x:T`	*variable*
`Γ,X`	*type variable*

System F: syntax

t ::= ...		***terms***
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
	λX.t	*type abstraction*
	t [T]	*type application*

T ::=		***types***
	X	*type variable*
	T → T	*function type*
	∀X.T	*universal type*

v ::= ...		***values***
	λx:T.t	*abstraction*
	λX.t	*type abstraction*

Γ ::=		***context***
	∅	*empty context*
	Γ,x:T	*variable*
	Γ,X	*type variable*

System F: application

$$(\lambda x.t_1)\ t_2 \longrightarrow [x \mapsto t_2]t_1$$

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

System F: application

$$(\lambda x.t_1)\ t_2 \longrightarrow [x \mapsto t_2]t_1$$

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

```
let id = λX. λx:X. x in id [Nat]
```

System F: application

$$(\lambda x.t_1)\ t_2 \longrightarrow [x \mapsto t_2]t_1$$

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

```
let id = λX. λx:X. x in id [Nat]
⟶  (λX. λx:X. x) [Nat]
```

System F: application

$$(\lambda x.t_1)\ t_2 \longrightarrow [x \mapsto t_2]t_1$$

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

```
let id = λX. λx:X. x in id [Nat]
⟶  (λX. λx:X. x) [Nat]
⟶  [X ↦ Nat] (λx:X. x)    =  λx:Nat. x
```

System F: universal types

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

System F: universal types

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

$$\frac{\Gamma \vdash t : \forall X.T_1}{\Gamma \vdash t\ [T_2] : [X \mapsto T_2]T_1}$$

System F: examples

```
let id = λX. λx:X. x                                        in
```

System F: examples

```
let id = λX. λx:X. x                                   in
let a = id [Nat]                                       in
```

System F: examples

```
let id = λX. λx:X. x                                in
let a = id [Nat]                                    in
let b = id [Nat] 0                                  in
```

System F: examples

```
let id = λX. λx:X. x                                   in
let a = id [Nat]                                       in
let b = id [Nat] 0                                     in
let twice = λX. λf:X→X. λx:X. f (f x)                  in
```

System F: examples

```
let id = λX. λx:X. x                              in
let a = id [Nat]                                  in
let b = id [Nat] 0                                in
let twice = λX. λf:X→X. λx:X. f (f x)             in
let c = twice [Nat] (λn:Nat. succ n) 0            in
```

System F: examples

```
let id = λX. λx:X. x                              in
let a = id [Nat]                                  in
let b = id [Nat] 0                                in
let twice = λX. λf:X→X. λx:X. f (f x)            in
let c = twice [Nat] (λn:Nat. succ n) 0           in
let selfApp = λx:(∀X.X→X). x [∀X.X→X] x
in
```

System F: examples

```
let id = λX. λx:X. x                           in
let a = id [Nat]                               in
let b = id [Nat] 0                             in
let twice = λX. λf:X→X. λx:X. f (f x)          in
let c = twice [Nat] (λn:Nat. succ n) 0         in
let selfApp = λx:(∀X.X→X). x [∀X.X→X] x
in
let fourTimes
  = λX. double [X→X] (double [X]) in
```

System F: examples

```
let id = λX. λx:X. x                                in
let a = id [Nat]                                    in
let b = id [Nat] 0                                  in
let twice = λX. λf:X→X. λx:X. f (f x)               in
let c = twice [Nat] (λn:Nat. succ n) 0              in
let selfApp = λx:(∀X.X→X). x [∀X.X→X] x
in
let fourTimes
  = λX. double [X→X] (double [X]) in
fourTimes [Nat] (λn:Nat. succ (succ n)) 0
```

System F: polymorphic lists

```
nil      : ∀X. List[X]
cons : ∀X. X → List[X] → List[X]
isempty : ∀X. List[X] → Bool
head : ∀X. List[X] → X
tail : ∀X. List[X] → List[X]
```

System F: polymorphic lists exercise

```
nil      : ∀X. List[X]
cons : ∀X. X → List[X] → List[X]
isempty : ∀X. List[X] → Bool
head : ∀X. List[X] → X
tail : ∀X. List[X] → List[X]
```

Exercise 12.1. Implement map for polymorphic lists:

```
map : ∀X. ∀Y. (X → Y) → List[X] → List[Y]
```

System F: polymorphic lists exercise (2)

Exercise 12.2.
Implement these functions for polymorphic lists:

$$\texttt{reverse} : \forall X.\ \texttt{List[X]} \to \texttt{List[X]}$$

$$\texttt{sort} : \forall X.\ (X \to X \to \texttt{Bool}) \to \texttt{List[X]} \to \texttt{List[X]}$$

System F: properties

Theorem 12.3 [Preservation].

Let $\Gamma \vdash$ `t : T` and `t` $\longrightarrow$ `t'`, then $\Gamma \vdash$ `t' : T`.

Theorem 12.4 [Progress].
If `t` is a closed, well-typed term, then either t is a value, or else there is some `t'` such that `t` $\longrightarrow$ `t'`.

Theorem 12.5 [Normalization].
Well-typed System F terms are normalizing.

System F: type erasure and reconstruction

```
erase(x)          = x
erase(λx:T.t)     = λx.erase(t)
erase(t₁ t₂)      = erase(t₁) erase(t₂)
erase(λX.t)       = erase(t)
erase(t [T])      = erase(t)
```

System F: type erasure and reconstruction

```
erase(x)          = x
erase(λx:T.t)     = λx.erase(t)
erase(t₁ t₂)      = erase(t₁) erase(t₂)
erase(λX.t)       = erase(t)
erase(t [T])      = erase(t)
```

Theorem 12.6.
It is *undecidable* whether, given a closed term `m` of untyped lambda calculus, there is some well-typed term `t` in System F such that `erase(t) = m`.

System F: impredicativity

The kind of polymorphism in System F is called ***impredicative***, since quantified type variables may range over all types, including the type being defined:

$$T = \forall X. X \to X$$

In some languages (e.g. ML), polymorphism is ***predicative***, stratifying the types (using ranks or via let-polymorphism).

Hindley-Milner Type System: syntax

`t ::= ...`	***terms***
`x`	*variable*
`λx:T.t`	*abstraction*
`t t`	*application*
`let` `x=t` **`in`** `t`	*let-binding*
`T ::=`	***types***
`X`	*type variable*
`T → T`	*function type*
`∀X.T`	*universal type*

Hindley-Milner Type System: specialization

Specialization Rule

$$\frac{\tau' = \{\alpha_i \mapsto \tau_i\}\tau \quad \beta_i \notin \mathrm{free}(\forall\alpha_1 \ldots \forall\alpha_n . \tau)}{\forall\alpha_1 \ldots \forall\alpha_n . \tau \sqsubseteq \forall\beta_1 \ldots \forall\beta_m . \tau'}$$

https://en.wikipedia.org/wiki/Hindley–Milner_type_system

Hindley-Milner Type System: typing

$$\frac{\Gamma \vdash_D e_0 : \sigma \qquad \Gamma, x : \sigma \vdash_D e_1 : \tau}{\Gamma \vdash_D \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau} \quad [\texttt{Let}]$$

$$\frac{\Gamma \vdash_D e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash_D e : \sigma} \quad [\texttt{Inst}]$$

$$\frac{\Gamma \vdash_D e : \sigma \quad \alpha \notin \mathrm{free}(\Gamma)}{\Gamma \vdash_D e : \forall\, \alpha\ .\ \sigma} \quad [\texttt{Gen}]$$

Summary

- ❏ System F
- ❏ Hindley-Milner Type System

See you next time!