

Spring 2018

# Polymorphisms

Yu Zhang

**Course web site:** <http://staff.ustc.edu.cn/~yuzhang/fopl>

# Recap

- General recursion

- Fixpoint operator:  $\text{fix}_\sigma : (\sigma \rightarrow \sigma) \rightarrow \sigma$

- Fixpoints:  $\text{fix}_\sigma F = F(\text{fix}_\sigma F)$

$F \triangleq \lambda(f : \text{int} \rightarrow \text{int}). \lambda(x : \text{int}). \text{if } x = 0 \text{ then } 1 \text{ else } x * f(x - 1)$

---

**Factorial function is the following fixpoint**

$\text{fix}_{\text{int} \rightarrow \text{int}} (\lambda(f : \text{int} \rightarrow \text{int}). \lambda(x : \text{int}). \text{if } x = 0 \text{ then } 1 \text{ else } x * f(x - 1))$

---

- Type inference

- unification

# References

- [PFPL](#)
  - Chapter 16 System F of Polymorphic Types
    - System F: polymorphic typed lambda calculus
  - Chapter 17 Abstract Types
  - Chapter 18 Higher kinds
- [TAPL](#)
  - Chapter 22 Type Reconstruction
  - Chapter 23 Universal Types
  - Chapter 24 Existential Types
- Modules in OCaml (L7-L11 in Cornell [CS 3110](#))
- Stanford CS242 [Notes](#)

# Discussion

- [Interpreter.ml](#)

- Some, None, option

```
36 let rec typecheck (t : Term.t) : Type.t option =
37     match t with
38     | Term.Z -> Some (Type.Int)
39     | Term.S t' ->
40         let tau = typecheck t' in
41         (match tau with
42         | Some Type.Int -> Some (Type.Int)
43         | _ -> None)
44     | Term.True -> Some (Type.Bool)
```

```
16 module Type = struct
17     type t =
18         | Int
19         | Bool
20 end
```

# Outline

- Various polymorphisms
- Polymorphic types:  $\forall X.\tau$ 
  - Procedural abstraction
- Data abstraction: existential types  $\exists X.\tau$ 
  - Abstract data types, Generic abstractions
- Overloading and type classes
- Subtyping

# Various Polymorphisms

**Static** polymorphism: binding at compile-time

- *Parametric* polymorphism (参数化多态)
  - **polymorphism** (FPL), **templates**, **generics** (OO)
- *Ad-hoc* polymorphism
  - **Overloading**: function or operator overloading
  - **Coercion**: implicit type conversion

**Dynamic** polymorphism: binding at run-time

- *Subtyping* (inclusion) polymorphism
  - **inheritance**, **virtual function**

# (Parametric) Polymorphism

- Example: identity function

- write **different function for different type**:

$\lambda(x : \text{int}).x$      $\lambda(x : \text{int} \rightarrow \text{int}).x$

- But in Ocaml, we can write the function:

```
let id = fun x -> x
```

id : 'a -> 'a where 'a is type variable

- Typed lambda calculus

- For any type  $\alpha$ ,  $\lambda(x : \alpha).x$      $\Lambda\alpha.\lambda(x : \alpha).x$

- Type application:

$(\Lambda\alpha.\lambda(x : \alpha).x) [\text{int}]$   
 $\mapsto [\text{int} / \alpha](\lambda(x : \alpha).x) = \lambda(x : \text{int}).x$

- Features

- **Single algorithm may be given many types**
- **Type variable may be replaced by any type**

# More Examples

- Polymorphism occurs frequently in data structures

```
type 'a tree = Node of 'a tree * 'a * 'a tree | Leaf
let x : int list = [1; 2] in
let y : string list = ["a"; "b"] in
let z : int tree = Node (Leaf, 3, Node(Leaf, 2, Leaf))
```

'a tree is a  
polymorphic type

- **tree** is not a type but a **type constructor**: takes a type as input and returns a type
  - int tree
  - string tree
  - (int \* string) tree
  - ...

# $\lambda^{\rightarrow, \forall}$ Formal Semantics

- Grammar

Type  $\tau ::= \dots$

- |  $X$  type variable
- |  $\forall X.\tau$  polymorphic type

Term  $t ::= \dots$

- |  $\Lambda X.t$  type function -- introduction form of  $\forall X.\tau$
- |  $t [\tau]$  type application -- elimination form of  $\forall X.\tau$

- Statics

$\Delta$  contains type variables and  $\Gamma$  contains term variables

# $\lambda^{\rightarrow, \forall}$ Formal Semantics

- Grammar
- Statics:

Type  $\tau ::= \dots \mid X \mid \forall X.\tau$   
 Term  $t ::= \dots \mid \Lambda X.t \mid t [\tau]$

Well-formed types

$$\frac{}{\Delta, X \text{ type} \vdash X \text{ type}}$$

$$\Delta, X \text{ type} \vdash \tau \text{ type}$$

$$\frac{}{\Delta \vdash \forall X.\tau \text{ type}}$$

Typing rules of terms

$$\frac{\Delta, X \text{ type}; \Gamma \vdash t : \tau}{\Delta; \Gamma \vdash \Lambda X.t : \forall X.\tau} \text{ (T-tfn)}$$

$$\frac{\Delta; \Gamma \vdash t : \forall X.\tau_1}{\Delta; \Gamma \vdash t [\tau_2] : [\tau_2 / X].\tau_1} \text{ (T-tapp)}$$

- Lemma (Substitution)

1. If  $\Delta, X \text{ type} \vdash \tau' \text{ type}$  and  $\Delta \vdash \tau \text{ type}$ , then  $\Delta \vdash [\tau / X]\tau' \text{ type}$ .
2. If  $\Delta, X \text{ type}; \Gamma \vdash t' : \tau'$  and  $\Delta \vdash \tau \text{ type}$ ,  
 then  $\Delta; [\tau / X] \Gamma \vdash [\tau / X]t' : [\tau / X]\tau'$ .
3. If  $\Delta; \Gamma, x : \tau \vdash t' : \tau'$  and  $\Delta; \Gamma \vdash t : \tau$ , then  $\Delta; \Gamma \vdash [t / x]t' : \tau'$ .

# $\lambda^{\rightarrow, \forall}$ Formal Semantics

Type  $\tau ::= \dots \mid X \mid \forall X.\tau$   
Term  $t ::= \dots \mid \Lambda X.t \mid t [\tau]$

- Grammar
- Statics
- Dynamics

$$\frac{}{\Lambda X.t \text{ val}} \text{ (D-tfn)} \quad \frac{t \mapsto t'}{t [\tau] : t' [\tau]} \text{ (D-tapp}_1\text{)} \quad \frac{}{(\Lambda X.t) [\tau] \mapsto [\tau / X] t} \text{ (D-tapp}_2\text{)}$$

- Lemma (Canonical Forms)

If  $t : \tau$  and  $t \text{ val}$ , then

1. If  $\tau = \tau_1 \rightarrow \tau_2$ , then  $t = \lambda x : \tau_1. t_2$  with  $x : \tau_1 \vdash t_2 : \tau_2$ .
2. If  $\tau = \forall X.\tau'$ , then  $t = \Lambda X.t'$  with  $X \text{ type} \vdash t' : \tau'$ .

- Theorem (Safety)

(Preservation) If  $t : \tau$  and  $t \mapsto t'$ , then  $t' : \tau$ .

(Progress) If  $t : \tau$ , then either  $t \text{ val}$  or  $t \mapsto t'$  for some  $t'$ .

# Example

- Polymorphic composition function
- Polymorphic composition function type

# Example

- Polymorphic composition function

$\Lambda X. \Lambda Y. \Lambda Z. \lambda f : Y \rightarrow Z. \lambda g : X \rightarrow Y. \lambda x : X. f \ g \ x$

- Polymorphic composition function type

$\forall X. \forall Y. \forall Z. \underline{(Y \rightarrow Z)} \rightarrow \underline{(X \rightarrow Y)} \rightarrow \underline{X} \rightarrow Z$   
 $= \forall X. \forall Y. \forall Z. \underline{(Y \rightarrow Z)} \rightarrow \underline{(X \rightarrow Y)} \rightarrow \underline{(X \rightarrow Z)}$

# Data Abstraction

- Interface
  - A contract between the client and the implementor
- Implementations
  - Satisfy the contract
  - One implementation can be replaced by another without affecting the behavior of the client

Data abstraction is formalized by

extending **System F** with *existential types*

- **Interfaces**: *existential types* that provide a collection of operations acting on abstract type
- **Implementations**: packages, the introduction form of existential types

# Modules in OCaml

- Different implementations of Counter

```
module IntCounter = struct
  type t = int
  let make (n : int) : t = n
  let incr (ctr : t) (n : int) : t = ctr + n
  let get (ctr : t) : int = ctr
end
```

```
let ctr : IntCounter.t = IntCounter.make 3 in
let ctr : IntCounter.t = IntCounter.incr ctr 5 in
assert((IntCounter.get ctr) = 8);
assert(ctr = 8)
```

```
module RecordCounter = struct
  type t = { x: int }
  let make (n : int) : t = {x = n}
  let incr (ctr : t) (n : int) : t = {x = ctr.x + n}
  let get (ctr : t) : int = ctr.x
end
```

# Modules in OCaml

- Interfaces

- module signature

```
module type Counter = sig
  type t
  val make : int -> t
  val incr : t -> int -> t
  val get : t -> int
end
```

- Implementations

- modules

```
module IntCounter : Counter = struct
  type t = int
  let make (n : int) : t = n
  let incr (ctr : t) (n : int) : t = ctr + n
  let get (ctr : t) : int = ctr
end
```

```
module RecordCounter : Counter = struct
  type t = { x: int }
  let make (n : int) : t = {x = n}
  let incr (ctr : t) (n : int) : t = {x = ctr.x + n}
  let get (ctr : t) : int = ctr.x
end
```

# Modules in OCaml

- Interfaces

- module signature

```
module type Counter = sig
  type t
  val make : int -> t
  val incr : t -> int -> t
  val get : t -> int
end
```

- Implementations

- modules

```
module IntCounter : Counter = struct
  type t = int
  let make (n : int) : t = n
  let incr (ctr : t) (n : int) : t = ctr + n
  let get (ctr : t) : int = ctr
end
```

```
let ctr : Counter.t = IntCounter.make 3 in
let ctr : Counter.t = IntCounter.incr ctr 5 in
assert((IntCounter.get ctr) = 8);
assert(ctr = 8) ← ❌
```

# Packing of a Package

IntCounter can be represented as

$$\{\mathbf{int}, (\underbrace{(\lambda n : \mathbf{int}.n)}_{\mathit{make}}, \underbrace{(\lambda c : \mathbf{int}.\lambda n : \mathbf{int}.c + n)}_{\mathit{incr}}, \underbrace{(\lambda c : \mathbf{int}.c)}_{\mathit{get}})\}$$

**as**  $\exists X.((\mathbf{int} \rightarrow X) \times (X \rightarrow \mathbf{int} \rightarrow X) \times (X \rightarrow \mathbf{int}))$

*packing of a “package”* -- introduction form

- Package

- **Implementation**: the second term in the curly braces

$$(\underbrace{(\lambda n : \mathbf{int}.n)}_{\mathit{make}}, \underbrace{(\lambda c : \mathbf{int}.\lambda n : \mathbf{int}.c + n)}_{\mathit{incr}}, \underbrace{(\lambda c : \mathbf{int}.c)}_{\mathit{get}})$$

- **Interface**: the type after **as** keyword

$$\exists X.((\mathbf{int} \rightarrow X) \times (X \rightarrow \mathbf{int} \rightarrow X) \times (X \rightarrow \mathbf{int}))$$

- **Abstracted type**: the first term in the curly braces, i.e. **int**

# Unpacking: eliminating a package

- Unpack: enable a client to use a package

```
unpack {X, p} = ({int, (...)} as  $\exists X...$ ) in  
  let c : X = p.L.L 3 in  
  let c : X = p.L.R c 5 in  
  p.R c
```

```
let ctr : IntCounter.t = IntCounter.make 3 in  
let ctr : IntCounter.t = IntCounter.incr ctr 5 in  
IntCounter.get ctr
```

# $\lambda^{\rightarrow, \forall, \exists}$ Formal Semantics

- Grammar

Type  $\tau ::= \dots$

|  $\exists X.\tau$  existential type, i.e.  $\text{some}(X.\tau)$

Term  $t ::= \dots$

|  $\{\rho, t\}$  as  $\exists X.\tau$  type pack, introduction form of  $\exists X.\tau$

$t$  is the implementation of the package

$\rho$  is the actual representation type of  $X$

$\exists X.\tau$  is the interface with the abstracted type  $X$

|  $\text{unpack } \{X, x\} = t_1 \text{ in } t_2$  type unpack, elimination form of  $\exists X.\tau$

$t_1$  is a package by binding its representation type to  $X$

and its implementation to  $x$

$t_2$  is the client code to use the methods of a package

# $\lambda^{\rightarrow, \forall, \exists}$ Formal Semantics

- Grammar

Type  $\tau ::= \dots \mid \exists X.\tau$   
Term  $t ::= \dots \mid \{\rho, t\} \text{ as } \exists X.\tau \mid \text{unpack } \{X, x\} = t_1 \text{ in } t_2$

- Statics

$$\frac{\Delta, X \text{ type} \vdash \tau \text{ type}}{\Delta \vdash \exists X.\tau \text{ type}}$$

$$\frac{\Delta \vdash \rho \text{ type} \quad \Delta, X \text{ type} \vdash \tau \text{ type} \quad \Delta; \Gamma \vdash t : [\rho / X]\tau}{\Delta; \Gamma \vdash \{\rho, t\} \text{ as } \exists X.\tau : \exists X.\tau} \text{ (T-pack)}$$

$$\frac{\Delta; \Gamma \vdash t_1 : \exists X.\tau \quad \Delta, X \text{ type}; \Gamma, x : \tau \vdash t_2 : \tau' \quad \Delta \vdash \tau' \text{ type}}{\Delta; \Gamma \vdash \text{unpack } \{X, x\} = t_1 \text{ in } t_2 : \tau'} \text{ (T-unpack)}$$

$\tau'$  是client代码的结果类型

# $\lambda^{\rightarrow, \forall, \exists}$ Formal Semantics

- Grammar  $\text{Type } \tau ::= \dots \mid \exists X.\tau$   
 $\text{Term } t ::= \dots \mid \{\rho, t\} \text{ as } \exists X.\tau \mid \text{unpack } \{X, x\} = t_1 \text{ in } t_2$
- Statics
- Dynamics  $\frac{}{\{\rho, t\} \text{ as } \exists X.\tau \text{ val}} \text{ (D-pack)}$   
 $\frac{t_1 \mapsto t_1'}{\text{unpack } \{X, x\} = t_1 \text{ in } t_2 \mapsto \text{unpack } \{X, x\} = t_1' \text{ in } t_2} \text{ (D-unpack}_1)$   
 $\frac{}{\text{unpack } \{X, x\} = \{\rho, t\} \text{ as } \exists X.\tau \text{ in } t_2 \mapsto [\rho / X, t / x] t_2} \text{ (D-unpack}_2)$

# \*Definability of Existential Types

- 引入存在类型的原因：对数据抽象进行建模
- 存在类型可由多态类型（全称类型）定义
  - 如下的client代码  $t_2$  是以表示类型  $X$  为参数的多态函数

$\text{unpack } \{X, x\} = t_1 \text{ in } t_2$

$$\frac{\Delta; \Gamma \vdash t_1 : \exists X. \tau \quad \Delta, X \text{ type}; \Gamma, x : \tau \vdash t_2 : \tau' \quad \Delta \vdash \tau' \text{ type}}{\Delta; \Gamma \vdash \text{unpack } \{X, x\} = t_1 \text{ in } t_2 : \tau'} \quad (\text{T-unpack})$$

- $t_1 : \exists X. \tau$  是一个package (具体实现),  $t_2 : \tau'$  是client代码
- Client代码本质上是类型为  $\forall X. \tau \rightarrow \tau'$  的多态函数,  $X$  可能出现在  $\tau$  中, 但是不会出现在  $\tau'$  中
- 存在类型是一个多态函数类型  $\exists X. \tau = \forall Y. (\forall X. \tau \rightarrow Y) \rightarrow Y$

# \*Definability of Existential Types

- 存在类型可由多态类型（全称类型）定义
  - 存在类型是一个多态函数类型  $\exists X.\tau = \forall Y.(\forall X.\tau \rightarrow Y) \rightarrow Y$

**打包**  $\{\rho, t\}$  as  $\exists X.\tau$  相当于  $\Lambda Y.\lambda y : (\forall X.\tau \rightarrow Y).y [\rho] t$

由表示类型  $\rho$  和实现  $t$  组成的包是一个多态函数，该函数在给定结果类型  $Y$  和 client 代码  $y$  时，用表示类型  $\rho$  实例化  $y$ ，再将实现  $t$  传递到其中 ( $y [\rho]$ )。

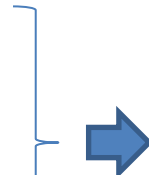
**解包**  $\text{unpack } \{X, x\} = t_1 \text{ in } t_2$  相当于  $t_1[\tau'] (\Lambda X.\lambda x : \tau.t_2)$

将 client 代码  $y$  打包成一个多态函数  $\Lambda X.\lambda x : \tau.t_2$ ，将存在类型的结果类型（即  $\forall Y.(\forall X.\tau \rightarrow Y) \rightarrow Y$  中的  $Y$ ）实例化为  $\tau'$ ，再将  $t_1[\tau']$  应用到多态 client 程序  $y$  上

$t_1$  最终为一个 pack 值，即为一个多态函数

$\Lambda Y.\lambda y : (\forall X.\tau \rightarrow Y).y [\rho] t$

# Type Quantification is not sufficient

- Quantification over types
  - $\forall X.\tau$  models generics
  - $\exists X.\tau$  models abstraction

Not sufficient to model many programming situations of practical interest.
- Examples (not just type quantification)
  - Abstract families of types
    - e.g.  $\tau$  list An infinite collection types sharing a common collection of operations on them
  - Interrelated abstract types
    - e.g. a type of trees whose nodes have a forest of child and a type of forests whose elements are trees

# \*Constructors and Kinds

- Quantification over **kinds**, than just types, e.g. over
  - **type constructors**: functions mapping types to types
  - **type structures**: tuples of types
- Kinds: classifying constructors
  - **Static layer**: use **kinds** to classify **constructors**
  - **Dynamic layer**: use **types** to classify **expressions(terms)**

The two-layer architecture models *phase distinction*.

- Constructors are the **static data** of the language.
- Expressions (terms) are the **dynamic data** of the language.

# $\lambda^{\rightarrow, \forall_k, \exists_k}$ Grammar

<b>Kind</b> $\kappa ::= \text{Type}$	the kind of types	<b>Type</b> $\tau ::= c$	
$\kappa_1 \rightarrow \kappa_2$	type constructors	<b>Term</b> $t ::= x$	variable
$\kappa_1 \times \kappa_2$	type structures	$\lambda x : \tau. t$	abstraction
<b>Cons</b> $c ::= X$	type variable	$t_1 t_2$	application
$c \rightarrow c$	function type	$\Lambda X :: \kappa. t$	type abstraction
$\forall X :: \kappa. c$	universal type	$t [c]$	type application
$\lambda X :: \kappa. c$	operator abstraction	...	
$c_1 [c_2]$	operator application		
$(c_1, c_2)$	operator pair		
$c.d$	operator projection, d: L R		
<b>unit</b>			

# \*Constructors and Kinds

- More details are not discussed in this course.
- But if you are interested, you need further learn to understand:
  - **More judgements** specifying static semantics of constructors and kinds
    - Constructor / type /expression formation
  - **Rules for constructor formation**
  - **Substitution lemma**
  - ...

# \*Modularity

- References: [PFPL Chapters 42-44]
- Syntax is divided into more levels
  - Expressions classified by types
  - Constructors classified by kinds
  - Modules classified by signatures

# Summary: Generic Abstractions

- Parameterize modules by types
- Create general implementations
  - Can be instantiated in many ways
- Language examples
  - Ada generic packages
  - C++ templates, e.g. C++ Standard Template Library(STL)
  - ML functors
  - ...