# Polymorphisms

Yu Zhang

**Course web site:** http://staff.ustc.edu.cn/~yuzhang/tpl

# Recap

- General recursion
  - Fixpoint operator: $\mathbf{fix}_\sigma : (\sigma \to \sigma) \to \sigma$
  - Fixpoints: $\mathbf{fix}_\sigma F = F(\mathbf{fix}_\sigma\ F)$

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

**Factorial function is the following fixpoint**

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

- 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 (pdf)
  - 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

```
let rec typecheck (t : Term.t) : Type.t option =
  match t with
  | Term.Z -> Some (Type.Int)
  | Term.S t' ->
    let tau = typecheck t' in
    (match tau with
      | Some Type.Int -> Some (Type.Int)
      | _ -> None)
  | Term.True -> Some (Type.Bool)
```

```
module Type = struct
  type t =
    | Int
    | Bool
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:

    $\lambda(x:\text{int}).x \quad \lambda(x:\text{int}\rightarrow\text{int}).x$
  - 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 $\alpha$, $\lambda(x:\alpha).x$ $\quad \Lambda\alpha.\lambda(x:\alpha).x$
    - 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

- System F - Syntax

**Type** $\tau$ ::= ...

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

**Term** $t$ ::= ...

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

- Statics

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

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

- Syntax

$$\textbf{Type } \tau ::= \ldots \mid X \mid \forall X.\tau$$
$$\textbf{Term } t ::= \ldots \mid \Lambda X.t \mid t\,[\tau]$$

- Statics:

Well-formed types

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

Typing rules of terms

$$\frac{\Delta, X \text{ type}; \Gamma \vdash t : \tau}{\Delta; \Gamma \vdash \Lambda X.t : \forall X.\tau}\textbf{(T-tfn)} \qquad \frac{\Delta; \Gamma \vdash t : \forall X.\tau_1}{\Delta; \Gamma \vdash t\,[\tau_2] : [\tau_2 / X].\tau_1}\textbf{(T-tapp)}$$

- 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** $\Delta; \Gamma \vdash [t / x]t' : \tau'$.

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

$$\text{Type } \tau ::= \ldots \mid X \mid \forall X.\tau$$
$$\text{Term } t ::= \ldots \mid \Lambda X.t \mid t\,[\tau]$$

- Syntax
- Statics
- Dynamics

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

- Lemma (Canonical Forms)

  If $t : \tau$ and $t$ val, then

  1. If $\tau = \tau_1 \to \tau_2$, 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 $X$ type $\vdash t' : \tau'$.

- Theorem (Safety)

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

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

# Example

- Polymorphic composition function

- Polymorphic composition function type

# Example

- Polymorphic composition function

$$\Lambda X.\Lambda Y.\Lambda Z.\lambda \underline{f : Y \to Z}.\lambda \underline{g : X \to Y}.\lambda \underline{x : X}.f\ g\ x$$

- Polymorphic composition function type

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

# 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

$$\{\mathbf{int}, ((\underbrace{\lambda n:\mathbf{int}.n}_{make}), (\underbrace{\lambda c:\mathbf{int}.\lambda n:\mathbf{int}.c+n}_{incr}), (\underbrace{\lambda c:\mathbf{int}.c}_{get}))\}$$
$$\textbf{as } \exists X.((\mathbf{int} \to X)\times(X \to \mathbf{int} \to X)\times(X \to \mathbf{int}))$$

*packing* of a "package" -- introduction form

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

    $$((\underbrace{\lambda n:\mathbf{int}.n}_{make}), (\underbrace{\lambda c:\mathbf{int}.\lambda n:\mathbf{int}.c+n}_{incr}), (\underbrace{\lambda c:\mathbf{int}.c}_{get}))$$
  - Interface: the type after **as** keyword

    $$\exists X.((\mathbf{int} \to X)\times(X \to \mathbf{int} \to X)\times(X \to \mathbf{int}))$$
  - Abstracted type: the first term in the curly braces, i.e. **int**

# Unpacking: eliminating a package

- Unpack: enable a client to use a package

$$\textbf{unpack}\ \{X, p\} = (\{\text{int}, (\ldots)\}\ \textbf{as}\ \exists X \ldots)\ \text{in}$$

$$\text{let}\ c : X = p.L.L\ 3\ \text{in}$$

$$\text{let}\ c : X = p.L.R\ c\ 5\ \text{in}$$

$$p.R\ c$$

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

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

- Syntax

Type $\tau$ ::= ...

$\mid \exists X.\tau$ existential type, i.e. some$(X.\tau)$

Term $t$ ::= ...

$\mid \{\rho, t\}$ 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 $X$

$\mid$ unpack $\{X, x\} = t_1$ 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

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

- Syntax

$$\textbf{Type } \tau ::= \ldots \mid \exists X.\tau$$

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

- 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

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

- Syntax

$$\textbf{Type } \tau ::= \ldots \mid \exists X.\tau$$

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

- Statics
- Dynamics

$$\frac{}{\{\rho, t\} \textbf{ as } \exists X.\tau \ \textbf{ val}} \textbf{(D-pack)}$$

$$\frac{t_1 \mapsto t_1'}{\textbf{unpack } \{X, x\} = t_1 \textbf{ in } t_2 \mapsto \textbf{unpack } \{X, x\} = t_1' \textbf{ in } t_2} \textbf{(D-unpack}_1\textbf{)}$$

$$\frac{}{\textbf{unpack } \{X, x\} = \{\rho, t\} \textbf{ as } \exists X.\tau \textbf{ in } t_2 \mapsto [\rho / X, t / x]\, t_2} \textbf{(D-unpack}_2\textbf{)}$$

# *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

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

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

  - $t_1 : \exists X.\tau$ 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 $\forall X.\tau \to \tau'$ 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 $\exists X.\tau = \forall Y.(\forall X.\tau \to Y) \to Y$

# *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 $\exists X.\tau = \forall Y.(\forall X.\tau \to Y) \to Y$

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

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** unpack $\{X, x\} = t_1$ 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 $\forall Y.(\forall X.\tau \to Y) \to Y$ 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

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

# 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.

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

| | |
|---|---|
| Kind $\kappa$ ::= Type | the kind of types |
| $\mid \kappa_1 \to \kappa_2$ | type constructors |
| $\mid \kappa_1 \times \kappa_2$ | type structures |
| Cons $c$ ::= $X$ | type variable |
| $\mid c \to c$ | function type |
| $\mid \forall X :: \kappa.c$ | universal type |
| $\mid \lambda X :: \kappa.c$ | operator abstraction |
| $\mid c_1[c_2]$ | operator application |
| $\mid (c_1, c_2)$ | operator pair |
| $\mid c.d$ | operator projection, d: L\|R |
| $\mid$ unit | |

Type $\tau$ ::= $c$

| | |
|---|---|
| Term $t$ ::= $x$ | variable |
| $\mid \lambda x : \tau.t$ | abstraction |
| $\mid t_1\ t_2$ | application |
| $\mid \Lambda X :: \kappa.t$ | type abstraction |
| $\mid t\ [c]$ | type application |
| $\mid \ldots$ | |

# *Constructors and Kinds

- More details are not discussed in this course.
- But if you are interest, 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
  - …