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: $\mathbf{fix}_\sigma F = F(\mathbf{fix}_\sigma F)$

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

    Factorial function is the following fixpoint

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

```
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: $\forall(t.\tau)$
  - Procedural abstraction
- Data abstraction: existential types $\exists(t.\tau)$
  - 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:
    $\boldsymbol{\lambda}(x:\mathbf{int}).x \quad \boldsymbol{\lambda}(x:\mathbf{int}\rightarrow\mathbf{int}).x$
  - 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 α, $\lambda(x{:}\alpha).x$ $\quad \Lambda(\alpha)\,\lambda(x{:}\alpha)x$
    - 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
    - …

System F Formal Semantics

- System F - Syntax

Typ	τ	::=	t	t	variable
			$\mathtt{arr}(\tau_1;\tau_2)$	$\tau_1\rightarrow\tau_2$	function
			$\mathtt{all}(t.\tau)$	$\forall(t.\tau)$	polymorphic
Exp	e	::=	x	x	
			$\mathtt{lam}\{\tau\}(x.e)$	$\lambda(x:\tau)\,e$	abstraction
			$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$	application
			$\mathtt{Lam}(t.e)$	$\Lambda(t)\,e$	type abstraction
			$\mathtt{App}\{\tau\}(e)$	$e[\tau]$	type application

- Statics

  Δ contains type variables and Γ contains term variables

System F Formal Semantics

- Syntax

  Typ τ ::= $\cdots \mid t \mid \forall(t.\tau)$
  Exp e ::= $\cdots \mid \Lambda(t)e \mid e[t]$
- Statics:

  Well-formed types $\dfrac{}{\Delta, t\ \text{type} \vdash t\ \text{type}}$ $\qquad \dfrac{\Delta, t\ \text{type} \vdash \tau\ \text{type}}{\Delta \vdash \forall(t.\tau)\ \text{type}}$

  Typing rules of terms $\dfrac{\Delta, t\ \text{type}\ \ \Gamma \vdash e:\tau}{\Delta\ \Gamma \vdash \Lambda(t)e:\forall(t.\tau)}$ $\qquad \dfrac{\Delta\ \Gamma \vdash e:\forall(t.\tau')\quad \Delta \vdash \tau\ \text{type}}{\Delta\ \Gamma \vdash e[\tau]:[\tau/t]\,\tau'}$
- Lemma (Substitution)
  - If $\Delta, t\ \text{type} \vdash \tau'\ \text{type}$, and $\Delta \vdash \tau\ \text{type}$, then $\Delta \vdash [\tau/t]\tau'\ \text{type}$
  - If $\Delta, t\ \text{type} \vdash e':\tau'$, and $\Delta \vdash \tau\ \text{type}$, then $\Delta\ [\tau/t]\Gamma \vdash [\tau/t]e':[\tau/t]\tau'$
  - If $\Delta\ \Gamma, x:\tau \vdash e':\tau'$, and $\Delta\ \Gamma \vdash e:\tau$, then $\Delta\ \Gamma \vdash [e/x]e':\tau'$

Formal Semantics

- Syntax
- Statics
- Dynamics

  Typ τ ::= $\cdots \mid t \mid \forall(t.\tau)$
  Exp e ::= $\cdots \mid \Lambda(t)e \mid e[t]$

  $\dfrac{}{\Lambda(t)e\ \text{val}}$ $\qquad \dfrac{e \mapsto e'}{e[\tau] \mapsto e'[\tau]}$ $\qquad \dfrac{}{(\Lambda(t)e)[\tau] \mapsto [\tau/t]e}$
- Lemma (Canonical Forms)

  If $e:\tau$ and e val
  - If $\tau = \tau_1 \rightarrow \tau_2$, then $e = \lambda(x:\tau_1)e_2$ with $x:\tau_1 \vdash e_2:\tau_2$
  - If $\tau = \forall(t.\tau')$, then $e = \Lambda(t)\,e'$ with $t\ \text{type} \vdash e':\tau'$
- Theorem (Safety)
  - Preservation: If $e:\tau$ and $e \rightarrow e'$, then $e':\tau$
  - Progress: If $e:\tau$, then either e val or there exists e' such that $e \rightarrow e'$

Example

- Polymorphic composition function
- Polymorphic composition function type

Example

- Polymorphic composition function

$$\Lambda(t_1)\Lambda(t_2)\Lambda(t_3)\lambda(f\colon t_2 \to t_3)\lambda(g\colon t_1 \to t_2)\lambda(x\colon t_1)\ f(g(x))$$

- Polymorphic composition function type

$$\forall(t_1.\forall(t_2.\forall(t_3.(t_2 \to t_3) \to (t_1 \to t_2) \to t_1 \to t_3)))$$
$$= \forall(t_1.\forall(t_2.\forall(t_3.(t_2 \to t_3) \to (t_1 \to t_2) \to (t_1 \to t_3))))$$

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
```

```
IntCounter.t = int
make: int → int
incr:int → int → int
get: int → int

RecorderCounter.t={x:int}
make: int → {x:int}
incr:{x:int} → int → {x:int}
get:{x:int}→ int
```


Modules in OCaml

- Interfaces
  - module signature
- Implementations
  - modules

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

```
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
- Implementations
  - modules

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

```
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

pack int **with**
$\langle \text{make} \hookrightarrow \lambda(n\colon \text{int})n, \text{incr} \hookrightarrow \lambda(c\colon \text{int})\,\lambda(n\colon \text{int})c + n, \text{get} \hookrightarrow \lambda(c\colon \text{int})c\rangle$
as $\exists t.\langle \text{make} \hookrightarrow \text{int} \to t, \text{incr} \hookrightarrow t \to \text{int} \to t, \text{get} \hookrightarrow t \to \text{int}\rangle$

packing of a "package" -- introduction form

- Package
  - Implementation: the second term in the curly braces
    $\langle \text{make} \hookrightarrow \lambda(n\colon \text{int})n, \text{incr} \hookrightarrow \lambda(c\colon \text{int})\,\lambda(n\colon \text{int})c + n, \text{get} \hookrightarrow \lambda(c\colon \text{int})c\rangle$
  - Interface: the type after **as** keyword
    $\exists t.\langle \text{make} \hookrightarrow \text{int} \to t, \text{incr} \hookrightarrow t \to \text{int} \to t, \text{get} \hookrightarrow t \to \text{int}\rangle$
  - Abstracted type: the first term in the curly braces, i.e. **int**

Unpacking: eliminating a package

- Unpack: enable a client to use a package

open
(**pack** int **with**
⟨make ↪ $\lambda(n:\text{int})n$, incr ↪ $\lambda(c:\text{int})\ \lambda(n:\text{int})c+n$, get ↪ $\lambda(c:\text{int})c$⟩
as $\exists t.$⟨make ↪ int → t, incr ↪ t → int → t, get ↪ t → int⟩)
as t **with** $x:\tau$
in let $c:t=x.$make 3 in let $c:t=x.$incr c 3 in $x.$get c

open e_1 **as** t **with** $x:\tau$ **in** e_2
打开一个包e_1，将其表示类型int绑定到t，将其实现绑定到x，以便在客户端 e_2中使用

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


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

- Syntax

Typ	τ	::=	some$(t.\tau)$	$\exists(t.\tau)$	interface
Exp	e	::=	pack$\{t.\tau\}\{\rho\}(e)$	pack ρ with e as $\exists(t.\tau)$	implementation
			open$\{t.\tau\}\{\rho\}(e_1;t,x.e_2)$	open e_1 as t with $x:\tau$ in e_2	client

pack ρ **with** e **as** $\exists(t.\tau)$
把表示类型ρ 和实现e打成满足接口$\exists(t.\tau)$的一个包，t为抽象类型

open e_1 **as** t **with** $x:\tau$ **in** e_2
打开一个包e_1，将其表示类型int绑定到t，将其实现绑定到x，
以便在客户端 e_2中使用

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

- Syntax

some$(t.\tau)$	$\exists(t.\tau)$
pack$\{t.\tau\}\{\rho\}(e)$	**pack** ρ **with** e **as** $\exists(t.\tau)$
open$\{t.\tau\}\{\tau_2\}(e_1;t;x.e_2)$	**open** e_1 **as** t **with** $x:\tau$ **in** e_2

- Statics

Well-formed types

$$\frac{\Delta, t\ \text{type} \vdash \tau\ \text{type}}{\Delta \vdash \text{some}(t.\tau)\ \text{type}}$$

Typing rules of terms

$$\frac{\Delta \vdash \rho\ \text{type} \quad \Delta, t\ \text{type} \vdash \tau\ \text{type} \quad \Delta\Gamma \vdash e:[\rho/t]\tau}{\Delta\,\Gamma \vdash \text{pack}\{t.\tau\}\{\rho\}(e):\text{some}(t.\tau)}$$

$$\frac{\Delta\ \Gamma \vdash e_1:\text{some}(t.\tau) \quad \Delta, t\ \text{type}\ \ \Gamma, x:\tau \vdash e_2:\tau_2 \quad \Delta \vdash \tau_2\ \text{type}}{\Delta\,\Gamma \vdash \text{open}\{t.\tau\}\{\tau_2\}(e_1;t;x.e_2):\tau_2}$$

τ_2是client代码的结果类型

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

- Syntax

some$(t.\tau)$	$\exists(t.\tau)$
pack$\{t.\tau\}\{\rho\}(e)$	**pack** ρ **with** e **as** $\exists(t.\tau)$
open$\{t.\tau\}\{\tau_2\}(e_1;t;x.e_2)$	**open** e_1 **as** t **with** $x:\tau$ **in** e_2

- Statics
- Dynamics

$$\frac{[e\ \text{val}]}{\text{pack}\{t.\tau\}\{\rho\}(e)\ \text{val}}$$

$$\left[\frac{e \mapsto e'}{\text{pack}\{t.\tau\}\{\rho\}(e) \mapsto \text{pack}\{t.\tau\}\{\rho\}(e')}\right]$$

$$\frac{e_1 \mapsto e_1'}{\text{open}\{t.\tau\}\{\tau_2\}(e_1;t;x.e_2) \mapsto \text{open}\{t.\tau\}\{\tau_2\}(e_1';t;x.e_2)}$$

$$\frac{[e\ \text{val}]}{\text{open}\{t.\tau\}\{\tau_2\}(\text{pack}\{t.\tau\}\{\rho\}(e);t;x.e_2) \mapsto [\rho,e/t,x]e_2}$$

*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 e_2 是以表示类型 X 为参数的多态函数

open$\{t.\tau\}\{\tau_2\}(e_1;t;x.e_2)$

$$\frac{\Delta\ \Gamma \vdash e_1:\text{some}(t.\tau) \quad \Delta, t\ \text{type}\ \ \Gamma, x:\tau \vdash e_2:\tau_2 \quad \Delta \vdash \tau_2\ \text{type}}{\Delta\,\Gamma \vdash \text{open}\{t.\tau\}\{\tau_2\}(e_1;t;x.e_2):\tau_2}$$

  - $e_1:\exists(t.\tau)$ 是一个package (具体实现)，$e_2:\tau_2$是client代码
  - Client代码本质上是类型为$\forall(t.\tau \to \tau_2)$ 的多态函数， t 可能出现在 τ 中，但是不会出现在τ_2中
  - 存在类型是一个多态函数类型

$$\exists(t.\tau) = \forall(u.\forall(t.\tau \to u) \to u)$$

*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$\exists(t.\tau) = \forall(u.\forall(t.\tau \to u) \to u)$

打包 **pack** ρ **with** e **as** $\exists(t.\tau)$相当于$\Lambda(u)\lambda(x:\forall(t.\tau \to u))x[\rho](e)$
由表示类型 ρ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 u 和 client代码 x 时，用 表示类型 ρ 实例化 t ，再将实现 e 传递到其中（ $x[\rho]$） .

解包 **open** e_1 **as** t **with** $x:\tau$ **in** e_2 相当于$e_1[\tau_2](\Lambda(t)\lambda(x:\tau)\ e_2)$
将client代码 e_2 打包成一个多态函数 $\forall(t.\tau \to u)$，将存在类型的结果类型（即$\forall(u.\forall(t.\tau \to u) \to u)$ 中的 u ）实例化为τ_2,再将 $e_1[\tau_2]$ 应用到多态client程序 e_2 上
e_1最终为一个pack值，即为一个多态函数

$$\Lambda(u)\lambda(x:\forall(t.\tau \to u))x[\rho](e)$$

Type Quantification is not sufficient

- Quantification over types
  - $\forall(t.\tau)$ models generics
  - $\exists(t.\tau)$ models abstraction

  ⇨ Not sufficient to model many programming situations of practical interest.
- Examples (not just type quantification)
  - Abstract families of types
    - e.g. τ 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.

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

Kind	κ	::=	`Type`	`T`	types
			`Unit`	1	nullary product
			`Prod`$(\kappa_1;\kappa_2)$	$\kappa_1 \times \kappa_2$	binary product
			`Arr`$(\kappa_1;\kappa_2)$	$\kappa_1 \rightarrow \kappa_2$	function
Con	c	::=	u	u	variable
			`arr`	$\rightarrow$	function constructor
			`all`$\{\kappa\}$	$\forall_\kappa$	universal quantifier
			`some`$\{\kappa\}$	$\exists_\kappa$	existential quantifier
			`proj[l]`(c)	$c \cdot$ `l`	first projection
			`proj[r]`(c)	$c \cdot$ `r`	second projection
			`app`$(c_1;c_2)$	$c_1[c_2]$	application
			`unit`	$\langle\rangle$	null tuple
			`pair`$(c_1;c_2)$	$\langle c_1,c_2\rangle$	pair
			`lam`$(u.c)$	$\lambda(u)\,c$	abstraction

*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
  - …