

# Polymorphism

---

Lecture 19

CS 565

4/17/08

# The Limitations of $F_1$ (simply-typed $\lambda$ -calculus)

---

- In  $F_1$  each function works exactly for one type
- Example: the identity function
  - $id = \lambda x:\tau. x : \tau \rightarrow \tau$
  - We need to write one version for each type
  - Even more important:  $sort : (\tau \rightarrow \tau \rightarrow bool) \rightarrow \tau \text{ array} \rightarrow unit$
- The various sorting functions differ only in typing
  - At runtime they perform exactly the same operations
  - We need different versions only to keep the type checker happy
- Two alternatives:
  - Circumvent the type system (see C, Java, ...), or
  - Use a more flexible type system that lets us write only one sorting function

# Polymorphism

---

- Informal definition

A function is polymorphic if it can be applied to “many” types of arguments

- Various kinds of polymorphism depending on the definition of “many”

- subtype (or bounded) polymorphism

“many” = all subtypes of a given type

- ad-hoc polymorphism

“many” = depends on the function

choose behavior at runtime (depending on types, e.g. sizeof)

- parametric predicative polymorphism

“many” = all monomorphic types

- parametric impredicative polymorphism

“many” = all types

# Parametric Polymorphism: Types as Parameters (System F)

---

- We introduce type variables and allow expressions to have variable types
- We introduce polymorphic types

$$\tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t \mid \forall t. \tau$$

$$e ::= x \mid \lambda x:\tau. e \mid e_1 e_2 \mid \Lambda t. e \mid e[\tau]$$

- $\Lambda t. e$  is type abstraction (or generalization)
- $e[\tau]$  is type application (or instantiation)
- Examples:
  - $\text{id} = \Lambda t. \lambda x:t. x \quad : \quad \forall t. t \rightarrow t$
  - $\text{id}[\text{int}] = \lambda x:\text{int}. x \quad : \quad \text{int} \rightarrow \text{int}$
  - $\text{id}[\text{bool}] = \lambda x:\text{bool}. x \quad : \quad \text{bool} \rightarrow \text{bool}$
  - "id 5" is invalid. Use "id [int] 5" instead

# Impredicative Polymorphism

- The typing rules:

$$\frac{x : \tau \text{ in } \Gamma}{\Gamma \vdash x : \tau} \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda t. e : \forall t. \tau} \quad t \text{ does not occur in } \Gamma$$

$$\frac{\Gamma \vdash e : \forall t. \tau'}{\Gamma \vdash e[\tau] : [\tau/t]\tau'}$$

# Impredicative Polymorphism (Cont.)

- Verify that “id [int] 5” has type int
- Note the side-condition in the rule for type abstraction
  - Prevents ill-formed terms like:  $\lambda x:t.\Lambda t.x$
- The evaluation rules are just like those of  $F_1$ 
  - This means that type abstraction and application are all performed at compile time
  - We do not evaluate under  $\Lambda$  ( $\Lambda t. e$  is a value)
  - We do not have to operate on types at run-time
  - This is called phase separation: type checking and execution

# Observations

---

- Based on the type of a term we can prove properties of that term
- There is only one value of type  $\forall t.t \rightarrow t$ 
  - The polymorphic identity function
- There is no value of type  $\forall t.t$
- Take the function:  $\text{reverse} : \forall t. t \text{ List} \rightarrow t \text{ List}$ 
  - This function cannot inspect the elements of the list
  - It can only produce a permutation of the original list
  - If  $L_1$  and  $L_2$  have the same length and let "match" be a function that compares two lists element-wise according to an arbitrary predicate
  - then "match  $L_1 L_2$ "  $\equiv$  "match (reverse  $L_1$ ) (reverse  $L_2$ )" !

# Expressiveness of Impredicative Polymorphism

---

- This calculus is called
  - $F_2$
  - system F
  - second-order  $\lambda$ -calculus
  - polymorphic  $\lambda$ -calculus
- Polymorphism is extremely expressive
- We can encode many base and structured types in  $F_2$

# Encoding Base Types in F<sub>2</sub>

---

## □ Booleans

- $\text{bool} = \forall t. t \rightarrow t \rightarrow t$  (given any two things, select one)
- There are exactly two values of this type!
  - $\text{true} = \Lambda t. \lambda x:t. \lambda y:t. x$
  - $\text{false} = \Lambda t. \lambda x:t. \lambda y:t. y$
- $\text{not} = \lambda b:\text{bool}. \Lambda t. \lambda x:t. \lambda y:t. b [t] y x$

## □ Naturals

- $\text{nat} = \forall t. (t \rightarrow t) \rightarrow t \rightarrow t$  (given a successor and a zero element, compute a natural number)
- $0 = \Lambda t. \lambda s:t \rightarrow t. \lambda z:t. z$
- $n = \Lambda t. \lambda s:t \rightarrow t. \lambda z:t. s (s (s \dots s(n)))$
- $\text{add} = \lambda n:\text{nat}. \lambda m:\text{nat}. \Lambda t. \lambda s:t \rightarrow t. \lambda z:t. n [t] s (m [t] s z)$
- $\text{mul} = \lambda n:\text{nat}. \lambda m:\text{nat}. \Lambda t. \lambda s:t \rightarrow t. \lambda z:t. n [t] (m [t] s) z$

# Expressiveness of $F_2$

---

- Polymorphic application
  - `selfApp =`  
 $\lambda x: \forall t. t \rightarrow t . \lambda y:t. x [\forall t. t \rightarrow t] x : (\forall t. t \rightarrow t) \rightarrow (\forall t. t \rightarrow t)$
  - `double =`  $\Lambda t. \lambda f:t \rightarrow t. \lambda a:t. f(f(a)) : (\forall t. t \rightarrow t) \rightarrow t \rightarrow t$
  - `quadruple =`  
 $\Lambda t. \text{double}[t \rightarrow t] (\text{double}[t]) : (\forall t. t \rightarrow t) \rightarrow t \rightarrow t$
- Recursive types?
  - We can encode primitive recursion but not full recursion

# Assessment

---

- Impredicative variant of System F very expressive:
  - can express complex polymorphic functions
    - type abstraction and application
  - limited form of self-application
  - prove interesting theorems based on type structure
  - complicated semantics
    - termination proof
    - type reconstruction

# Predicative Polymorphism

---

- Restriction: type variables can be instantiated only with monomorphic types
- This restriction can be expressed syntactically

$$\tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid \dagger$$
$$\sigma ::= \tau \mid \forall t. \sigma \mid \sigma_1 \rightarrow \sigma_2$$
$$e ::= x \mid e_1 e_2 \mid \lambda x:\sigma. e \mid \Lambda t.e \mid e [\tau]$$

- Type application is restricted to mono types
  - Cannot apply "id" to itself anymore
- 
- Same typing rules
  - Simple semantics and termination proof
  - Type reconstruction still undecidable
  - Must restrict further !

# Prenex Predicative Polymorphism

---

- Restriction: polymorphic type constructor at top level only
- This restriction can also be expressed syntactically

$$\tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t$$

$$\sigma ::= \tau \mid \forall t. \sigma$$

$$e ::= x \mid e_1 e_2 \mid \lambda x:\tau. e \mid \Lambda t.e \mid e [\tau]$$

- Type application is restricted to mono types (i.e., predicative)
- Abstraction only on mono types
- The only occurrences of  $\forall$  are at the top level of a type

$(\forall t. t \rightarrow t) \rightarrow (\forall t. t \rightarrow t)$  is not a valid type

- Same typing rules
- Simple semantics and termination proof
- Decidable type inference !