

Introduction to System F

Lecture 18

CS 565

4/20/09

The Limitations of F_1 (simply-typed λ -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 Λ ($\Lambda t. e$ 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 λ -calculus
 - polymorphic λ -calculus
 - Polymorphism is extremely expressive
 - We can encode many base and structured types in F_2
-

Encoding Base Types in F₂

Encoding Base Types in F₂

□ 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)
 - $0 = \Lambda t. \lambda s:t \rightarrow t. \lambda z:t. z$
 - $n = \Lambda t. \lambda s:t \rightarrow t. \lambda z:t. s (s \dots s(n))$
 - $\text{add} = \lambda n:\text{nat}. \lambda m:\text{nat}. \Lambda t. \lambda s:t \rightarrow t. \lambda z:t. n [t] s (m [t] s z)$
 - $\text{mul} = \lambda n:\text{nat}. \lambda m:\text{nat}. \Lambda t. \lambda s:t \rightarrow t. \lambda z:t. n [t] (m [t] s) z$
-

Expressiveness of F_2

□ We can encode similarly:

- $\tau_1 + \tau_2$ as $\forall t. (\tau_1 \rightarrow t) \rightarrow (\tau_2 \rightarrow t) \rightarrow t$
- $\tau_1 \times \tau_2$ as $\forall t. (\tau_1 \rightarrow \tau_2 \rightarrow t) \rightarrow t$
- unit as $\forall t. t \rightarrow t$

□ Polymorphic application:

- $\text{selfApp} = \lambda x:\forall t.t \rightarrow t. x[\forall t. t \rightarrow t] x : (\forall t.t \rightarrow t) \rightarrow (\forall t.t \rightarrow t)$
- $\text{double} = \Lambda t.\lambda f:t \rightarrow t.\lambda a:t. f(f(a)) : \forall t.(t \rightarrow t) \rightarrow t \rightarrow t$
- $\text{quadruple} = \Lambda t.\text{double} [t \rightarrow t] (\text{double} [t]) : \forall t. (t \rightarrow t) \rightarrow t \rightarrow t$

□ We cannot encode $\mu t.\tau$

- We can encode primitive recursion but not full recursion
 - All terms in F_2 have a termination proof in second-order Peano arithmetic (Girard, 1971): *strongly normalizing*
 - This is the set of naturals defined using zero, successor, induction along with quantification both over naturals and over sets of naturals
-

Predicative Polymorphism

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

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

- 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 !
-

Expressiveness of Prenex Predicative F_2

- We have simplified too much !

- Not expressive enough to encode nat, bool
 - $\text{bool} = \forall t. t \rightarrow t \rightarrow t$
 - $\text{true} = \Lambda t. \lambda x:t. \lambda y:t. x$
 - $\text{false} = \Lambda t. \lambda x:t. \lambda y:t. y$
 - But such encodings are only of theoretical interest anyway

- Is it expressive enough in practice?
 - Almost
 - Cannot write something like
$$(\lambda s:\forall t.\tau. \dots s [\text{nat}] x \dots s [\text{bool}] y) (\Lambda t. \dots \text{code for sort})$$
 - Because the type of formal argument s cannot be polymorphic

ML's Polymorphic Let

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{let } x : \sigma = e_1 \text{ in } e_2 : \tau}$$

ML's Polymorphic Let

- ML solution: slight extension of the predicative F_2
 - Introduce "let $x : \sigma = e_1$ in e_2 "
 - With the semantics of " $(\lambda x : \sigma. e_2) e_1$ "
 - And typed as " $[e_1/x] e_2$ "

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{let } x : \sigma = e_1 \text{ in } e_2 : \tau}$$

- This lets us write the polymorphic sort as

let

$s : \forall t. \tau = \Lambda t. \dots$ code for polymorphic sort ...

in

$\dots s [\text{nat}] x \dots s [\text{bool}] y$

- Surprise: this was a major ML design flaw!
-

ML Polymorphism and References

- let is evaluated using call-by-value but is typed using call-by-name
 - What if there are side effects ?

- Example:

```
let x :  $\forall t. (t \rightarrow t)$  ref =  $\Delta t. \text{ref } (\lambda x : t. x)$ 
```

```
in
```

```
  x [bool] :=  $\lambda x : \text{bool}. \text{not } x$ 
```

```
  (! x [int]) 5
```

```
end
```

- Will apply "not" to 5
 - Similar examples can be constructed with exceptions
- It took 10 years to find and agree on a clean solution
-

The Value Restriction in ML

- A type in a let is generalized only for syntactic values

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{let } x : \sigma = e_1 \text{ in } e_2 : \tau} \quad \begin{array}{l} e_1 \text{ is a syntactic} \\ \text{value or } \sigma \text{ is} \\ \text{monomorphic} \end{array}$$

- Since e_1 is a value, its evaluation cannot have side-effects
 - In this case call-by-name and call-by-value are the same
 - In the previous example $\text{ref } (\lambda x:t. x)$ is not a value
 - This is not too restrictive in practice !
-