

CS6848 - Principles of Programming Languages

Principles of Programming Languages

V. Krishna Nandivada

IIT Madras

Parametric Polymorphism - System F

- System F discovered by Jean-Yves Girard (1972)
- Polymorphic lambda-calculus by John Reynolds (1974)
- Also called second-order lambda-calculus - allows quantification over types, along with terms.



Recap

- Extensions to simply typed lambda calculus.
- Pairs, Tuples and records



System F

- Definition of System F - an extension of simply typed lambda calculus.

Lambda calculus recall

- Lambda abstraction is used to abstract terms out of terms.
- Application is used to supply values for the abstract types.

System F

- A mechanism for abstracting types of out terms and fill them later.
- A new form of abstraction:
 - $\lambda X.e$ – parameter is a type.
 - Application – $e[t]$
 - called type abstractions and type applications (or instantiation).



Type abstraction and application

-

$$(\lambda X.e)[t_1] \rightarrow [x \rightarrow t_1]e$$

Examples

-

$$id = \lambda X.\lambda x : X.x$$

Type of id : $\forall X.X \rightarrow X$

$$applyTwice = \lambda X.\lambda f : X \rightarrow X.\lambda a : X.f (f a)$$

Type of $applyTwice$: $\forall X.(X \rightarrow X) \rightarrow X \rightarrow X$



Evaluation

-

$$\text{type application 1} \quad \frac{e_1 \rightarrow e'_1}{e_1[t_1] \rightarrow e'_1[t_1]}$$

-

$$\text{type application 2} \quad (\lambda X.e_1)[t_1] \rightarrow [X \rightarrow t_1]e_1$$



Extension

- Expressions:

$$e ::= \dots | \lambda X.e | e[t]$$

- Values

$$v ::= \dots | \lambda X.e$$

- Types

$$t ::= \dots | \forall X.t$$

- typing context:

$$A ::= \phi | A, x : t | A, X$$



Typing rules

-

$$\text{type abstraction} \quad \frac{A, X \vdash e_1 : t_1}{A \vdash \lambda X.e_1 : \forall X.t_1}$$

-

$$\text{type application} \quad \frac{A \vdash e_1 : \forall X.t_1}{A \vdash e_1[t_2] : [X \rightarrow t_2]t_1}$$



Examples

- $id = \lambda X. \lambda x : X. x$

$id : \forall X. X \rightarrow X$

type application: $id [Int] : Int \rightarrow Int$

value application: $id [Int] 0 = 0 : Int$

- $applyTwice = \lambda X. \lambda f : X \rightarrow X. \lambda a : X. f (f a)$

$ApplyTwiceInts = applyTwice [Int]$

$applyTwice [Int] (\lambda x : Int. succ(succx)) 3$
 $= 7$



Example

- Recall: Simply typed lambda calculus - we cannot type $\lambda x.x x$.
- How about in System F?
- $selfApp : (\forall X. X \rightarrow X) \rightarrow (\forall X. X \rightarrow X)$



Polymorphic lists

List of uniform members

- $nil : \forall X. List \rightarrow X$
- $cons : \forall X. X \rightarrow List X \rightarrow List X$
- $isnil? : \forall X. List X \rightarrow bool$
- $head : \forall X. List X \rightarrow X$
- $tail : \forall X. List X \rightarrow List X$



(Recall) Type inference algorithm (Hindley-Milner)

Input: G : set of type equations (derived from a given program).

Output: Unification σ

- 1 failure = false; $\sigma = \{\}$.
- 2 while $G \neq \emptyset$ and \neg failure do
 - 1 Choose and remove an equation e from G . Say $e\sigma$ is $(s = t)$.
 - 2 If s and t are variables, or s and t are both Int then continue.
 - 3 If $s = s_1 \rightarrow s_2$ and $t = t_1 \rightarrow t_2$, then $G = G \cup \{s_1 = t_1, s_2 = t_2\}$.
 - 4 If $(s = Int$ and t is an arrow type) or vice versa then failure = true.
 - 5 If s is a variable that does not occur in t , then $\sigma = \sigma \circ [s := t]$.
 - 6 If t is a variable that does not occur in s , then $\sigma = \sigma \circ [t := s]$.
 - 7 If $s \neq t$ and either s is a variable that occurs in t or vice versa then failure = true.
- 3 end-while.
- 4 if (failure = true) then output "Does not type check". Else o/p σ .



- $a = \lambda x. \lambda y. x$
- $b = \lambda f. (f\ 3)$
- $c = \lambda x. (+(\text{head } x)\ 3)$
- $d = \lambda f. ((f\ 3), (f\ \lambda y. y))$



- Ensures that we get finite types.
- If we allow recursive types - the occurs check can be omitted.
 - Say in $(s = t)$, $s = A$ and $t = A \rightarrow B$. Resulting type?
 - Example term?

