CS6848 - Principles of Programming Languages

Principles of Programming Languages

V. Krishna Nandivada

IIT Madras

Recap

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

Polymorphism - motivation

- AppTwiceInt = $\lambda f : \text{Int} \rightarrow \text{Int} . \lambda x : \text{Int} . f\ (f\ x)$
  AppTwiceRcd = $\lambda f : (l : \text{Int}) \rightarrow (l : \text{Int}) . \lambda x : (l : \text{Int}) . f\ (f\ x)$
  AppTwiceOther =
  $\lambda f : (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int}) . \lambda x : (\text{Int} \rightarrow \text{Int}) . f\ (f\ x)$
- Breaks the idea of abstraction: Each significant piece of functionality in a program should be implemented in just one place in the source code.

Polymorphism - variations

- Type systems allow single piece of code to be used with multiple types are collectively known as polymorphic systems.
- Variations:
  - Parametric polymorphism: Single piece of code to be typed generically (also known as, let polymorphism, first-class polymorphism, or ML-style polymorphic).
    - Restricts polymorphism to top-level `let` bindings.
    - Disallows functions from taking polymorphic values as arguments.
    - Uses variables in places of actual types and may instantiate with actual types if needed.
    - Example: ML, Java Generics

```
(let ((apply  lambda f. lambda a (f a)))
  (let ((a (apply succ 3)))
    (let ((b (apply zero? 3))) ...
```

  - Ad-hoc polymorphism - allows a polymorphic value to exhibit different behaviors when viewed using different types.
    - Example: function Overloading, Java `instanceof` operator.
  - subtype polymorphism: A single term may get many types using subsumption.

polymorphism.

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.

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

- $$(\lambda X.e)[t_1] \to [X \to t_1]e$$

Examples

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

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

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

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

Extension

- Expressions:
  $$e ::= \cdots | \lambda X.e | e[t]$$
- Values
  $$v ::= \cdots | \lambda X.e$$
- Types
  $$t ::= \cdots | \forall X.t$$
- typing context:
  $$A ::= \phi | A, x : t | A, X$$

Evaluation

- $$\text{type application 1} \;\frac{e_1 \to e_1'}{e_1[t_1] \to e_1'[t_1]}$$
- type appliation 2 — $(\lambda X.e_1)[t_1] \to [X \to t_1]e_1$

Typing rules

- $$\text{type abstraction} \;\frac{A, X \vdash e_1 : t_1}{A \vdash \lambda X.e_1 : \forall X.t_1}$$
- $$\text{type application} \;\frac{A \vdash e_1 : \forall X.t_1}{A \vdash e_1[t_2] : [X \to t_2]t_1}$$

Examples

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

  $id : \forall X.X \to X$

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

  value application: id[Int] $0 = 0$: Int
- $applyTwice = \lambda X.\lambda f : X \to X.\lambda a : Xf\ (f\ a)$

  $ApplyTwiceInts = applyTwice$ [Int]

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

Polymorphic lists

List of uniform members

- `nil` : $\forall X.List\ \ X$
- `cons`: $\forall X.X \to List\ X \to List\ X$
- `isnil`: $\forall X.List\ X \to bool$
- `head`: $\forall X.List\ X \to X$
- `tail`: $\forall X.List\ X \to List\ X$

Example

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

Church literals

Booleans

- `tru` = $\lambda t.\lambda f.t$
- `fls` = $\lambda t.\lambda f.f$

- Idea: A predicate will return `tru` or `fls`.
- We can write `if pred s1 else s2` as (`pred s1 s2`)

Building on booleans

- `and` = $\lambda b.\lambda c.b\ c$ `fls`
- `or` = ? $\lambda b.\lambda c.b$ `tru` c
- `not` = ?

Building pairs

- `pair` = $\lambda f.\lambda s.\lambda b.b\ f\ s$
- To build a pair: `pair v w`
- `fst` = $\lambda p.p$ `tru`
- `snd` = $\lambda p.p$ `fls`

Church numerals

- $c_0 = \lambda s.\lambda z.\ z$
- $c_1 = \lambda s.\lambda z.\ s\ z$
- $c_2 = \lambda s.\lambda z.\ s\ s\ z$
- $c_3 = \lambda s.\lambda z.\ s\ s\ s\ z$

Intuition

- Each number n is represented by a combinator c_n.
- c_n takes an argument s (for successor) and z (for zero) and apply s, n times, to z.
- c_0 and `fls` are exactly the same!
- This representation is similarto the unary representation we studies before.
- `scc` = $\lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$

(Recall) Type inference algorithm (Hindley-Milner)

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

1. failure = `false`; σ = $\{\}$.
2. while $G \neq \phi$ 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 $s = s_1 \to s_2$ and $t = t_1 \to t_2$, 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 $\sigma = \sigma\ o\ [s := t]$.
   6. If t is a variable that does not occur in s, then $\sigma = \sigma\ o\ [t := s]$.
   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.

Examples - derive the types

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

"Occurs" check

- Ensures that we get finite types.
- If we allow recursive types - the occurs check can be omitted.
  - Say in $(s = t)$, $s = A$ and $t = A \to B$. Resulting type?
- What if we are interested in System F - what happens to the type inference? (undecidable in general)

Self study.