Tutorial on Parametric Polymorphism

Hongseok Yang
Seoul National University

Polymorphism in OOL

- Means that a single method definition is used for multiple types.
- Current technology: subtyping, type testing, type casting.

```
class Y { Object m(Object s) {
            if (s instanceof Int) { return(2); }
            return(s); } }
Y y; 4 + ((Int)y.m(3)); "Ba" + ((String)y.m("Bo"));
```

- Coming technology: generics.

```
class Y { C m<C>(C s) { return(s); } }
Y y; 4 + (y.m<int>(3)); "Ba" + (y.m<string>("Bo"));
```

- Question:
  1. Can you implement the above generic method in Java?
  2. What are so special about generic methods?

Strachey and Reynolds's answer for the second question

- Strachey classifies polymorphic methods (or functions) into two categories:
  - Parametrically polymorphic method: behaves the same for all types.
  - Ad-hoc polymorphic method: behaves differently depending on types.
- Strachey's answer: generic methods are precisely parametrically polymorphic ones.
- Reynolds formalized Strachey's intuition using relational parametricity in his 1974,1983 papers.
- Goal: to understand Reynolds's formal answer.

Polymorphic Type

- Language for constructing "sets."

  t ::= int | t!t | X | 8X. t | ΠX. t
- «int!int¬: functions from integers to integers.
- «ΠX. X!X¬: all polymorphic functions from X to X.
  - Given a set S, f(S) is a function from S to S.
  - E.g. ΛS. λs. if (S = int) then 2 else s

    ΛS. λs. s
  - Formally, «ΠX. X!X¬ = Π_{S2SET}(S!S).
- «8X. X!X¬: parametrically polymorphic functions from X to X.
- Fact: «8X. X!X¬ is a subset of «ΠX. X!X¬.

Relational Parametricity

- Ex: Define a parametricity condition for f 2 «ΠX. X!X¬.
- Hint1: Use the below examples.
  - non-parametric fn: adhoc = ΛS. λs. if (S = int) then 2 else s
  - parametric fn: id = ΛS. λs. s
- Hint2: Parametric fns do not look at the set parameter X. Formalize this using relations between sets.
- Relational parametricity: preservation of all relations.

  8S,S'2SET. 8R:S$S'.

  8s2S,s'2S'. (s[R]s') f(S)(s)[R]f(S')(s'))
- Ex: Show that id is parametric, but adhoc is not.
- «8X. X!X¬ = { f2«ΠX. X!X¬ | f is rel. parametric }

Benefits of Parametricity

- Free theorem: id is the only element.

  «8X. X!X¬ = { id }
- The compilers can use such a fact:
  - Every generic method "X m<X>(X x)" immediately returns the parameter. (Slight exaggeration ☺)
  - Thus, "m<C>(o)" can be optimized by "o".
  - No need to look at the implementation of m.
- Ex: Prove the free theorem.
  - Given f 2 «8X. X!X¬, set S, and element s2S.
  - Need to show f(S)(s) = s.
  - Now, it is your turn.
  - Hint: Use the relation R:S${0} = {(s,0)}.

Parametricity for Other Types

- f2«ΠX. X!int¬: Given a set S, f(S) is a function from S to integers.
  - E.g. "ΛS.λs. 3", "ΛS.λs. if (S=int) then s+1 else 4".
  - Ex: Define the para. condition for «8X.X!int¬.
  - Hint: Use the identity relation {(n,n)| n2int} for int.
- f2«ΠX. X!(X!X)¬: Given a set S, f(S) take two values from S, and returns an integer.
  - E.g "ΛS. λs_1. λs_2. s_1", "ΛS. λs_1. λs_2. s_2"
  - Ex: Define the para. condition for «8X.X!(X!X)¬.
- HW:
  - Define the para. condition for «8X. (int!X)!X¬.
  - Show that «8X. (int!X)!X¬ = { ΛS.λk.(k n) j n 2 int}

Polymorphic Lambda Calculus

- Invented by Girard and Reynolds independently.

t ::= X | t!t | 8X.t | int

M ::= x | M M | λx:t. M | M[t] | ΛX. M | n

- Contains type abstraction and type application.
- Does not include type test and type casting.
- Its type system ensures that only parametrically polymorphic functions are definable in the language.
- E.g.
  - (ΛX. λx:X. x) : 8X.X!X
  - (λf: (8X.X!X). f[int!int] (f[int] 3)) : int
- Forms the basis of generics in the coming Java and C#, and polymorphism in ML and HASKELL.

Theoretical Results

Core polymorphic lambda calculus (AKA system F):

t ::= X | t!t | 8X.t

M ::= x | M M | λx:t. M | M[t] | ΛX. M

- Strongly normalizing: all terms terminate no matter how you evaluate them.
- Very expressive:
  - Intergers can be encoded by Church numerals.
  - In that encoding, most of the "useful" total recursive functions can be expressed in the language.
    - All primitive recursive functions.
    - Ackerman functions.

Further Development

- Rel. parametricity as a device for ensuring independence:
  - Haskell: used to safely incorporate imperative computation:

    runST t: (8s. ST s t) ! t
  - Information flow: used to formalize that high-level security values are not used in a computation. [Abadi et. al].
- Data abstraction by the "dual" of rel. parametricity:
  - Abstract data type by existential type (9X. t) [Plotkin&Mitchell].
  - The dual of rel. param. ensures the soundness of simulation.
- Type inference of system F: Almost done.
  - undecidable, but when limited to let-polymorphism, decidable.
- Parametricity semantics of system F: Still active.
  - No easy set-theoretic semantics [Plotkin&Reynolds].
  - Various category-theoretic semantics [Birkedal&Rasmus, Dunphy&Reddy, etc].

Conclusion

- Explained parametric polymorphism, relational parametricity, and system F.
- Instance of Reynolds's comment about what a type system is:

  "A type system is a syntactic discipline for ensuring a level of abstraction."
- The type system of system F ensures that only param. polymorphic fns can be defined.
- Param. polymorphic fns maintain the abstraction of type parameter X:
  - Never ask what a type (or set) parameter X is.
  - Formally, satisfy relational parametricity.