

Type inference in examples

©Andrey Kruglyak, 2010

Wednesday, April 28, 2010

1

Overall algorithm

- ◀ Type inference is done separately in each module
- ◀ The goal is to find a single (unambiguous, possibly polymorphic) principal (most useful in all contexts) type for each top-level name and each term
- ◀ Several stages:
 - ◀ collect type information from this module and all imported modules (including Prelude): types, typeclasses, type instances, subtyping information
 - ◀ collect type constraints for terms, solve the constraints and compare calculated types to declared types (type signatures)

Wednesday, April 28, 2010

2

Overall algorithm

- ◀ Several stages (cont'd):
 - ◀ simplify typing constraints (e.g. remove reflexive subtyping pairs $a < a$, use transitivity of subtyping $a < b$ & $b < c \Rightarrow a < c$, use co/contravariance $a \rightarrow b \ \backslash \ a < A, B < b \Rightarrow A \rightarrow B$)
 - ◀ "defaulting":
 - eliminate constraints of type $a < A$ (a is replaced by A) to prevent type explosion
 - utilize the default relation between instances of typeclasses to eliminate ambiguity (e.g. in " $\backslash x \rightarrow \text{show (read x)}$ ")Defaulting changes the type and possibly the behavior of the term!

Wednesday, April 28, 2010

3

In the examples below...

```
struct S1 where
  getX :: Request Int
  incX :: Action
```

```
struct S2 < S1 where
  setX :: Int -> Action
```

```
struct S3 < S1 where
  id :: Int
  getY :: Request String
  setY :: String -> Action
```

```
struct S4 < S2,S3 where
  reset :: Action
```

```
instance s3Eq :: Eq S3 where
  a == b = a.id == b.id
  a /= b = a.id /= b.i
```

Wednesday, April 28, 2010

4

Example 1

```
< fcn1 x y = x+y

x :: t1 (we assume that t1 is a fresh free type variable)
y :: t2 (we assume that t2 is a fresh free type variable)
(+) :: a -> a -> a \Num a, a
=> t1 = t2, Num t1
=> x+y :: t1
=> fcn1 :: t -> t -> t \Num t, t (t here is a bound variable - for any t)

< If we add a type signature:
fcn1 :: Int -> Int -> Int => Num Int exists => OKAY, simply limits
applicability of the function
fcn1 :: S1 -> S1 -> S1 => Cannot solve typing constraint Num S1
fcn1 :: Int -> Float -> Int => Cannot solve typing constraint Float < Int
```

Examples 2 and 3

```
< fcn2 x y = x/y

x :: t1
y :: t2
(/) :: Float -> Float -> Float
=> t1 = t2 = Float
=> fcn2 :: Float -> Float -> Float

< fcn3 x = x+1

x :: t1
(+) :: a -> a -> a \Num a, a
literal 1 can be Int or Float => 1 :: IntLiteral a
=> Num t1, IntLiteral t1
=> fcn3 :: t -> t \Num t, IntLiteral t, t
```

Example 4

```
< f2 a = a.setX 5

a :: t1
a.setX => t1 < S2
setX :: Int -> Action, 5 :: Int => a.setX 5 :: Action
f2 :: t -> Action \t < S2, t

After simplification (t is in a contravariant position):
f2 :: S2 -> Action
```

Example 5

```
< f3 a = (a.getX, a.getY)

a :: t1
a.getX => t1 < S1
a.getY => t1 < S3
getX :: Request Int, getY :: Request String =>
(a.getX, a.getY) :: (Request Int, Request String)
f3 :: t -> (Request Int, Request String) \t < S1, t < S3, t

After simplification (using S3 < S1):
f3 :: t -> (Request Int, Request String) \t < S3, t

After simplification (since t is in a contravariant position):
f3 :: S3 -> (Request Int, Request String)
```

Example 6

```
< proc1 a1 a2 = do
  if (a1 /= a2) then
    a1.reset
    a2.reset

a1 :: t1
a2 :: t2
(/=) :: a -> a -> Bool \\Eq a => t1 = t2, Eq t1 and
"if (a1 /= a2) ..." is OKAY
a1.reset => t1 < S4
a2.reset => t1 < S4
a1.reset :: Action => OKAY
a2.reset :: Action => OKAY
```

Wednesday, April 28, 2010

9

Example 6 (cont'd)

- < Unifying the constraints "Eq t1" and "t1 < S4" we identify the typeclass instance eqs3 :: Eq S3 (which forces application of a coercion function c :: S4 -> S3 to arguments of (\=))
- < After simplification we obtain:
proc1 :: S4 -> S4 -> Cmd s () \\s
Here the result type is () since the compiler will automatically complete our code with "result ()" at the end
- < Note that if add a type signature
proc1 :: S1 -> S1 -> Cmd s () \\s
we get a compilation error: "Cannot solve typing constraint S1 < S4"

Wednesday, April 28, 2010

10

Example 7

```
< proc2 a = do
  v <- a.getY
  a.setX (length v)

a :: t1
a.getY => t1 < S3
getY :: Request String => v :: String (and "v <- a.getY" is OKAY)
length :: [a] -> Int \\a => length v :: Int
a.setX => t1 < S2
setX :: Int -> Action => a.setX (length v) :: Action => OKAY

< Unifying t < S3 and t < S2 gives t < S4 or simply t = S4. Hence:
proc2 :: S4 -> Cmd s () \\s
```

Wednesday, April 28, 2010

11

Example 8

```
< proc3 a b = do
  result if a==b then [a,b] else []

a :: t1
b :: t2
(==) :: a -> a -> Bool \\Eq a => t1 = t2, Eq t1 and
"if a==b ..." is OKAY
a :: t1 & b :: t1 => [a,b] :: [t1]

Checking result types of the two branches for consistency:
[a,b] :: [t1] and [] :: [t3] => result must be of type t4 > [t1] and t4 > [t3]

After simplification (t4 is in covariant position and [a] is invariant)
proc :: t -> t -> Cmd s [t] \\Eq t, t, s
```

Wednesday, April 28, 2010

12

Home assignment

- Describe the type derivation and determine the type of:

```
proc4 x y c = do
  o1 = new c x
  o2 = new c (y*2)
  rs = f3 o1
  o2.reset
  v <- f2 o1
```

Home assignment

- Answer:

```
proc4 :: a -> a -> (a -> Class S4) -> Cmd b () \\ Num a, IntLiteral a, a, b
```

```
proc4 x y c = do
  o1 = new c x
  o2 = new c (y*2)
  rs = f3 o1
  o2.reset
  v <- f2 o1
```