Advanced Functional Programming

Tim Sheard

- Polymorphism
- Hindley-Milner Polymorphism
- Rank 2 polymorphism

Polymorphism

- A function is polymorphic if it can work on any kind of argument.

```
f x = (x,x)

Main> :t f
f :: a -> (a,a)
```

- In essence it makes no reference to the value of its argument, it only manipulates it abstractly.

Local Polymorphism

Polymorphism can be scoped.

What type does f have?
forall b . b -> (a,b)

```
g x = let f = \ y -> (x,y)
          w1 = f "z"
          w2 = f True
      in (x,f)

Main> :t g
g :: a -> (a,b -> (a,b))
```

Let as function application

Let is often defined in terms of application

```
let x = e in y  == (\ x -> y) e
```

But there are difference in how let is typed.

```
g x = (\ f -> let w1 = f "z"
                  w2 = f True
              in (x,f))
      (\ y -> (x,y))
```

```
g x = let f = \ y -> (x,y)
          w1 = f "z"
          w2 = f True
      in (x,f)
```

```
ERROR " (line 12): Type error in application
*** Expression     : f True
*** Term           : True
*** Type           : Bool
*** Does not match : [Char]
```

Let polymorphism

Let-bound functions can be polymorphic, but lambda-bound arguments cannot.

This is the essence of Hindley-Milner polymorphism.

This means

- no function can be defined to take an argument which must be polymorphic
- No argument can ever be used in more than none polymorphic context.
- All types have the forall on the outermost

  forall a . (x -> (a -> b) -> (x,b))

- as opposed to

  x -> (forall a . a -> b) -> (x,b)

Example

```
h f x = let w1 = f "z"
            w2 = f True
        in (w1,w2)

ERROR (line 18): Type error in application
*** Expression     : f True
*** Term           : True
*** Type           : Bool
*** Does not match : [Char]
```

Rank 2 polymorphism

Rank 2 polymorphism relaxes some of this restriction.

```
h :: (forall a . a -> a) -> x -> (x,Bool)
h f x = let w1 = f x
            w2 = f True
        in (w1,w2)
```

forall's can be in the back-end of an arrow, but never the front end.

```
(forall ...) -> ((forall ...) -> z)
```

Type inference

Type inference of polymorphic arguments is undecidable.

If we want rank 2 polymorphism, we must use type annotations. Type-checking of rank 2 polymorphism is decidable

What kind of annotations must we give?

- The answer to this is hard to find.
- Giving the full signature of *every* function is enough.
- Is there any compromise using less information?

Full application

- In order to do type checking, rank 2 functions must be *fully applied*. That is all polymorphic arguments must be supplied.

```
ex2 = (4,h)
(line 28): Use of h requires at least 1 argument
```

- Arguments to rank 2 functions must really be polymorphic.

```
ex4 = h id 5                Main> :t ex4
                            ex4 :: (Integer,Bool)

ex3 = h ( \ x -> 1) 5
ERROR (line 33): Cannot justify constraints in
  application
*** Expression    : \x -> 1
*** Type          : b -> b
*** Given context : ()
*** Constraints   : Num b
```

Rank 2 Data Constructors

Data Constructors with polymorphic components give enough information to do type inference.

```
data Test x = C (forall a . a -> x -> (a,x)) x

ex5 = C (\ a x -> (a,x+1)) 3

ex6 = C (\ a x -> (a,not x)) True

f3 (C h n) w = h "z" w
```

What is the type of `ex5`, `ex6`, and `f3` ?

Church Numerals

Recognize the data type definition for natural numbers

```
data Nat = Z | S Nat
```

The catamorphism for Nat is the natural recursion pattern for Nat (sometimes called the fold)

```
cataNat zobj sfun Z = zobj
cataNat zobj sfun (S x) =
        sfun (cataNat zobj sfun x)
```

Many functions on Nat can be defined in terms of cataNat

```
plus x y = cataNat y S x
ex7 = plus (S Z) (S (S Z))
Main> ex7
S (S (S Z))
```

CataNat for multiplication

```
times x y = cataNat Z (plus x) y

one = S Z
two = S one
three = S two

ex8 = times two three

Main> ex8
S (S (S (S (S (S Z)))))
```

Nat as a rank 2 function

```
data N = N (forall z . z -> (z -> z) -> z)

cataN zobj sfun (N f) = f zobj sfun

n0 = N(\ z s -> z)
n1 = N(\ z s -> s z)
n2 = N(\ z s -> s(s z))
n3 = N(\ z s -> s(s(s z)))
n4 = N(\ z s -> s(s(s(s z))))

n2Int n = cataN 0 (+1) n
ex9 = n2Int n3


Main> ex9
3
```

Plus in data type N

```
--plus x y = cataNat y S x

succN :: N -> N
succN (N f) = N(\ z s -> s(f z s))

plusN :: N -> N -> N
plusN x y = cataN y succN x

ex10 = n2Int (plusN n2 n3)

Main> ex10
5
```

Church Numerals for List

```
data L1 a = L1 (forall b . b -> (a -> b -> b) -> b)

-- [1,2,3,4]
ex1 = L1 ( \ n  c -> c 1 (c 2 (c 3 (c 4 n))))

toList (L1 f) = f [] (:)

ex11 = toList ex1

Main> :t ex11
ex11 :: [Integer]
Main> ex11
[1,2,3,4]
```

Append in "church numeral" lists

```
cataList nobj cfun [] = nobj
cataList nobj cfun (x:xs) =
         cfun x (cataList nobj cfun)

cataL nobj cfun (L1 f) = f nobj cfun

cons x (L1 f) = L1(\ n c -> c x (f n c))

app x y = cataL y cons x

ex12 = app ex1 ex1
ex13 = toList ex12

Main> ex13
[1,2,3,4,1,2,3,4]
```

lists, fusion, and rank 2 polymorphism

- This form of rank 2 polymorphism has been exploited to justify fusion or deforestation.
- Consider

```
sum(map (+1) (upto 3))
sum(map (+1) [1,2,3])
sum[2,3,4]
9
```

- Produces, then consumes a bunch of intermediate lists, which never needed to be produced at all

Discovering fusion

How can we take an arbitrary expression about lists like:

```
sum(map (+1) (upto 3))
```

and discover an equivalent expression that does not build the intermediate lists?

Answer: write functions in terms of abstract recursion patterns, and rank-2 representations of lists.

```
cata : b -> (a -> b -> b) -> [a] -> b
build: (forall b . b -> (a -> b -> b) -> b) -> [a]
```

with the law: `cata n c (build f) == f n c`

```
build :: (forall b . b -> (a -> b -> b) -> b) -> [a]
build f = f [] (:)
cata nobj cfun [] = nobj
cata nobj cfun (x:xs) = cfun x (cata nobj cfun xs)
upto x =
 build(\ n c ->
         let h m = if m>x
                     then n
                     else c m (h (m+1))
         in h 1)

mapX f x =
  build(\ n c -> cata n (\ y ys -> c (f y) ys) x)
sumX xs = cata 0 (+) xs
```

```
sum(map (+1) (upto 3))   ==
sum(map (+1)
      (build(\ n c ->
               let h m = if m>3
                         then n
                         else c m (h (m+1))
               in h 1)   ==
sum(build(\ n c ->
            cata n (\ y ys -> c (f y) ys)
              (build(\ n c ->
                        let h m = if m>3
                                     then n
                                     else c m (h (m+1))
                        in h 1)))   ==
```

```
sum(build(\ n c ->
            let h m = if m>3
                         then n
                         else c (f m) (h (m+1))
            in h 1))  ==
cata 0 (+)
     (build(\ n c ->
              let h m = if m>3
                           then n
                           else c (f m) (h (m+1))
              in h 1)) ==
let h m = if m>3
             then 0
             else (f m) + (h (m+1))]
in h 1 ==  sum(map (+1) (upto 3))
```

We can encode this as such

```
data List a
  = Nil
  | Cons a (List a)
  | Build (forall b . b -> (a -> b -> b) -> b)

cataZ nobj cfun Nil = nobj
cataZ nobj cfun (Cons y ys) = cfun y (cataZ nobj cfun ys)
cataZ nobj cfun (Build f) = f nobj cfun

uptoZ x =
 Build(\ n c -> let h m = if m>x
                            then n
                            else c m (h (m+1))
                in h 1)
mapZ f x =
  Build(\ n c -> cataZ n (\ y ys -> c (f y) ys) x)
sumZ xs = cataZ 0 (+) xs
```

Results

```
ex14 = sumZ(mapZ (+1) (uptoZ 3))
ex15 = sum(map (+1) ([1..3]))


Main> ex14
9
(81 reductions, 177 cells)
Main> ex15
9
(111 reductions, 197 cells)
```

Type inference and Hindley-Milner

How is type inference done?
- Structural recursion over a term.
- Uses an environment which maps variables to their types
- Returns a computation in a monad
- type infer :: Exp -> Env -> M Type

- What does the Env look like
  - partial function from Name -> Scheme
  - Scheme is an encoding of a Hindley-Milner polymorphic type. All the forall's to the outermost position.
  - Often implemented as a list

How is Env used

```
g x = let f = \ y -> (x,y)
          w1 = f "z"
          w2 = f True
      in (x,f)
```

Every instance of a variable is given a new instance of its type.

Let Capital letters (A,B,C,A1,B1,C1, ...) indicate new fresh type variables.

In the box

suppose `f:: forall a . a -> (x,a)`

Instantiation

```
g x = let f = \ y -> (x,y)
          w1 = f "z"
          w2 = f True
      in (x,f)

the f in (f "z")
   A1 -> (x,A1)    A1 gets "bound" to String

the f in (f True)
   A2 -> (x,A2)    A2 gets "bound" to Bool

the f in (x,f)
    A3 -> (x,A3)   A3 remains "unbound"
```

Binding Introduction

```
g x = let f = \ y -> (x,y)
          w1 = f "z"
          w2 = f True
      in (x,f)
```

Every Bound program variable is assigned a new fresh type variable

- {g::E1}
- {g::E1, x::A1}
- {g::E1, x::A1, f::B1, y::C1 }
- {g::E1, x::A1, f::B1, w1::D1}
- {g::E1, x::A1, f::B1, w1::D1, w2::F1}

Type inference

```
g x = let f = \ y -> (x,y)
          w1 = f "z"
          w2 = f True
      in (x,f)
```

`{g::E1, x::A1, f::B1}`

As type inference proceeds type variables become "bound", thus the type of

`(\ y -> (x,y))`

becomes

`C1 -> (A1,C1)`

Since `f = (\ y -> (x,y))`

the type variable `B1` could be bound to `C1 -> (A1,C1)`

Generalization

But the rules of Hindley-Milner type inference say for every let-bound variable generalize it on all the type variables not in the current scope.

```
g x = let f = ((\ y -> (x,y)) :: C1 -> (A1,C1))
          w1 = f "z"
          w2 = f True
      in (x,f)
```

```
{g::E1, x::A1, f::B1}
```

Since `C1` does not appear in the types of the current scope, it is generalized and the type of `f` (`B1`) becomes polymorphic.

```
{g::E1, x::A1, f::forall c . c -> (A1,c)}
```

The monad of Type Inference

Methods required

```
unify:: Type -> Type -> M ()
lambdaExt :: Name -> Env -> M(Env,Type)
letExt:: Name -> Env -> M(Env,Scheme)
lookup:: Name -> Env -> Scheme
instantiate:: Scheme -> M Type
generalize:: Type -> Env -> M Scheme
freshTypeVar:: M Type
```

Rank 2 polymorphism

- The Type of runSt is a rank 2 polymorphic type
  - `runST :: ∀a . (∀s . ST s a) -> a`
- The forall is not all the way to the outside.
- There are other uses of rank 2 types.