Run-time Type Analysis in Haskell
with an awful lot of newtypes

Stephanie Weirich
University of Pennsylvania

WG 2.8, Crans-Montana

Type-indexed functions

- There are many neat tricks for encoding type-indexed functions in statically-typed languages.
- Examples:
  - printf
  - polymorphic equality
  - type-safe cast, dynamic types
  - run-time type representations
- These are all *first-order* type-indexed functions.

Higher-order-type-indexed functions

- Functions indexed by type constructors instead of types.
- Polytypic programming examples: size, map, zip, flatten, fold
  - size[List]
  - map[Tree]
  - zip[Maybe]

- Today: how to encode higher-order-type analysis in a statically typed language.

Caveat

- This talk is not very practical. The encoding I'm going to show you is *too annoying* to use in practice.
- But it's still cool. And it argues for more expressive type systems (first-class polymorphism, higher-order polymorphism) and more aggressive type inference.

Outline

- Example of a first-order type-indexed functions
- Digression on existential types
- Encoding in Haskell++
  - by encoding Lambda-R
- Example of a higher-order-type-analyzing function
- Encoding in Haskell++
  - by changing the encoding of Lambda-R

Redefining datatypes

```
type Unit = ()
type Plus a b = Either a b
type Times a b = (a, b)

type Maybe a = Plus Unit a
nothing = Left ()
just a = Right a

newtype List a = List (Plus Unit (Times a (List a)))
nil = List (Left ())
cons a b = List (Right (a, b))

newtype Tree a = Tree (Plus a (Times (Tree a)(Tree a)))
leaf a = Tree (Left a)
node t1 t2 = Tree (Right (t1,t2))
```

Representing types

```
type R a

rint   :: R Int
rchar  :: R Char
runit  :: R Unit
rtimes :: forall a b. R a -> R b -> R (Times a b)
rplus  :: forall a b. R a -> R b -> R (Plus a b)
rname  :: forall a b.(String,a->b,b->a)
             -> R a -> R b

rlistchar :: R (List Char)
rlistchar =
  rname (“List”, List, unList)
        (rplus runit (rtimes rchar rlistchar))
```

Representing type constructors

```
newtype List a
   = List (Plus Unit (Times a (List a))
unList (List x) = x

rlist :: R a -> R (List a)
rlist ra = rname ("List", List, unList)
  rplus runit (rtimes ra (rlist ra))

rlistchar :: R (List Char)
rlistchar = rlist rchar
```

Typecase

```
type Branches c = Branches {
   int   :: c Int,
   unit  :: c Unit,
   char  :: c Char,
   times :: forall a b. R a -> R b -> c (Times a b)
   plus  :: forall a b. R a -> R b -> c (Plus a b)
   name  :: forall a b. (String,a->b,b->a)
                     -> R a -> c b
}
typecase :: R a -> Branches c -> c a
```

Example

- Write a program to count the number of characters stored in any data structure.

```
count_chars (rmaybe rchar) nothing = 0
count_chars (rmaybe rchar) (just 'a') = 1

count_chars (rlist rchar)
   (cons 'a' (cons 'b' (cons 'c' nil))) = 3

count_chars (rtree rchar)
   (node (leaf 'a') (node (leaf 'b') (leaf 'c')))
   = 3
```

Type indexed functions

```
count_chars :: R a -> a -> Int
count_chars ra = typecase ra
 Branches {
   int   = const 0,
   unit  = const 0,
   char  = const 1,
   times = \ra -> \rb -> \(a,b) ->
             count_chars ra a + count_chars rb b,
   plus  = \ra -> \rb -> \x ->
             case x of
               (Inl a) -> count_chars ra a
               (Inr b) -> count_chars rb b,
   name  = \(in,out) -> ra -> \x ->
             count_chars ra (out x)
   }

count_chars (rlist rchar)
   (cons `a' (cons `b' (cons `c' nil))) = 3
```

can't unify return type

```
c Int vs. Int -> Int
c Unit vs. Unit -> Int
c Char vs. Char -> Int
```

Type indexed functions

```
newtype Count a = S (a -> Int)    unS(S a) = a
count_chars_aux :: R a -> Count a
count_chars_aux = \ra -> typecase ra
  Branches {
   int   = S $ const 0,
   unit  = S $ const 0,
   char  = S $ const 1,
   times = \ra -> \rb -> S $ \(a,b)->
            count_chars ra a + count_chars rb b,
   plus  = \ra -> \rb -> S $ \x ->
            case x of
             (Inl a) -> count_chars ra a
             (Inr b) -> count_chars rb b,
   name  = \(s,i,o) -> ra -> S $ \x ->
            count_chars ra (o x)
   }
count_chars :: R a -> a -> Int
count_chars = unS . count_chars_aux
```

Existential types

```
data Ex a = forall b. Ex (R b, a b)
```

New representation form

```
rex  :: forall a.
          (forall b. R b -> R (a b))
          -> R (Ex a)
```

New branch for count_chars_aux

```
ex = \ra -> S $ \(Ex x) ->
       let (rep, w) = x in
             count_chars (ra rep) w
```

Using Existential Types

```
type Hidden = Ex (Times Char)
rhidden :: R (Hidden)
rhidden = rex (rtimes rchar)

hidden1 = Ex (rint,  ('a', 3))
hidden2 = Ex (rchar, ('a','b'))

count_chars rhidden hidden1 = 1
count_chars rhidden hidden2 = 2
```

$\exists a.\ (\text{char} \times a)$

Scott encoding of representations

```
typecase :: R a -> Branches c -> c a

newtype R a = R (forall c. Branches c -> c a)
typecase (R rep) brs = rep brs

rint :: R Int
rint = R int
rchar :: R Char
rchar = R char
rtimes :: R a -> R b -> R (Times a b)
rtimes r1 r2 = R $ \brs -> times brs r1 r2
rname :: (String,a->b,b->a) -> R a -> R b
rname info rep = R $ \brs -> name brs info rep
rex :: (forall b. R b -> R (a b)) -> R (Ex a)
rex r = R $\brs -> ex brs r
```

Higher-order-type analysis

What if we want to be polymorphic over what we count?

- if we have something of type List a, can we count everything of type a?
- i.e. like list length

Type indexed functions : count

```
newtype Count a = S (a -> Int)    unS(S a) = a
count_aux :: R a -> Count a
count_aux = \ra -> typecase ra
(Branches {
   int   = S $ const 0,
   unit  = S $ const 0,
   char  = S $ const 0,
   times = \ra -> \rb -> S $ \(a,b)->
            count ra a + count rb b,
   plus  = \ra -> \rb -> S $ \x ->
            case x of
              (Inl a) -> count ra a
              (Inr b) -> count rb b,
   name  = \(s,i,o) -> ra -> S $ \x ->
            count_chars ra (o x)
   })
count :: R a -> a -> Int
count rep = unS (count_aux rep)
```

Idea: Add a place holder

Instead of :

```
length = count (rlist rchar)
```

we use:

```
length = count (rlist (rplace  ( S $ const 1)))

rplace :: a -> R b
```

such that `typecase (rplace e) brs = e`

How to implement the place holder?

Want:

```
rplace e = \brs -> e
rplace :: c a -> Branches c -> c a
```

```
R b = forall c.
Branches c -> c b
```

Parameterize R type

```
newtype R c a = R (Branches c -> c a)
typecase :: R c a -> Branches c -> c a
typecase (R rep) brs = rep brs

rint :: R c Int
rint = R int
rtimes :: R c a -> R c b -> R c (Times a b)
rtimes ra rb = R $ \brs -> times brs r1 r2

rplace :: c a -> R c a
rplace x = \brs -> x
```

Don't have to change original definitions, just their types

Example

```
rlist :: R c a -> R c (List a)
rlist ra = rname ("List",List,unList)
           (rplus runit (rtimes ra (rlist ra))

rep :: R Count (List a)
rep = rlist (rplace (S $ const 1))

length :: List a -> Int
length = count rep
length (cons 1 (cons 2 (cons 3 nil))) = 3

type Bar a = (Char, a)
rbar ra = rtimes rchar ra

count (rbar (rplace (S $ const 1))) ('a','b') = 1
```

Higher-kinded constructors

- rplace is a right inverse to typecase

```
typecase<*> brs = typecase brs
untypecase<*> brs = rplace
typecase<k1 -> k2> brs =
  typecase<k2> brs . untypecase<k1> brs
untypecase<k1 -> k2> brs =
  untypecase<k1> brs . typecase<k1> brs

typecase<* -> *>
= typecase<*> brs . rplace
= \x -> typecase brs (rplace x)
```

Existential types

```
data Ex f = forall a. Ex (forall c. (R c a, f a))

rex :: (forall b. R c b -> R c (a b)) -> R c (Ex a)
rex t1 = R $ \brs -> ex brs t1

type Hidden a = Ex (Times a)
rhidden :: R c a -> R c (Hidden a)
rhidden rep = rex (rtimes a)

hidden :: Hidden Char
hidden = Ex (rchar, (`a', `b'))

count (rhidden (rplace (S $ const 1))) hidden = 1
```

Existential types

```
new branch
 ex  :: forall a. (forall b. R c b -> R c (a b))
                  -> c (Ex a)

count
 ex = \ra -> S $ \(Ex w) ->
          let (rep,z) = w in
            count (ra (rplace (count_aux rep)) z

another version
ex = \ra -> S $ \(Ex w) ->
          let (rep,z) = w in
            count (ra (rplace (S $ const 0))) z
```

Existential types

```
newtype Show a = S (a -> String)
show :: R Show a -> Show a
show rep = typecase rep
  Branches {
    int  = S $ int2string,
    unit = S $ const "unit",
    …
    ex   = \ra -> S $ \(Ex w) ->
             let (rep,x) = w in
             let s = show rep in
             "Ex[" + unS (show (ra (rplace s))) +
  "]"
  }
show (rhidden rint) hidden_int = "Ex[ (3,4) ]"
```

Existential types

```
show rep = typecase rep
   Branches {
     …
     ex = \ra -> S $ \(Ex w) ->
        let (rep,x) = w in
        let s = S $ const "<hidden>" in
         "Ex[" + unS (show (ra (rplace s)) + "]"
   }

show (rhidden rint) hidden_int
   = "Ex[ (<hidden>,4) ]"
```

The Bad

- Need new representations to implement map: the types of the branches are different.
- To be fully expressive, each type must have several representations.
- The terms are exactly the same as before, except for their typing annotations.

Map

```
map :: R a1 a2 -> a1 -> a2
map rep = typecase2 rep $ Branches {
   int2   = id,
   unit2  = id,
   char2  = id,
   times2 = \ra rb (a,b) -> (map ra a, map rb b)
   plus2  = \ra rb x ->
              case x of
              (Left y) -> Left (map ra y)
              (Right y) -> Right (map rb y)
   name2 = \(in1,out1)(in2,out2) ra ->
              in1 . (map ra) . out2
}
listmap :: (a -> b) -> (List a) -> (List b)
listmap f = map (rlist (rplace f))
```

Types of the branches are different

```
type Branches2 c = Branches2 {
   int2  :: c Int Int,
   unit2 :: c Unit Unit,
   char2 :: c Char Char,
   times2:: forall a1 a2 b1 b2. R a1 a2 -> R b1 b2
              -> c (Times a1 b1) (Times a2 b2)
   plus2 :: forall a1 a2 b1 b2. R a1 a2 -> R b1 b2
              -> c (Plus a1 b1) (Plus a2 b2)
   name2 :: forall a1 a2 b1 b2. (a1->b1,b1->a1)
              -> (a2->b2,b2->a2) -> R a1 a2
              -> c b1 b2
}
typecase2 :: R2 c a1 a2 -> Branches2 c -> c a1 a2
type R2 c a1 a2 = R2 (Branches2 c -> c a1 a2)
```

Representations have same defs

```
typecase2 :: R2 c a1 a2 -> Branches2 c -> c a1 a2
type R2 c a1 a2 = R2 (Branches2 c -> c a1 a2)
rint2 :: R2 c Int Int
rint2 = R2 int2

runit2 :: R2 c Unit Unit
runit2 = R2 unit2

rtimes2 :: R2 c a1 a2 -> R2 c b1 b2
             -> R2 c (Times a1 a2) (Times b1 b2)
rtimes2 r1 t2 = R2 $ \brs -> times2 t1 t2

rplace2 = c a b -> R2 c a b
rplace2 x = R2 $ \brs -> x
```

Wishlist

- More higher-order type inference.

- Record subtyping or row-polymorphism
  - Prohibit invalid args to type-indexed functions.
  - often **`->`** is not defined
  - disallow **`count_chars(rarrow rint rint )`**

  - Allow users to pass in their own dictionaries.