The Type System of Haskell 98

Type and Module Systems, June 29, 00.

First-Order Types
(= no quantification over types)
The type constructor $\rightarrow$
(Haskell)

```
twice :: (Int->Int) -> (Int->Int)
twice f x = f (f x)
twice = \f -> \x -> f (f x)

addtwo :: Int -> Int
addtwo = twice succ
```

The type constructor $\rightarrow$ (fun)

```
twice : (Int->Int) -> (Int->Int)
twice = fun(f) fun(x) f( f (x))

addtwo : Int -> Int
addtwo = twice(succ)
```

Universal type quantification

```
twice2 :: (a->a)->(a->a)
twice2 f x = f (f x)

twice2 id   -- ok
twice2 succ -- ok
```

ML Polymorphism

(intersections)

the type of twice above is only most general
iff there is no subtyping involved. OHaskell.

Universal quantification (fun)

```
twice2 : forall a . ((a->a)->(a->a))
twice2 = fun(f) fun(x) f( f (x))

twice2 (id)   -- ok
twice2 (succ) -- ok

twice1 : (forall a . a->a) -> (forall a . a->a)
twice1 = fun(f) fun(x) f( f (x))

twice1 (id)   -- ok
twice1 (succ) -- ill-typed.
```

Parametric Polymorphism

Restricted Universal Quantification

```
gcd x y
    | deg y <= 0 = x
    | deg y > 0  = gcd y (modulo x y)
```

1. `gcd :: Integer -> Integer -> Integer`

   Cannot be used, for example, for polynomials.

2. `gcd :: a -> a -> a`

   Rejected. Would allow us to compute

   ```
   gcd  "two" "eight"
   ```

3. `gcd :: EuclideanDomain a => a -> a -> a`

   `EuclideanDomain` is a *type class*.

```haskell
class Group a where
  zero :: a
  add :: a -> a -> a
  neg :: a -> a
class Group a => Ring a where
  one :: a
  mul :: a->a->a
class Ring a => EuclideanDomain a where
  deg :: a->Integer
  divide :: a->a->a
  modulo :: a->a->a
gcd :: EuclideanDomain a => a -> a -> a
gcd x y
    | deg y <= 0 = x
    | deg y > 0  = gcd y (modulo x y)
double :: Group a => a -> a
double x = add x x
```

```
instance Group Integer where
  zero = 0
  add x y = x+y
  neg x = -x
instance Ring Integer where
  one = 1
  mul x y = x*y
instance EuclideanDomain Integer where
  deg x | x >= 0 = x
    | x <  0 = -x
  divide = div
  modulo = mod
test :: Integer
test = gcd  70 130
```

```
class Group a where
  zero :: a
  add :: a -> a -> a
  neg :: a -> a

instance Group Integer where
  zero = 0
  add x y = x+y
  neg x = -x

double :: Group a => a-> a
double x = add x x

doubletest = double (4::Integer)
```

```
zero (zeroD,addD,negD) = zeroD
add (zeroD,addD,negD)  = addD
neg (zeroD,addD,negD)  = negD

doubleD dict x = add dict x x

dictGroupInteger = (zeroI,addI,negI)
zeroI :: Integer
zeroI = 0
addI :: Integer->Integer->Integer
addI x y = x+y
negI :: Integer->Integer
negI x = -x

doubletest = doubleD dictGroupInteger (4::Integer)
```

“Algebraic Datatypes” (Haskell)

Algebraic datatypes are variants of records)

```
data Mammal = VarHuman { name  :: String,
                         age   :: Int->Int }
            | VarBear  { color :: Color,
                         age   :: Int->Int }

tom :: Mammal
tom = VarHuman {name = "Thomas",
                age = \y -> y - 1975}

mammaltoString :: Int -> Mammal -> String
mammaltoString = \now -> \m -> case m of
  VarHuman {name=n, age=a} ->
    n ++ ", "    ++ intToString (a now)
    ++ " years."
  VarBear  {color=c, age=a} ->
    "A bear of " ++ intToString (a now)
    ++ " years."
```

“Algebraic Datatypes”
(fun)

```
type Mammal =
 [ VarHuman : { name  : String,
                age   : Integer->Integer }
   VarBear  : { color : Color,
                age   : Integer->Integer } ]

tom : Mammal
value tom =
  [ VarHuman = {name = "Thomas"
                age  = lambda y. y - 1975}]

mammalToString : Int -> Mammmal -> String
value mammalToString =
  lambda now .lambda m . case m of
  [VarHuman = rec] rec.name ++ ", "
                ++ intToString (rec.age now)
                ++ " years."
  [VarBear  = rec] "A bear of "
                ++ intToString (rec.age now)
                ++ " years."
```

Examples of Algebraic Datatypes from the Prelude

```
data Bool     = False | True

(&&), (||)  :: Bool -> Bool -> Bool
False && x   = False
True  && x   = x
False || x   = x
True  || x   = True

not         :: Bool -> Bool
not True     = False
not False    = True
```

Examples of Algebraic Datatypes from the Prelude

To *add a single value* (like "error") to a type `a`, one uses the type `Maybe a`.

```
data Maybe a = Nothing | Just a

maybe :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  = n
maybe n f (Just x) = f x
```

To form the *disjoint union* of two types `a` and `b`, one uses the type `Either a b`.

```
data Either a b = Left a | Right b

either :: (a -> c) -> (b -> c)
          -> Either a b -> c
either l r (Left x)  = l x
either l r (Right y) = r y
```

Type Equivalence

```
data BW1 = Black{}
         | White{}
data BW2 = Black{}
         | White{}
```

is rejected by

```
ERROR: Repeated definition for
       constructor function "Black"
```

The question "BW1 = BW2"

```
c1 :: BW1
c2 :: BW2
c2 = c1
```

does not arise.

“Recursive Algebraic” Datatypes
(Trees)

```
data TreeBool =
    Leaf
  | Fork { content :: Bool,
           left, right :: TreeBool }

tb1 :: TreeBool
tb1 = Fork {
  content = True,
  left    = Leaf,
  right   = Fork {
    content = False,
    left    = Leaf,
    right   = Leaf }}
```

“Recursive Algebraic” Datatypes (Trees) (continued)

```
equalTreeBool :: TreeBool -> TreeBool -> Bool
equalTreeBool
  Leaf
  Leaf = True
equalTreeBool
  Fork { content=c1, left=l1, right=r1 }
  Fork { content=c2, left=l2, right=r2 } =  equalBool c1 c2
                   && equalTreeBool l1 l2
                   && equalTreeBool r1 r2
```

Modules (1)

Haskell Modules control namespaces.

Haskell modules are *not* first class values (unlike fun packages whicht *are* first class values. fun package can be passed as arguments to functions, returned by functions etc.)

Abstract Data Types (1)

Legal use:

```
import PointModule1
       (Point, makepoint, getx, gety)
p :: Point
p = makepoint 3.0 4.0
dist :: Point->Float
dist p = sqrt (sqr (getx p) + sqr (gety p))
main :: Float
main = dist p
```

Illegal, rejected by the compiler:

```
import PointModule1
       (Point, makepoint, getx, gety)
p :: Point
p = makepoint 3.0 4.0
fst :: (a,b)->a
fst (x,y) = x
main :: Float
main = fst p
```

Abstract Data Types (2)

```
module PointModule1        -- name
       (Point, makepoint,
        getx, gety)        -- exports
       where
data Point = Pt {x::Float, y:: Float}
makepoint :: Float->Float->Point
makepoint x1 y1 = Pt {x=x1, y=y1}
getx, gety :: Point->Float
getx p = x p
gety p = y p
```

Hiding the implementation of the type Point is achieved by excluding `Pt`, `x` and `y` from the exports.

Abstract Data Types (3)

```
module PointModule2
       (Point, makepoint, getx, gety)
       where
type Point = (Float, Float)
makepoint :: Float->Float->Point
makepoint x1 y1 = (x1, y1)
getx, gety :: Point->Float
getx (x1, y1) = x1
gety (x1, y1) = y1
```

Now the implementation of Point is *not hidden any more.* In particular,

```
import PointModule2 (makepoint, getx, gety)
p :: Point
p = makepoint 3.0 4.0
fst :: (a,b)->a
fst (x,y) = x
main :: Float
main = fst p
```

is legal.

Abstract Data Types (4)

How to separate the interface from the implementation.

```
module PointInterface ( PointClass
  (makepoint, getx, gety ) ) where
class PointClass p where
   makepoint :: Float -> Float -> p
   getx :: p -> Float
   gety :: p -> Float

module PointImplementation
  (PointType) where
import PointInterface
data PointType =
  PointValueConstructor {x, y:: Float} deriving Show
instance PointClass PointType where
  makepoint x1 y1 =
    PointValueConstructor {x=x1, y=y1}
  getx p = x p
  gety p = y p
```

```
import PointInterface
  (PointClass(makepoint, getx, gety))
import PointImplementation (PointType)
p :: PointType
p = makepoint 3.0 4.0
dist p = sqrt (  sqr (getx p)
                +sqr (gety p) )
test = dist p
```

Sets as an ADT (1)

```
import SetInterface
      (SetClass(emptySet,insertSet,memberSet))
import SetImplementation0 (SetType0)
import SetImplementation1 (SetType1)

s0 :: SetType0 Char
s0 = insertSet 'b' (insertSet 'a' emptySet)
test0 = memberSet 'a' s0

s1 :: SetType1 Char
s1 = insertSet 'b' (insertSet 'a' emptySet)
test1 = memberSet 'a' s1
```

Sets as an ADT (2)

```
module SetInterface
      (SetClass(emptySet,insertSet,memberSet))
      where
  class Eq et => SetClass stc et where
    emptySet  :: stc et
    insertSet :: et -> stc et -> stc et
    memberSet :: et -> stc et -> Bool
  -- et  (element type)
  --   is the type of the elements in the set.
  -- stc (set type constructor)
  --   is a unary type constructor.
  -- stc et
  --   is the type of sets with elements in t.
```

Sets as an ADT (3)

```
module SetImplementation1 (SetType1) where
-- implementation of sets as unbalanced trees.

  import SetInterface(SetClass(emptySet,
                      memberSet, insertSet))

  instance Ord a => SetClass SetType1 a where
    emptySet = emptySet1
    memberSet = memberSet1
    insertSet = insertSet1

  type SetType1 = Tree
  data Tree a = E
              | T { content :: a,
            left,right :: (Tree a) }

  emptySet1 :: SetType1 a
  emptySet1 = E

  memberSet1 x E = False
  memberSet1 x (T {left=a, content=y, right=b}) =
```

```
    if x < y then memberSet1 x a
    else if x > y then memberSet1 x b
    else True

  insertSet1 x E = T {left=E, content=x, right=E}
  insertSet1 x s@(T {left=a, content=y, right=b}) =
    if x < y then T {left=(insertSet1 x a), content=y, right=b}
    else if x > y then T {left=a, content=y, right=(insertSet1 x b)}
    else s

module SetImplementation0 (SetType0) where
-- implementation of sets as functions.

  import SetInterface(SetClass(..))

  data SetType0 a = MkSetFunction {f :: a->Bool}

  instance Ord a => SetClass SetType0 a where
    emptySet = MkSetFunction {f = \x->False}

    memberSet x (MkSetFunction {f = f1}) = f1 x

    insertSet x (MkSetFunction {f = f1}) =
    MkSetFunction {f = \y -> (y==x) || f1 y}
```