Functional Programming

Type System

H. Turgut Uyar

2013-2016

License

© 2013-2016 H. Turgut Uyar

You are free to:

- Share – copy and redistribute the material in any medium or format
- Adapt – remix, transform, and build upon the material

Under the following terms:

- Attribution – You must give appropriate credit, provide a link to the license, and indicate if changes were made.
- NonCommercial – You may not use the material for commercial purposes.
- ShareAlike – If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

For more information:
https://creativecommons.org/licenses/by-nc-sa/4.0/

Read the full license:

https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode

Topics

Type Checks

- Haskell has strong and static typing
- strong: operands/parameters of incorrect types not allowed
- static: types checked at compile-time

Type Inference

- type annotations may be omitted
- types inferred by the language processor
- assign every binding a type such that type checking succeeds
- fail if no such assignment can be found

Type Inference Examples

```
nand x y = not (x && y)
-- nand :: Bool -> Bool -> Bool

limitedLast s n =
    if (length s > n) then s !! (n - 1) else (last s)
-- limitedLast :: [a] -> Int -> a

capitalize s = toUpper (s !! 0) : drop 1 s
-- capitalize :: [Char] -> [Char]

foo1 x y = if x then y else x + 1
-- type inference fails

foo2 f g x = (f x, g x)
-- (a -> b) -> (a -> c) -> a -> (b, c)
```


Type Inference Examples

```
f (x, y) = (x, ['a' .. y])

g (m, zs) = m + length zs

h = g . f
```

- exercise: what are the types of `f`, `g`, and `h`?

Type Classes

check whether an item is an element of a list

```
elemChar :: Char -> [Char] -> Bool
elemChar _ []     = False
elemChar x (c:cs) =
    if x == c then True else elemChar x cs
```

- different function for every type?
- better if we can write it as:
  `a -> [a] -> Bool`
- provided that type a supports equality check

Type Classes

- type class: collection of types
  over which some functions are defined
- every type belonging to the class must implement its functions
- member of type class: instance

equality class

```
class Eq a where
  (==) :: a -> a -> Bool
```

- `Bool`, `Char`, `Integer`, `Float` are instances of `Eq`
- tuples and lists are also instances of `Eq`

Context

- type signature can contain context ("provided that")
- a type being an instance of a class

```
elem :: Eq a => a -> [a] -> Bool
elem _ []     = False
elem x (c:cs) = if x == c then True else elem x cs
```


Instance Example

```
data Move = Rock | Paper | Scissors

instance Eq Move where
  (==) Rock     Rock     = True   -- Rock == Rock = True
  (==) Paper    Paper    = True
  (==) Scissors Scissors = True
  (==) _        _        = False

-- elem Paper    [Rock, Paper, Rock] ~> True
-- elem Scissors [Rock, Paper, Rock] ~> False
```


Derived Instances

- default equality check: derive from `Eq`

```
data Move = Rock | Paper | Scissors
            deriving Eq
```

- default string conversion: derive from `Show`

```
class Show a where
  show :: a -> String

data Move = Rock | Paper | Scissors
            deriving (Eq, Show)
```

Instance Example

rational numbers

```
data Rational = Fraction Integer Integer

instance Eq Rational where
  Fraction n1 d1 == Fraction n2 d2 = n1 * d2 == n2 * d1

instance Show Rational where
  show (Fraction n d) = show n ++ "/" ++ show d
```


Default Definitions

- classes can contain default definitions for functions
- defaults can be overridden by instances

example: equality class

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  x /= y = not (x == y)
  x == y = not (x /= y)
```

- defining one of == and /= is enough

Derived Classes

- type classes can depend on other type classes

example: order class

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool

  x <= y = (x < y || x == y)
  x >  y = y < x
```


Type Class Example

```
instance Ord Rational where
  Fraction n1 d1 < Fraction n2 d2 = n1 * d2 < n2 * d1

qSort :: Ord a => [a] -> [a]
qSort []     = []
qSort (x:xs) = qSort smaller ++ [x] ++ qSort larger
  where
    smaller = [a | a <- xs, a <= x]
    larger  = [b | b <- xs, b >  x]
```

Order Class

```
data Ordering = LT | EQ | GT

class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min             :: a -> a -> a
  compare              :: a -> a -> Ordering

  compare x y
    | x == y    = EQ
    | x <= y    = LT
    | otherwise = GT
```

- define one of `compare` and <=

Order Default Definitions

```
x <= y = compare x y /= GT
x <  y = compare x y == LT
x >= y = compare x y /= LT
x >  y = compare x y == GT

max x y
  | x <= y    = y
  | otherwise = x

min x y
  | x < y     = x
  | otherwise = y
```


Numeric Class

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate        :: a -> a
  abs, signum   :: a -> a
  fromInteger   :: Integer -> a

  x - y = x + negate y
```

- exercise: make `Rational` an instance of `Num`

Recursive Algebraic Types

- types can be described in terms of themselves

example: expressions

```
data Expr = Lit Integer |
            Add Expr Expr |
            Mul Expr Expr
```

Recursive Algebraic Types

example: evaluating an expression

```
eval :: Expr -> Integer
eval e = case e of
    Lit n     -> n
    Add e1 e2 -> (eval e1) + (eval e2)
    Mul e1 e2 -> (eval e1) * (eval e2)
```


Recursive Algebraic Types

example: converting an expression to a string

```
instance Show Expr where
  show e = case e of
      Lit n     -> show n
      Add e1 e2 -> show e1 ++ "+" ++ show e2
      Mul e1 e2 -> show e1 ++ "*" ++ show e2
```

- precedences incorrect, try with:
  `Mul (Add (Lit 2) (Lit 3)) (Lit 5)`
- exercise: write a correct implementation
  with minimal use of parentheses in string

Polymorphic Algebraic Types

example: integer binary tree

```
data BinTree = Nil | Node Integer BinTree BinTree

depth :: BinTree -> Int
depth Nil            = 0
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)

sumTree :: BinTree -> Integer
sumTree Nil            = 0
sumTree (Node n t1 t2) = n + sumTree t1 + sumTree t2
```

- different tree for every type?

Polymorphic Algebraic Types

example: polymorphic binary tree

```
data BinTree a = Nil | Node a (BinTree a) (BinTree a)

depth :: BinTree a -> Int
depth Nil            = 0
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)

sumTree :: Num a => BinTree a -> a
sumTree Nil            = 0
sumTree (Node n t1 t2) = n + sumTree t1 + sumTree t2
```

Maybe Type

- returning an error value

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord, Read, Show)
```

example

```
errDiv :: Integer -> Integer -> Maybe Integer
errDiv m n
  | n == 0    = Nothing
  | otherwise = Just (m `div` n)
```


Maybe Example

maximum of a list

```
maxList :: Ord a => [a] -> Maybe a
maxList [] = Nothing
maxList xs = Just (foldl1 max xs)
```


Maybe Type Handling

- transmitting an error through a function

```
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe g Nothing  = Nothing
mapMaybe g (Just x) = Just (g x)

-- mapMaybe length Nothing          ~> Nothing
-- mapMaybe length (Just "haskell") ~> Just 7
```


Maybe Type Handling

- trapping an error

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

-- maybe 0 length Nothing          ~> 0
-- maybe 0 length (Just "haskell") ~> 7
```

Either Type

- one of two types

```
data Either a b = Left a | Right b
                  deriving (Eq, Ord, Read, Show)
```

example

```
isFalse :: Either Integer [a] -> Bool
isFalse (Left  0)  = True
isFalse (Right []) = True
isFalse _          = False
```


Either Type Handling

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

-- either length abs (Left  "haskell") ~> 7
-- either length abs (Right (-8))      ~> 8
```


Either Type Handling

```
join f g (Left  x) = Left  (f x)
join f g (Right y) = Right (g y)
```

- exercise: what is the type of `join`?
- how can it be invoked?
- express `join` in the form:
  `join f g = either ___ ___`

References

Required Reading: Thompson

- Chapter 13: Overloading, type classes and type checking
- Chapter 14: Algebraic types