Advanced Functional Programming, 1DL450

Lecture 8, 2012-11-26

Cons T Åhs

Types revisited

- Haskell is *statically typed*
  - Type information is derived and checked at *compile time*
  - Programs have to be type correct at compile time
  - Better information up front, less checking at run time
- Erlang and Lisp are *dynamically typed*
  - Type information is checked at *run time*
  - Very little is done at compile time, even for obvious type errors
- Static or dynamic typing affects programming style very much
  - static forces discipline by refusing to compile incorrectly typed programs - this is good
  - dynamic requires discipline if you do not want to end up with very large union types - this is bad
  - “productivity” might be quite different
    - Why/when should you choose one or the other?
    - Why do we have both, really?

Overloading

- For a strictly typed language each function (operator) will have a well defined type
  - good for type inference and understanding
  - impractical for “standard” functions, such as equality
- Many languages introduce overloading to make it more practical
  - drawback is loss of precision during static analysis
  - auto conversion between types may take behind the scenes
- Examples
  - numbers
    - Int, Integer, Float
  - arithmetic
    - add, subtract, ..
  - equality
    - what does it mean for two objects to be equal?
  - ordering
  - printing

To type or not to type

```
fac :: Integer -> Integer
fac 0 = 1
fac n = n * fac (n - 1)

*Main> :type fac
fac :: Integer -> Integer

*Main> :type length
length :: [a] -> Int
*Main> fac (length [1,2,3])

<interactive>:281:6:
    Couldn't match expected type `Integer' with actual type `Int'
    In the return type of a call of `length'
    In the first argument of `fac', namely `(length [1, 2, 3])'
    In the expression: fac (length [1, 2, 3])
```

- If we give a type Haskell will use that type and only complain if the definition is not of the same type
- We can ask for the type of an expression using `:type` (interactively)
- Types mismatch and Haskell complains

To type or not to type

```
-- Don't specify type
fac 0 = 1
fac n = n * fac (n - 1)

*Main> :type fac
fac :: (Eq a, Num a) => a -> a

*Main> fac (length [1,2,3])
6
```

What's this?

Ok!

- Not giving a type makes Haskell more happy
- The type, however, might be surprising
- Enter type classes

Type Classes

- Type classes can be seen as similar to interfaces in Java
  - declare name, type dependence and functions to be implemented
  - other types can then be made to be instances of the type class
  - types are now conditionalised on belonging to type classes

```
-- Don't specify type
fac 0 = 1
fac n = n * fac (n - 1)

*Main> :type (-)
(-) :: Num a => a -> a -> a

*Main> :type fac
fac :: (Eq a, Num a) => a -> a
```

Note: this version of fac allows floats, which is questionable..

equality defined for a

a is a number

conditional/implication

Type Class `Eq`

- Determining equality means defining when two instances are equal
- Equality, i.e., being an instance of `Eq`, can be defined automatically if want it.
  - You will then get the simplest possible equality, i.e., isomorphic structures and members.

```
data Set a = EmptySet | SetAdd a (Set a) | Union (Set a) (Set a)

*Main> EmptySet == EmptySet

<interactive>:332:10:
    No instance for (Eq (Set a0))
      arising from a use of `=='
    Possible fix: add an instance declaration for (Eq (Set a0))
    In the expression: EmptySet == EmptySet
    In an equation for `it': it = EmptySet == EmptySet

data Set a = EmptySet | SetAdd a (Set a) | Union (Set a) (Set a)
             deriving Eq

*Main> EmptySet == EmptySet
True
```

Type Class `Eq`

- Determining equality means defining when two instances are equal
  - define equality for the type
  - make it be part of type class `Eq` by extending/overloading == to handle the new type
  - Read as "..if a is an equality type then two sets of type a are equal when.."
  - Reading an recursive instance of `Eq` is a good exercise in operator precedence..

```
data Set a = EmptySet | SetAdd a (Set a) | Union (Set a) (Set a)

instance Eq a => Eq (Set a) where
         EmptySet == EmptySet = True
         (SetAdd x EmptySet) == (SetAdd y EmptySet) = x ==y
         (SetAdd x EmptySet) == (Union (SetAdd y EmptySet) EmptySet)
              = x == y
-- more clauses needed..
```

Defining Type Classes

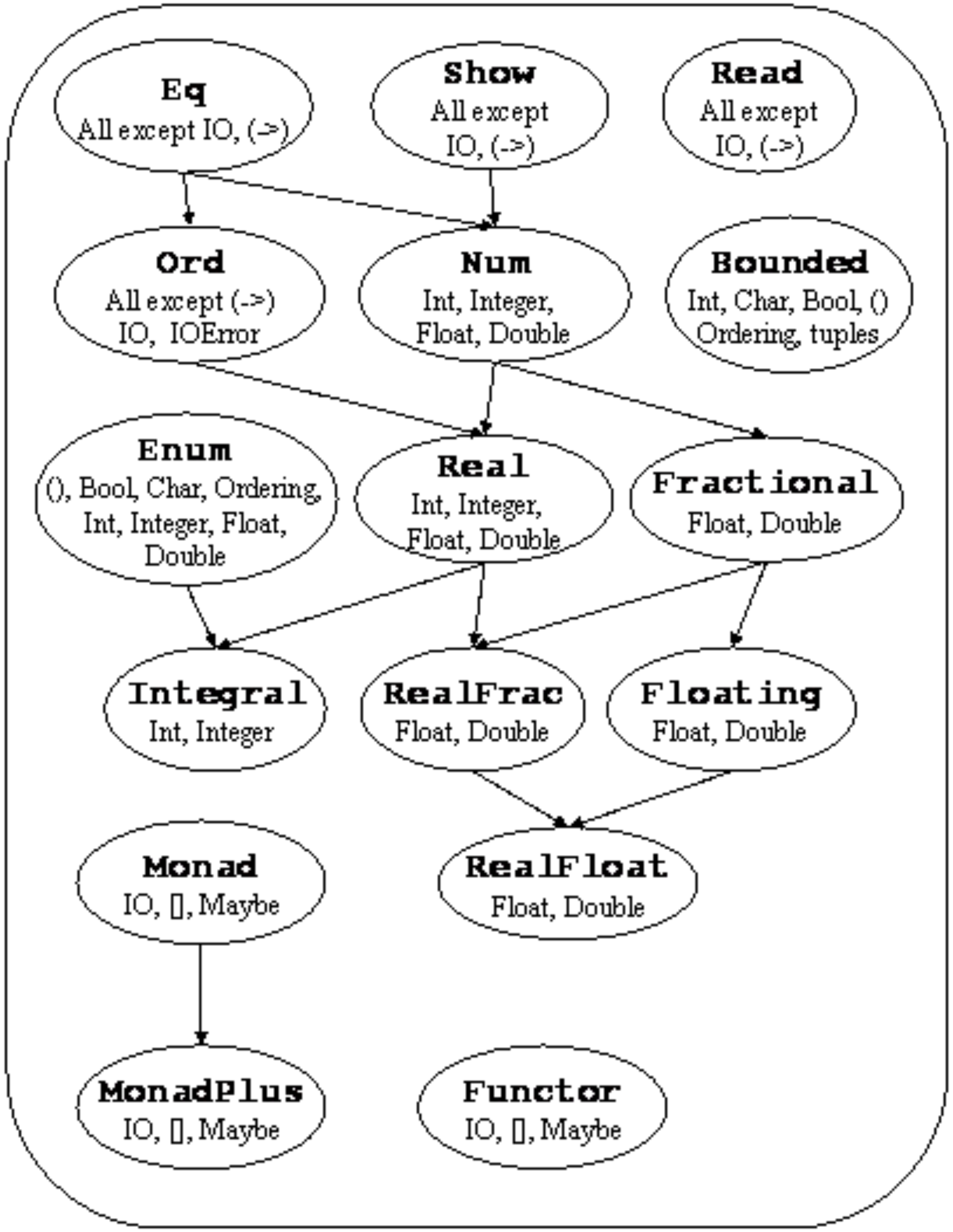
```
class Complexity a where
  complexity :: a -> Integer

instance Complexity Integer where
  complexity x = x

instance Complexity Int where
  complexity x = toInteger x

instance Complexity a =>  Complexity [a] where
  complexity l = (toInteger (length l))
               + (foldl (\w e -> complexity e + w) 0 l)
```

- Type class contains one required function
- Define instances
  - Instances can be recursive

Predefined Type Classes

- `Eq` - equality
- `Show` - printing instances
  - solves problem of printing value of computation
- `Read` - reading instances
- `Enum` - enumerate (only possible for certain types)
- `Ord` - extension of Eq for total ordering
- .. and some more ..
- Together they define a type class hierarchy

Type class hierarchy



Automagic Deriving

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday
           | Saturday | Sunday
           deriving (Eq, Enum, Show, Read, Ord, Bounded)

nextday :: Day -> Day
nextday day = toEnum (fromEnum day + 1)

weekday day = elem day [Monday .. Friday]

*Main> Monday < Wednesday
True
*Main> (toEnum 2) :: Day
Wednesday
*Main> nextday Tuesday
Wednesday
*Main> nextday Friday == Sunday
False
*Main> (read "Saturday") :: Day
Saturday
*Main> nextday (read "Saturday") :: Day
Sunday
*Main> minBound :: Day
Monday
```

Extending Show

```
data Sexpr a = Leaf a | Cons (Sexpr a) (Sexpr a)

showSexpr :: (Show a) => Sexpr a -> String
showSexpr (Leaf x) = show x
showSexpr (Cons car cdr) =
  "(" ++ showSexpr car ++ " . " ++ showSexpr cdr ++ ")"

instance Show a => Show (Sexpr a) where
  show s = showSexpr s

*Main> (Cons (Cons (Leaf 3)(Leaf 2)) (Leaf 4))
((3 . 2) . 4)
```

- Extending `Read` allows you to define you input syntax as well

The Show Class

```
type  ShowS   = String -> String

class  Show a  where
    showsPrec :: Int -> a -> ShowS
    show      :: a -> String
    showList  :: [a] -> ShowS

    showsPrec _ x s   = show x ++ s
    show x            = showsPrec 0 x ""
    -- ... default decl for showList given in Prelude
```

- `showsPrec` is for converting to strings using precedence
- `ShowS` is used to produce accumulating implementations of show, making it more efficicient

Revisiting `show` for Sexprs

```
data Sexpr a = Leaf a | Cons (Sexpr a) (Sexpr a)

showsSexpr :: (Show a) => Sexpr a -> ShowS
showsSexpr (Leaf x) = shows x
showsSexpr (Cons car cdr) =
  ('(':) . showsSexpr car . (" . "++) . showsSexpr cdr . (')':)

instance Show a => Show (Sexpr a) where
  show s = showsSexpr s ""

*Main> (Cons (Cons (Leaf 3)(Leaf 2)) (Leaf 4))
((3 . 2) . 4)
```

- Return a function with an accumulator instead
- Linear complexity instead of quadratic
- Note compact representation with use of function composition

Class Enum

```
class Enum a  where
    succ, pred     :: a -> a
    toEnum         :: Int -> a
    fromEnum       :: a -> Int
    enumFrom       :: a -> [a]                -- [n..]
    enumFromThen   :: a -> a -> [a]           -- [n,n'..]
    enumFromTo     :: a -> a -> [a]           -- [n..m]
    enumFromThenTo :: a -> a -> a -> [a]      -- [n,n'..m]
```

- Introduce convenient functions and notations for enumerations

Class `Eq`

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

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

- Only one needs to be defined
- Standard definitions exist for both exists
- One can be defined in terms of the other

Ordering

```
data Ordering = LT | EQ | GT
                deriving (Eq, Ord, Bounded, Enum, Read, Show)

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

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

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
```

Class Num

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

class  (Num a, Ord a) => Real a  where
    toRational ::  a -> Rational

class  (Real a, Enum a) => Integral a  where
    quot, rem, div, mod :: a -> a -> a
    quotRem, divMod     :: a -> a -> (a,a)
    toInteger           :: a -> Integer
```

Class Num

```
class  (Num a) => Fractional a  where
    (/)           :: a -> a -> a
    recip         :: a -> a
    fromRational :: Rational -> a

class  (Fractional a) => Floating a  where
    pi                   :: a
    exp, log, sqrt       :: a -> a
    (**), logBase        :: a -> a -> a
    sin, cos, tan        :: a -> a
    asin, acos, atan     :: a -> a
    sinh, cosh, tanh     :: a -> a
    asinh, acosh, atanh :: a -> a
```