Advanced Programming
Handout 4

A Taste of Infinity

Infinite Lists

- Lists in Haskell need not be finite. E.g.:

```
list1 = [1..]           -- [1,2,3,4,5,6,...]

f x = x : (f (x+1))
list2 = f 1             -- [1,2,3,4,5,6,...]

list3 = 1:2:list3       -- [1,2,1,2,1,2,...]
```

Working with Infinite Lists

- Of course, if we try to perform an operation that requires consuming *all* of an infinite list (such as printing it or finding its length), our program will never yield a result.
- However, a program that only consumes a *finite part* of an infinite list will work just fine.

```
take 5 [10..]  ➔ [10,11,12,13,14]
```

Lazy Evaluation

- The feature of Haskell that makes this possible is *lazy evaluation*.
- Only the portion of a list that is actually needed by other parts of the program will actually be constructed at run time.
- We will discuss the mechanics of lazy evaluation in much more detail later in the course.

More About Higher-Order Functions

(SOE Chapter 9)

Multi-Arg Functions in Haskell

- What is the difference between

  `f x y = x*y+5`

  and

  `f (x,y) = x*y+5`

  ?

Multi-Arg Functions in Haskell

```
f :: Integer -> Integer -> Integer
f x y = x*y+5

f :: (Integer,Integer) -> Integer
f (x,y) = x*y+5
```

Multi-Arg Functions in Haskell

When we write

```
f :: Integer -> Integer -> Integer
```

what we really mean is:

```
f :: Integer -> (Integer -> Integer)
```

Multi-Arg Functions in Haskell

- The observation that an n-argument function can equivalently be considered as a 1-argument function that returns an (n-1)-argument function is called *Currying* (after the great early-20th-century logician Haskell B. Curry!)

Use of Currying

```
listSum, listProd :: [Integer] -> Integer
listSum  xs          = foldr (+) 0 xs
listProd xs          = foldr (*) 1 xs
```

⇓

```
listSum              = foldr (+) 0
listProd             = foldr (*) 1
```

```
and, or :: [Bool] -> Bool
and xs   = foldr (&&) True  xs
or  xs   = foldr (||) False xs
```

⇓

```
and      = foldr (&&) True
or       = foldr (||) False
```

Be Careful Though ...

Consider:

```
f x = g (x+2) y x
```

This is not the same as:

```
f = g (x+2) y
```

because the remaining occurrence of `x` becomes unbound. (Or, in fact, it might be bound by some outer definition!)

In general:

```
f x = e x
```

is the same as

```
f = e
```

only if `x` *does not appear free in* `e`.

Simplifying Definitions

Recall:

```
reverse xs = foldl revOp [] xs
  where revOp acc x = x : acc
```

In the prelude we have: `flip f x y = f y x`. (what is its type?) Thus:

```
revOp acc x = flip (:) acc x
```

or even better:

```
revOp = flip (:)
```

And thus:

```
reverse xs = foldl (flip (:)) [] xs
```

or even better:

```
reverse = foldl (flip (:)) []
```

Anonymous Functions

- So far, all of our functions have been defined using an *equation*, such as the function `succ` defined by:
  ```
  succ x = x+1
  ```
- This raises the question: Is it possible to define a *value* that behaves just like `succ`, but has no name? Much in the same way that `3.14159` is a value that behaves like `pi`?
- The answer is *yes*, and it is written `\x -> x+1`. Indeed, we could rewrite the previous definition of `succ` as:
  ```
  succ = \x -> x+1.
  ```

Sections

- Sections are like currying for infix operators. For example:

  ```
  (+5) = \x -> x + 5
  (4-) = \y -> 4 - y
  ```

  So in fact `succ` is just `(+1)` !

- Note the section notation is consistent with the fact that `(+)`, for example, is equivalent to `\x -> \y -> x+y`.

- Although convenient in many situations, sections are less expressive than anonymous functions. For example, it's hard to represent
  `\x -> (x+1)/2` as a section.

- You can also pattern match using an anonymous function, as in `\(x:xs) -> x`, which is the `head` function.

Function Composition

- Very often we would like to combine the effect of one function with that of another. *Function composition* accomplishes this for us, and is easily defined as the infix operator `(.)`:

```
(f . g) x = f (g x)
        -- i.e.: (.) f g x = f (g x)
```

- So `f.g` means the same thing as `\x -> f (g x)`.

- Function composition can be used to simplify some of the previous definitions:

```
totalSquareArea sides
  = sumList (map squareArea sides)
  = (sumList . map squareArea) sides
```

  Combining this with currying simplification yields:

```
totalSquareArea = sumList . map squareArea
```

Qualified Types

(SOE Chapter 12)

Motivation

- What should the principal type of (+) be?
  - `Int -> Int -> Int` -- too specific
  - `a -> a -> a` -- too general
- It seems like we need something "in between", that restricts "a" to be from the set of all number types, say Num = {Int, Integer, Float, Double, etc.}.
- The type a -> a -> a
  is really shorthand for (∀ a) a -> a -> a
- *Qualified types* generalize this by qualifying the type variable, as in (∀ a ∈ Num) a -> a -> a,
  which in Haskell we write as Num a => a -> a -> a

Type Classes

- “Num” in the previous example is called a *type class*, and should not be confused with a type constructor or a value constructor.
- “Num T” should be read “T is a member of (or an instance of) the type class Num”.
- Haskell’s type classes are one of its most innovative features.
- This capability is also called “overloading”, because one function name is used for potentially very different purposes.
- There are many pre-defined type classes, but you can also define your own.

Example: Equality

- Like addition, equality is not defined on all types (how would we test the equality of two functions, for example?).
- So the equality operator (==) in Haskell has type Eq a => a -> a -> Bool. For example:

```
42 == 42            ➔  True
'a' == 'a'          ➔  True
'a' == 42           ➔  << type error! >>
                           (types don't match)
(+1) == (\x->x+1)   ➔  << type error! >>
                           ((->) is not an instance of Eq)
```

- Note: the type errors occur *at compile time!*

Equality, cont'd

- Eq is defined by this *type class declaration:*

```
class Eq a  where
        (==), (/=)        :: a -> a -> Bool
        x /= y            =  not (x == y)
        x == y            =  not (x /= y)
```

- The last two lines are *default methods* for the operators defined to be in this class.
- A type is made an instance of a class by an *instance declaration.* For example:

```
instance Eq Int where
    x == y = intEq x y   -- primitive equality for Ints
instance Eq Float where
    x == y = floatEq x y -- primitive equality for Floats
```

Equality, cont'd

- *User-defined* data types can also be made instances of Eq. For example:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

instance         Eq (Tree a) where
    Leaf a1      == Leaf a2       = a1 == a2
    Branch l1 r1 == Branch l2 r2 = l1==l2 && r1==r2
    _            == _             = False
```

- But something is strange here: is "**`a1 == a2`**" on the right-hand side correct? How do we know that equality is defined on the type "**`a`**"???

Equality, cont'd

- *User-defined* data types can also be made instances of Eq. For example:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

instance Eq a => Eq (Tree a) where
     Leaf a1        == Leaf a2       = a1 == a2
     Branch l1 r1 == Branch l2 r2 = l1==l2 && r1==r2
     _              == _             = False
```

- But something is strange here: is "**a1 == a2**" on the right-hand side correct? How do we know that equality is defined on the type "**a**"???
- Answer: Add a constraint that requires **a** to be an equality type.

Constraints / Contexts are Propagated

- Consider this function:

```
x `elem`  []        =  False
x `elem` (y:ys)     =  x==y || x `elem` ys
```

- Note the use of (==) on the right-hand side of the second equation. So the principal type for elem is:

```
elem :: Eq a => a -> [a] -> Bool
```

- This is inferred automatically by Haskell, but, as always, it is recommended that you provide your own type signature for all top-level functions.

Classes for Regions

- Useful slogan:

  "polymorphism captures similar structure over different values, while type classes capture similar operations over different structures."
- For a simple example, recall from Chapter 8:

```
containsS :: Shape -> Point -> Bool
containsR :: Region -> Point -> Bool
```

- These are similar ops over different structures. So:

```
class PC t where
        contains :: t -> Point -> Bool
instance PC Shape where
        contains = containsS
instance PC Region where
        contains = containsR
```

Numeric Classes

- Haskell's numeric types are embedded in a very rich, hierarchical set of type classes.
- For example, the Num class is defined by:

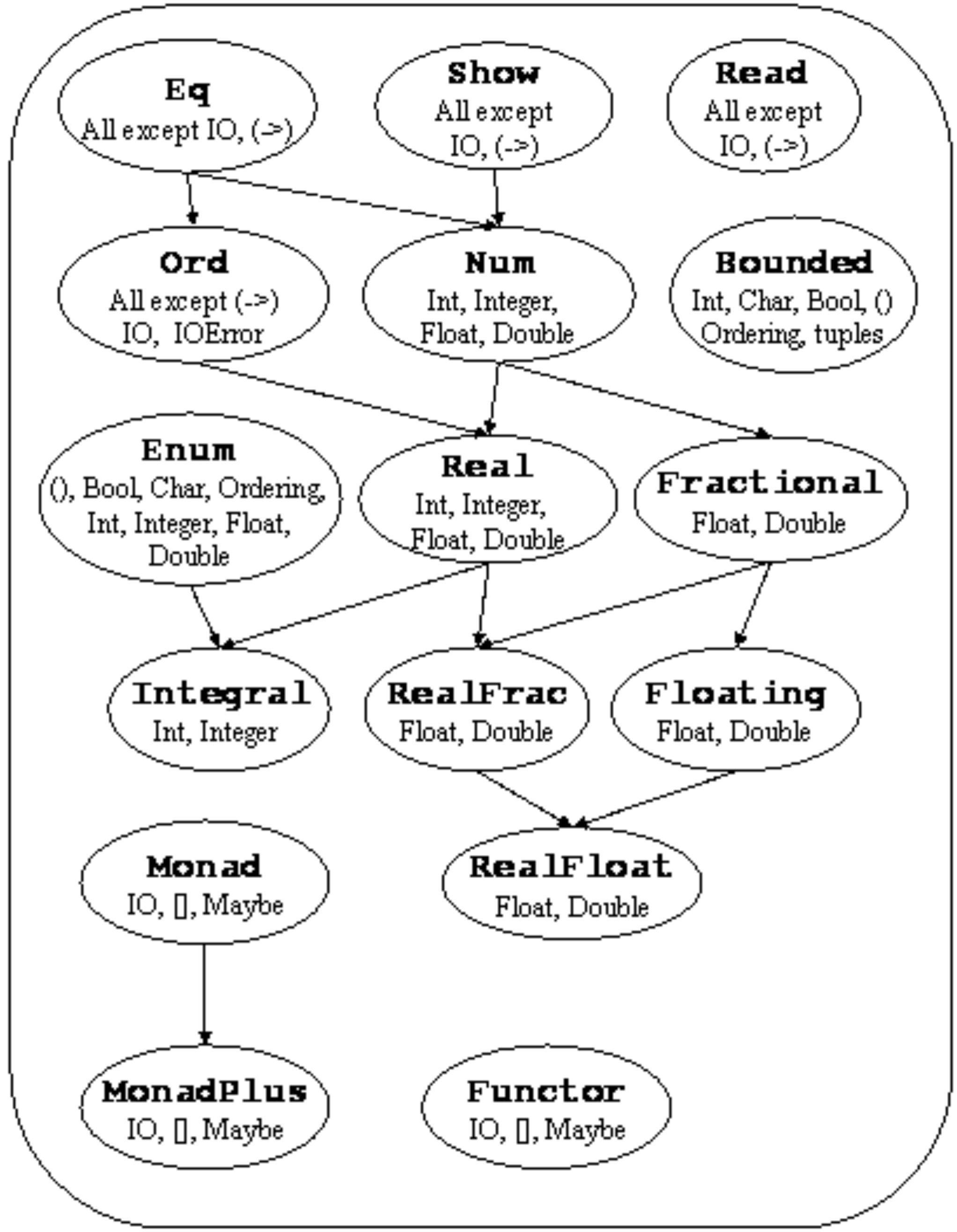
```
class  (Eq a, Show a) => Num a  where
        (+), (-), (*) :: a -> a -> a
        negate :: a -> a
        abs, signum :: a -> a
        fromInteger :: Integer -> a
```

- ...where Show is a type class whose main operator is

```
show :: Show a => a -> String
```

- See the Numeric Class Hierarchy in the Haskell Report on the next slide.

Haskell's Standard Class Hierarchy



Coercions

- Note this method in the class Num:
  ```
  fromInteger :: Num a => Integer -> a
  ```
- Also, in the class Integral:
  ```
  toInteger :: Integral a => a -> Integer
  ```
- This explains the definition of intToFloat:
  ```
  intToFloat :: Int -> Float
  intToFloat n = fromInteger (toInteger n)
  ```
- These generic coercion functions avoid a quadratic blowup in the number of coercion functions.
- Also, every integer literal, say "42", is really shorthand for "fromInteger 42", thus allowing that number to be typed as *any* member of Num.

Derived Instances

- Instances of the following type classes may be automatically *derived*:

  Eq, Ord, Enum, Bounded, Ix, Read, and Show

- This is done by adding a *deriving* clause to the data declaration. For example:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
        deriving (Show, Eq)
```

- This will automatically create an instance for Show (Tree a) as well as one for Eq (Tree a) that is precisely equivalent to the one we defined earlier.

Derived vs. User-Defined

- Suppose we define an implementation of finite sets in terms of lists, like this:

```
data Set a = Set [a]

insert (Set s) x = Set (x:s)

member (Set s) x = elem x s

union (Set s) (Set t) = Set (s++t)
```

Derived vs. User-Defined

- We can automatically derive an equality function just by adding “deriving Eq” to the declaration.

```
data Set a = Set [a]
   deriving Eq

insert (Set s) x = Set (x:s)

member (Set s) x = elem x s

union (Set s) (Set t) = Set (s++t)
```

But is this really what we want??

Derived vs. User-Defined

- No!
- E.g.,

  (Set [1,2,3]) == (Set [1,1,2,2,3,3]) ➔ False

A Better Way

```
data Set a = Set [a]

instance Eq a => Eq (Set a) where
  s == t  =  subset s t && subset t s

subset (Set ss) t = all (member t) ss
```

Haskell Classes <> OO Classes

- Warning…
  - The terminology used in Haskell (classes, instances, inheritance, etc.) is obviously intended to have something to do with Object-Oriented Programming.
  - However, the exact correspondence is a bit tricky.
  - I recommend *not* trying to think about this for the time being.

Reasoning About Type Classes

- Most type classes implicitly carry a set of *laws*.
- For example, the Eq class is expected to obey:

```
(a /= b)  =  not (a == b)
(a == b) && (b == c)  ⊇  (a == c)
```

- Similarly, for the Ord class:

```
a <= a
(a <= b) && (b <= c)  ⊇  (a <= c)
(a <= b) && (b <= a)  ⊇  (a == b)
(a /= b)  ⊇  (a < b) || (b < a)
```

- These laws capture the properties of an *equivalence class* and a *total order*, respectively.
- Unfortunately, there is nothing in Haskell that *enforces* the laws – its up to the programmer!