Contents

A Better Dictionary

A lookup table as a list of pairs is very inefficient. Potentially, we have to search through the entire table in order to find the one we want. If we assume that the elements can be kept in some kind of order then the definition can be improved.

A *binary search tree* (`Tree`) will improve efficiency. We will want to store a root node for some type `nd` and two subtrees.

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

We can immediately write `map` and `fold` for trees

Tree Map

Remember - we leave the stucture alone and modify the content Type Signature:

```
mapTree :: (a -> b)  -- function to change content
           -> Tree a -- input tree
           -> Tree b -- output tree
```

Base Case (Leaf):

```
mapTree f Leaf = Leaf
```

Recursive Case (Branch):

```
mapTree f (Branch ltree n rtree)
  = Branch (mapTree f ltree)
           (f n)
           (mapTree f rtree)
```

Tree Fold

Remember - we leave the content alone and modify the structure

```
Leaf :: Tree nd
Branch :: Tree nd -> nd -> Tree nd -> Tree nd
```

Type signature

```
foldTree :: a                        -- replacement for Leaf
            -> (a -> nd -> a -> a) -- replacement for Branch
            -> Tree nd               -- input truee
            -> a                     -- final value
```

First constructor (Leaf):

```
foldTree zero f Leaf = zero
```

Second constructor (Branch):

```
foldTree zero f (Branch ltree n rtree)
  = f (foldTree zero f ltree)
      n
      (foldTree zero f rtree)
```

Using a Binary Tree as a Dictionary

We take te node type (`nd` previously) to be a key-data pair, where the key is a string and we have integer data

```
type TDict = Tree (String, Int)
```

If all we have is a `Leaf`, then we build a `Branch` with empty sub-trees

```
insert Leaf key value = Branch Leaf (key, value) Leaf
```

Otherwise we compare keys to decide where the insert should occur:

```
insert (Branch l (k, v) r) key value
  | key < k  = Branch (insert l key value) (k, v) r
  | key > k  = Branch l (k, v) (insrt r key value)
  | key == k = Branch l (k, value) r
```

Lookup up Binary Tree

A `Leaf` has no key-data contents:

```
lookup Leaf _ = Nothing
```

With a `Branch` we compare keys to see where to search

```
lookup (Branch l (k, v) r) key
  | key < k  = lookup l key
  | key > k  = lookup r key
  | key == k = Just v
```

Instantiating Example

- Consider the following non-polymorphic Binary Tree type:

```
data STree = Lf String | Br STree STree
```

- We shall define instances for `STree` for classes `Eq`, `Org`, and `Show`
- `Tree nd`, is polymorphic, so for now we start with a simpler example

STree instance for Eq

We state our intention to create an instance:

```
instance Eq STree where
```

`Lf` are equal only if their strings are:

```
(Lf s1) == (Lf s2) = s1 == s2
```

`Br` are equal only if their components are

```
(Br t11 t12) == (Br t21 t22)
              = t11 == t21 && t12 == t22
```

Any other combination yields `False`:

```
_ == _ = False
```

The definition of `/=` wil be derived by the compiler from that for `==`

STree instance for Org

We assume `Lf` are smaller than `Br`

```
instance Ord STree where
  compare (Lf s1) (Lf s2) = compare s1 s2
  compare (Lf _) (Br _ _) = LT
  compare (Br _ _) (Lf _) = GT
  compare (Br t11 t12) (Br t21 t22)
    | t11 = t21 = compare t12 t22
    | otherwise = compare t11 t21
```

We only need to define `compare` - the others are derivable from it.

STree instance for Show

We display the tree here using the same Haskell syntax we use to create the values

```
instance Show STree where
  show (Lf s) = "(Lf " ++ s ++ ")"
  show (Br t1 t2)
    = "(Br " ++ show t1 ++ " " ++ show t2 ++ ")"
```

Again, `showsPrec` and `showList` can be derived

Deriving Instances

- For certain (standard) type-classes, we can ask Haskell to automatically generate instances for user-defined types
- So the following code replaces all the instance declarations shown above

```
data STree = Lf String | Br STree STree
  deriving (Eq, Ord, Show)
```

- `deriving` is another Haskell keyword
- Deriving can be done for the following classes among others: `Eq, Org, Enum, Bounded, Show` and `Read`

Not the Whole Story

There are some aspects of the typeclass system that haven't been discussed yet

- Some classes depend on other classes
- Some classes are themselves polymorphic
- Some classes are associated with type constructors

Classes based on other Classes

- Here is part of the class declaration for `Ord`:

```
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
```

- The notation `(Eq a =>` is a *context*, stating that the `Ord` class depends on the `Eq` class (why?)
- In order to define `compare`, we have to use `==`
  - So, for a type to belong to `Ord`, it must belong to `Eq`
  - Think of it as a form of inheritance

"Polymorphic" Type Classes

How might we define an `Eq` instance of lists?

- For `[Bool]`

```
instance Eq [Bool] where
  [] == []             = True
  (b1:bs1) == (b2:bs2) = b1 == b2 && bs1 == bs2
  _ == _               = False
```

- For `[Int]`

```
instance Eq [Int] where
  [] == []               = True
  (i1:is1) == (i2:is2) = i1 == i2 && is1 == is2
  _ == _                 = False
```

- Can't we do this polymorphically?

```
instance (Eq a) => Eq [a] where
  [] == []                 = True
  (x1:xs1 == x2 == xs2) = x1 == x2 && xs1 == xs2
  _ == _                   False
```

- We can define equality on `[a]` provided we have equality set up for `a`
- Here we are defining equality for a type constructor (`[]` for lists) applied to a type `a`:
  - So the class refers to a type built with a constructor

Type-Constructor Classes

- Consder the class declaration for `Functor`

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

- Here we are associating a class with a *type-constructor* `f`
  - Not with a type
  - See how in the type signature `f` is applied to type variables `a` and `b`
  - So, `f` is something that takes a type as argument to produce a type

Examples

- The `Maybe` type-constructor

```
data Maybe a = Nothing | Just a
```

- The `IO` type-constructor

```
data IO a = ...
```

- The `[]` type-constructor

The type we usually write as `[a]` can be written at `[] a`, i.e. the application of list constructor `[]` to a type `a`

Instances of Functor

- `Maybe` as a `Functor`

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

- `[]` as a `Functor`

```
instance Functor [] where
  fmap = map
```

- Both the above are straight from the Prelude

Class Contexts

- We have seen notation in class declarations stating the one class depends on another, e.g. `(Eq a) => Ord a`
- However, we also see such contexts in function type signatures:

```
sum :: (Num a) => [a] -> a
```

- What are they telling us about such functions?

In Function Types

- Consider a possible definition of `sum`:

```
sum [] = 0
sum (n:ns) = n + sum ns
```

- The function is almost molymorphic, but for the use of `+`
- It will work provided the element type belongs to the `Num` class
- This is exactly what the type signature says

```
sum :: (Num a) => [a] -> a
```

> "`sum` transforms a list of `a`s to a result of type `a`, provided type `a` is in the `Num` class (i.e. supports `+`)

Instances and Type Declarations

- A type can only have one instance of any given class
- A type synonym therefore cannot have its own instance declaration
  - `type MyType a = ...`
  - It simply is a shorthand for an existing type
- A user-defined algebraic datatype can have instance declarations
  - `data MyData a = ...`
  - In general, we need to do this for `Eq, Show` in any case
- A user-cloned (new) type can also have instance declarations
  - `newtype MyNew a = ...`
  - A key use of !newtype! is to allow instance declarations for existing types (now "re-badged")