11 Types and trees

Recall (from lecture 3) that a data type declaration like

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

introduces three things:

- a type-value function of types, *Either* that constructs a new type for any two types a and b;
- an injection $Left :: a \to Either\ a\ b$, and
- an injection $Right :: b \to Either\ a\ b$, in such a way that every proper value of $Either\ a\ b$ is either in the image of *Left* or the image of *Right*, but never both.

The functions *Left* and *Right* are *constructors* which are necessarily invertible. We could define functions

```
> left  (Left x)  = x
> right (Right y) = y
```

as their inverses (known as *selectors*), but Haskell provides a convenient abbreviation:

```
> data Either a b = Left {left :: a} | Right {right :: b}
```

Care: the inverses are partial functions, and the claim is that the constructors have inverses on the left.

More generally any contructor can have any number of arguments, for example

```
> data Pair a = Pair { first :: a,  second :: a }
```

The type $Pair\ a$ is not quite the same as the type (α, β) of pairs (x, y) of elements $x :: \alpha$ and $y :: \beta$ because a $Pair\ \alpha$ has elements both of which are α. The pun between the name of the type function *Pair* and its unique constructor $Pair :: \alpha \to \alpha \to Pair\ \alpha$ is common, but some people prefer a different name like *mkPair* for the constructor.

It is worth mentioning the predefined type

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

where different constructors have different numbers of arguments. I am reasonably confident *Maybe* was named by Mike Spivey in a paper published in 1990. He has been known talk about *Adverb-Oriented Programming*.

Constructors with no arguments are distinct constants, for example

```
> data Bool = False | True
> data Day = Sunday | Monday   | Tuesday | Wednesday |
>                    Thursday | Friday  | Saturday
```

These are not strings: you have to explain how to print them, for example by

```
> instance Show Day where
>     show Sunday = "Sunday"
>     show Monday = "Monday"
>     ...
```

(Of course, you might want `show Sunday = "Sul"`, etc.)

Even though they are distinct there is no equality test unless you instantiate

```
> instance Eq Day where
>     Sunday == Sunday = True
>     Monday == Monday = True
>                     ...
>     _      == _      = False
```

although it would perhaps be better to define

```
> daynum :: Day -> Int
> daynum Sunday = 0
> daynum Monday = 1
>              ...

> instance Eq Day where
>     d1 == d2 = daynum d1 == daynum d2
```

11.1 Strictness and polynomial types

These data types are made of sums (coproducts or alternatives) of products (or tuples). Constructors are never strict: the construction of a tuple necessaily distinguishes (say) $\perp :: Pair\ \alpha$ and $Pair\ \perp\ \perp$ and so on. Since the predefined pairs (and other tuples) are also products, the same is true for them.

However, pattern matching on constuctors is strict, even when there is only one constructor:

```
> zero :: (a,b) -> Int
> zero (x,y) = 0
```

is strict in its argument: $zero\ \perp = \perp$. The less strict function is defined by

```
> zero' :: (a,b) -> Int
> zero' xy = 0
```

These types are called *lifted* because the values you want have all been 'lifted' up the ($\sqsubseteq$) ordering by putting a $\perp$ underneath them.

Sometimes you really do not need the lifted type. A type like

```
> data Value a = Value a
> data Count a = Count a
```

might be used to distinguish a *Count Int* used for one purpose from a *Value Int* used for another. The typechecker would prevent one from being confused with the other.

However you would like the type of *Value Int* to be the same as *Int*: you do not want to distinguish *Value* $\perp$ from $\perp$. Just for this there is a declaration

```
> newtype Value a = Value a
```

which makes the constructor *Value* be strict. (The practical effect if this is that a *Value Int* can be represented at run time by an *Int*: all the type checking happens at compile time, and there is no run time cost either in space or time.)

11.2 Recursive types

We have seen types like

```
> List a = Nil | Cons a (List a)
```

(which is isomorphic to $[\alpha]$ where the contructors are $[\,]$ and $(:)$). These include values with arbitrary numbers of constructors, like infinite lists.

More generally, we might represent sets of αs by

```
> data Set a = Empty | Singleton a | Union (Set a) (Set a)
```

but the values of these are trees. If you really wanted them to behave like sets, you would want something like

```
> instance Eq a => Eq (Set a) where
>   xs == ys = (xs `subset` ys) && (ys `subset` xs)
```

and then you would need to define a recursive function *subset*.

11.3 Maps

The function $map :: (\alpha \to \beta) \to ([\alpha] \to [\beta])$ can be generalised from lists (of α) to other datatypes with one argument. One of the properties is that $map\ id$ is the identity on $[\alpha]$, another that $map\ f \cdot map\ g = map\ (f \cdot g)$.

The map for *Pair* is

```
> mapPair :: (a -> b) -> (Pair a -> Pair b)
> mapPair f (Pair x y) = Pair (f x) (f y)
```

that for *Maybe* is

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

that for *Set* is

```
> mapSet :: (a -> b) -> (Set a -> Set b)
> mapSet f  Empty        = Empty
> mapSet f (Singleton x) = Singleton (f x)
> mapSet f (Union xs ys) = Union (mapSet f xs) (mapSet f ys)
```

It would be good to be able to capture this commonality. The type class *Functor* does roughly this:

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

It does not capture the invariants that *fmap* preserves identity and composition: these are expected properties of any function that you define as an instance of *fmap*. (The predefined class also contains $(<\$) :: b \ \to f\ a \to f\ b$.)

Notice that f is not a type: it is a *type* $\to$ *type* function, so the instances are for example

```
> instance Functor [] where
>    fmap f    []  = []
>    fmap f (x:xs) = f x : fmap f xs

> instance Functor Pair where
>    fmap f (Pair x y) = Pair (f x) (f y)

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

The effect of these definitions is that if $f :: \alpha \to \beta$ you can use *fmap f* as a $[\alpha] \to [\beta]$ or as a $Maybe\ \alpha \to Maybe\ \beta$ and so on.

There is no instance of *Functor Either*, because the type constructor expects too many arguments, but there is an instance of *Functor* (*Either* α), as a functor on the second type. (I am pretty sure I think there should not be.)

11.4 Deriving standard instances

It can be tedious to define obvious instances of classes for types, so Haskell can provide these automatically. For example

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

makes default definitions for *Eq* (*Either* α β) and so on. The instances of *Read* makes $read :: String \to Either\ \alpha\ \beta$ be the function that parses your input to the interpreter to turn it unto an *Either a b*, and the *Show* makes $show :: Either\ \alpha\ \beta \to String$ produce the representation that the interpreter would use to print an answer.

The equality defined by deriving *Eq* is structural equality, which is what you want for *Either* α β, and will itself require *Eq* α and *Eq* β; but structural equality is not what you want for *Set* α.

The ordering derived for *Either* α β makes all *Left x* be less than all *Right y* and otherwise compares $Left\ x < Left\ y$ iff $x < y$ and so on.

The class *Enum* α includes

```
> class Enum a where
>     succ     :: a -> a
>     pred     :: a -> a
>     toEnum   :: Int -> a
>     fromEnum :: a -> Int
```

used in translating list enumerations like $[x \mathinner{..} y]$. The derived instance for *Day* would make $fromEnum\ Tuesday = 2$ and $succ\ Friday = Saturday$, and so on.

The class *Bounded* α includes

```
> class Bounded a where
>    minBound, maxBound :: a
```

The derived instance of *Bounded Day* would make $minBound = Sunday$ and $maxBound = Saturday$.

11.5 Folds

Recall that the *fold* for a data type T acts to replace each of the constructors of T with the arguments of the fold function, and when applied to the constructors returns the identity $T \to T$.

Where the type is not recursive neither is the fold function:

```
> foldPair :: (a -> a -> t) -> Pair a -> t
> foldPair pair (Pair x y) = pair x y

> foldMaybe :: b -> (a -> b) -> Maybe a -> b
> foldMaybe nothing just Nothing  = nothing
> foldMaybe nothing just (Just x) = just x

> foldEither :: (a -> c) -> (b -> c) -> Either a b -> c
> foldEither left right (Left x)  = left x
> foldEither left right (Right y) = right y
```

The standard prelude defines *maybe* and *either* which are the folds on those last two types (but the designers of the prelude probably did not think of them as folds, rather as generalised deconstructors).

The fold function for a recursive type like

```
> data BTree a = Leaf a | Fork (BTree a) (BTree a)
```

would be recursive

```
> foldBTree leaf fork (Leaf x)   = leaf x
> foldBTree leaf fork (Fork l r) = fork (foldBTree leaf fork l)
>                                       (foldBTree leaf fork r)
```

or perhaps more sensibly

```
> foldBTree :: (a -> b) -> (b -> b -> b) -> BTree a -> b
> foldBTree leaf fork = f
>   where f (Leaf x)   = leaf x
>         f (Fork l r) = fork (f l) (f r)
```

Here is another kind of tree, a *rose tree*:

```
> data RTree a = RTree a [RTree a]
```

which is never empty, and where each node can have any number of children. There is a natural *map* on rose trees

```
> instance Functor RTree where
>       fmap f (RTree a ts) = RTree (f a) (map (fmap f) ts)
```

The corresponding fold function is

```
> foldRTree :: (a -> [b] -> b) -> RTree a -> b
> foldRTree node (RTree x ts) = node x (map (foldRTree node) ts)
```

These ideas generalise to a bushy tree with other arrangments of children, provided the *map* function on the list of children can be replaced by the appropriate *fmap*

```
> data Bush f a = Bush a (f (Bush f a))
```

so that *Bush* [ ] *a* is essentially the same as *RTree a*

```
> instance Functor f => Functor (Bush f) where
>       fmap f (Bush a ts) = Bush (f a) (fmap (fmap f) ts)
```

where on the right hand side the first, outer, call of *fmap* is that for the functor f, and the second, inner one is a recursive call of *fmap* for *Bush* f, and

```
> foldBush :: Functor f => (t -> f b -> b) -> Bush f t -> b
> foldBush bush (Bush a ts) = bush a (fmap (foldBush bush) ts)
```

Just as with lists, once the *foldBush* function is defined, *fmap* could be defined by

```
> instance Functor f => Functor (Bush f) where
>       fmap f = foldBush (Bush . f)
```

There is no type class with a general folding function in it, because folds have very different types from each other, depending on the numbers and types of constructors. (This is *not* what *Foldable* is! That is sadly something much more confusing.)

However, in general it is possible to write a map function as an instance of the corresponding fold.

Exercises

11.1 What are the natural folds on *Bool* and

```
> data Day = Sunday | Monday   | Tuesday | Wednesday |
>                     Thursday | Friday  | Saturday
```

11.2 Given that the ordering on *Bool* is the one that would be obtained by **deriving**(*Ord*), to what logical function of two variables does (<=) correspond?

11.3 Write out the fold function for the data type

```
> data Set a = Empty | Singleton a | Union (Set a) (Set a)
```

and use it to define a function

```
> isIn :: Eq a => a -> Set a -> Bool
```

which tests whether an element appears as a value in the tree. Hence define a function

```
> subset :: Eq a => Set a -> Set a -> Bool
```

which tests whether all the elements of the first set are elements of the second. Use this to implement

```
> (==) :: Eq a => Set a -> Set a -> Bool
```

for equality of the sets represented by two trees from *Set*.

11.4 Define a function

```
> find :: Eq a => a -> BTree a -> Maybe Path
```

which searches for a value in the leaves of a *BTree*,

```
> data BTree a = Leaf a | Fork (BTree a) (BTree a)
```

returning a path, a sequence of *go left* and *go right* instructions, from the root to the leftmost occurrence of the value, if there is one, where

```
> data Direction = L | R
> type Path = [ Direction ]
```

You should aim to make use of folds and maps where possible.

11.5 A total version $head' :: [a] \to Maybe\ a$ of the standard function *head* might be defined by

```
> head'    []  = Nothing
> head'(x:xs) = Just x
```

Show, by deriving a definition, that there is a unique function $(\oplus)$ satisfying

$$head'(xs +\!\!+ ys) \quad = \quad head'\ xs \oplus head'\ ys$$