Contents

Map and Fold, or: Iteration in Functional Style

- We have seen some built-in and user-defined datatypes (e.g. Lists (`[t]`), `Maybe`), and examples of function definitions that use them
- We are now going to take a ore systematic look at the relationship between datatypes and code

Haskell Lists Reconsidered

- In Haskell, the type "list of type t" is written `[t]`
  - So, in a type-expresson, `[_]` is a *type-constructor*, a type-valued function taking a type as argument and returning a type as result
- A "list of t" (`[t]`) has two forms
  1. "empty", or "nil", written `[]`
  2. a non-empty "cons" node with two sub-components, the first (`x`) of type `t`, the second (`xs`) of type `[t]`, written `x:xs`
- The notations `[]` and `:` are *data-constructors* for the type `[t]`. That is, they are both *functions* from zero or more arguments to result of type `[t]`

  - `[]` is a data-constrctor taking no argument: `[] :: [t]`
  - `:` is a data-constructor taking two arguments: `(:) :: t -> [t] -> [t]`

Structure vs Contents

- Consider the list `[4, 2, 7]`, or written without syntactic sugar: `4:(2:(7:[]))`
- It has a *structure*, defined by the pattern of data-constructors used
  - 4**:**(2**:**(7**:**[]))
- It has *contents*, defined by data values supplied as arguments to data-constructors
  - **4**:(**2**:(**7**:[]))
- We can define two key functions: one that changes contents whilst leaving structure unchanged, whilst the other does the opposite

Map

Replace Content

- We can define a function that allows us to systematically transform content, whilst leaving the structure intact: `map`
- A map function can be defined for every `data`-type we have or define
- It typically takes two arguments
  1. A function that transforms the content
  2. An instance of the datatype
- It returns an instance of the datatype with the structure unchanged, but the **contents** replaced by the result of applying the function

For Haskell Lists

- `map` makes a function changing list elements (values) and a list as arguments, and returns the modified list as result:

```
map :: (s -> t) -> [s] -> [t]
```

- With the empty list, all we have is structure, so there is no change

```
map f [] = []
```

- With a cons-node, we can modify the first value, and recurse to do the rest:

```
map f (x:xs) = (f x):(map f xs)
```

- Example:

```
map (+1) (4:(2:(7:[]))) = 5:(3:(8:[]))
```

Fold

Replace Content

- We can define a function that allows us to systematically transform structure, whilst leaving the values "almost intact": `fold`
- A fold function can be defined for every `data`-type we have or define
- It typically takes the following arguments:
    1. Functions/values to replace data-constructors (structure)
- It returns an instance of the datatype with the **data constructors** replaced by the corresponding functions
- The results expression has contents unchanged initially (thanks to laziness), but ocne evaluated, everything will collapse ("fold") into some final value

For Haskell Lists

- `fold` takes functions replacing `[]` and `:` and a list as arguments, and returns a matching expressions structure as a result:

```
fold :: t -> (s -> t -> t) -> [s] -> t
```

- With the empty list, we replace the zero-argument constructor with the (nullary) replacement function (value):

```
fold x op [] = z
```

- With a cons-node, we replace the two argument constructor with the two argument replacement function, and recurse:

```
fold z op (x:xs) = x `op` (fold z op xs)
```

- Example:

```
fold 0 (+) (4:(2:(7:[]))) = 4+(2+(7+0))
```

Map is a Fold

- We can implement `map` as a `fold`

```
mapAsFold :: (a -> b) -> [a] -> [b]
mapAsFold f = foldr (mop f)

mop :: (a -> b) -> a -> [b] -> [b]
mop f x xs = f x : xs
```