COSC 3015: Lecture 10

Lecture given by Prof. Caldwell and scribed by Sunil Kothari

September 25, 2008

1 HW recap

unique_stable preserves the order and keeps the earlier elements whereas unique keeps the last occurrences of elements.

2 More functions

2.1 member

$$\begin{aligned} member1 :: a &\rightarrow [a] \rightarrow Bool \\ member1\ y\ [] &= False \\ member1\ y\ (x : xs) &= y == x \mid\mid member1\ y\ xs \end{aligned}$$

The "||" has short-circuit evaluation.i.e. if the first argument evaluates to true then it doesn't check for the value of second argument.

We can also write member as.

$$\begin{aligned} member'\ y\ [] &= False \\ member'\ y\ (x : xs) &= y == x || member'\ y\ xs \end{aligned}$$

member' evaluates as follows:

$$\begin{aligned} member'\ 1\ [1,1] &\leadsto member'\ 1\ [1] || 1 == 1 \\ &\leadsto member'\ 1\ [] || 1 == 1 || 1 == 1 \\ &\leadsto False || True || True \\ &\leadsto True | True \\ &\leadsto True \end{aligned}$$

On the other hand, *member1* evaluates as follows:

$$\begin{aligned} member1\ 1\ [1,1] &\leadsto 1 == 1 || member1\ 1\ [1] \\ &\leadsto True || member1\ 1\ [1] \\ &\leadsto True \end{aligned}$$

member is *elem* in the Haskell prelude *elem x m* is written *'elem'* *m* to simulate $x \in m$

2.2 zip

The zip (in our case zip1) function is defined as:

$$\begin{aligned} zip1\ []\ m &= [] \\ zip1\ m\ [] &= [] \\ zip1\ (x : xs)\ (y : ys) &= (x, y) : zip1\ xs\ ys \end{aligned}$$

and has the type

```
zip1:: [a] -> [b] -> [(a,b)]
```

What is the design decision for zip1 ? For example, $zip1\ [1,2,3]\ [a,b] = [(1,a),(2,b)]$

Let's do a shorter one. $zip1\ [3]\ [] \rightarrow []$

Note: The zip in the theorem below refers to the Haskell prelude's zip.

Theorem 1. $\forall m : [a], \forall n : [b]. length\ (zip\ m\ n) == min\ (length\ m)(length\ n)$

Theorem 2. $\forall m : [a], \forall n : [b]. length\ (zip'\ m\ n) == max\ (length\ m)(length\ n)$

Since the arguments to *zip1* are lists of arbitrary types, what will you do when you run out of values.

1. Pass a special value for type a and type b. These special values must be unique.
2. Define a value called *Nothing*.

Consider the *Maybe* data type:

$$data\ Maybe\ a = Just\ a\ \mid Nothing$$

Nothing is like a special value that means null - but works for any type.

So let's define *zip1* now:

$$\begin{aligned} zip1[][] &= [] \\ zip1\ []\ (x:xs) &= (Nothing, x) : zip1\ []\ xs \\ zip1\ (x:xs)\ [] &= (x, Nothing) : zip1\ xs\ [] \\ zip1\ (x:xs)\ (y:ys) &= (x, y) : zip1\ xs\ ys \end{aligned}$$

```
Main> :t zip1
zip1 :: [Maybe a] -> [Maybe b] -> [(Maybe a,Maybe b)]
```

But if we define *zip1* slightly differently as:

$$\begin{aligned} zip1\ []\ [] &= [] \\ zip1\ []\ (x:xs) &= (Nothing, Just\ x) : zip1\ []\ xs \\ zip1\ (x:xs)\ [] &= (Just\ x, Nothing) : zip1\ xs\ [] \\ zip1\ (x:xs)\ (y:ys) &= (Just\ x, Just\ y) : zip1\ xs\ ys \end{aligned}$$

```
Main> :t zip1
zip1 :: [a] -> [b] -> [(Maybe a,Maybe b)]
```

$$\begin{aligned} zip1\ []\ [1] &\leadsto (Nothing, Just\ 1) : Zip1\ []\ [] \\ &\leadsto (Nothing, Just\ 1) : [] \\ &\leadsto [(Nothing, Just\ 1)] \end{aligned}$$

We can use type class to give the following expressions a meaningful semantics:
Just 5 + 7
Nothing + 7
But we can also use *case* statement

```
case m of
    Nothing -> error "..."
    Just k -> k
```

But this definition has an error.

2.3 Either

The data type *Either* can be defined as: $data\ Either\ a\ b = Left\ a|Right\ bderivingShow$

The either type is also called disjoint union

```
> Left 5
Left 5 :: Num a => Either a b

> :t Right "xyxxy"
Either a [char]
```

It helps us put together two different types. The constructors are like tags on data values -

Either Int [char] - Taking the union of Int and [char] So what are the inhabitants of this type ?
$\{Left\, 0, Left\ -1, Right\, "a", Right\, "ab", \ldots\}$

We can deconstruct the disjoint union by the case statement.

```
case m of
  left x -> ...x...
  right y -> ...y...
```

where,
x - an integer
y - string

```
 case (Left 5) of
   Left x -> x + 1
   Right y -> length y
```

would evaluate to 6.

What about the following expression ?

```
case (Left 5) of
   Right y -> length y
   Left x -> x + 1
```

would evaluate to 6.

2.4 filter

In Haskell, *filter* is defined as:

```
 Hugs> :t filter
filter :: (a -> Bool) -> [a] -> [a]
```

And so we can define our *filter1* is defined as

$$\begin{aligned} filter1\, p\, [] &= [] \\ filter1\, p\, (x : xs) &= \text{if p x then x:filter p xs else filter p xs} \end{aligned}$$

```
filter1(/= 5) [1,5,2,5,3,4,5,5]
[1,2,3,4]
```

where $(/=5)\backslash x \rightarrow x+1$.
Recall, in HW we have *remove_all*:
$remove\_all\ x\ m = filter\ (/=x)\ m$
can also be written as :
$remove_all\ x = filter\ (/=x)$

```
Main> :t remove_all
remove_all :: Eq a => a -> [a] -> [a]
```

What about $filter\ (==(+))\ [(+),(*),(-)]$?
Compiler will give an error since functions cannot be compared.

2.5 Finite functions as lists of pairs

$[(1,"xy"),(1,"zw")]$ - It is not a function since one domain element is getting mapped to different range values.

But, $[(1,"xy"),(1,"xy")]$ is functional - since the element in domain are mapped to the same values in the co-domain.

What does it mean for two sets to be equal ?
$S = T \stackrel{\text{def}}{=} \forall x.x \in S \leftrightarrow x \in T$

So, what's the difference between a list and a set ?
As lists,$[1,1] \neq [1]$
As sets, $\{1,1\} = \{1\}$
Also,
As lists, $[1,2] \neq [2,1]$
As sets, $\{1,2\} = \{2,1\}$

So for lists order and multiplicity of elements is significant.
For sets- the only significant factor is membership.

2.6 Implementing sets as lists

Sets are defined as:
$data\ Set\ a = S\ [a]$

```
Main> :t S []
S [] :: Set a
Main> :t S
S :: [a] -> Set a
Main> :t S [1,2,3,4]
S [1,2,3,4] :: Num a => Set a
```

What if we change the datatype to

$$data\ Set\ a = S\ [a]\ deriving\ Eq$$

```
Main> :t S[]== S []
S [] == S [] :: Eq (Set a) => Bool

seteq [] [] = True
seteq [] (x:xs) = False
seteq (x:xs) [] = False
seteq (x:xs) m = x `elem` m && seteq (remove_all x xs) (remove_all x m)

instance Eq a => Eq (Set a) where
 m == n = seteq m n


Main> S[] == S[1]
False
Main> S[1,1,2] == S[2,2,2,1]
False
```

$$\begin{aligned} map\ f\ [] &= [] \\ map\ f\ (x : xs) &= f\ x : map\ f\ xs \end{aligned}$$

and *domain* can be defined as:

$$domain\ f = map\ fst\ f$$