COSC 3015: Lecture 9

Lecture given by Prof. Caldwell and scribed by Sunil Kothari

September 23, 2008

1 Functions on lists

Lists are the bread and butter of functional programming. At least they started out that way. The first functional programming language, LISP - List Processing, 1960 by John McCarthy, had list as the fundamental data-structure. LISP is great at symbolic processing - the AI programming language.

Using the Haskell notation we can write list as data Nat [a] = [] | a:[a]

so we can say

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

1.1 null function

The term cons actually means list constructor.

$$\begin{aligned} null2\ [] &= True \\ null2\ \_ &= False \end{aligned}$$

"_" is a specification that matches any pattern

Alternative:

If we define *null* as

$$null3\ x = (x == [])$$

then the type is

```
Main> :t null
null1 :: Eq [a] => [a] -> Bool
```

`if (x==[]) then True else False` is a really dumb way of defining the above function.

if b then e1 else e2, where

- b - bool
- $e1, e2$ - both must have the same type.

Evaluation for if-then-else
if True then e1 else e2 $\leadsto e1$
if False then e1 else e2 $\leadsto e2$

In the book *null* is defined as:

$$null2 \; [] = True$$
$$null2 \; (x : xs) = False$$

How is this different from one above ?

$null1 \perp = \perp$

So- null is strict - i.e. if it is applied to $\perp$ then the result is $\perp$. $null[(+), (-)]$

```
:t  [(+),(-)]
[(+),(-)] :: Num a => [a -> a -> a]
Main>
```

But we cannot do so for $null3$, it generates a type error.

```
Main> null3 [(+),(-)]
ERROR - Unresolved overloading
*** Type       : (Num a, Eq (a -> a -> a)) => Bool
*** Expression : null3 [(+),(-)]
```

But, *null2* and *null1* behave differently than *null3*.

```
Main> null2 [(+),(-)]
False
Main> null1 [(+),(-)]
False
```

So *null1* and *null2* apply to a wider class of list types - those that are instances of the Eq type class and those that are not.

1.2 Append (++)

We want that [1,2]++[3,4,5] = [1,2,3,4,5] If we had some reflection we can do it. So how can we do it ? We are gonna define it by recursion on the first argument.

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

So how dowe think about it. We have a cons and we want to glue together it with ys. So what is thew first element of what we wwant ? The first element is x. So the pattern is almost always same. In this case we take out x and recurse down on the smaller structure.

$$\begin{aligned} [1, 2] ++ [3, 4, 5] &= 1 : ([2] ++ [3, 4, 5]) \\ &= 1 : (2 : ([] ++ [3, 4, 5])) \\ &= 1 : (2 : [3, 4, 5]) \\ &= [1, 2, 3, 4, 5] \end{aligned}$$

Let's try another recursive definition.

1.3 length

What about *length* ? It's defined as

```
Main> :t length
length :: [a] -> Int
Main>
```

$$\begin{aligned} length [] &= 0 \\ length (x : xs) &= 1 + (length xs) \end{aligned}$$

1.4 reverse

Let's try reverse:

```
Main> :t reverse
reverse :: [a] -> [a]
```

and is defined as

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

1.5 concat

Let's do concat

```
Main> :t concat
concat :: [[a]] -> [a]
```

concat [[1,2],[],[3,4,5]] = [1,2,3,4,5] and is defined as

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

How do we compute with the concat function ?

$$\begin{aligned} concat[[1,2,3]] &= [1,2,3] ++ concat\ [] \\ &= [1,2,3] ++ [] \\ &= [1,2,3] \end{aligned}$$

Another example:

$$\begin{aligned} concat[[1,2,3],[],[4,5]] &= [1,2,3] ++ concat\ [[],[4,5]] \\ &= [1,2,3] ++ [] ++ concat\ [[4,5]] \\ &= [1,2,3] ++ ([] ++ [4,5] ++ concat\ []) \\ &= [1,2,3] ++ ([] ++ ([4,5] ++ [])) \\ &= \ldots \\ &= [1,2,3,4,5] \end{aligned}$$

1.6 zip

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

We define it by recursion on the first argument.
Here's an alternate definition

$$\begin{aligned} zip\ []ys &= [] \\ zip\ (x:xs)\ ys &= (x, headys) : zip\ xs\ (tailys) \\ zip\ (x:xs)\ (y:ys) &= (x,y) : zip\ xs\ ys \end{aligned}$$

Design choice for zip - what is the right length ??

$$\begin{aligned} length\ (zip\ xs\ ys) &\stackrel{?}{=} min\ (length\ xs, length\ ys) \\ &\stackrel{?}{=} max(length\ xs, length\ ys) \\ &\stackrel{?}{=} \text{if length xs <> length ys then error else length xs} \\ &\stackrel{?}{=} length\ xs\ or\ \text{if length xs > length ys then error} \end{aligned}$$

zip1 is strict in both argument

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

$$\begin{aligned} zip'\ []\ \perp = [] \\ zip'\ \perp\ [] = \perp \\ zip\ []\ \perp = \perp \end{aligned}$$

zip' is not strict in its second argument.

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

In Haskll there is intersting notation

- An infinits list of ints:
  $[1..] = [1, 2, 3, 4, \ldots]$
- Partial lists
  1. $\perp$
  2. $1 : \perp$
  3. $1 : 2 : \perp$
- Finite lists $[\perp, \perp]$ - the type of lists is [a]

1.7 last

$$\begin{aligned} last\ [] &= error \\ last\ (x:xs) &= \text{if (null xs) then x else last xs} \end{aligned}$$

$$\begin{aligned} last\ [1,2,3] &= last\ [2,3] \\ &= last\ [3] \\ &= 3 \end{aligned}$$

Here's another way of doing it

$$\begin{aligned} last\ [x] &= x \\ last\ x : xs &= last\ xs \end{aligned}$$

yet another way is:
$last = head \cdot reverse$

```
Main> :t error
error :: String -> a
```

In a way the result of error is like bottom.