COSC 3015: Lecture 11

Lecture given by Prof. Caldwell and scribed by Sunil Kothari

September 30, 2008

1 cons vs. append

cons is the list constructor, whereas append glues two lists

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

A slightly pathological case is x:⊥.

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

append glues lists on the right. cons can only add things to the left.
$[1, 2, 3] + +[4] \leadsto [1, 2, 3, 4]$
Remeber, append is defined as:

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

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

Cons can be implemented using append but is less efficient.
$(x : xs) = [x] + +xs$

2 More list functions

We talked about the map function earlier

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

The following will not work as a definition since "++" is not a constructor for lists.
$map\ f\ (xs + +ys) = (map\ f\ xs) + +(map\ f\ ys)$.

Note::Pattern-matching works on patterns specified using data-type constructors.
snd (x:y:m) = y

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

$$\begin{aligned} prod\ [] &= 0 \\ prod\ (x : xs) &= x * prod\ xs \end{aligned}$$

$$concat1\ [] = []concat1\ (xs + +xss) = xs + +concat\ xss$$

All these functions follow the following general pattern

$$\begin{aligned} f\ [] = e \\ f\ (x : xs) = x\ `op`\ fxs \end{aligned}$$

$$\begin{aligned} sum\ [1, 2, 3] &= 1 + sum\ [2, 3] \\ &= 1 + (2 + sum\ [3]) \\ &= 1 + (2 + (3 + sum\ [])) \\ &= 1 + (2 + (3 + 0)) \end{aligned}$$

$$prod[1, 2, 3] = 1 * (2 * (3 * 1))$$

$$concat[[1], [], [2, 3]] = [1] + +([] + +([2, 3] + +[]))$$

The pattern associates the "'op"' to the right.

$$\begin{aligned} foldr\ op\ e\ [] &= e \\ foldr\ op\ e\ (x : xs) &= x\ `op`\ (foldr\ op\ x\ xs) \end{aligned}$$

So if we had macros, we could just plug the functions and identity element in th macros to get the above definition.

Now, we can define the above functions using $foldr$
$sum = foldr\ (+)\ 0$
$prod = foldr\ (*)\ 1$
$concat = foldr\ (++)\ []$

What about the operators that don't associate to the right ?
For example, $a - (b - c) \neq (a - b) - c$
What if you want a left associative pattern ?
$sum'\ [1, 2, 3] = ((0 + 1) + 2) + 3$

We need $foldl$

$$\begin{aligned} foldl\ op\ e\ [\,] &= [\,] \\ foldl\ op\ e\ (x : xs) &= foldl\ op\ (e\ `op`\ x)xs \end{aligned}$$

The idea is carry the results completed so fat in e - starting with e being the identity. $sum' = foldl\ (+)\ 0\ sum'\ [1, 2, 3]$
$= foldl\ (1)\ 0\ [1, 2, 3]$
$= foldl\ (1)\ (0 + 1)\ [2, 3]$
$= foldl\ (1)\ ((0 + 1) + 2)\ [3]$
$= foldl\ (1)\ (((0 + 1) + 2) + 3)\ []$
$= (((0 + 1) + 2) + 3)$

Note: for associative operators $\oplus$ with identity e, $foldr\ op\ e = foldl\ op\ e$. In general $foldl$ is more efficient than $foldr$. We can do a $foldr$ computation as:

$$\begin{aligned} sum[1, 2, 3] &= foldr\ (+)\ 0\ [1, 2, 3] \\ &= 1 + (foldr\ (+)\ 0\ [2, 3]) \\ &= 1 + (2 + foldr\ (+)\ 0\ [3]) \\ &= 1 + (2 + (3 + foldr\ (+)\ 0\ [])) \\ &= 1 + (2 + (3 + 0)) \\ &= 1 + (2 + 3) \\ &= 1 + 5 \\ &= 6 \end{aligned}$$

3 List Comprehensions

In Haskell, strings are just list of chars.

```
Main> [(x,y)| x <- [1..3], y <- "abc"]
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c'),(3,'a'),(3,'b'),(3,'c')]
```

If we want to do this in the normal way we have

```
all_pairs [] ys = []
all_pairs (x:xs) ys = (map ( y -> (x,y)) ys) ++ all_pairs xs ys
```

$(map\ (\backslash y \rightarrow (1, y))\ [{}'a','b'])$
$= ((\backslash y \rightarrow (1, y))\ 'a') : (\ (\backslash y \rightarrow (1, y))\ 'b') : []$
$= (1,'a') : ((1,'b') : [])$
$= [(1,'a'), (1,'b')]$

$$\begin{aligned} all\_pairs1\ (x : xs)\ ys &= \text{map}\ p\ ys + +all\_pairs1\ xs\ ys \\ \text{where}\ p\ y &= (x, y) \end{aligned}$$

```
Main> [x*x| x <- [1..5], odd x]
[1,9,25]
```

- [1..5] - generators
- *odd x* - guard

So what is list comprehension ? $\{x \in S \mid P(x)\}$
In set theory, this is definition by comprehension.
In general, $[e \mid Q]$ is a list comprehension

$$\begin{array}{lll} where & e & -\text{is a Haskell expression} \\ & Q & -\text{is a comma separated list of generations and guards} \end{array}$$

Generators look like $x \leftarrow xs$ where x is a variable and xs is a list valued expression and a guard is a boolean valued expression.
List comprehensions are very expressive but add no computational power.
What is the semnatics ? There are two rules :

$$[e \mid x \leftarrow xs, Q] = concat\ (map\ f\ x) where\ f\ x = [e \mid Q]$$

Note:: I had to leave early and so I missed last 5-7 mins. of lecture material.