COSC 3015: Lecture 12

Lecture given by Prof. Caldwell and scribed by Sunil Kothari

October 2, 2008

1 HW queries

```
data FinFun a b = FF [(a,b)]
```

update (x,y) (FF m) where,

- FF is the constructor
- a is the domain
- b is the range

apply f x = case f of ....

Here's what we can do to get a hold of m - which is the list list of pairs for that finite function. apply (FF m) x =

```
:t apply
apply ::(FinFun a b) -> a -> Maybe b
```

apply is just *lookup*. We want to go inside and check if the first component of some pair in m matches x.

$$\begin{array}{rcl} apply\ (FF\ [])\ x & = & Nothing \\ apply\ (FF\ ((z,y) : xys)) & = & \text{if } z == x\text{then } (Just\ y) \text{ else } apply\ (FF\ xys)\ x \end{array}$$

Here's a different way of writing the above code

```
apply (FF m) x =
   case m of
     [] -> Nothing
     ((z,y)::zys) -> if z == x then Just y else apply (FF zys) x
```

The finite map can also be oregnaized in a binary tree then the lookup can be done in $O(log\ n)$ time. With lists, it is $O(n)$.

The map is called association lists in Haskell. So, lookup can be used for the apply function above.

```
 Main> :t lookup
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```

Q.What should update do on the following ? $update\ (5, "xyzzy")(FF\ [(5, "x"), (5, "y")])$
A. Then the map is not functional. If we do not consider the functional aspect, we can update the first occurrence - this is consistent with search where you return the first occurrence.

One of the axioms for update is
$apply\ (update\ (x, y)\ (update\ (x, z)\ f)) = apply\ (update\ (x, y)\ f)$

2 List Induction

Remember that

$$Nat ::= Zero \mid Succ\ Nat$$

and the induction principle for Nat is:

$$((P(zero) \wedge \forall k : nat.P(k)) \Rightarrow P(Succk)) \Rightarrow \forall m : Nat.P(m)$$

Lists are defined as the following datatype

$$List\ a := [] \mid Cons\ a\ (List\ a)$$

The induction principle for list is

$$(P([]) \wedge (\forall xs : (List\ a).P(xs) \Rightarrow (\forall x : a.P(x : xs)))) \Rightarrow \forall xs : (Lista).P(xs)$$

This is called "Structural induction" because it follows the "structure" of the datatype.

$$data\ Tree\ a = Leaf\ a \mid Node\ (Tree\ a)\ a\ (Tree\ a)$$

Some examples:

- (Leaf 5)
- (Node (Leaf 5) 4 (Leaf 3))

The induction principle for tree is:

$$((\forall x : a.P(Leaf\ a) \wedge \forall t1, t2 : (Tree\ a), (P(t1) \wedge P(t2))) \Rightarrow \forall x : a.P(Node\ t1\ x\ t2)) \Rightarrow \forall t : Tree\ a.P(t)$$

The point is if you define a datatype you get this powerful principle for your data type which helps you in reasoning about the structure. There are some extensions in Haskell which can allow you to refine the structure. For example for the finite maps we can define the functionality condition as:

```
data FinFun a b = FF [(a,b)]
   condition (FF m) => (functional m)
```

Note: the syntax used might be different in Haskell.
Coming back to list induction, the book says it as follows:

1. $P([])$ - base case
2. $P(x : xs)$ - assume P(xs) and show P(x:xs) for arbitrary x.
3. $P(\perp)$ - add this to show that P holds for partial lists. For example, [1..].

$reverse\ [] = []$
$reverse\ (x : xs) = reverse\ xs +\!+ [x]$

$[] +\!+ ys = ys$
$(x : xs) +\!+ ys = x : (xs +\!+ ys)$

$length\ [] = 0$
$length\ (x : xs) = 1 + length\ xs$

From now on we will use | to denote the list length.

Theorem 1. $\forall xs, ys : [a].\ | xs + +ys | = | xs | + | ys |$

Proof. Choose arbitrary ys and do induction on xs.
$P(xs) \overset{\text{def}}{=} | xs{+}{+}ys | = | xs | + | ys |$

case []: we must show $| []{+}{+}ys | = | [] | + | ys |$. On the L.H.S. $| []{+}{+}ys | = | ys |$. On the R.H.S. $| xs | + | ys | = 0 + | ys | = | ys |$ Sp the base case holds

case $x : xs$: assume P(xs) i.e. I.H. $| xs{+}{+}ys | = | xs | + | ys |$
Show $P(x : xs)$ i.e. show
$| x : xs +\!+ ys | = | x : xs | + | ys |$
Left side :
$| x : xs +\!+ ys | \overset{<<+>>}{=} | x : (xs +\!+ ys) | \overset{<<\text{length}>>}{=} | x | + | xs + +ys | = 1 + (| xs | + | ys |)$
Right side :
$| x : xs | + | ys | = (1 + | xs |) + | ys |$.
Now since + is associative, the induction step holds.

□

Q: which one should you choose to do induction ?
A: You can do it on both but look at the def. of append. It is defined by induction on the first argument. Then there is another issue. we don't need forall on the ys.

Strategy 1. *Strategy: Arrange quantifiers so that the variable you want to do induction on is innermost. For example,*
$\forall x, \forall y. P(x,y) \Leftrightarrow \forall y \forall x. P(x,y)$
Consider $\forall xs, ys : [a]. Q(xs, ys)$
Then - if you do induction on xs - the property proved by induction is
$P(xs) \stackrel{\text{def}}{=} \forall ys : [a]. Q(xs, ys)$

Often, we do not need this generality.

But, suppose we want to prove
$\forall ys, xs : [a]. Q(xs, ys)$

- choose arb. ys and prove
$\forall xs : [a]. Q(xs, ys)$

by induction, then
$P(xs) = Q(xs, ys)$

Theorem 2. $\forall xs, ys : [a]. rev(xs \mathbin{++} ys) = (rev\ ys) \mathbin{++} (rev\ xs)$

Remark:The above theorem will require that append is associative.