Lecture 30 January 2018

Datatype -- Tree

A user-defined recursive datatype.

> The **recursive** structure allows easy recursion and induction proof.

- We want to be able to represent **empty trees** (as base case).

```
    1
  2 _
_ _    # empty sub-trees - we have to somehow represent those
```

- We want to represent "Nodes" that contain a **left** sub-tree, an integer, and a **right** sub-tree.
- For the moment, that's it.

How do we tell ML about all this?

Introduction of Datatype

```
datatype tree = Empty | Node of tree * int * tree

(* A constant constructor: it constructs a value called "Empty" with type "tree". *)
(* It is convention to capitalize constant constructor value. *)
(* "of" says we want some more stuff for the constructor Node *)

(* If we re-define the datatype of the same name, the old definition is shadowed. *)
(* No mutation after a "Node" is constructed. *)

(* "Node" is also a function -- Node : tree * int * tree -> tree *)
```

Some example values of this new datatype:

```
Empty : tree
val t1 : tree = Node(Node(Empty, 2, Empty), 1, Empty)
val t2 : tree = Node(Empty, 3, Node(Empty, 4, Empty))
val T : tree = Node(t1, 5, t2)
```

Suppose we want "mutation" at run-time:

```
fun change_to_6 (Node(left, _, right) : tree) : tree = Node(left, 6, right)
  | change_to_6 Empty = Empty
```

> Note: There is no actual mutation. The new defined Node shadows the old Node.

More with Trees

Consider a function to define the height of a tree:

```
(* height : tree -> int *)
fun height (Empty : tree) : int = 0
  | height (Node(left, _, right)) = 1 + Int.max(height left, height right)

val 3 = height T
```

> Note: `Int` is a structure, and `Int.max` is a built-in function.

Question: could we run into something cyclical?

> By an inductive proof, we can show there could be **no** cycles. Most likely in functional programming, we build something from base case and a well-defined constructor. Then we can prove totality. If you somehow want to have cycles: 1) there will be reference cells, 2) there will be "stream" that can somehow mimic cyclical behaviors, 3) as for graphs, we represent the data structure with functions, and that normally will give us cycles.

Theorem. `height` is total. (Essentially, we have to prove it terminates.)

Proof. We prove this by structural induction on the datatype.

Base Case. `T = Empty`.

Want-to-Show. `height(Empty) ===> (some value)`.

```
    height(Empty)                [by clause 1 of height]
===> 0 (* which is a value *)
```

Induction Step. `T <> Empty` by assumption, so `T = Node(left, x, right)` for some integer `x` and some trees `left` and `right`.

Induction Hypothesis. `height left ===> (some value) h_l` and `height right ===> (some value) h_r`.

> Be clear about what you're doing. Be pedantic if necessary.

Want-to-Show. `height T ===> (some value)`.

```
     height T
==> height Node(left, x, right)          [* by referential transparency]
==> 1 + Int.max(height left, height right)    [by clause 2 of height]
==> 1 + Int.max(h_l, h_r)                [by IH, and h_l, h_r are some values]
==> 1 + h_max                            [h_max of some value, assuming Int.max is total]
==> (some value)                         [assuming + is total]
```

> Note: When we do these proofs, we automatically assume the possible values **typecheck**.
>
> Note: In this case, we want a **reduction** instead of an **equivalence**. An argument for **equivalence** is in fact weaker.

Question: Do we only use structural induction to prove totality?

> No. We can use structural induction to prove many things. It's powerful!

> Note: **Reduction** automatically gives you **extensional equivalence**. But you have to be careful. Especially, if you only have **equivalence** in the **IH**, then there could be problems.

Another Type of Trees

Suppose we only want data be held on leaves:

```
  datatype tree = Leaf of int | Node of tree * tree
```

Consider a function to convert this type of trees into a list (in-order traversal):

```
fun flatten (Leaf x : tree) : int list = [x]
  | flatten (Node(left, right)) = flatten left @ flatten right
```

Question: What is the time-complexity of this function definition?

> Linear-time if tree is weighted to the right, and quadratic-time if weighted to the left.

Can we do better? Can we ensure linear-time complexity?

> Yes! We can use tail recursion with an accumulator.

```
(* flatten' : tree * int list -> int list
 * REQUIRES: true
 * ENSURES: flatten'(T, acc) === flatten(T) @ acc
 *)
fun flatten' (Leaf x : tree, acc : int list) : int list = x::acc  (* consistent with spec *)
  | flatten' (Node(left, right), acc) = flatten' (left, flatten' (right, acc))
```

> Note: This new function will ensure linear time.
>
> Note: We do **not** have Empty in this type of tree! Otherwise, the complexity is more complicated.

Computational Trees

How is this useful?

> Consider an operator-operand tree.

```
    *
  max (4)
(3) (7)
```

> Consider a computational tree.

```
datatype optree = Val of int
                | Op of optree * (int * int -> int) * optree
val computation : optree = Op(Op(Val 3, Int.max, Val 7), op*, Val 4)
(* For op*, we can also put a lambda expression. *)
```

> Note: `*` is basically a syntactic sugar for `op *`.

Consider a function to evaluate a computational tree:

```
(* eval : optree -> int
 * REQUIRES: all function in T total
 * ENSURES: eval T gives the expected computational result
 *)
fun eval (Val x : optree) : int = x
  | eval (Op(left, func, right)) = func (eval left, eval right)
```