Lecture 25 January 2018

This lecture talks about recursion, evaluation trace, and proving extensional equivalence.
This lecture also talks about some list operations.

The most important topics are **tail recursion** and **list operations**.

Recursion, Evaluation Trace, Function Specs, and Tail Recursion

Consider function `length` :

```
(* length : int list -> int *)
fun length ([] : int list) : int = 0
  | length (_::L) = 1 + length L
```

Evaluation Trace

Consider when we evaluate the expression `length ([1, 2, 3])` :

1. Evaluate function value `length`
2. `[1, 2, 3]` is already a value (It is a syntactic sugar of `1::2::3::nil`)
3. Pattern match the second clause of function `length`
4. Evaluate the corresponding function body in the function closure with the formal parameter binding

```
    length ([1, 2, 3])
==> 1 + length ([2, 3])
==> 1 + (1 + length ([3]))
==> 1 + (1 + (1 + length nil)) # matching the first clause of function length
==> 1 + (1 + (1 + 0))
==> ...
```

Tail Recursion and Accumulator

> Note: Space increases in the process of recursive calls, and space decreases when **pending computations** are one-by-one evaluated. We can, however, do (in this case) the additions instantly upon each call. We can use an **accumulator argument** to hold all the computation values.

```
(* tlength : int list * int -> int *)
fun tlength ([] : int list, acc : int) : int = acc (* base case gives the accumulated computations *)
  | tlength (_::L) = tlength (L, acc + 1)          (* do the addition right away *)

(* length' : int list -> int
 * REQUIRES: true
 * ENSURES: length' === length // Extensionally equivalent to the original length function
 *)
fun length' (L : int list) : int = tlength (L, 0)
```

Consider the evaluation trace of the expression `length' ([1, 2, 3])`:

```
    length' [1, 2, 3]
==> tlength ([1, 2, 3], 0)
==> tlength ([2, 3], 1)
==> tlength ([3], 2)
==> tlength (nil, 3)
==> 3
```

Now the space-time relation is basically constant.

> Note: This is in fact a **tail recursion**. It may sometimes save space, or sometimes save time. It may sometimes save nothing but is simply easier to write.

Function Specs

Consider the spec for function `tlength`:

```
REQUIRES: true
ENSURES: tlength (L, acc) === length L + acc
```

Proving a Relative Correctness

We'll prove that two functions have the same behavior (extensional equivalence).

> Theorem: For all values `L : int list` and `acc : int`, `tlength (L, acc) === length L + acc`.

```
Proof. By structural induction on L:

Base Case. L = nil
  N.T.S. For all acc, tlength (nil, acc) === length nil + acc

Note. If e1 ==> v and e2 ==> v, then e1 === e2
      If e1 ==> e and e2 ==> e, then e1 === e2
      In fact, if e1 ==> e, e1 === e

  Showing:
    tlength (nil, acc)
==> acc                 (* first clause of tlength *)
    length nil + acc
==> 0 + acc             (* first clause of length *)
==> acc

Induction Step. L != nil, so L = x::xs for some x : int and some xs : int list

Induction Hypothesis: For all acc', for the given xs : int list, tlength (xs, acc') === length xs + acc'
  N.T.S. tlength (x::xs, acc) === length xs + acc [For all acc]

  Showing:
    tlength (x::xs, acc)
=== tlength (xs, 1 + acc) (* second step of tlength *)
=== length xs + (1 + acc) (* by IH, where acc' = 1 + acc [+ is total => (1 + acc) is a VALUE] *)
=== 1 + length xs + acc   (* by Math [including assuming correct implementations of arithmetics] *)
=== length (x::xs) + acc  (* reverse of second clause of length *)
```

> Important: In the proving of the base case, we use reductions on both sides and show that the reduced expression/value is equivalent. In the proving of the induction step, we use a chain of equivalence (NOT reductions).

> Note: We are assuming the totality and correctness of the basic arithmetics.

List Operations

The Append Function

Consider the function `append` that joins two lists:

```
(* append : int list * int list -> int list *)
fun append ([] : int list, L2 : int list) : int list = L2
  | append (x::xs, L2) = x::(append (xs, L2))
```

The evaluation trace of the expression `append ([2, 3], [~5, 7, 10])` :

```
    append ([2, 3], [~5, 7, 10])
==> 2::(append ([3], [~5, 7, 10]))
==> 2::(3::(append (nil, [~5, 7, 10])))
==> 2::(3::([~5, 7, 10]))
==> 2::([3, ~5, 7, 10])
==> [2, 3, ~5, 7, 10]
```

> Note: There is no mutation of lists here. We are constructing a new list with cons.

The complexity of function `append (L1, L2)` is $O(L1)$.

In ML, `append` is predefined as a right-associative infix operator `@`.

The Reversal Function

```
(* rev : int list -> int list *)
(* rev [2, 3, 4] ==> [4, 3, 2] *)
fun rev ([] : int list) : int list = []
  | rev (x::xs) = (rev xs) @ [x]
```

The complexity of this function is $O(L^2)$.

Tail Recursion of Reversal Function

We can again use a *tail recursion* to save, this time, time.

```
(* trev : int list * int list -> int list
 * REQUIRES: true
 * ENSURES: [extensional equivalence]
 *)
fun trev ([] : int list, acc : int list) : int list = acc
  | trev (x::xs, acc) = trev (xs, x::acc)

(* wrapper for user *)
fun reverse ([] : int list) : int list = trev (L, nil)
```

By using tail recursion, this implementation of reversal runs in linear time.