

COMP302: Programming Languages and Paradigms

Prof. Brigitte Pientka (Sec 01)

`bpientka@cs.mcgill.ca`

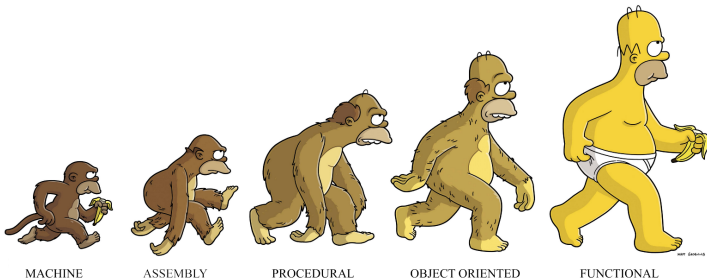
Francisco Ferreira (Sec 02)

`fferre8@cs.mcgill.ca`

School of Computer Science

McGill University

Week 3-1, Fall 2017



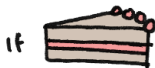
Functional Tidbit: Cake!

How do I prove that all slices of cake are tasty using structural induction?

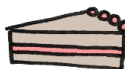
Step 1. Define a set of cake slices recursively.



is cake.



and



are cake, then both of them
put together is still cake :



Functional Tidbit: Cake is tasty!

Step 2. Prove that a single piece of cake is tasty.



Functional Tidbit: Cake is tasty!

Step 2. Prove that a single piece of cake is tasty.



Step 3. Use the recursive definition of the set to prove that all slices are tasty.

Functional Tidbit: Cake is tasty!

Step 2. Prove that a single piece of cake is tasty.



Step 3. Use the recursive definition of the set to prove that all slices are tasty.

Step 4. Conclude all slices of cake are tasty.



More on Structural Induction

Two programs: Do they compute the same value?

Program A (naive)

```
1 (* rev : 'a list -> 'a list *)
2 let rec rev l = match l with
3   | []      -> []
4   | x::l    -> (rev l) @ [x]
```

Program B (tail-recursive)

```
1 (* rev' : 'a list -> 'a list *)
2 let rev' l =
3   (* rev_tr : 'a list -> 'a list -> 'a list *)
4   let rec rev_tr l acc = match l with
5     | []      -> acc
6     | h::t    -> rev_tr t (h::acc)
7   in
8   rev_tr l []
```

Two programs: Do they compute the same value?

Program A (naive)

```
1 (* rev : 'a list -> 'a list *)
2 let rec rev l = match l with
3   | []      -> []
4   | x::l    -> (rev l) @ [x]
```

Theorem: For all lists l . $\text{rev } l = \text{rev}' l$

Program B (tail-recursive)

```
1 (* rev' : 'a list -> 'a list *)
2 let rev' l =
3   (* rev_tr : 'a list -> 'a list -> 'a list *)
4   let rec rev_tr l acc = match l with
5     | []      -> acc
6     | h::t    -> rev_tr t (h::acc)
7   in
8   rev_tr l []
```


What to prove? – Finding the invariant!

Program A (naive)

```
1 (* rev : 'a list -> 'a list *)
2 let rec rev l = match l with
3   | []      -> []
4   | x::l    -> (rev l) @ [x]
```

What is the relationship between `l`, `acc` and `rev_tr l acc`?

Program B (tail-recursive)

```
1 (* rev' : 'a list -> 'a list *)
2 let rev' l =
3   (* rev_tr : 'a list -> 'a list -> 'a list *)
4   let rec rev_tr l acc = match l with
5     | []      -> acc
6     | h::t    -> rev_tr t (h::acc)
7   in
8     rev_tr l []
```

What to prove? – Finding the invariant!

Program A (naive)

```
1 (* rev : 'a list -> 'a list *)
2 let rec rev l = match l with
3   | []      -> []
4   | x::l    -> (rev l) @ [x]
```

For all l , acc , $(rev\ l) @ acc \Downarrow v$ and $rev_tr\ l\ acc \Downarrow v$

Program B (tail-recursive)

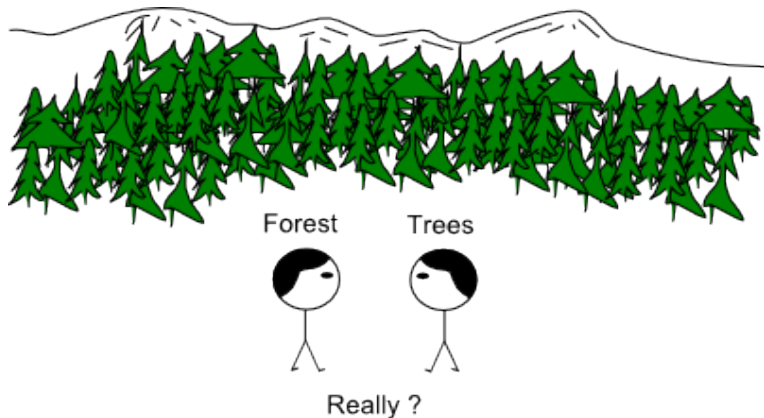
```
1 (* rev' : 'a list -> 'a list *)
2 let rev' l =
3   (* rev_tr : 'a list -> 'a list -> 'a list *)
4   let rec rev_tr l acc = match l with
5     | []      -> acc
6     | h::t    -> rev_tr t (h::acc)
7   in
8   rev_tr l []
```

Theorem: For all l , acc ,
 $length\ (rev_tr\ l\ acc) \Downarrow v$ and $length\ l + length\ acc \Downarrow v$.

We often simply write instead:

For all l , acc ,
 $length\ (rev_tr\ l\ acc) = length\ l + length\ acc$

Can't see the forest for the trees



Inductive definition of a binary tree

- The empty binary tree `Empty` is a binary tree
- If `l` and `r` are binary trees and `v` is a value of type `'a` then `Node(v, l, r)` is a binary tree.
- Nothing else is a binary tree.

Inductive definition of a binary tree

- The empty binary tree `Empty` is a binary tree
- If `l` and `r` are binary trees and `v` is a value of type `'a` then `Node(v, l, r)` is a binary tree.
- Nothing else is a binary tree.

How can we define a data type that describes binary trees?

Forest and trees

Example of a mutual recursive data type definition:

```
1 type 'a forest = Forest of ('a tree) list
2 and 'a tree = Empty | Node of 'a * 'a forest
```

Remember the slice of cake?

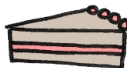
Step 1. Define a set of cake slices recursively.



is cake.



and



are cake, then both of them
put together is still cake :



Give an OCaml data type definition for cake!