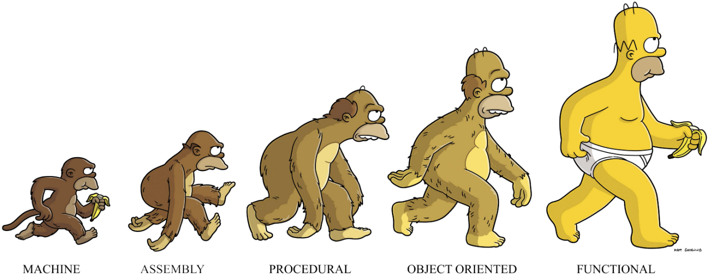
COMP302: Programming Languages and Paradigms

Prof. Brigitte Pientka (Sec 01)
bpientka@cs.mcgill.ca

Francisco Ferreira (Sec 02)
fferre8@cs.mcgill.ca

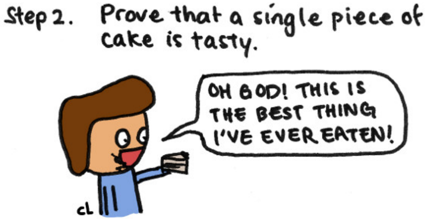
School of Computer Science
McGill University

Week 3-1, Fall 2017



Functional Tidbit: Cake!

Functional Tidbit: Cake is tasty!



Functional Tidbit: Cake is tasty!

Functional Tidbit: Cake is tasty!

Topic

More on Structural Induction

Two programs: Do they compute the same value?

Program A (naive)

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

Program B (tail-recursive)

```
(* rev' : 'a list -> 'a list *)
let rev' l =
  (* rev_tr : 'a list -> 'a list -> 'a list *)
  let rec rev_tr l acc = match l with
    | []   -> acc
    | h::t -> rev_tr t (h::acc)
  in
    rev_tr l []
```

Two programs: Do they compute the same value?

Program A (naive)

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

Theorem: For all lists `l`. `rev l` = `rev' l`

Program B (tail-recursive)

```
(* rev' : 'a list -> 'a list *)
let rev' l =
  (* rev_tr : 'a list -> 'a list -> 'a list *)
  let rec rev_tr l acc = match l with
    | []    -> acc
    | h::t -> rev_tr t (h::acc)
  in
    rev_tr l []
```

What to prove? – Finding the invariant!

Program A (naive)

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

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

Program B (tail-recursive)

```
(* rev' : 'a list -> 'a list *)
let rev' l =
  (* rev_tr : 'a list -> 'a list -> 'a list *)
  let rec rev_tr l acc = match l with
    | []    -> acc
    | h::t -> rev_tr t (h::acc)
  in
    rev_tr l []

```

What to prove? – Finding the invariant!

Program A (naive)

```
(* rev : 'a list -> 'a list *)
let rec rev l = match l with
  | []    -> []
  | 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)

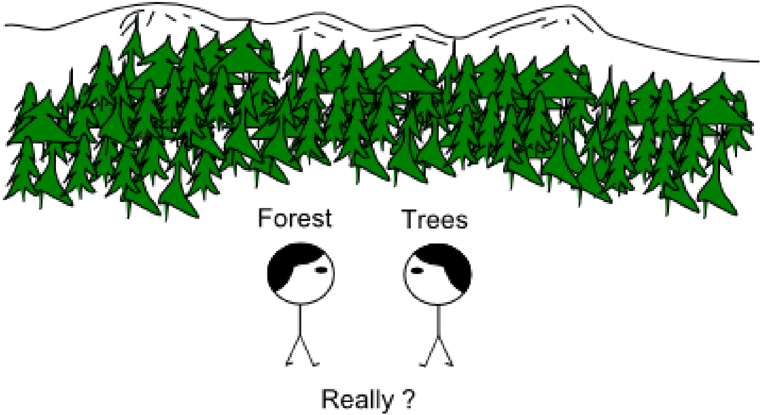
```
(* rev' : 'a list -> 'a list *)
let rev' l =
  (* rev_tr : 'a list -> 'a list -> 'a list *)
  let rec rev_tr l acc = match l with
    | []    -> acc
    | h::t -> rev_tr t (h::acc)
  in
    rev_tr l []
```

Exercise

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:

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

Remember the slice of cake?

Give an OCaml data type definition for cake!