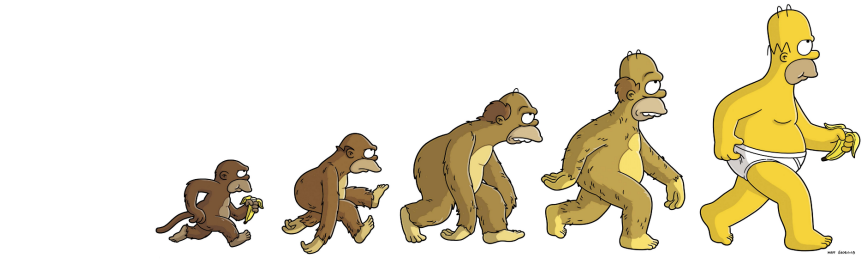
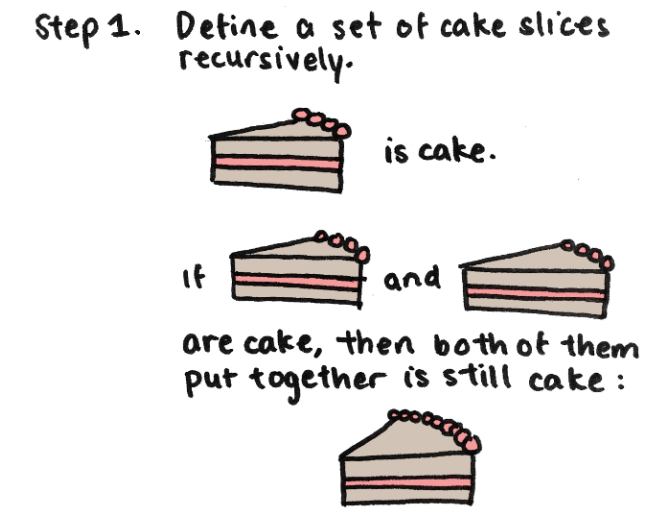


COMP 302: Programming Languages and Paradigms

Prof. B. Pientka, McGill University McGill

Week 3 (Part 1): Induction

Rember ...



Our goal this week: Prove that indeed all cakes are tasty!

How? – Induction!

Proofs are all around you!

"On theories such as these, rely, we cannot. Proof, we need. Proof!"

Maybe someone still has memories of McGill?

Inductive Definitions and Proofs by Induction

Recall: Inductive definition of a list

- The empty list `[]` is a list of type `'a list`.
- If `x` is an element of type `'a` and `xs` is a list of type `'a list` then `x::xs` is a list of type `'a list`.
- Nothing else is a list of type `'a list`.

In OCaml, lists are defined as a recursive data type.

```
1 type 'a list =
2   | []
3   | (::) of 'a * 'a list
4
```

How to prove properties about lists?

Analyze their structure!

Recipe for proving properties about lists ...

To prove that a property `P` holds for a list `l`

Base Case: `l = []`
Show `P([])` holds

Step Case: `l = x::xs`

IH	`P(xs)`	Assume the property `P` holds for lists smaller than `l`.
Show	`P(x::xs)` holds	Show the property `P` holds for the list `x::xs`.

Example

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

```
(* length: 'a list -> int *)
let rec length l = match l with
  | [] -> 0
  | _ :: xs -> 1 + length xs
```

Theorem. *For all* `l`. `length(rev l) = length l`

But what *is* equality???

- When we say `length(rev l)` $=$ `length l`, we are *not* saying that these are *the same* program!

- Technically, we should write:
  "There exists `v` and `w` such that `length(rev l)` $\Downarrow$ `v` and `length l` $\Downarrow$ `w` and `v` $=$ `w`"

- The notation $\boxed{e \Downarrow v}$ means "e evaluates to the value v".

- We will however write $\boxed{e = e'}$ as an abuse of notation to say "e and e' evaluate to the same value".

Let's revisit the example and prove the theorem!

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

```
(* length: 'a list -> int *)
let rec length l = match l with
  | [] -> 0
  | _ :: xs -> 1 + length xs
```

Theorem. *For all* `l. length(rev l) = length l`

proof by structural induction l

- base case: l = []
  length (rev []) → length [] → 0
  length [] → 0

- step case l = x::xs
  IH: length (rev xs) = length xs
  length (rev x::xs)
  → length (rev xs @ [x])
  → length (rev xs) + length([x]) by lemma: length l1 + length l2 = length (l1 @ l2)
  → length (xs) + 1
  → 1 + length(xs) by commutativity of +
  → length (x::xs)

Another example: Do they compute the same value?

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

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

Theorem. *For all lists* `l`. `rev l = rev_tr l []`

IH : rev xs = rev_tr xs []

RHS : rev_tr (x :: xs) []

⇒ rev_tr xs (x :: []) (✗) Wrong!

[x]

Use the next theorem with acc = []

(rev l) @ [] = rev_tr l []

⇒ rev l = rev_tr l []

Generalize the statement

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

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

Theorem. *For all* `l`, `acc`, *we have that* `(rev l) @ acc = rev_tr l acc`

By structural induction on l.

- base case $l = []$

  RHS: rev_tr [] acc → acc

  LHS: (rev []) @ acc → [] @ acc → acc

- step case $l = x :: xs$

  IH: for all acc, we have that (rev xs) @ acc = rev_tr xs acc

  RHS: rev_tr (x::xs) acc
  → rev_tr xs (x:: acc)
  → (rev xs) @ (x:: acc)

  LHS: rev (x::xs) @ acc
  → ((rev xs) @ [x]) @ acc
  → (rev xs) @ ([x] @ acc)
  → (rev xs) @ (x :: acc)

Take Away

- Inductive definitions such as recursive data types directly give us a way of thinking recursively.
- Inductive definitions give us induction principles
- Recursion and induction go hand in hand.

IH: ∀ acc, length (rev_tl t acc) = length t + length acc

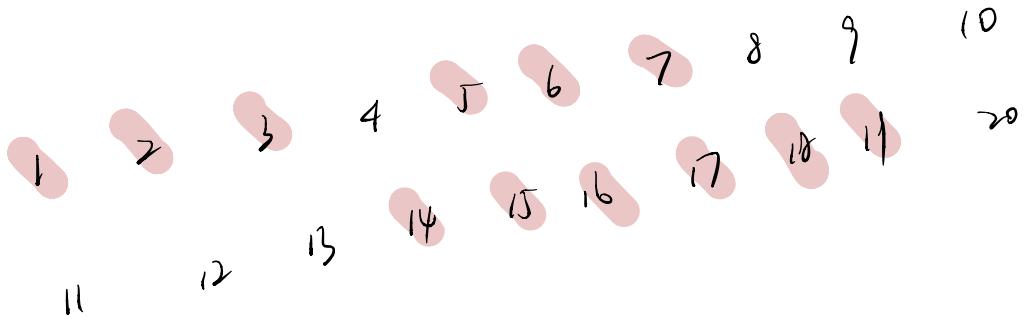
length (rev_tl (h::t) acc) = length (h::t) + length acc

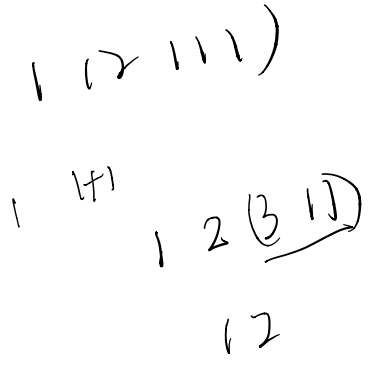
↓

(rev_tl t (h::acc))

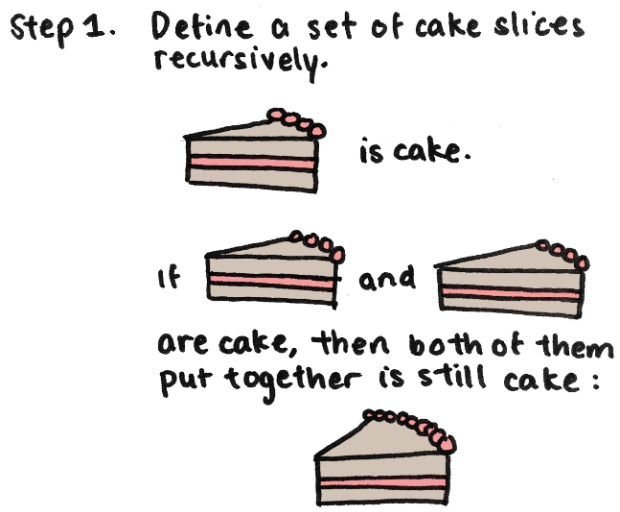
IH ↓

length t + length (h::acc)





Proving that all cakes are tasty! Rember ...



This directly gives rise to the following OCaml data type:

```
type ingredient = Chocolate | Almonds | Orange

type cake = Slice of ingredient
|  Cake of cake * cake

let tasty_ingredient i = match i with
| Chocoloate | Almonds | Orange -> true

let rec is_tasty c = match c with
| Slice _ -> true
| Cake (s1, s2) -> is_tasty s1 && is_tasty s2
```

A proof .... (sort of)

A better proof ....

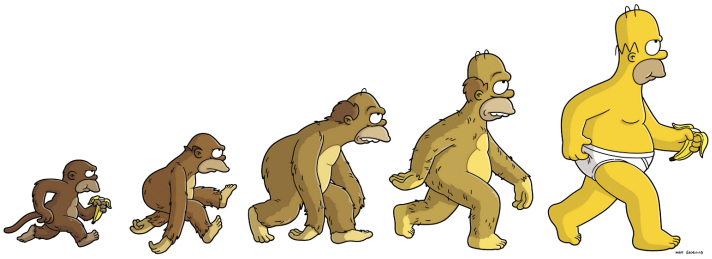
```
type ingredient = Chocolate | Almonds | Orange

type cake = Slice of ingredient
|  Cake of cake * cake

let tasty_ingredient i = match i with
| Chocoloate | Almonds | Orange -> true

let rec is_tasty c = match c with
| Slice _ -> true
| Cake (s1, s2) -> is_tasty s1 && is_tasty s2
```

Theorem. *For all cakes* c, $is\_tasty\ c = true$.



COMP 302: Programming Languages and Paradigms

Prof. B. Pientka, McGill University

Week 3 (Part 2): Induction

Programming with and Reasoning about Binary Trees

Inductive definition of a binary tree

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

How to define a recursive data type for trees in OCaml?

```
1 type 'a tree =
2   | Empty
3   | Node of 'a tree * 'a * 'a tree
```

How to reason inductively about trees?

Analyze their structure!

The recipe

To prove a property `P(t)` holds for a binary tree `t : 'a tree`

Base Case:	Suppose `t = Empty`	
Show	`P(Empty)` holds	
Step Case:	Suppose `t = Node(l, x, r)`	
IH	`P(l)`	Assume the property `P` holds
IH	`P(r)`	for trees smaller than `t`.
Show	`P(Node(l, x, r))` holds	Show the property `P` holds for the tree `Node(l, x, r)`.

Let's prove something

```
(* insert:                                    *)
let rec insert (k, v) t = match t with
  | Empty -> Node (Empty, (k, v), Empty)
  | Node (l, (k', v'), r) ->
    if k = k' then
      Node (l, (k, v), r)
    else
     if k < k' then
      Node (insert (k, v) l, (k', v'), r)
     else
      Node (l, (k', v'), insert (k, v) r)
```

```
(* lookup:                                    *)
let rec lookup k t = match t with
  | Empty -> None
  | Node (l, (k', v), r) ->
    if k = k' then
      Some v
    else
      if k < k' then
       lookup k l
      else lookup k r
```

Theorem: For any tree `t`, key `k`, and data `v`, `lookup k (insert (k, v) t)` = `Some v`

By structural induction on tree t

- base: t = empty

  lookup k (insert (k,v) empty)

  → lookup k (Node (Empty, (k,v), Empty))

  → Some v

- step: t = Node (l, (k',v'), r)

  IH 1: lookup k (insert (k,v) l) = Some v

  IH 2: lookup k (insert (k,v) r) = Some v

  subcase: k = k'

  lookup k (insert (k,v) (Node (l, (k',v'), r))

  → lookup k (Node ((l, (k,v), r))

  → Some v

  subcase: k > k'

  lookup k (insert (k,v) (Node (l, (k',v'), r))

  → lookup k (Node (l, (k',v'), insert (k,v) r))

  → lookup k (insert (k,v) r)

  → Some v

Let's have some delicious cake!

Rember ...

This directly gives rise to the following OCaml data type:

```
type ingredient = Chocolate | Almonds | Orange

type cake = Slice of ingredient
|  Cake of cake * cake

let tasty_ingredient i = match i with
| Chocoloate | Almonds | Orange -> true

let rec is_tasty c = match c with
| Slice _ -> true
| Cake (s1, s2) -> is_tasty s1 && is_tasty s2
```

Prove that indeed all cakes are tasty!

How? – Induction!

Proving that all cakes are tasty!

A proof .... (sort of)

A better proof ....

```
type ingredient = Chocolate | Almonds | Orange

type cake = Slice of ingredient
|  Cake of cake * cake

let tasty_ingredient i = match i with
| Chocoloate | Almonds | Orange -> true

let rec is_tasty c = match c with
| Slice _ -> true
| Cake (s1, s2) -> is_tasty s1 && is_tasty s2
```

Theorem. *For all cakes* `c`, `is_tasty c` = `true`.

By structural induction on c.

- base : c = Slice i

  is_tasty (Slice i) → true

- step : c = Cake (c1, c2)

  IH 1 : is_tasty c1 = true

  IH 2 : is_tasty c2 = true

  is_tasty (Cake (c1, c2))

  → is_tasty c1 && is_tasty c2

  → true && true

  → true

Take Away

- Inductive definitions such as recursive data types directly give us a way of thinking recursively.
- Inductive definitions give us induction principles
- Recursion and induction go hand in hand.