Induction proofs

October 18th

UNIVERSITY OF MINNESOTA
Driven to Discover®

Example of a recursive proof

- let rec append lst1 lst2 = match lst1 with
  - | [] -> lst2
  - | (h::tl) -> h::append tl lst2
- We prove (for this definition) that
  append (append a b) c = append a (append b c)
  by induction on a
- case a = []
- append (append a b) c
  = …

Example of a recursive proof

- let rec append lst1 lst2 = match lst1 with
  - | [] -> lst2
  - | (h::tl) -> h::append tl lst2
- We prove (for this definition) that
  append (append a b) c = append a (append b c)
  by induction on a
- case a = []
- append (append a b) c
  = {case}
  …

Example of a recursive proof

- let rec append lst1 lst2 = match lst1 with
  - | [] -> lst2
  - | (h::tl) -> h::append tl lst2
- We prove (for this definition) that
  append (append a b) c = append a (append b c)
  by induction on a
- case a = []
- append (append a b) c
  = {case}
  append (append [] b) c
  = …

Example of a recursive proof

- let rec append lst1 lst2 = match lst1 with
  - | [] -> lst2
  - | (h::tl) -> h::append tl lst2
- We prove (for this definition) that
  append (append a b) c = append a (append b c)
  by induction on a
- case a = []
- append (append a b) c
  = {case}
  append (append [] b) c
  = {definition append}
  append (match [] with | [] -> b | (h::tl) -> h::append tl b) c

Example of a recursive proof

- We prove (for this definition) that
  append (append a b) c = append a (append b c)
  by induction on a
- case a = []
- append (append a b) c
  = {case}
  append (append [] b) c
  = {definition append}
  append (match [] with | [] -> b | (h::tl) -> h::append tl b) c
  = …

Example of a recursive proof

- We prove (for this definition) that
  append (append a b) c = append a (append b c)
  by induction on a
- case a = []
- append (append a b) c
  = {case}
  append (append [] b) c
  = {definition append}
  append (match [] with | [] -> b | (h::tl) -> h::append tl b) c
  = {matching pattern}
  append b c
  = {…}

Example of a recursive proof

- We prove (for this definition) that
  append (append a b) c = append a (append b c)
  by induction on a
- case a = []
- append (append a b) c
  = {case}
  append (append [] b) c
  = {definition append}
  append (match [] with | [] -> b | (h::tl) -> h::append tl b) c
  = {matching pattern}
  append b c
  = {…}

Switch strategies: prove bottom to top…

Example of a recursive proof

- append (append a b) c
  = {case}
  append (append [] b) c
  = {definition append}
  append (match [] with | [] -> b | (h::tl) -> h::append tl b) c
  = {matching pattern}
  append b c
  = {matching pattern}
  match [] with | [] -> append b c
  | h::tl -> h::append tl (append b c)
  = {definition append}
  append [] (append b c)
  = {case}
  append a (append b c)

Back to the overall proof…

- let rec append lst1 lst2 = match lst1 with
  - | [] -> lst2
  - | (h::tl) -> h::append tl lst2
- We prove (for this definition) that
  append (append a b) c = append a (append b c)
  by induction on a
- case a = [] on previous slides
- Our second proof will have two extra assumptions!
  - ‘case’: a = h :: tl
  - ‘Inductive hypothesis’ (IH):
    append (append tl b) c = append tl (append b c)

Inductive case…

- ‘case’: a = h :: tl
- IH: append (append tl b) c = append tl (append b c)

- append (append a b) c
  = {case}
  append (append (h :: tl) b) c
  = …

Inductive case…

- ‘case’: a = h :: tl
- IH: append (append tl b) c = append tl (append b c)

- append (append a b) c
  = {case}
  append (append (h :: tl) b) c
  = {append definition}
  append (match h :: tl with [] -> b | h :: tl -> h :: append tl b) c
  =

Note: the ‘h’ and ‘tl’ match up perfectly here,
we won’t always be so lucky…

Inductive case…

- ‘case’: a = h :: tl
- IH: append (append tl b) c = append tl (append b c)

- append (append a b) c
  = {case}
  append (append (h :: tl) b) c
  = {append definition}
  append (match h :: tl with [] -> b | h :: tl -> h :: append tl b) c
  = …

Inductive case…

- ‘case’: a = h :: tl
- IH: append (append tl b) c = append tl (append b c)

- append (append a b) c
  = {case}
  append (append (h :: tl) b) c
  = {append definition}
  append (match h :: tl with [] -> b | h :: tl -> h :: append tl b) c
  = {match fits pattern}
  …

Inductive case…

- ‘case’: a = h :: tl
- IH: append (append tl b) c = append tl (append b c)

- append (append a b) c
  = {case}
  append (append (h :: tl) b) c
  = {append definition}
  append (match h :: tl with [] -> b | h :: tl -> h :: append tl b) c
  = {match fits pattern}
  append (h :: append tl b) c
  = {…}

Inductive case…

- ‘case’: a = h :: tl
- IH: append (append tl b) c = append tl (append b c)

- append (append a b) c
  = {case}
  append (append (h :: tl) b) c
  = {append definition}
  append (match h :: tl with [] -> b | h :: tl -> h :: append tl b) c
  = {match fits pattern}
  append (h :: append tl b) c
  = {append definition}
  match (h :: append tl b) with [] -> c | h :: tl2 -> h :: append tl2 c

Note: I’ve renamed tl to tl2, so we don’t get too confused
(the h still lines up nicely)

Inductive case…

- ‘case’: a = h :: tl
- IH: append (append tl b) c = append tl (append b c)
- append (append a b) c
  = {case}
  append (append (h :: tl) b) c
  = {append definition}
  append (match h :: tl with [] -> b | h :: tl -> h :: append tl b) c
  = {match fits pattern}
  append (h :: append tl b) c
  = {append definition}
  match (h :: append tl b) with [] -> c | h :: tl2 -> h :: append tl2 c
  = {…}

Inductive case…

- ‘case’: a = h :: tl
- IH: append (append tl b) c = append tl (append b c)
- append (append a b) c
  = {case}
  append (append (h :: tl) b) c
  = {append definition}
  append (match h :: tl with [] -> b | h :: tl -> h :: append tl b) c
  = {match fits pattern}
  append (h :: append tl b) c
  = {append definition}
  match (h :: append tl b) with [] -> c | h :: tl2 -> h :: append tl2 c
  = {match fits pattern}
  …

Inductive case…

- ‘case’: a = h :: tl
- IH: append (append tl b) c = append tl (append b c)
- append (append a b) c
  = {case}
  append (append (h :: tl) b) c
  = {append definition}
  append (match h :: tl with [] -> b | h :: tl -> h :: append tl b) c
  = {match fits pattern}
  append (h :: append tl b) c
  = {append definition}
  match (h :: append tl b) with [] -> c | h :: tl2 -> h :: append tl2 c
  = {match fits pattern}
  h :: append (append tl b) c
  = {…}

Inductive case…

- ‘case’: a = h :: tl
- IH: append (append tl b) c = append tl (append b c)
- append (append a b) c
  = {case}
  append (append (h :: tl) b) c
  = {append definition}
  append (match h :: tl with [] -> b | h :: tl -> h :: append tl b) c
  = {match fits pattern}
  append (h :: append tl b) c
  = {append definition}
  match (h :: append tl b) with [] -> c | h :: tl2 -> h :: append tl2 c
  = {match fits pattern}
  h :: append (append tl b) c
  = {IH}
  …

Inductive case…

- ‘case’: a = h :: tl
- IH: append (append tl b) c = append tl (append b c)
- append (append a b) c
  = {case}
  append (append (h :: tl) b) c
  = {append definition}
  append (match h :: tl with [] -> b | h :: tl -> h :: append tl b) c
  = {match fits pattern}
  append (h :: append tl b) c
  = {append definition}
  match (h :: append tl b) with [] -> c | h :: tl2 -> h :: append tl2 c
  = {match fits pattern}
  h :: append (append tl b) c
  = {IH}
  h :: append tl (append b c)

Inductive case…

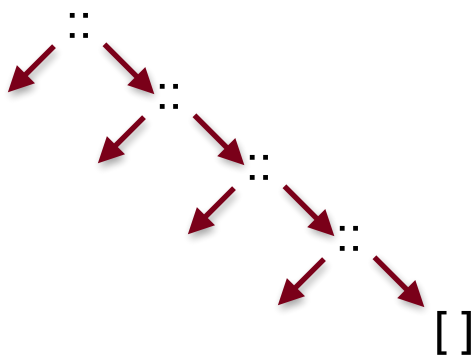
- append (append a b) c
  = {case}
  append (append (h :: tl) b) c
  = {append definition}
  append (match …) c
  = {match fits pattern}
  append (h :: append tl b) c
  = {append definition}
  match …
  = {match fits pattern}
  h :: append (append tl b) c
  = {IH}
  h :: append tl (append b c)
  = {match fits pattern}
  match h :: tl with
  | [] -> (append b c)
  | h :: tl -> h :: append tl (append b c)
  = {append definition}
  append (h :: tl) (append b c)
  = {case}
  append a (append b c)

Finishing up…

- append (append a b) c
  = {case}
  append (append (h :: tl) b) c
  = {append definition}
  append (match …) c
  = {match fits pattern}
  append (h :: append tl b) c
  = {append definition}
  match …
  = {match fits pattern}
  h :: append (append tl b) c
  = {IH}
  h :: append tl (append b c)
  = {match fits pattern}
  match h :: tl with
  | [] -> (append b c)
  | h :: tl -> h :: append tl (append b c)
  = {append definition}
  append (h :: tl) (append b c)
  = {case}
  append a (append b c)

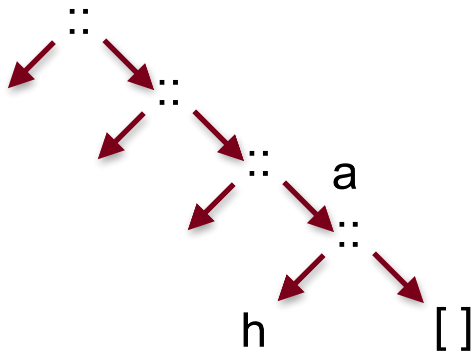
- This proves the theorem using induction on a :: 'a list.
- Let's call it: append is associative

What’s going on?



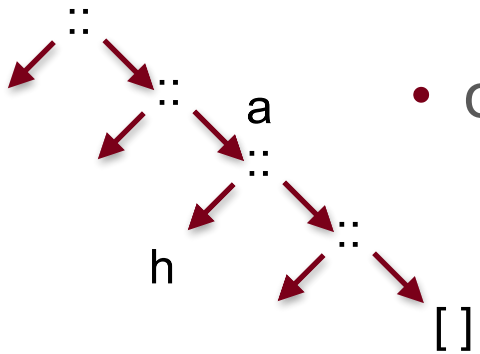
- We first prove the [ ] case
- This uses the assumption a = [ ]

What’s going on?



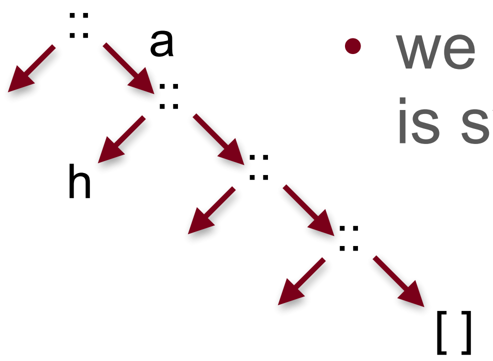
- We’ve proven the [ ] case, so we can use it in a proof for a = h::[ ]

What’s going on?



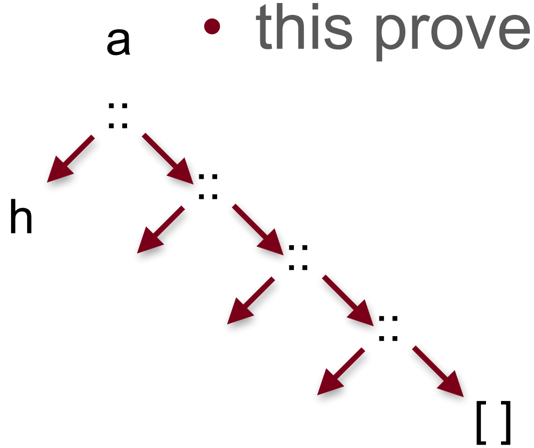
- once we’ve proven the _ :: [ ] case,
  we can use it in a proof for
  a = h::[_]

What's going on?



- we can repeat the pattern with what is syntactically the same proof

What's going on?



- this proves the statement for all (finite) lists

A faulty proof …

- What's wrong with this proof?
- We prove (for this definition) that
  append (append a b) c = append a (append b c)
  by induction on a
- case a = [] as before
- case a = h :: tl
  IH: append (append tl b) = append tl (append b c)

  append (append a b) c
  = {IH (substitute tl for a)}
  append a (append b c)   (and we are done)

A faulty proof …

- What’s wrong with this proof?
- We prove (for this definition) that
  append (append a b) c = append a (append b c)
  by induction on a
- case a = [] as before
- case a = h :: tl
  IH: append (append tl b) = append tl (append b c)

   append (append a b) c
  = {IH (substitute tl for a)}   <— This step is wrong (obviously)
   append a (append b c)   (and we are done)

A faulty proof …

- What's wrong with this proof?
- We prove (for this definition) that
  append (append a b) c = append a (append b c)
  by induction on a
- case a = [] as before
- case a = h :: tl
  IH: append (append tl b) = append tl (append b c)

 append (append a b) c
= {IH (substitute tl for a)}   <— tl is not a variable in the IH!
 append a (append b c)   (and we are done)

Variables vs non-variables

- When using an assumption, we have to distinguish variables from non-variables
- Usually, the difference is obvious:
  $x + 0 = x$
- variable: x
  non-variable: (+) and 0
- We can only substitute variables.

Another induction proof…

- let rec rev x = match x with
  [] -> [] | h::tl -> append (rev tl) [h]
- we prove that rev (rev x) = x
- case x = []
  - rev (rev [])
    = …

Another induction proof…

- let rec rev x = match x with
  [] -> [] | h::tl -> append (rev tl) [h]
- we prove that rev (rev x) = x
- case x = []
  - rev (rev [])
    = {rev definition}
    rev (match [] with [] -> [] | h :: tl -> append (rev tl) [h])
    = …

Another induction proof…

- let rec rev x = match x with
  [] -> [] | h::tl -> append (rev tl) [h]
- we prove that rev (rev x) = x
- case x = []
  - rev (rev [])
    = {rev definition}
    rev (match [] with [] -> [] | h :: tl -> append (rev tl) [h])
    = {match pattern}
    rev []
    = {rev definition}
    (match [] with [] -> [] | h :: tl -> append (rev tl) [h])
    = {match pattern}
    []

case x = []

Another induction proof…

- let rec rev x = match x with
  [] -> [] | h::tl -> append (rev tl) [h]
- we prove that rev (rev x) = x
- case x = h :: tl, IH: rev (rev tl) = tl
- rev (rev x)
  = {…}

Another induction proof…

- let rec rev x = match x with
  [] -> [] | h::tl -> append (rev tl) [h]
- we prove that rev (rev x) = x
- case x = h :: tl, IH: rev (rev tl) = tl
- rev (rev x)
  = {case}
  rev (rev (h :: tl))
  = {rev definition}
  rev (match h :: tl with [] -> [] | h::tl -> append (rev tl) [h])
  = {match}
  rev (append (rev tl) [h])
  = {…}

Another induction proof…

- let rec rev x = match x with
  [] -> [] | h::tl -> append (rev tl) [h]
- we prove that rev (rev x) = x
- case x = h :: tl, IH: rev (rev tl) = tl
- rev (rev x)
  = {case}
  rev (rev (h :: tl))
  = {rev definition}
  rev (match h :: tl with [] -> [] | h::tl -> append (rev tl) [h])
  = {match}
  rev (append (rev tl) [h])
  = {…}

Hey… are we stuck?

… let's add an unproven assumption, we can try proving it later

- let rec rev x = match x with
  [] -> [] | h::tl -> append (rev tl) [h]
- we prove that rev (rev x) = x
- case x = h :: tl, IH: rev (rev tl) = tl
- rev (rev x) case x = h :: tl
  = {case}
  rev (rev (h :: tl))
  = {rev definition}
  rev (match h :: tl with [] -> [] | h::tl -> append (rev tl) [h])
  = {match}
  rev (append (rev tl) [h])
  = {rev (append a b) = append (rev b) (rev a)}

inductive case

- let rec rev x = match x with
  [] -> [] | h::tl -> append (rev tl) [h]
- we prove that rev (rev x) = x
- case x = h :: tl, IH: rev (rev tl) = tl
- rev (rev x)
  = {case}
  rev (rev (h :: tl))
  = {rev definition}
  rev (match h :: tl with [] -> [] | h::tl -> append (rev tl) [h])
  = {match}
  rev (append (rev tl) [h])
  = {rev (append a b) = append (rev b) (rev a)}
  append (rev [h]) (rev (rev tl))
  = …

… powering through

- let rec rev x = match x with
  [] -> [] | h::tl -> append (rev tl) [h]
- we prove that rev (rev x) = x
- case x = h :: tl, IH: rev (rev tl) = tl
- rev (rev x)
  = {case}
  rev (rev (h :: tl))
  = {rev definition}
  rev (match h :: tl with [] -> [] | h::tl -> append (rev tl) [h])
  = {match}
  rev (append (rev tl) [h])
  = {rev (append a b) = append (rev b) (rev a)}
  append (rev [h]) (rev (rev tl))
  = {rev [x] = [x]}

… powering through

- let rec rev x = match x with
  [] -> [] | h::tl -> append (rev tl) [h]
- we prove that rev (rev x) = x
- case x = h :: tl, IH: rev (rev tl) = tl
- rev (rev x)
  = {case}
  rev (rev (h :: tl))
  = {rev definition}
  rev (match…)
  = {match}
  rev (append (rev tl) [h])

= {rev (append a b) = append (rev b) (rev a)}
append (rev [h]) (rev (rev tl))
= {rev [x] = [x]}
append [h] (rev (rev tl))
= {IH}
append [h] tl
= {append [x] y = x::y}
h :: tl
= {case}
x

… powering through

- let rec rev x = match x with
  [] -> [] | h::tl -> append (rev tl) [h]
- we prove that rev (rev x) = x
- case x = h :: tl, IH: rev (rev tl) = tl
- rev (rev x)
  = {case}
  rev (rev (h :: tl))
  = {rev definition}
  rev (match…)
  = {match}
  rev (append (rev tl) [h])
  = {rev (append a b) = append (rev b) (rev a)}
  append (rev [h]) (rev (rev tl))
  = {rev [x] = [x]}
  append [h] (rev (rev tl))
  = {IH}
  append [h] tl
  = {append [x] y = x::y}
  h :: tl
  = {case}
  x
- This completes the inductive proof

Let’s assess the damage:

- We’ve completed a proof by induction, but we need to prove the following things first:
  - rev (append a b) = append (rev b) (rev a)
  - rev [x] = [x]
  - append [x] y = x::y
- The last two can be done with simple evaluation proofs
- The first of these can be done with induction on a
  - You’ll get stuck again (in the base case this time)
  - You need ‘append x [] = x’ as an assumption
    - This you can prove using induction on x!

On hard induction proofs

- Textbooks change the order of the proofs
- You hardly ever encounter these in exams
- They occasionally show up in homework
- You encounter these in real life all the time

- btw, the proof of this is easier:
  rev_append (rev_append x xs) ys
  = append x (rev_append xs ys)
- (sometimes, the more efficient code is also easier to reason about)

Induction on other structures

- Natural numbers can be defined as follows:

- type nat = Z | S of nat

- If we prove L(x) = R(x) by induction on x : nat, we get:
  - case x = Z
  - case x = S y, IH: L(y) = R(y)

- Looks familiar?

Recall lecture 8 on fold_right

```
type exercise =
    | Int of int
    | Div of exercise * exercise
    | Add of exercise * exercise
```

- Three constructors, so our fold gets:

```
let fold_exercise f_int f_div f_add : exercise -> 'b =
  let rec fold_exercise' = function
    | Int i -> f_int i
    | Div (e1, e2) -> f_div (fold_exercise' e1)
                            (fold_exercise' e2)
    | Add (e1, e2) -> f_add (fold_exercise' e1)
                            (fold_exercise' e2)
  in fold_exercise’

val fold_exercise : (int -> 'b) -> ('b -> 'b -> ‘b)
      -> ('b -> 'b -> 'b) -> exercise -> 'b = <fun>
```

Recursive data-types

- Fold_right:
  - Takes a function per constructor
  - Non-recursive arguments to the constructor will be function arguments of the same type
  - Recursive arguments to the constructor have type ‘a
  - Result of constructor is type ‘a
- Inductive proof:
  - Takes a case per constructor
  - Recursive arguments to the constructor give an additional property

A property about your code

```
type exercise =
    | Int of int
    | Div of exercise * exercise
    | Add of exercise * exercise
```

- We prove, by induction on e, that:
  eval (filter_nontrivial e) = eval e
  - case e = Int i
  - case e = Div e1 e2
    IH1: eval (filter_nontrivial e1) = eval e1
    IH2: eval (filter_nontrivial e2) = eval e2
  - case e = Add e1 e2
    IH1: eval (filter_nontrivial e1) = eval e1
    IH2: eval (filter_nontrivial e2) = eval e2

Some concluding remarks

- Structural induction is just like case analysis…
  - but with an inductive hypothesis for the recursive part of the datastructure
  - treat the replaced elements as constants in the IH
  - there can be multiple IHs if there are multiple recursive parts
- You might get stuck and in need of a helper-lemma
  - complete the proof, then try your hands on any helper-lemmas