Midterm 2 practice

UNIVERSITY OF MINNESOTA
Driven to Discover®

Two questions, 5 x 20 points

- Prove that: or x y = or y x
- or (contains x lst1) (contains x lst2)
  = (contains x (append lst1 lst2))
  - base case
  - inductive part
- or (elem e a) (elem e b) = elem e (union2 a b)
- elem e a = elem e (foo a)

Definitions AND rules

- let or a b = if a then true else b
- or-t: or true b = true
- or-f: or false b = b

- The rules suffice, and save you a proof-step

Proof by cases

- or x y = or y x
- 4 cases, almost fits on one slide:
- x = true, y = true
  x = true, y = false
  x = false, y = true
  x = false, y = false
- case: x = y
  - or x y
    = {case}
    or x x
    = {case}
    or y x
- case: x = true, y = false
  - or x y
    = {case}
    or true false
    = {or-t}
    true
    = {or-f}
    or false true
    = {case}
    of y x
- case: x = false, y = true
  - (as above in other order)

Append / contains

```
let rec append lst1 lst2 = match lst1 with
  | [] -> lst2
  | hd::tl -> hd :: append tl lst2

let rec contains x lst =
  match lst with
  | [] -> false
  | hd::tl -> if hd = x then true
              else contains x tl

ap-nil: append [] lst2 = lst2
ap-c:   append (hd::tl) lst2 = hd :: append tl lst2
cts-n:   contains x [] = false
cts-c:   contains x (hd::tl)
           = if hd = x then true else contains x tl
```

Append / contains

```
ap-nil: append [] lst2 = lst2
ap-c:   append (hd::tl) lst2 = hd :: append tl lst2
cts-n:   contains x [] = false
cts-c:   contains x (hd::tl)
           = if hd = x then true else contains x tl

Prove:
or (contains x lst1) (contains x lst2)
= (contains x (append lst1 lst2))
```

Append / contains

```
ap-nil: append [] lst2 = lst2
ap-c:   append (hd::tl) lst2 = hd :: append tl lst2
cts-n:   contains x [] = false
cts-c:   contains x (hd::tl)
           = if hd = x then true else contains x tl

case: lst1 = []
or (contains x lst1) (contains x lst2)
= {case}
or (contains x []) (contains x lst2)
= {cts-n}
or false (contains x lst2)
= {or-f}
contains x lst2
= {ap-nil}
contains x (append [] lst2)
= {case}
contains x (append lst1 lst2)
```

Append / contains

```
ap-nil: append [] lst2 = lst2
ap-c:   append (hd::tl) lst2 = hd :: append tl lst2
cts-n:   contains x [] = false
cts-c:   contains x (hd::tl)
           = if hd = x then true else contains x tl

case: lst1 = h::tl, IH : ??
or (contains x lst1) (contains x lst2)
= {case}
or (contains x (h::tl)) (contains x lst2)
= {cts-c}
or (if h = x then true else contains x tl)
   (contains x lst2)
= {???}
(if h = x then true else contains x (append tl lst2))
= {cts-c}
contains x (h :: append tl lst2)
= {ap-c}
contains x (append (h::tl) lst2)
= {case}
contains x (append lst1 lst2)
```

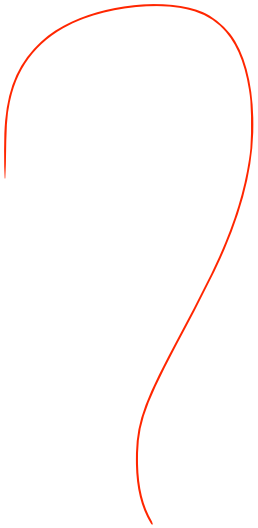
Append / contains

```
ap-nil: append [] lst2 = lst2
ap-c:   append (hd::tl) lst2 = hd :: append tl lst2
cts-n:   contains x [] = false
cts-c:   contains x (hd::tl)
           = if hd = x then true else contains x tl
IH : or (contains x tl) (contains x lst2)
      = contains x (append tl lst2)
case: lst1 = h::tl
proof of lemma, case (h = x) = true

or (if h = x then true else contains x tl)
   (contains x lst2)
= {case}
or (if true then true else contains x tl)
   (contains x lst2)
= {ite-t}
or true (contains x lst2)
= {or-t}
true
= {or-t}
(if true then true else contains x (append tl lst2))
```

the result of processing the right stuck, which was the conclusion from being stuck

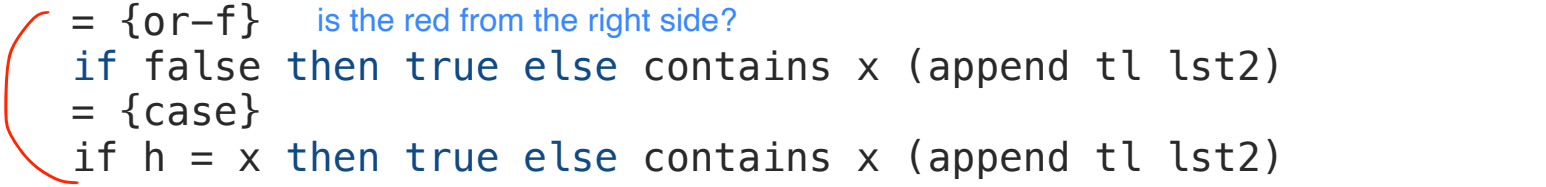
Append / contains

```
ap-nil: append [] lst2 = lst2
ap-c:   append (hd::tl) lst2 = hd :: append tl lst2
cts-n:   contains x [] = false
cts-c:   contains x (hd::tl)
           = if hd = x then true else contains x tl
IH : or (contains x tl) (contains x lst2)
      = contains x (append tl lst2)
case: lst1 = h::tl
proof of lemma, case (h = x) = true

or (if h = x then true else contains x tl)
   (contains x lst2)
= {case}
or (if true then true else contains x tl)
   (contains x lst2)
= {ite-t}
or true (contains x lst2)
= {or-t}
true
= {or-t}  is the red from the right side?
if true then true else contains x (append tl lst2)
= {case}
if h = x then true else contains x (append tl lst2)
```



Append / contains

```
ap-nil: append [] lst2 = lst2
ap-c:   append (hd::tl) lst2 = hd :: append tl lst2
cts-n:   contains x [] = false
cts-c:   contains x (hd::tl)
           = if hd = x then true else contains x tl
IH : or (contains x tl) (contains x lst2)
      = contains x (append tl lst2)
case: lst1 = h::tl
proof of lemma, case (h = x) = false

or (if h = x then true else contains x tl)
   (contains x lst2)
= {case}
or (if false then true else contains x tl)
   (contains x lst2)
= {ite-f}
or (contains x tl) (contains x lst2)
= {IH}
contains x (append tl lst2)
= {or-f}   is the red from the right side?
if false then true else contains x (append tl lst2)
= {case}
if h = x then true else contains x (append tl lst2)
```



Append / contains proof

- 2 x 20 points
- Induction on lst1 is required by midterm question
- Case distinction was not mentioned on mock midterm

- … a bit longer than ideal for a midterm

Question two

```
module type lattice = sig
  type 'a t
  elem : 'a -> 'a t -> bool
  (** must satisfy:
   [ or (elem e a) (elem e b)
   = elem e (union a b) ] **)
  union : 'a t -> 'a t -> 'a t
end
```

We'll assume these properties

Question two

```
module type lattice = sig
  type 'a t
  elem : 'a -> 'a t -> bool
  (** must satisfy:
   [ or (elem e a) (elem e b)
   = elem e (union a b) ] **)
  union : 'a t -> 'a t -> 'a t
end
```

```
module Converse (L : lattice) = struct
  type 'a t = 'a L.t
  let elem = L.elem
  let union2 x y
    = L.union y x
  let foo x
    = L.union x x
end
```

1. Prove that union2 satisfies:
or (elem e a) (elem e b)
= elem e (union2 a b)

2. Prove that foo satisfies:
elem e a = elem e (foo a)

union2

```
given:
or (elem e a) (elem e b) = elem e (union a b)

def union2: let union2 x y = L.union y x

exercise 1: or x y = or y x

prove: or (elem e a) (elem e b) = elem e (union2 a b)

elem e (union2 a b)
= {def}
elem e (L.union b a)
= {given}
or (elem e b) (elem e a)
= {ex 1}
or (elem e a) (elem e b)
```

foo

```
let foo x
  = L.union x x
prove: elem e a = elem e (foo a)
```

foo

```
lemma: or x x = x

case: x = true
or x x
= {case, twice}
or true true
= { or-t}
true
= {case}
x
```

```
case: x = false
or x x
= {case, twice}
or false false
= { or-f}
false
= {case}
x
```

foo

```
let foo x
  = L.union x x
prove: elem e a = elem e (foo a)

elem e a
= {lemma}
or (elem e a) (elem e a)
= {given by signature}
elem e (L.union a a)
= {foo-def}
elem e (foo a)
```