Did I get it right?

Part 3: Induction for Lists

Speaker: David Walker
COS 326
Princeton University

http://~cos326/notes/evaluation.php
http://~cos326/notes/reasoning.php

Last Time --> This Time

Last time, we saw some proofs can be done by induction over natural numbers

It turns out the structure of natural numbers is similar in many ways to the structure of lists.

In this lecture, we'll take a look at how to do a similarly structured proofs over lists.

A Couple of Useful Functions


```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
  match xs1 with
  | [] -> xs2
  | hd::tl -> hd :: cat tl xs2
```

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof strategy:

- Proof by induction on the list xs
  - recall, a list may be of these two things:
    - [] (the empty list)
    - hd::tl (a non-empty list, where tl is shorter)
  - a proof must cover both cases: [ ] and hd :: tl
  - in the second case, you will often use the inductive hypothesis on the smaller list tl
  - otherwise as before:
    - use folding/eval of OCaml definitions
    - use your knowledge of OCaml evaluation
    - use lemmas/properties you know of basic operations like :: and +

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

case xs = [ ]:

```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
  match xs1 with
  | [] -> xs2
  | hd::tl -> hd :: cat tl xs2
```

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

case xs = [ ]:

length (cat [ ] ys) (LHS of theorem)

```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
  match xs1 with
  | [] -> xs2
  | hd::tl -> hd :: cat tl xs2
```

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

case xs = [ ]:

length (cat [ ] ys)	(LHS of theorem)
= length ys	(evaluate cat)

```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
  match xs1 with
  | [] -> xs2
  | hd::tl -> hd :: cat tl xs2
```

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

case xs = [ ]:

length (cat [ ] ys)	(LHS of theorem)
= length ys	(evaluate cat)
= 0 + (length ys)	(arithmetic)

```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
  match xs1 with
  | [] -> xs2
  | hd::tl -> hd :: cat tl xs2
```

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

case xs = [ ]:

```
   length (cat [ ] ys)              (LHS of theorem)
 = length ys                        (evaluate cat)
 = 0 + (length ys)                  (arithmetic)
 = (length [ ]) + (length ys)       (eval length)
```

case done!

```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
   match xs1 with
   | [] -> xs2
   | hd::tl -> hd :: cat tl xs2
```

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

case xs = hd::tl

```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
  match xs1 with
  | [] -> xs2
  | hd::tl -> hd :: cat tl xs2
```

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

case xs = hd::tl
  IH: length (cat tl ys) = length tl + length ys

```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
  match xs1 with
  | [] -> xs2
  | hd::tl -> hd :: cat tl xs2
```

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

case xs = hd::tl
IH: length (cat tl ys) = length tl + length ys

```
   length (cat (hd::tl) ys)          (LHS of theorem)
==
```

```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
  match xs1 with
  | [] -> xs2
  | hd::tl -> hd :: cat tl xs2
```

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

```
case xs = hd::tl
  IH: length (cat tl ys) = length tl + length ys

   length (cat (hd::tl) ys)        (LHS of theorem)
== length (hd :: (cat tl ys))      (evaluate cat, take 2nd branch)
==
```

```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
  match xs1 with
  | [] -> xs2
  | hd::tl -> hd :: cat tl xs2
```

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

case xs = hd::tl

IH: length (cat tl ys) = length tl + length ys

```
   length (cat (hd::tl) ys)        (LHS of theorem)
== length (hd :: (cat tl ys))      (evaluate cat, take 2nd branch)
== 1 + length (cat tl ys)          (evaluate length, take 2nd branch)
==
```

```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
  match xs1 with
  | [] -> xs2
  | hd::tl -> hd :: cat tl xs2
```

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

case xs = hd::tl

IH: length (cat tl ys) = length tl + length ys

```
   length (cat (hd::tl) ys)          (LHS of theorem)
== length (hd :: (cat tl ys))        (evaluate cat, take 2nd branch)
== 1 + length (cat tl ys)            (evaluate length, take 2nd branch)
== 1 + (length tl + length ys)       (by IH)
==
```

```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
   match xs1 with
   | [] -> xs2
   | hd::tl -> hd :: cat tl xs2
```

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

case xs = hd::tl

IH: length (cat tl ys) = length tl + length ys

length (cat (hd::tl) ys)	(LHS of theorem)
== length (hd :: (cat tl ys))	(evaluate cat, take 2nd branch)
== 1 + length (cat tl ys)	(evaluate length, take 2nd branch)
== 1 + (length tl + length ys)	(by IH)
== length (hd::tl) + length ys	(reparenthesizing and evaling length in reverse we have RHS with hd::tl for xs)

case done!

```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
   match xs1 with
   | [] -> xs2
   | hd::tl -> hd :: cat tl xs2
```

Proofs About Lists

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof strategy:

- Proof by induction on the list xs? why not on the list ys?
  - answering that question, may be the hardest part of the proof!
  - it tells you how to split up your cases
  - sometimes you just need to do some trial and error

```
let rec length xs =
  match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
```

```
let rec cat xs1 xs2 =
  match xs1 with
  | [] -> xs2
  | hd::tl -> hd :: cat tl xs2
```

a clue:
pattern matching
on first argument.
In the theorem:
cat xs ys
Hence induction
on xs. Case split
the same way
as the program

Be careful with the Induction Hypothesis!

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

In COS 326, the induction hypothesis is a function of <u>one</u> variable (in this case, xs)

case xs = hd::tl
IH: length (cat tl ys) = length tl + length ys

The <u>use</u> of the IH must be at a <u>smaller</u> value (in this case, “tl” is smaller than “xs”)

length (cat (hd::tl) ys)
== length (hd :: (cat tl ys))
== 1 + length (cat tl ys)
== 1 + (length tl + length ys) (by IH)
== length (hd::tl) + length ys (reparenthesizing and evaling length in reverse

case

In your proofs, it should be <u>really obvious</u>

- which variable the IH is supposed to be a function of
- that your induction is on that variable
- that you’re applying the IH at <u>smaller</u> values

If you’re not sure it’s obvious, just say explicitly in your proof: which variable it is, and why you claim you’re applying it at smaller values

xs2

Be careful with the Induction Hypothesis!

Theorem: For all lists xs and ys,

length(cat xs ys) = length xs + length ys

Proof: By induction on xs.

In COS 326, the induction hypothesis will typically be a function of *one variable* (in this case, xs)

In more complicated proofs, the induction hypothesis is a function of *one structure* where the ordering of elements in the structure is *well-founded* (there are no infinite descending chains).

EG: Induction on pairs of naturals (x, y)
where pairs are ordered lexicographically:

(x1, y1) > (x2, y2)
iff
x1 > x2 or (x1 = x2 and y1 > y2)

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = [ ]:

```
    add_all (add_all [] a) b             (LHS of theorem)
==
```

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = [ ]:

```
   add_all (add_all [] a) b          (LHS of theorem)
== add_all [ ] b                     (by evaluation of add_all)
==
```

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = [ ]:

```
   add_all (add_all [] a) b          (LHS of theorem)
== add_all [ ] b                     (by evaluation of add_all)
== [ ]                               (by evaluation of add_all)
==
```

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = [ ]:

add_all (add_all [] a) b	(LHS of theorem)
== add_all [ ] b	(by evaluation of add_all)
== [ ]	(by evaluation of add_all)
== add_all [ ] (a + b)	(by evaluation of add_all)

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = hd :: tl:

```
    add_all (add_all (hd :: tl) a) b            (LHS of theorem)
==
```

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = hd :: tl:

```
   add_all (add_all (hd :: tl) a) b          (LHS of theorem)
== add_all ((hd+a) :: add_all tl a) b        (by eval inner add_all)
==
```

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = hd :: tl:

add_all (add_all (hd :: tl) a) b	(LHS of theorem)
== add_all ((hd+a) :: add_all tl a) b	(by eval inner add_all)
== (hd+a+b) :: (add_all (add_all tl a) b)	(by eval outer add_all)
==	

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = hd :: tl:

add_all (add_all (hd :: tl) a) b	(LHS of theorem)
== add_all ((hd+a) :: add_all tl a) b	(by eval inner add_all)
== (hd+a+b) :: (add_all (add_all tl a) b)	(by eval outer add_all)
== (hd+a+b) :: add_all tl (a+b)	(by IH)

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = hd :: tl:

```
   add_all (add_all (hd :: tl) a) b        (LHS of theorem)
== add_all ((hd+a) :: add_all tl a) b      (by eval inner add_all)
== (hd+a+b) :: (add_all (add_all tl a) b)  (by eval outer add_all)
== (hd+a+b) :: add_all tl (a+b)            (by IH)
== (hd+(a+b)) :: add_all tl (a+b)          (associativity of + )
```

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = hd :: tl:

add_all (add_all (hd :: tl) a) b	(LHS of theorem)
== add_all ((hd+a) :: add_all tl a) b	(by eval inner add_all)
== (hd+a+b) :: (add_all (add_all tl a) b)	(by eval outer add_all)
== (hd+a+b) :: add_all tl (a+b)	(by IH)
== (hd+(a+b)) :: add_all tl (a+b)	(associativity of +)
== add_all (hd::tl) (a+b)	(by (reverse) eval of add_all)

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Template for Inductive Proofs on Lists

Theorem: For all lists xs, property of xs.

Proof: By induction on lists xs.

Case: xs == [ ]:

...

Case: xs == hd :: tl:

...

There are other ways to cover all lists:
case for [], case for x1::[], case for x1::x2::tl'

cases must cover all lists

But that's the same as covering [] and x1::tl ...

... and then just splitting x1::tl into 2 additional cases
where tl is [] or tl is x2::tl' ...

Template for Inductive Proofs on ***any datatype***

type ty = A of … | B of … | C of … | D

Theorem: For all ty x, property of x.

Proof: By induction on x of type ty.

Case: x == A(…):
…
Case: x == B(…):
…
Case: x == C(…):
…
Case: x == D:
…

cases must cover all the constructors of the datatype

SUMMARY

Summary of Proof Techniques

Proofs about programs are structured similarly to the programs:

- types tell you the kinds of values your proofs/programs operate over
- types suggest how to break down proofs/programs into cases
- when programs use recursion on smaller values they terminate and their proofs appeal to the inductive hypothesis on smaller values

Key proof ideas:

- expression evaluation: if e evaluates to e' then e == e
- substitution of equals for equals
- use well-established axioms about primitives (+, -, %, etc)
- use proof by induction to prove correctness of recursive functions
- split proofs about complex data into cases; be sure to cover all cases