Did I get it right?

COS 326
Andrew W. Appel
Princeton University

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

Did I get it right?

"Did I get it right?"

– Most fundamental question you can ask about a computer program

Techniques for answering:

Grading

- hand in program to TA
- check to see if you got an A
- (does not apply after school is out)

Testing

- create a set of sample inputs
- run the program on each input
- check the results
- how far does this get you?
  - has anyone ever tested a homework and not received an A?
  - why did that happen?

Proving

- consider all legal inputs
- show every input yields correct result
- how far does this get you?
  - has anyone ever proven a homework correct and not received an A?
  - why did that happen?

Program proving

The basic, overall *mechanics* of proving functional programs correct is not particularly hard.

- You are already doing it to some degree.
- The real goal of this lecture to help you further organize your thoughts and to give you a more systematic means of understanding your programs.
- Of course, it can certainly be hard to prove some specific program has some specific property -- just like it can be hard to write a program that solves some hard problem

We are going to focus on proving the correctness of *pure expressions*

- their meaning is determined exclusively by the value they return
- don't print, don't mutate global variables, don't raise exceptions
- always terminate
- another word for "pure expression" is "valuable expression"
- but I want you to understand why the presence of possibly non-terminating programs complicates rigorous reasoning about program correctness

“Expressions always terminate”

Two key concepts:

- *A valuable expression*
  - an expression that always terminates (without side effects) and produces a value, provided we substitute values for free variables in the expression
- A *total function* with type t1 -> t2
  - a function that terminates on all args : t1, producing a value of type t2
  - the “opposite” of a total function is a *partial function*
    - terminates on some (possibly all) input values

Many reasoning rules depend on expressions being valuable and hence the functions that are applied being total.

Unless told otherwise, you can assume all functions are total and expressions are valuable. (Such facts can typically be proven by induction.)

Example Theorems

We'll prove properties of OCaml expressions, starting with equivalence properties:

```
let easy x y z =
  x * (y + z)
```

Theorem: easy 1 20 30 == 50

```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem:

for all natural numbers n,

exp n == 2^n

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

Theorem:

for all lists xs, ys,

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

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

Things to Watch For

The types are going to guide us in our theorem proving, just like they guided us in our programming

Things to Watch For

The types are going to guide us in our theorem proving, just like they guided us in our programming

- when *programming* with lists, *functions* (often) have 2 cases:
  - [ ]
  - hd :: tl
- when *proving* with lists, *proofs* (often) have 2 cases:
  - [ ]
  - hd :: tl

Things to Watch For

The types are going to guide us in our theorem proving, just like they guided us in our programming

- when *programming* with lists, *functions* (often) have 2 cases:
  - [ ]
  - hd :: tl
- when *proving* with lists, *proofs* (often) have 2 cases:
  - [ ]
  - hd :: tl
- when *programming* with natural numbers, *functions* have 2 cases:
  - 0
  - k + 1
- when *proving* with natural numbers, *proofs* have 2 cases:
  - 0
  - k + 1

This is not a fluke! Proof structure often related to program structure.

Things to Watch For

More structure:

- when *programming* with lists:
  - [ ] is often easy
  - hd :: tl often requires a *recursive function call* on tl
    - we *assume* our recursive function behaves correctly on tl
- when *proving* with lists:
  - [ ] is often easy
  - hd :: tl often requires appeal to an *induction hypothesis* for tl
    - we *assume* our property of interest holds for tl

Things to Watch For

More structure:

- when *programming* with lists:
  - [ ] is often easy
  - hd :: tl often requires a *recursive function call* on tl
    - we *assume* our recursive function behaves correctly on tl
- when *proving* with lists:
  - [ ] is often easy
  - hd :: tl often requires appeal to an *induction hypothesis* for tl
    - we *assume* our property of interest holds for tl
- when *programming* with natural numbers:
  - 0 is often easy
  - k + 1 often requires a *recursive call* on k
- when *proving* with natural numbers:
  - 0 is often easy
  - k + 1 often requires appeal to an *induction hypothesis* for k

Key Ideas

Idea 1: The fundamental definition of when programs are equal.

two expressions are equal if and only if:

- they both evaluate to the same value, or
- they both raise the same exception, or
- they both infinite loop

we will use what we learned about OCaml evaluation

Key Ideas

Idea 1: The fundamental definition of when programs are equal.

two expressions are equal if and only if:

- they both evaluate to the same value, or
- they both raise the same exception, or
- they both infinite loop

this is the principle of "substitution of equals for equals"

Idea 2: A fundamental proof principle.

if two expressions e1 and e2 are equal
and we have a third complicated expression FOO (x)
then FOO(e1) is equal to FOO (e2)

super useful since we can do a small, local proof
and then use it in a big program: modularity!

The Workhorse: Substitution of Equals for Equals

if two expressions e1 and e2 are equal
and we have a third complicated expression FOO (x)
then FOO(e1) is equal to FOO (e2)

An example: I know 2+2 == 4.

I have a complicated expression: bar (foo (____)) * 34

Then I also know that bar (foo (2+2)) * 34 == bar (foo (4)) * 34.

If expressions contain things like mutable references, this proof principle breaks down. That's a big reason why I like functional programming and a big reason we are working primarily with pure expressions.

Important Properties of Expression Equality

Other important properties:

(reflexivity) every expression e is equal to itself: e == e

(symmetry) if e1 == e2 then e2 == e1

(transitivity) if e1 == e2 and e2 == e3 then e1 == e3

(evaluation) if e1 --> e2 then e1 == e2.

(congruence, aka substitution of equals for equals) if two expressions are equal, you can substitute one for the other inside any other expression:

- if e1 == e2 then e[e1/x] == e[e2/x]

Function evaluation

If: f == fun x -> e

and if: a is a valuable expression

then: f a == e[a/x]

we say, "by evaluation of f"

EASY EXAMPLES

Easy Examples

Most of our proofs will use what we know about the substitution model of evaluation. Eg:

a function definition

Given:

```
let easy x y z = x * (y + z)
```

Easy Examples

Most of our proofs will use what we know about the substitution model of evaluation. Eg:

Given:

```
let easy x y z = x * (y + z)
```

Theorem: easy 1 20 30 == 50

Easy Examples

Most of our proofs will use what we know about the substitution model of evaluation. Eg:

Given:

```
let easy x y z = x * (y + z)
```

Theorem: easy 1 20 30 == 50

Proof:

easy 1 20 30 (left-hand side of equation)

Easy Examples

Most of our proofs will use what we know about the substitution model of evaluation. Eg:

Given:

```
let easy x y z = x * (y + z)
```

Theorem: easy 1 20 30 == 50

Proof:

```
    easy 1 20 30          (left-hand side of equation)
==  1 * (20 + 30)         (by evaluating easy 1 step) actually 3 steps
```

Easy Examples

Most of our proofs will use what we know about the substitution model of evaluation. Eg:

Given:

```
let easy x y z = x * (y + z)
```

Theorem: easy 1 20 30 == 50

Proof:

```
   easy 1 20 30          (left-hand side of equation)
== 1 * (20 + 30)         (by evaluating easy 1 step)
== 50                    (by math)
```

QED.

Easy Examples

Most of our proofs will use what we know about the substitution model of evaluation. Eg:

Given: `let easy x y z = x * (y + z)`

facts go on the left

Theorem: easy 1 20 30 == 50

justifications on the right

Proof:

easy 1 20 30	(left-hand side of equation)
== 1 * (20 + 30)	(by evaluating easy 1 step)
== 50	(by math)

notice the 2-column proof style

QED.

Easy Examples

We can use *symbolic values* in in our proofs too. Eg:

Given:

```
let easy x y z = x * (y + z)
```

Theorem: for all integers n and m, easy 1 n m == n + m

Proof:

easy 1 n m (left-hand side of equation)

Easy Examples

We can use *symbolic values* in in our proofs too. Eg:

Given:

```
let easy x y z = x * (y + z)
```

Theorem: for all integers n and m, easy 1 n m == n + m

Proof:

easy 1 n m (left-hand side of equation)

When asked to prove something “for all n : t”, one way to do that is to consider *arbitrary* elements n of that type t. In other words, all you get to assume is that you have an element of the given type. You don’t get to assume any extra properties of n.

Easy Examples

We can use *symbolic values* in in our proofs too. Eg:

Given:

```
let easy x y z = x * (y + z)
```

Theorem: for all integers n and m, easy 1 n m == n + m

Proof:

easy 1 n m (left-hand side of equation)

== 1 * (n + m) (by evaluating easy)

Easy Examples

We can use *symbolic values* in in our proofs too. Eg:

Given:

```
let easy x y z = x * (y + z)
```

Theorem: for all integers n and m, easy 1 n m == n + m

Proof:

```
   easy 1 n m          (left-hand side of equation)
==  1 * (n + m)        (by evaluating easy)
== n + m               (by math)
```

QED.

Easy Examples

We can use *symbolic values* in in our proofs too. Eg:

Given:

```
let easy x y z = x * (y + z)
```

Theorem: for all integers n, m, k, easy k n m == easy k m n

Proof:

easy k n m (left-hand side of equation)

Easy Examples

We can use *symbolic values* in in our proofs too. Eg:

Given:

```
let easy x y z = x * (y + z)
```

Theorem: for all integers n, m, k, easy k n m == easy k m n

Proof:

easy k n m	(left-hand side of equation)
== k * (n + m)	(by evaluating easy)

Easy Examples

We can use *symbolic values* in in our proofs too. Eg:

Given: let easy x y z = x * (y + z)

Theorem: for all integers n, m, k, easy k n m == easy k m n

Proof:

easy k n m	(left-hand side of equation)
== k * (n + m)	(by evaluating easy)
== k * (m + n)	(by math, subst of equals for equals)

I'm not going to mention this from now on

Easy Examples

We can use *symbolic values* in in our proofs too. Eg:

Given:

```
let easy x y z = x * (y + z)
```

Theorem: for all integers n, m, k, easy k n m == easy k m n

Proof:

```
   easy k n m          (left-hand side of equation)
== k * (n + m)         (by evaluating easy)
== k * (m + n)         (by math)
== easy k m n          (by evaluating easy)
```

QED.

Easy Examples

We can use *symbolic values* in in our proofs too. Eg:

Given: let easy x y z = x * (y + z)

Theorem: for all integers n, m, k, easy k n m == easy k m n

Proof:

easy k n m	(left-hand side of equation)
== k * (n + m)	(eval)
== k * (m + n)	(by math)
== easy k m n	(eval)

QED.

substitution/
evaluating/
“eval”
a definition

the reverse:
“folding” a definition
back up

An Aside: Symbolic Evaluation

One last thing: we sometimes find ourselves with a function, like easy, that has a symbolic argument like k+1 for some k and we would like to evaluate it in our proof. eg:

easy x y (k+1)
== x * (y + (k+1))   (by evaluation of easy .... I hope)

However, that is not how OCaml evaluation works. OCaml evaluates its arguments to a *value* first, and then calls the function.

Don't worry: if you know that the expression *will* evaluate to a value (and will not infinite-loop or raise an exception) then you can substitute the symbolic expression for the parameter of the function

*To be rigorous, you should **prove** it will evaluate to a value, not just "know" ... but we won't require you prove that in this class ...*

An Aside: Symbolic Evaluation

An interesting example:

let const x = 7

const (exp) == 7 (By evaluation of const?)

does this work for any expression?

An Aside: Symbolic Evaluation

An interesting example:

```
let const x = 7
```

const (n / 0) == 7 (By *careless*, *wrong!* proof)

An Aside: Symbolic Evaluation

An interesting example:

```
let const x = 7
```

const (n / 0) == 7   (By *careless*, *wrong!* evaluation of const)

- n / 0 raises an exception
- so const (n / 0) raises an exception
- but 7 is just 7 and doesn't raise an exception
- an expression that raises an exception is not equal to one that returns a value!

An Aside: Symbolic Evaluation

An interesting example:

```
let const x = 7
```

const (exp) == 7 (By evaluation of const?)

does this work for any expression ***that doesn't raise an exception?***

An Aside: Symbolic Evaluation

An interesting example:

let const x = 7

const (loop 0) == 7 when let rec loop(x:int) = loop x ?

more careless, wrong evaluation ...

equations:

(1) (fun x -> e1) e2 == e1[e2/x]

(2) (f e2) == e1[e2/x] when let rec f x = e1

and when e2 evaluates to a value
(not an exception or infinite loop)

An Aside: Symbolic Evaluation

An interesting example:

```
let const x = 7
```

const (f 0) == 7 when let f i = print_endline "hello"; 6 in

?

equations:

(1) (fun x -> e1) e2 == e1[e2/x]

(2) (f e2) == e1[e2/x] when let rec f x = e1

and when e2 evaluates to a value

without side effects, raising an exception, or infinite loops

Summary so far: Proof by simple calculation

Some proofs are very easy and can be done by:

- eval definitions (ie: using forwards evaluation)
- using lemmas or facts we already know (eg: math)
- folding definitions back up (ie: using reverse evaluation)

Eg:

Definition:
let easy x y z = x * (y + z)

given this

we do this proof

Theorem: easy a b c == easy a c b

Proof:

easy a b c

== a * (b + c) (by def of easy)

== a * (c + b) (by math)

== easy a c b (by def of easy)

INDUCTIVE PROOFS

A problem

Theorem: For all natural numbers n,

exp(n) == 2^n.

```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

A problem


```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is
either 0 or it is k+1 (where k is also a natural number).
Hence, we follow the structure of the data and do
our proof in two cases.

A problem


```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is either 0 or it is k+1 (where k is also a natural number). Hence, we follow the structure of the data and do our proof in two cases.

Proof:

Case: n = 0:

exp 0

A problem


```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is either 0 or it is k+1 (where k is also a natural number). Hence, we follow the structure of the data and do our proof in two cases.

Proof:

Case: n = 0:

```
   exp 0
== match 0 with 0 -> 1 | n -> 2 * exp (n -1)    (by eval exp)
```

A problem


```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is either 0 or it is k+1 (where k is also a natural number). Hence, we follow the structure of the data and do our proof in two cases.

Proof:

Case: n = 0:

exp 0	
== match 0 with 0 -> 1 | n -> 2 * exp (n -1)	(by eval exp)
== 1	(by evaluating match)
== 2^0	(by math)

A problem


```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is either 0 or it is k+1 (where k is also a natural number). Hence, we follow the structure of the data and do our proof in two cases.

Proof:

Case: n == k+1:

exp (k+1)

A problem


```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is either 0 or it is k+1 (where k is also a natural number). Hence, we follow the structure of the data and do our proof in two cases.

Proof:

Case: n == k+1:

    exp (k+1)
== match (k+1) with 0 -> 1 | n -> 2 * exp (n -1)   (by eval exp)

A problem


```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is
either 0 or it is k+1 (where k is also a natural number).
Hence, we follow the structure of the data and do
our proof in two cases.

Proof:

Case: n == k+1:

```
   exp (k+1)
== match (k+1) with 0 -> 1 | n -> 2 * exp (n -1)      (by eval exp)
== 2 * exp (k+1 - 1)                                  (by evaluating case)
```

A problem


```
let rec exp n =
  match n with
   | 0 -> 1
   | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is either 0 or it is k+1 (where k is also a natural number). Hence, we follow the structure of the data and do our proof in two cases.

Proof:

Case: n == k+1:

```
   exp (k+1)
== match (k+1) with 0 -> 1 | n -> 2 * exp (n -1)        (by eval exp)
== 2 * exp (k+1 - 1)                                   (by evaluating case)
== ??
```

A problem


```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is either 0 or it is k+1 (where k is also a natural number). Hence, we follow the structure of the data and do our proof in two cases.

Proof:

Case: n == k+1:

```
   exp (k+1)
== match (k+1) with 0 -> 1 | n -> 2 * exp (n -1)          (by eval exp)
== 2 * exp (k+1 - 1)                                      (by evaluating case)
== 2 * (match (k+1-1) with 0 -> 1 | n -> 2 * exp (n -1)) (by eval exp)
```

A problem


```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is either 0 or it is k+1 (where k is also a natural number). Hence, we follow the structure of the data and do our proof in two cases.

Proof:

Case: n == k+1:

```
   exp (k+1)
== match (k+1) with 0 -> 1 | n -> 2 * exp (n -1)          (by eval exp)
== 2 * exp (k+1 - 1)                                      (by evaluating case)
== 2 * (match (k+1-1) with 0 -> 1 | n -> 2 * exp (n -1)) (by eval exp)
== 2 * (2 * exp ((k+1) - 1 - 1))                          (by evaluating case)
```

A problem


```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is either 0 or it is k+1 (where k is also a natural number). Hence, we follow the structure of the data and do our proof in two cases.

Proof:

Case: n == k+1:

```
   exp (k+1)
== match (k+1) with 0 -> 1 | n -> 2 * exp (n -1)          (by eval exp)
== 2 * exp (k+1 - 1)                                        (by evaluating case)
== 2 * (match (k+1-1) with 0 -> 1 | n -> 2 * exp (n -1))   (by eval exp)
== 2 * (2 * exp ((k+1) - 1 - 1))                           (by evaluating case)
== ... we aren't making progress ... just unrolling the loop forever ...
```

Induction

When proving theorems about recursive functions, we usually need to use *induction*.

- In inductive proofs, in a case for object X, we assume that the theorem holds *for all objects smaller than X*
  - this assumption is called the *induction hypothesis* (IH for short)
- Eg: When proving a theorem about natural numbers by induction, and considering the case for natural number k+1, we get to assume our theorem is true for natural number k (because k is smaller than k+1)
- Eg: When proving a theorem about lists by induction, and considering the case for a list x::xs, we get to assume our theorem is true for the list xs (which is a shorter list than x::xs)

Back to the Proof


```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is either 0 or it is k+1 (where k is also a natural number). Hence, we follow the structure of the data and do our proof in two cases.

Proof:

Case: n == k+1:

```
   exp (k+1)
== match (k+1) with 0 -> 1 | n -> 2 * exp (n -1)     (by eval exp)
== 2 * exp (k+1 - 1)                                  (by evaluating case)
```

Back to the Proof

```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is either 0 or it is k+1 (where k is also a natural number). Hence, we follow the structure of the data and do our proof in two cases.

Proof:

Case: n == k+1:

exp (k+1)	
== match (k+1) with 0 -> 1 | n -> 2 * exp (n -1)	(by eval exp)
== 2 * exp (k+1 - 1)	(by evaluating case)
== 2 * exp (k)	(by math)

Back to the Proof


```
let rec exp n =
  match n with
    | 0 -> 1
    | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is
either 0 or it is k+1 (where k is also a natural number).
Hence, we follow the structure of the data and do
our proof in two cases.

Proof:

Case: n == k+1:

exp (k+1)	
== match (k+1) with 0 -> 1 | n -> 2 * exp (n -1)	(by eval exp)
== 2 * exp (k+1 - 1)	(by evaluating case)
== 2 * exp (k)	(by math)
== 2 * 2^k	(by IH!)

Back to the Proof


```
let rec exp n =
  match n with
   | 0 -> 1
   | n -> 2 * exp (n-1)
```

Theorem: For all natural numbers n,

exp(n) == 2^n.

Recall: Every natural number n is
either 0 or it is k+2 (where k is also a natural number).
Hence, we follow the structure of the data and do
our proof in two cases.

Proof:

Case: n == k+1:

exp (k+1)	
== match (k+1) with 0 -> 1 | n -> 2 * exp (n -1)	(by eval exp)
== 2 * exp (k+1 - 1)	(by evaluating case)
== 2 * exp (k)	(by math)
== 2 * 2^k	(by IH!)
== 2^(k+1)	(by math)
QED!	

Another example

Theorem: For all natural numbers n, even(2*n) == true.

```
let rec even n =
  match n with
   | 0 -> true
   | 1 -> false
   | n -> even (n-2)
```

Recall: Every natural number n is either 0 or k+1, where k is also a natural number.

Case: n == 0:

...

Case: n == k+1:

...

Another example

Theorem: For all natural numbers n, even(2*n) == true.

Recall: Every natural number n is either 0 or k+1, where k is also a natural number.

Case: n == 0:

even (2*0)

==

```
let rec even n =
  match n with
    | 0 -> true
    | 1 -> false
    | n -> even (n-2)
```

Another example

Theorem: For all natural numbers n,
even(2*n) == true.

```
let rec even n =
  match n with
    | 0 -> true
    | 1 -> false
    | n -> even (n-2)
```

Recall: Every natural number n is either 0 or k+1, where k is also a natural number.

Case: n == 0:

even (2*0)
== even (0) (by math)
==

Another example

Theorem: For all natural numbers n, even(2*n) == true.

```
let rec even n =
  match n with
    | 0 -> true
    | 1 -> false
    | n -> even (n-2)
```

Recall: Every natural number n is either 0 or k+1, where k is also a natural number.

Case: n == 0:

```
   even (2*0)
== even (0)                                                (by math)
== match 0 of (0 -> true | 1 -> false | n -> even (n-2))   (by eval even)
== true                                                    (by evaluation)
```

Another example

Theorem: For all natural numbers n,

even(2*n) == true.

Recall: Every natural number n is either 0 or k+1, where k is also a natural number.

Case: n == k+1:   IH: even(2*k)==true

    even (2*(k+1))

==

```
let rec even n =
  match n with
    | 0 -> true
    | 1 -> false
    | n -> even (n-2)
```

Another example

Theorem: For all natural numbers n, even(2*n) == true.

Recall: Every natural number n is either 0 or k+1, where k is also a natural number.

```
let rec even n =
  match n with
   | 0 -> true
   | 1 -> false
   | n -> even (n-2)
```

Case: n == k+1: IH: even(2*k)==true

even (2*(k+1))
== even (2*k+2) (by math)
==

Another example

Theorem: For all natural numbers n,

even(2*n) == true.

Recall: Every natural number n is either 0 or k+1, where k is also a natural number.

```
let rec even n =
  match n with
    | 0 -> true
    | 1 -> false
    | n -> even (n-2)
```

Case: n == k+1   IH: even(2*k)==true

```
   even (2*(k+1))
== even (2*k+2)                                             (by math)
== match 2*k+2 with (0 -> true | 1 -> false | n -> even (n-2))   (by eval even)
== even ((2*k+2)-2)                                         (by evaluation)
== even (2*k)                                               (by math)
```

Another example

Theorem: For all natural numbers n, even(2*n) == true.

Recall: Every natural number n is either 0 or k+1, where k is also a natural number.

```
let rec even n =
  match n with
    | 0 -> true
    | 1 -> false
    | n -> even (n-2)
```

Case: n == k+1 IH: even(2*k)==true

```
   even (2*(k+1))
== even (2*k+2)                                        (by math)
== match 2*k+2 with (0 -> true | 1 -> false | n -> even (n-2))   (by eval even)
== even ((2*k+2)-2)                                    (by evaluation)
== even (2*k)                                          (by math)
== true                                                (by IH)
QED.
```

Template for Inductive Proofs on Natural Numbers

Theorem: For all natural numbers n, property of n.

Proof: By induction on natural numbers n.

proof methodology.
write this down.

Case: n == 0:
...

Case: n == k+1: IH: ...(k)...
...

justifications to use:

- simple math
- eval, reverse eval, "by def"
- IH

cases must cover all natural numbers

Template for Inductive Proofs on Natural Numbers

Theorem: For all natural numbers n, property of n.

Proof: By induction on natural numbers n.

Case: n == 0:

...

Case: n == k+1: IH: ...(k)...

...

cases must cover all natural numbers

Note there are other ways to cover all natural numbers:

- eg: case for 0, case for 1, case for k+2

PROOFS ABOUT LIST-PROCESSING FUNCTIONS

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 induction 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

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:

IH: add_all (add_all tl a) b == add_all tl (a+b)

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:
      IH: add_all (add_all tl a) b == add_all tl (a+b)
    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:

IH: add_all (add_all tl a) b == add_all tl (a+b)

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:

IH: add_all (add_all tl a) b == add_all tl (a+b)

```
   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:

IH: add_all (add_all tl a) b == add_all tl (a+b)

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:

IH: add_all (add_all tl a) b == add_all tl (a+b)

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:

IH: ...(tl)...

...

cases must cover all lists

Note there are other ways to cover all lists:

- eg: case for [], case for x1::[], case for x1::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 the constructors of ty.

Case: x == A(…):
… IH ? *[zero or more induction hyps]*
Case: x == B(…):
… IH ? *[zero or more induction hyps]*
Case: x == C(…):
… IH ? *[zero or more induction hyps]*
Case: x == D:
…

cases must cover all the constructors of the datatype

SUMMARY

Summary

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 in to cases
- when programs that use recursion on smaller values, their proofs appeal to the inductive hypothesis on smaller values

Key proof ideas:

- two expressions that evaluate to the same value are equal
- substitute equals for equals
- use calculation (evaluation) to reason about simple equalities
- use well-established axioms about primitives (+, -, %, etc)
- use proof by induction to prove correctness of recursive functions