Did I get it right?

COS 326
David Walker
Princeton University

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

slides copyright 2013-2015 David Walker and Andrew W. Appel
permission granted to reuse these slides for non-commercial educational purposes
“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”
“Expressions always terminate”

Two key concepts:

– A **valuable expression**
  - an expression that always terminates (without side effects) and produces a value
– A **total function** with type $t1 \rightarrow t2$
  - a function that terminates on all arguments with type $t1$, 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 functions are total and expressions are valuable. (Such facts can typically be proven by induction.)
We'll prove properties of OCaml expressions, starting with equivalence properties:

**Theorem:** easy 1 20 30 == 50

Let

```
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

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

**Theorem:**
for all lists xs, ys,

\(\text{length (cat xs ys)} == \text{length xs} + \text{length ys}\)

Let

```
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! Proofs usually follow the structure of programs.
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
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
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

Idea 2: A fundamental proof principle.

if two expressions $e_1$ and $e_2$ are equal
and we have a third complicated expression $\text{FOO} (x)$
then $\text{FOO}(e_1)$ is equal to $\text{FOO} (e_2)$

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

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]
EASY EXAMPLES
Easy Examples

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

Given: \text{let easy x y z = x * (y + z)}
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
Most of our proofs will use what we know about the substitution model of evaluation. Eg:

Given: \[ \text{let easy } x \ y \ z = x * (y + z) \]

Theorem: \[ \text{easy } 1 \ 20 \ 30 = 50 \]

Proof:
\[ \text{easy } 1 \ 20 \ 30 \] (left-hand side of equation)
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)
Most of our proofs will use what we know about the substitution model of evaluation. Eg:

**Given:** \[ \text{let easy} \ x \ y \ z = x \ast (y + z) \]

**Theorem:** \[ \text{easy} \ 1 \ 20 \ 30 \ == \ 50 \]

**Proof:**

\[
\begin{align*}
\text{easy} \ 1 \ 20 \ 30 & \quad \text{(left-hand side of equation)} \\
== \ 1 \ast (20 + 30) & \quad \text{(by evaluating easy 1 step)} \\
== 50 & \quad \text{(by math)} \\
\end{align*}
\]

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

**Given:** \( \text{let easy } x \ y \ z = x \ast (y + z) \)

**Theorem:** easy 1 20 30 == 50

**Proof:**
- easy 1 20 30 (left-hand side of equation)
- == 1 \ast (20 + 30) (by evaluating easy 1 step)
- == 50 (by math)
- QED.

facts go on the left

justifications on the right

notice the 2-column proof style
We can use *symbolic values* 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)
We can use *symbolic values* 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:**

\[
\text{easy 1 n m} \quad \quad \text{(left-hand side of equation)} \\
== 1 * (n + m) \quad \quad \text{(by evaluating easy)}
\]
We can use *symbolic values* 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:

\[
\begin{align*}
\text{easy } 1 \ n \ m & \quad \text{(left-hand side of equation)} \\
== \ 1 \ * \ (n + m) & \quad \text{(by evaluating easy)} \\
== \ n + m & \quad \text{(by math)} \\
\end{align*}
\]

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

**Given:** \(\text{let easy } x \ y \ z = x \times (y + z)\)

**Theorem:** for all integers \(n, m, k\), easy \(k \ n \ m \equiv \text{easy } k \ m \ n\)

**Proof:**

\[\text{easy } k \ n \ m \quad \text{(left-hand side of equation)}\]
Easy Examples

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

Given: \[
\text{let easy } x y z = x \times (y + z)
\]

Theorem: *for all integers* \( n, m, k \), easy \( k n m \) == easy \( k m n \)

Proof:
\[
\text{easy } k n m = k \times (n + m) \quad \text{(left-hand side of equation)}
\]
\[
\text{== } k \times (n + m) \quad \text{(by evaluating easy)}
\]
We can use *symbolic values* 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:**

\[
\begin{align*}
\text{easy } k \text{ n m} & \quad \text{(left-hand side of equation)} \\
== k \ast (n + m) & \quad \text{(by evaluating easy)} \\
== k \ast (m + n) & \quad \text{(by math, subst of equals for equals)}
\end{align*}
\]

I'm not going to mention this from now on.
Easy Examples

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

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

*Theorem:*  
for all integers \(n, m, k\), easy \(k \ast n \ast m\) == easy \(k \ast m \ast n\)

*Proof:*  
easy \(k\ n\ m\) \hspace{1cm} \text{(left-hand side of equation)}

== \(k \ast (n + m)\) \hspace{1cm} \text{(by evaluating easy)}

== \(k \ast (m + n)\) \hspace{1cm} \text{(by math)}

== easy \(k\ m\ n\) \hspace{1cm} \text{(by evaluating easy)}

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

Given: \( \text{let easy } x, y, z = x * (y + z) \)

Theorem: *for all integers* \( n, m, k \), easy \( k n m \) == easy \( k m n \)

Proof:

\[
\begin{align*}
\text{easy } k n m & \quad (\text{left-hand side of equation}) \\
== k * (n + m) & \quad (\text{by def of easy}) \\
== k * (m + n) & \quad (\text{by math}) \\
== \text{easy } k m n & \quad (\text{by def of easy})
\end{align*}
\]

QED.

---

substitution/evaluating/“unfolding” 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:

\[
\text{easy x y (k+1)}
\]
\[
== x * (y + (k+1)) \quad \text{(by evaluation of easy .... I hope)}
\]

However, that is not how O’Caml evaluation works. O’Caml evaluates it’s arguments to a \textit{value} first, and then calls the function.

Don’t worry: if you know that the expression \textit{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.

\textit{To be rigorous, you should prove it will evaluate to a value, not just guess ... typically we will take this for granted ...}
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:

```javascript
let const x = 7

const ( n / 0 ) == 7   (By careless, wrong! evaluation of const)
```
An Aside: Symbolic Evaluation

An interesting example:

```javascript
let const x = 7
```

c\(\text{const } ( n \div 0 ) == 7 \) (By careless, wrong! evaluation of const)

- \(n \div 0\) raises an exception
- so \(\text{const } (n \div 0)\) 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:

```javascript
let const x = 7
```

\[ \text{const } (n / 0) \equiv 7 \]  (By *careless, wrong!* evaluation of const)

**what to remember:**

\[ f(e) \equiv \text{body_of_f_with_e_substituted_for_f_parameter} \]

whenever \( e \) evaluates to a value (not an exception or infinite loop)
Summary so far: Proof by simple calculation

• Some proofs are very easy and can be done by:
  – unfolding 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
Theorem: For all natural numbers $n$, $\text{exp}(n) = 2^n$.

```ocaml
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.

let rec exp n =
  match n with
  | 0 -> 1
  | n -> 2 * exp (n-1)
Theorem: For all natural numbers n, 
\( \text{exp}(n) = 2^n \).

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: \( n = 0 \):

\( \text{exp} \, 0 \)
Theorem: For all natural numbers n,
\( \text{exp}(n) = 2^n \).

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: \( n = 0 \):
\[
\text{exp } 0 = \text{match } 0 \text{ with } \begin{cases} 0 \rightarrow 1 \\ n \rightarrow 2 \times \text{exp}(n-1) \end{cases} \quad \text{(by unfolding exp)}
\]
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: \( n = 0 \):
\[
\begin{align*}
\exp 0 &= \text{match } 0 \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \ast \exp (n-1) \\
&= 1 \\
&= 2^0
\end{align*}
\]
(by unfolding \( \exp \))
(by evaluating match)
(by math)
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:
\[ \exp(k+1) \]
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 unfolding exp)
Theorem: For all natural numbers \( n \),
\[
\text{exp}(n) = 2^n.
\]

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: \( n = k+1 \):

\[
\text{exp} (k+1)
\]
\[
= \text{match} (k+1) \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \ast \text{exp} (n-1) \quad \text{(by unfolding exp)}
\]
\[
= 2 \ast \text{exp} (k+1-1) \quad \text{(by evaluating case)}
\]
A problem

Theorem: For all natural numbers n,

\[ \text{exp}(n) = 2^n. \]

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: \( n = k+1 \):

\[
\begin{align*}
\text{exp}(k+1) & = \text{match } (k+1) \text{ with } 0 \to 1 \mid n \to 2 \times \text{exp}(n-1) \\
& = 2 \times \text{exp}(k+1 - 1) \quad \text{(by evaluating case)} \\
& = ?? \quad \text{(by unfolding exp)}
\end{align*}
\]
Theorem: For all natural numbers n,

\[ \text{exp}(n) = 2^n. \]

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: \( n = k+1 \):

\[ \text{exp}(k+1) \]
\[ = \text{match (k+1) with 0 -> 1 | n -> 2 * exp (n -1)} \] (by unfolding exp)
\[ = 2 * \text{exp}(k+1 - 1) \] (by evaluating case)
\[ = 2 * (\text{match (k+1-1) with 0 -> 1 | n -> 2 * exp (n -1)}) \] (by unfolding exp)
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: \( n = k+1 \):
\[
\exp(k+1)
\]

\[
= \text{match } (k+1) \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \times \exp(n-1) \quad \text{(by unfolding } \exp) \\
= 2 \times \exp(k+1-1) \quad \text{(by evaluating case)} \\
= 2 \times (\text{match } (k+1-1) \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \times \exp(n-1)) \quad \text{(by unfolding } \exp) \\
= 2 \times (2 \times \exp((k+1) - 1 - 1)) \quad \text{(by evaluating case)}
\]
A problem

Theorem: For all natural numbers n, 
\( \text{exp}(n) = 2^n \).

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: \( n = k+1 \):

\[
\begin{align*}
\text{exp}(k+1) \\
&= \text{match} \ (k+1) \ \text{with} \\
&\quad | \ 0 \rightarrow 1 \\
&\quad | \ n \rightarrow 2 \star \text{exp}(n-1) \\
&= 2 \star \text{exp}(k+1-1) \\
&= 2 \star \left( \text{match} \ (k+1-1) \ \text{of} \\
&\quad | \ 0 \rightarrow 1 \\
&\quad | \ n \rightarrow 2 \star \text{exp}(n-1) \right) \\
&= 2 \star \left( 2 \star \text{exp} \ ((k+1)-1-1) \right) \\
&= \ldots \text{we aren't making progress} \ldots \text{just unrolling the loop forever} \ldots
\end{align*}
\]
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 *inductive 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 $x::xs$)
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 unfolding exp)
== 2 * exp (k+1 - 1) (by evaluating case)
Theorem: For all natural numbers n, 
\[ \text{exp}(n) == 2^n. \]

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: \( n == k+1: \)

\[
\begin{align*}
\text{exp} \ (k+1) &= \text{match} \ (k+1) \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 * \text{exp} \ (n-1) \\
&= 2 * \text{exp} \ (k+1 - 1) \\
&= 2 * \text{exp} \ (k)
\end{align*}
\]

(by unfolding exp) (by evaluating case) (by math)
**Theorem:** For all natural numbers n, 
\[ \text{exp}(n) = 2^n. \]

**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:
\[
\begin{align*}
\text{exp}(k+1) \\
= \text{match } (k+1) \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \ast \text{exp}(n-1) \\
= 2 \ast \text{exp}(k+1 - 1) \\
= 2 \ast \text{exp}(k) \\
= 2 \ast 2^k
\end{align*}
\]
(by unfolding exp) (by evaluating case) (by math) (by IH!)
**Theorem:** For all natural numbers $n$, 
\[ \text{exp}(n) = 2^n. \]

**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:** $n = k+1$:

\[
\begin{align*}
\text{exp}(k+1) &= \text{match } (k+1) \text{ with }\quad 0 \rightarrow 1 \mid n \rightarrow 2 \times \text{exp} (n - 1) \\
&= 2 \times \text{exp} (k+1 - 1) \\
&= 2 \times \text{exp} (k) \\
&= 2 \times 2^k \\
&= 2^{(k+1)}
\end{align*}
\]

(by unfolding $\text{exp}$)  
(by evaluating case)  
(by math)  
(by IH!)  
(by math) 
QED!
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:** \( n == 0 \):  
\[
\ldots
\]

**Case:** \( n == k+1 \):  
\[
\ldots
\]

let rec even n = match n with 
| 0 -> true 
| 1 -> false 
| n -> even (n-2)
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 = 
let rec even n =
match n with
| 0 -> true
| 1 -> false
| n -> even (n-2)
Theorem: For all natural numbers \( n \),
\[
even(2*n) == \text{true}.
\]

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

Case: \( n == 0 \):
\[
even (2*0)
\]
\[
== even (0)
\]
\[
== \quad \text{(by math)}
\]
Another example

**Theorem:** For all natural numbers \( n \),
\[
\text{even}(2*n) = \text{true}.
\]

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

**Case:** \( n == 0 \):
\[
\begin{align*}
even (2*0) & \quad \text{(by math)} \\
== \text{even} (0) & \quad \text{(by def of even)} \\
== \text{match 0 of (0 -> true | 1 -> false | n -> even (n-2))} & \quad \text{(by evaluation)} \\
== \text{true}
\end{align*}
\]
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:  
    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 \( n \),

\[ \text{even}(2*n) == \text{true}. \]

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

**Case:** \( n == k+1: \)

\[
\begin{align*}
\text{even } (2*(k+1)) \\
== \text{even } (2*k+2) \\
==
\end{align*}
\]

(by math)

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

Theorem: For all natural numbers \( n \), 
\[ \text{even}(2 \times n) = \text{true}. \]

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

Case: \( n = k+1 \): 
\[
\begin{align*}
\text{even} \ (2 \times (k+1)) &= \text{even} \ (2 \times k + 2) \\
&= \text{match} \ 2 \times k + 2 \ \text{of} \ (0 \rightarrow \text{true} \ | \ 1 \rightarrow \text{false} \ | \ n \rightarrow \text{even} \ (n-2)) \\
&= \text{even} \ ((2 \times k + 2) - 2) \\
&= \text{even} \ (2 \times k)
\end{align*}
\]
(by math) 
(by def of even) 
(by evaluation) 
(by math)

let rec even n = 
match n with 
| 0 -> true 
| 1 -> false 
| n -> even (n-2)
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:
   even (2*(k+1))
== even (2*k+2)
== match 2*k+2 of (0 -> true | 1 -> false | n -> even (n-2))
== even ((2*k+2)-2)
== even (2*k)
== true
QED.
Template for Inductive Proofs on Natural Numbers

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

**Proof:** By induction on natural numbers $n$.

**Case:** $n == 0$:

... write this down.

**Case:** $n == k+1$:

... justifications to use:

- simple math
- evaluation, reverse evaluation
- IH

cases must cover all natural numbers

proof methodology.
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: \( n == k+1 \):

\[ ... \]

Note there are other ways to cover all natural numbers:
- eg: case for 0, case for 1, case for \( k+2 \)

cases must cover all natural numbers
PROOFS ABOUT LIST-PROCESSORS
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
Theorem: For all lists $xs$ and $ys$,
\[
\text{length(cat } xs \ ys) = \text{length } xs + \text{length } ys
\]

Proof strategy:

- Proof by induction on the list $xs$? or 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
  | head::tail -> head :: cat tail 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
Proofs About Lists

Theorem: For all lists \( xs \) and \( ys \),

\[
\text{length(cat} \; xs \; ys) = \text{length} \; xs + \text{length} \; ys
\]

Proof strategy:

• Proof by induction on the list \( xs \)
  
  – recall, a list may be of these two things:
    
    • \([\,]\) (the empty list)
    
    • \( \text{hd}::\text{tl} \) (a non-empty list, where \( \text{tl} \) is shorter)
  
  – a proof must cover both cases: \([\,]\) and \( \text{hd}::\text{tl} \)
  
  – in the second case, you will often use the inductive hypothesis
    on the smaller list \( \text{tl} \)
  
  – otherwise as before:
    
    • use folding/unfolding 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 $ys$,

$$\text{length}(\text{cat } xs \text{ ys}) = \text{length } xs + \text{length } ys$$

**Proof:** By induction on $xs$.

```plaintext
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 $ys$,

$$\text{length}(\text{cat } xs \; ys) = \text{length } xs + \text{length } ys$$

**Proof:** By induction on $xs$.

case $xs = []$:

$$\text{length (cat } [] \; ys)$$ (LHS of theorem)

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

let rec cat $xs1 \; xs2 =$
match $xs1$ with
| [] -> $xs2$
| $\text{hd}::\text{tl}$ -> $\text{hd} :: \text{cat } \text{tl} \; xs2$
Proofs About Lists

**Theorem:** For all lists \(xs\) and \(ys\),

\[
\text{length(cat } xs \text{ ys)} = \text{length } xs + \text{length } ys
\]

**Proof:** By induction on \(xs\).

- **case** \(xs = [ ]\):
  - \(\text{length (cat [ ] ys)}\) \quad \text{(LHS of theorem)}
  - \(\text{length } ys\) \quad \text{(evaluate cat)}

```ml
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`,

\[ \text{length(cat } xs \ ys) = \text{length } xs + \text{length } ys \]

**Proof:** By induction on `xs`.

case `xs = []`:

\[
\begin{align*}
\text{length (cat } [ ] \ ys) &= \text{length } ys & \text{(LHS of theorem)} \\
= 0 + (\text{length } ys) &= \text{(evaluate cat)} \\
= 0 + (\text{length } ys) &= \text{(arithmetic)}
\end{align*}
\]
Proofs About Lists

Theorem: For all lists $xs$ and $ys$,
\[
\text{length(cat } xs \; ys) = \text{length } xs + \text{length } ys
\]

Proof: By induction on $xs$.

case $xs = []$:
\[
\begin{align*}
\text{length (cat } [ ] \; ys) & \quad \text{(LHS of theorem)} \\
= \text{length } ys & \quad \text{(evaluate cat)} \\
= 0 + (\text{length } ys) & \quad \text{(arithmetic)} \\
= (\text{length } [ ]) + (\text{length } ys) & \quad \text{(fold length)}
\end{align*}
\]

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 \(ys\),
\[
\text{length(cat } xs \text{ ys) = length } xs + \text{ length } ys
\]

**Proof:** By induction on \(xs\).

```plaintext
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 \( ys \),

\[
\text{length}(\text{cat } xs \ ys) = \text{length } xs + \text{length } ys
\]

**Proof:** By induction on \( xs \).

\[
\text{case } xs = \text{hd}::\text{tl} \\
\text{IH: length (cat tl ys) = length tl + length ys}
\]

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 $ys$,
\[
\text{length(\text{cat} \; xs \; ys)} = \text{length} \; xs + \text{length} \; ys
\]

**Proof:** By induction on $xs$.

case $xs = \text{hd}::\text{tl}$

IH: $\text{length (\text{cat} \; \text{tl} \; ys)} = \text{length} \; \text{tl} + \text{length} \; ys$

\[
\text{length (\text{cat} \; (\text{hd}::\text{tl}) \; ys)} \quad \text{(LHS of theorem)}
\]

==

```
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,  
\[
\text{length}(\text{cat } \text{xs } \text{ys}) = \text{length } \text{xs} + \text{length } \text{ys}
\]

**Proof:** By induction on xs.

case xs = hd::tl
  IH: \(\text{length } (\text{cat } \text{tl } \text{ys}) = \text{length } \text{tl} + \text{length } \text{ys}\)

  \[
  \text{length } (\text{cat } (\text{hd}::\text{tl}) \text{ys}) = (\text{LHS of theorem})
  \]

  \[
  = \text{length } (\text{hd} :: (\text{cat } \text{tl} \text{ys})) \quad \text{(evaluate cat, take 2\textsuperscript{nd} branch)}
  \]

  \[
  = \text{let rec length xs =}
  \]

let rec cat xs1 xs2 =  
match xs1 with
  | [] -> 0
  | hd::tl -> hd :: cat tl xs2

match xs with
  | [] -> 0
  | x::xs -> 1 + length xs
Proofs About Lists

**Theorem:** For all lists $xs$ and $ys$,

$$\text{length}(\text{cat } xs \ ys) = \text{length } xs + \text{length } ys$$

**Proof:** By induction on $xs$.

\[
\begin{align*}
\text{case } xs &= \text{hd}::\text{tl} \\
\text{IH: length } (\text{cat } \text{tl } ys) &= \text{length } \text{tl} + \text{length } ys
\end{align*}
\]

\[
\begin{align*}
\text{length } (\text{cat } (\text{hd}::\text{tl}) \ ys) &= (\text{LHS of theorem}) \\
&= \text{length } (\text{hd} :: (\text{cat } \text{tl } ys)) \\
&= 1 + \text{length } (\text{cat } \text{tl } ys) \\
&= \\
\end{align*}
\]

\[
\begin{align*}
\text{let rec length } xs &= \\
&= \text{match } xs \text{ with} \\
&\quad | [] \rightarrow 0 \\
&\quad | x::xs \rightarrow 1 + \text{length } xs
\end{align*}
\]

\[
\begin{align*}
\text{let rec cat } xs1 \ xs2 &= \\
&= \text{match } xs1 \text{ with} \\
&\quad | [] \rightarrow xs2 \\
&\quad | \text{hd}::\text{tl} \rightarrow \text{hd} :: \text{cat } \text{tl } xs2
\end{align*}
\]
Proofs About Lists

**Theorem:** For all lists $xs$ and $ys$,

\[ \text{length}(\text{cat} \; xs \; ys) = \text{length} \; xs + \text{length} \; ys \]

**Proof:** By induction on $xs$.

\[
\text{case } xs = \text{hd}:\text{tl} \\
\text{IH: } \text{length} \; (\text{cat} \; \text{tl} \; ys) = \text{length} \; \text{tl} + \text{length} \; ys
\]

\[
\text{length} \; (\text{cat} \; (\text{hd}:\text{tl}) \; ys) \\
== \text{length} \; (\text{hd} :: (\text{cat} \; \text{tl} \; ys)) \\
== 1 + \text{length} \; (\text{cat} \; \text{tl} \; ys) \\
== 1 + (\text{length} \; \text{tl} + \text{length} \; ys) \\
==
\]

\[
\text{let rec length xs =} \\
\quad 
\begin{cases} 
\quad | [] -> 0 \\
\quad | x::xs -> 1 + \text{length} \; xs 
\end{cases}
\]

\[
\text{let rec cat xs1 xs2 =} \\
\quad 
\begin{cases} 
\quad | [] -> xs2 \\
\quad | hd::tl -> hd :: cat tl xs2 
\end{cases}
\]
Proofs About Lists

**Theorem:** For all lists $xs$ and $ys$,

$$\text{length(cat } xs \text{ ys)} = \text{length } xs + \text{length } ys$$

**Proof:** By induction on $xs$.

\[
\begin{align*}
\text{case } xs &= \text{hd}::\text{tl} \\
\text{IH: } \text{length } (\text{cat } \text{tl } ys) &= \text{length } \text{tl} + \text{length } ys
\end{align*}
\]

\[
\begin{align*}
\text{length } (\text{cat } (\text{hd}::\text{tl}) \text{ ys}) &= \text{LHS of theorem} \\
\text{== length } (\text{hd } :: (\text{cat } \text{tl } ys)) &= \text{evaluate cat, take } 2^{\text{nd}} \text{ branch} \\
\text{== 1 + length } (\text{cat } \text{tl } ys) &= \text{evaluate length, take } 2^{\text{nd}} \text{ branch} \\
\text{== 1 + (length } \text{tl } + \text{length } ys) &= \text{by IH} \\
\text{== length } (\text{hd}::\text{tl}) + \text{length } ys &= \text{reparenthesizing and evaling length in reverse we have RHS with hd}::\text{tl for xs}
\end{align*}
\]

\[\text{case done!}\]

\[
\begin{align*}
\text{let rec length } xs &= \\
&\text{match } xs \text{ with} \\
&| [] -> 0 \\
&| x::xs -> 1 + \text{length } xs
\end{align*}
\]

\[
\begin{align*}
\text{let rec cat } xs1 \text{ xs2} &= \\
&\text{match } xs1 \text{ with} \\
&| [] -> xs2 \\
&| \text{hd}::\text{tl} -> \text{hd } :: \text{cat } \text{tl } xs2
\end{align*}
\]
Be careful with the Induction Hypothesis!

**Theorem:** For all lists xs and ys,

\[ \text{length}(\text{cat } xs \text{ ys}) = \text{length } xs + \text{length } ys \]

**Proof:** By induction on xs.

\[
\begin{align*}
\text{case } xs &= \text{hd} :: \text{tl} \\
\text{IH: } \text{length } (\text{cat } \text{tl } ys) &= \text{length } \text{tl} + \text{length } ys
\end{align*}
\]

\[
\begin{align*}
\text{length } (\text{cat } (\text{hd} :: \text{tl}) \text{ ys}) &= \text{length } (\text{hd } :: (\text{cat } \text{tl } ys)) \\
&= 1 + \text{length } (\text{cat } \text{tl } ys) \\
&= 1 + (\text{length } \text{tl} + \text{length } ys) \\
&= \text{length } (\text{hd} :: \text{tl}) + \text{length } ys
\end{align*}
\]

\[
\begin{align*}
\text{length } (\text{cat } (\text{hd} :: \text{tl}) \text{ ys}) &= 1 + \text{length } (\text{cat } \text{tl } ys) \\
&= 1 + (\text{length } \text{tl} + \text{length } ys) \\
&= \text{length } (\text{hd} :: \text{tl}) + \text{length } ys
\end{align*}
\]

Induction hypothesis is a function of one variable (in this case, xs)

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

In your proofs, it should be really obvious
- 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 smaller 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
Be careful with the Induction Hypothesis!

**Theorem:** For all lists \(xs\) and \(ys\),

\[
\text{length}(\text{cat} \; xs \; ys) = \text{length} \; xs + \text{length} \; ys
\]

**Proof:** By induction on \(xs\).

Induction hypothesis is 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, we could do induction on pairs of naturals \((x, y)\) where pairs are ordered lexicographically. ie:

\[
(x1, y1) > (x2, y2) \iff x1 > x2 \text{ or } (x1 = x2 \text{ and } y1 > y2)
\]
Another List example

Theorem: For all lists xs,
\[
\text{add\_all}(\text{add\_all}\ xs\ a)\ b\ ==\ \text{add\_all}\ xs\ (a+b)
\]

Proof: By induction on xs.

case xs = []:

\[
\text{add\_all}(\text{add\_all}\ []\ a)\ b\quad \text{(LHS of theorem)}
\]

==

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

**Theorem:** For all lists $xs$, 

$$\text{add\_all}\ (\text{add\_all}\ xs\ a)\ b\ ==\ \text{add\_all}\ xs\ (a+b)$$

**Proof:** By induction on $xs$.

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

*Case $xs = [ ]$:*

$$\text{add\_all}\ (\text{add\_all}\ [ ]\ a)\ b\quad (\text{LHS of theorem})$$

$$==\ \text{add\_all}\ [ ]\ b\quad (\text{by evaluation of }\ add\_all)$$

$$==$$
Another List example

Theorem: For all lists \(xs\),

\[
\text{add\_all} (\text{add\_all} \ xs \ a) \ b \equiv \text{add\_all} \ xs \ (a+b)
\]

Proof: By induction on \(xs\).

case \(xs = [\ ]\):

\[
\text{add\_all} (\text{add\_all} \ [\ ] \ a) \ b
\equiv \text{add\_all} \ [\ ] \ b
\equiv [\ ]
\equiv
\]

let rec add\_all \ xs \ c =
    match \ xs \ with
    | [ ] -> [ ]
    | \ hd::tl \ -> (hd+c)::add\_all \ tl \ c
Theorem: For all lists \(xs\),
\[
\text{add\_all\ (add\_all\ \text{xs}\ a)\ b\ ==\ add\_all\ \text{xs}\ (a+b)}
\]

Proof: By induction on \(xs\).

\[
\begin{align*}
\text{case } xs &= [ ]: \\
\quad \text{add\_all\ (add\_all\ [ ]\ a)\ b} &\quad \text{(LHS of theorem)} \\
&=\ add\_all\ [ ]\ b &\quad \text{(by evaluation of add\_all)} \\
&= [ ] &\quad \text{(by evaluation of add\_all)} \\
&=\ add\_all\ [ ]\ (a + b) &\quad \text{(by evaluation of add\_all)}
\end{align*}
\]

let rec add_all xs c =
    match xs with
        | [ ] -> [ ]
        | hd::tl -> (hd+c)::add_all tl c
Theorem: For all lists \( xs \),
\[
\text{add\_all \ (\text{add\_all \ } xs \ \ a) \ \ b \ \ == \ \ \text{add\_all \ } xs \ \ (a+b)
\]

Proof: By induction on \( xs \).

\[
\text{case } xs = \text{hd :: tl:}
\]
\[
\text{add\_all \ (\text{add\_all \ } (\text{hd :: tl}) \ \ a) \ \ b} \quad \text{(LHS of theorem)}
\]

\[
==
\]

let rec add\_all \ xs \ c =
match \ xs \ with
| [ ] -> [ ]
| \ text{hd::tl} \ -> (\text{hd+c)::add\_all \ tl \ c}
Another List example

Theorem: For all lists \( xs \),
\[
\text{add\_all} \ (\text{add\_all} \ xs \ a) \ b \ == \ \text{add\_all} \ xs \ (a+b)
\]

Proof: By induction on \( xs \).

case \( xs = \text{hd} :: \text{tl} \):

\[
\text{add\_all} \ (\text{add\_all} \ (\text{hd} :: \text{tl}) \ a) \ b \ == \text{add\_all} \ ((\text{hd}+a) :: \text{add\_all} \ \text{tl} \ a) \ b
\]

(\text{LHS of theorem})

(by eval inner add\_all)

==

\[
\text{let rec add\_all} \ xs \ c =
\text{match} \ xs \ \text{with}
\ |
\ [] \ -> \ []
\ |
\ \text{hd} :: \text{tl} \ -> \ (\text{hd}+c) :: \text{add\_all} \ \text{tl} \ c
\]

87
Another List example

Theorem: For all lists \(xs\),
\[
\text{add\_all (add\_all \(xs\) \(a\)) \(b\) == add\_all \(xs\) (\(a+b\))}
\]

Proof: By induction on \(xs\).

case \(xs = \text{hd :: tl}\):

\[
\begin{align*}
\text{add\_all (add\_all (\text{hd :: tl}) \(a\)) \(b\)} & \quad \text{(LHS of theorem)} \\
== \text{add\_all ((hd+a) :: add\_all \(tl\) \(a\)) \(b\)} & \quad \text{(by eval inner add\_all)} \\
== (hd+a+b) :: (add\_all (add\_all \(tl\) \(a\)) \(b\)) & \quad \text{(by eval outer add\_all)} \\
== \\
\end{align*}
\]

let rec add_all \(xs\) \(c\) =
match \(xs\) with
| [ ] -> [ ]
| \(\text{hd::tl}\) -> (hd+c)::add_all \(tl\) \(c\)
Another List example

Theorem: For all lists \(xs\),

\[
\text{add\_all} \ (\text{add\_all} \ xs \ a) \ b \ == \ \text{add\_all} \ xs \ (a+b)
\]

Proof: By induction on \(xs\).

\[
\text{case } xs = \text{hd} :: \text{tl}:
\]

\[
\text{add\_all} \ (\text{add\_all} \ (\text{hd} :: \text{tl}) \ a) \ b \quad \text{(LHS of theorem)}
\]
\[
== \ \text{add\_all} \ ((\text{hd}+a) :: \text{add\_all} \ \text{tl} \ a) \ b \quad \text{(by eval inner add\_all)}
\]
\[
== (\text{hd}+a+b) :: (\text{add\_all} (\text{add\_all} \ \text{tl} \ a) \ b) \quad \text{(by eval outer add\_all)}
\]
\[
== (\text{hd}+a+b) :: \text{add\_all} \ \text{tl} (a+b) \quad \text{(by IH)}
\]

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,
\[ \text{add}_\text{all} (\text{add}_\text{all} \text{a} \text{b}) = \text{add}_\text{all} \text{a} (\text{a+b}) \]

Proof: By induction on xs.

case \(\text{xs} = \text{hd} :: \text{tl}\):

\[
\begin{align*}
\text{add}_\text{all} (\text{add}_\text{all} (\text{hd} :: \text{tl}) \text{a} \text{b}) & \quad \text{(LHS of theorem)} \\
\text{== add}_\text{all} ((\text{hd+a}) :: \text{add}_\text{all} \text{tl} \text{a} \text{b}) & \quad \text{(by eval inner add}_\text{all}\text{)} \\
\text{== (hd+a+b) :: (add}_\text{all} (\text{add}_\text{all} \text{tl} \text{a}) \text{b}) & \quad \text{(by eval outer add}_\text{all}\text{)} \\
\text{== (hd+a+b) :: add}_\text{all} \text{tl} (\text{a+b}) & \quad \text{(by IH)} \\
\text{== (hd+(a+b)) :: add}_\text{all} \text{tl} (\text{a+b}) & \quad \text{(associativity of + )}
\end{align*}
\]

let rec add_all xs c =
match xs with
| [ ] -> [ ]
| hd::tl -> (hd+c)::add_all tl c
Theorem: For all lists $xs$, 

$$\text{add\_all\ (add\_all\ xs\ a)\ b} \equiv \text{add\_all\ xs\ (a+b)}$$

Proof: By induction on $xs$.

case $xs = \text{hd :: tl}$:

$$\begin{align*}
\text{add\_all\ (add\_all\ (\text{hd :: tl})\ a)\ b} & \quad \text{(LHS of theorem)} \\
\equiv & \quad \text{add\_all\ ((hd+a) :: add\_all\ tl\ a)\ b} \quad \text{(by eval inner add\_all)} \\
\equiv & \quad (hd+a+b) :: (\text{add\_all\ (add\_all\ tl\ a)\ b}) \quad \text{(by eval outer add\_all)} \\
\equiv & \quad (hd+a+b) :: \text{add\_all\ tl\ (a+b)} \quad \text{(by IH)} \\
\equiv & \quad (hd+(a+b)) :: \text{add\_all\ tl\ (a+b)} \quad \text{(associativity of + )} \\
\equiv & \quad \text{add\_all\ (hd::tl)\ (a+b)} \quad \text{(by (reverse) eval of add\_all)}
\end{align*}$$

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: $xs == \text{hd :: tl}$:
...

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*

```plaintext
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(...):

...  
Case: x == B(...):

...  
Case: x == C(...):

...  
Case: x == D:

...

cases must cover all the constructors of the datatype
SUMMARY
Summary

• Proofs about programs are structured similarly to the programs themselves:
  – types tell you what 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 proof by induction to prove correctness of recursive functions