Did I get it right?

COS 326
Andrew W. Appel
Princeton University

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

slides copyright 2018 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?
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 \( t_1 \rightarrow t_2 \)
  - a function that terminates on all args : \( t_1 \), producing a value of type \( t_2 \)
  - 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.)
We'll prove properties of OCaml expressions, starting with equivalence properties:

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

**Theorem:**
for all natural numbers n,
exp n == 2^n

**Theorem:**
for all lists xs, ys,
length (cat xs ys) == length xs + length ys

**Example**

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

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

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

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.
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
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
More structure:

– when *programming* with lists:
  • \([\ ]\) is often easy
  • \(\text{hd} :: \text{tl}\) often requires a *recursive function call* on \(\text{tl}\)
    – we *assume* our recursive function behaves correctly on \(\text{tl}\)

– when *proving* with lists:
  • \([\ ]\) is often easy
  • \(\text{hd} :: \text{tl}\) often requires appeal to an *induction hypothesis* for \(\text{tl}\)
    – we *assume* our property of interest holds for \(\text{tl}\)

– 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

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

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

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

I have a complicated expression: $\text{bar (foo ( \_\_\_ ))} \times 34$

Then I also know that $\text{bar (foo (2+2))} \times 34 == \text{bar (foo (4))} \times 34$.

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

Given: \hspace{1cm} \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: \( \text{let easy } x \ yi z = x * (y + z) \)

Theorem: \( \text{easy } 1 \ 20 \ 30 == 50 \)

Proof:
\[ \text{easy } 1 \ 20 \ 30 \quad \text{(left-hand side of equation)} \]
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:

   \( \text{easy } 1 \ 20 \ 30 \)  \hspace{1cm} (left-hand side of equation)
   \( == \ 1 \ast (20 + 30) \)  \hspace{1cm} (by evaluating easy 1 step)
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.
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  
== 1 * (20 + 30)  
== 50  
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)
Easy Examples

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

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

**Theorem:** for all integers \( n \) and \( m \), easy 1 \( n \ m \) == \( n + m \)

**Proof:**

\[
\text{easy 1 n m} \quad \text{(left-hand side of equation)}
\]

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 \( n \).
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:**

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

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

**Given:** 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:**

\[
\text{easy } k \ n \ m \quad \text{(left-hand side of equation)}
\]
We can use *symbolic values* in our proofs too. Eg:

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

**Theorem:**  
for all integers n, 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 \; \ast \; (n + m) & \quad \text{(by evaluating } \text{easy})
\end{align*}
We can use *symbolic values* in our proofs too. Eg:

Given:  

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

Theorem:  

for all integers \( n, m, k \), \( \text{easy } k \ n \ m == \text{easy } k \ m \ n \)

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 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:**

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

QED.
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:**

\[
\begin{align*}
\text{easy } k & \ n \ m \\
& = k \ast (n + m) \\
& = k \ast (m + n) \\
& = \text{easy } k \ m \ n
\end{align*}
\]

QED.
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 it’s 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 guess ... but we won’t require you prove that in this class ...
An Aside: Symbolic Evaluation

An interesting example:

```javascript
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
```

\(\text{const ( n / 0 ) interfere with } 7\) (By careless, wrong! evaluation of \text{const})
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:

```plaintext
let rec loop(x:int) = loop x in
const ( loop 0 ) == 7  (By careless, wrong! evaluation of const)

what to remember:

f (e) == body_of_f_with_e_substituted_for_f_parameter

whenever e evaluates to a value (not an exception or infinite loop)
An Aside: Symbolic Evaluation

An interesting example:

```plaintext
let r = ref 0 in let f(i) = (let s = !r in r := s+i; s) const ( f 1 ) == 7 (By evaluation of const?)
```

what to remember:

f (e) == body_of_f_with_e_substituted_for_f_parameter

whenever e evaluates to a value, without side effects, without raising an exception, without infinite loop
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)

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

```
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, 
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
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: \( n = 0 \):
\[
\text{exp} 0 = \text{match } 0 \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \times \text{exp} (n-1) \quad \text{(by eval exp)}
\]
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:
exp 0
== match 0 with 0 -> 1 | n -> 2 * exp (n -1) (by eval exp)
== 1 (by evaluating match)
== 2^0 (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: \( n = k+1 \):
\[ \text{exp} (k+1) \]
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: \( n = k+1 \):
\[
\begin{align*}
\text{exp} \ (k+1) \\
= \text{match} \ (k+1) \ \text{with} \ 0 \to 1 \ | \ n \to 2 \times \text{exp} \ (n-1) \quad \text{(by eval exp)}
\end{align*}
\]
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: \( n = k+1 \):
\[ \exp(k+1) \]
\[ = \text{match } (k+1) \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \ast \exp(n-1) \] (by eval \( \exp \))
\[ = 2 \ast \exp(k+1-1) \] (by evaluating case)
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: \( n = k+1 \):

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

\[
\begin{align*}
\text{exp}(k+1) &= \text{match } (k+1) \text{ with } 0 \to 1 \mid n \to 2 \times \text{exp}(n-1) \quad \text{(by eval exp)} \\
&= 2 \times \text{exp}(k+1-1) \quad \text{(by evaluating case)} \\
&= 2 \times (\text{match } (k+1-1) \text{ with } 0 \to 1 \mid n \to 2 \times \text{exp}(n-1)) \quad \text{(by eval exp)}
\end{align*}
\]
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: \( n = k+1 \):

\[ \begin{align*}
\text{exp} (k+1) & = \text{match } (k+1) \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \times \text{exp} (n-1) \quad \text{(by eval exp)} \\
& = 2 \times \text{exp} (k+1-1) \quad \text{(by evaluating case)} \\
& = 2 \times (\text{match } (k+1-1) \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \times \text{exp} (n-1)) \quad \text{(by eval exp)} \\
& = 2 \times (2 \times \text{exp} ((k+1)-1-1)) \quad \text{(by evaluating case)}
\end{align*} \]
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: \( n = k+1 \):
\[
\begin{align*}
\text{exp}(k+1) &= \text{match } (k+1) \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \cdot \text{exp}(n-1) \quad \text{(by eval exp)} \\
&= 2 \cdot \text{exp}(k+1-1) \quad \text{(by evaluating case)} \\
&= 2 \cdot (\text{match } (k+1-1) \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \cdot \text{exp}(n-1)) \quad \text{(by eval exp)} \\
&= 2 \cdot (2 \cdot \text{exp}((k+1) - 1 - 1)) \quad \text{(by evaluating case)} \\
&= \ldots \text{we aren’t making progress \ldots just unrolling the loop forever \ldots}
\end{align*}
\]
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 eval exp)
== 2 * exp (k+1 - 1) (by evaluating case)
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: \( n = k+1 \):
\[
\begin{align*}
\exp(k+1) & = \text{match } (k+1) \text{ with } \begin{cases} 0 \to 1 \\ n \to 2 \ast \exp(n-1) \end{cases} \\
& = 2 \ast \exp(k+1-1) \\
& = 2 \ast \exp(k) 
\end{align*}
\]
Back to the Proof

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!)
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!
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: \( n == 0 \):
...

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

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

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

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

**Case:** \(n == 0:\)
\[
\text{even } (2*0) ==
\]
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: \( n == 0 \):
   \( \text{even } (2*0) \)
== even (0)
==

let rec even n =
  match n with
  | 0 -> true
  | 1 -> false
  | n -> even (n-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.

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

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:** \( n == k+1: \)
\[ \text{even} (2*(k+1)) \]

```ocaml
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) (by math)
==
Another example

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

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

(by eval even)

(by evaluation)

(by math)
Theorem: For all natural numbers n, 
\( \text{even}(2n) = \text{true} \).

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

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

\[
\text{even} \ (2(k+1)) \\
= \text{even} \ (2k+2) \\
= \text{match} \ 2k+2 \ \text{with} \ (0 \rightarrow \text{true} \ | \ 1 \rightarrow \text{false} \ | \ n \rightarrow \text{even} \ (n-2)) \\
= \text{even} \ ((2k+2)-2) \\
= \text{even} \ (2k) \\
= \text{true} \\
\]

QED.

```plaintext
let rec even n = 
  match n with 
  | 0 -> true 
  | 1 -> false 
  | n -> even (n-2)
```
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:** \( n = k+1 \):
...  

proof methodology. write this down.

justifications to use:
- simple math
- eval, reverse eval, "by def"
- IH

cases must cover all 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 $k+2$
PROOFS ABOUT
LIST-PROCESSING FUNCTIONS
let rec cat xs1 xs2 =
  match xs1 with
  | [] -> xs2
  | hd::tl -> hd :: cat tl xs2

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

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

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

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/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 \(ys\),

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

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

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

```comp
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 $xs = [\; ]$:
  \[
  \text{length}\; (\text{cat}\; [\; ]\; ys) \quad \text{(LHS of theorem)}
  \]

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

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

**case** $xs = [ ]$:

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

Proof: By induction on \(xs\).

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

**case** $xs = [\ ]$:

- $\text{length}\ (\text{cat}\ [\ ]\ ys)$  \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{(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 $ys$,

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

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

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

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.

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

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

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

\[
===
\]

Let \(\text{rec length xs =}
\)

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

Let \(\text{rec cat xs1 xs2 =}
\)

\[
\begin{align*}
\text{match } xs1 \text{ with} \\
| [] &\rightarrow xs2 \\
| \text{hd}::\text{tl} &\rightarrow \text{hd} :: \text{cat} \; \text{tl} \; xs2
\end{align*}
\]
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:** \( length (cat \, tl \, ys) = length \, tl + length \, ys \)

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

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

\[
eq
\]

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

\[
let \, rec \, cat \, xs1 \, xs2 =
match \, xs1 \, with
| \, [] \, -> \, xs2
| \, hd::tl \, -> \, hd :: cat \, tl \, xs2
\]
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: \(\text{length } (\text{cat } \text{tl } ys) = \text{length } \text{tl} + \text{length } ys\)

\[
\begin{align*}
\text{length } (\text{cat } (\text{hd} :: \text{tl}) \ ys) & \quad \text{(LHS of theorem)} \\
= & \quad \text{(evaluate cat, take 2\textsuperscript{nd} branch)} \\
= & \quad \text{(evaluate length, take 2\textsuperscript{nd} branch)} \\
= & \quad \text{length } (\text{cat } \text{tl } ys) + 1 \\
= & \quad \text{length } \text{tl} + \text{length } ys + 1
\end{align*}
\]

```
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:** $\text{length} \left( \text{cat} \, \text{tl} \, ys \right) = \text{length} \, \text{tl} + \text{length} \, ys$

  $$\begin{align*}
  \text{length} \left( \text{cat} \, (\text{hd} :: \text{tl}) \, ys \right) & \quad \text{(LHS of theorem)} \\
  == \text{length} \left( \text{hd} :: (\text{cat} \, \text{tl} \, ys) \right) & \quad \text{(evaluate cat, take 2\textsuperscript{nd} branch)} \\
  == 1 + \text{length} \left( \text{cat} \, \text{tl} \, ys \right) & \quad \text{(evaluate length, take 2\textsuperscript{nd} branch)} \\
  == 1 + (\text{length} \, \text{tl} + \text{length} \, ys) & \quad \text{(by IH)} \\
  == & \quad \\
  \end{align*}$$

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

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

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

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

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

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

a clue: pattern matching on first argument. In the theorem: $\text{cat } xs \; ys$
Hence induction on $xs$. Case split the same way as the program.
Another List example

Theorem: For all lists xs,

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

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 $xs$. 

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

Proof: By induction on \( xs \).

\text{case} \; xs \; = \; [ \; ]:

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

==
**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** $xs = [ ]$:

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

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

$$==$$

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

Proof: By induction on \( xs \).

case \( xs = [ ] \):

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

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 \( xs \).

```plaintext
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 \( xs \).

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

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

\[
= \text{let rec add\_all} \ xs \ c =
\text{match} \ xs \ \text{with}
\begin{cases} 
| [] & \rightarrow [] \\
| \text{hd}::\text{tl} & \rightarrow (\text{hd}+c)::\text{add\_all} \ \text{tl} \ c
\end{cases}
\]
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 \( xs = \text{hd} :: \text{tl} \):

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

(by eval inner add\_all)

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

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

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

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

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

$$\text{add}_\text{all} \ (\text{add}_\text{all} \ (\text{hd} :: \text{tl}) \ a) \ b \quad \text{(LHS of theorem)}$$

$$= \text{add}_\text{all} \ ((\text{hd}+a) :: \text{add}_\text{all} \ \text{tl} \ a) \ b \quad \text{(by eval inner add}_\text{all}})$$

$$= (\text{hd}+a+b) :: (\text{add}_\text{all} \ (\text{add}_\text{all} \ \text{tl} \ a) \ b) \quad \text{(by eval outer add}_\text{all}})$$

$$= \text{add}_\text{all} \ xs \ (a+b)$$
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 \( xs = \text{hd :: tl} \):

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

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 $xs$.

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

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

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

cases must cover all the constructors of the datatype
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 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