Did I get it right?

COS 326
David Walker
Princeton University

http://.../cos326/notes/evaluation.php
http://.../cos326/notes/reasoning.php
We'll prove properties of OCaml expressions, starting with equivalence properties:

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

**Theorem:**
for all natural numbers \( n \),
\[ \text{exp} \ n \ =\ =\ 2^n \]

**Theorem:**
for all lists \( xs \), \( ys \),
\[ \text{length} \ (\text{cat} \ xs \ ys) \ =\ =\ \text{length} \ xs \ +\ \text{length} \ ys \]
• 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

we will use what we learned about OCaml evaluation
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 $\text{FOO}(e_2)$

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: $\text{bar} \ (\text{foo} \ (\_\_\_)) \ * \ 34$

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

*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 \(e_1 == e_2\) then \(e_2 == e_1\)

*(transitivity)* if \(e_1 == e_2\) and \(e_2 == e_3\) then \(e_1 == e_3\)

*(evaluation)* if \(e_1 \to e_2\) then \(e_1 == e_2\).

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

– if \(e_1 == e_2\) then \(e[e_1/x] == e[e_2/x]\)
Most of our proofs will use what we know about the substitution model of evaluation. Eg:

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

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

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

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

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

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

Theorem:  
\[
easy 1 \ 20 \ 30 \ == \ 50
\]

Proof:  
\[
easy 1 \ 20 \ 30 \quad \text{(left-hand side of equation)}
\]
\[
== \ 1 \times (20 + 30) \quad \text{(by evaluating \ easy 1 step)}
\]
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:**

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

QED.
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 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 $m$, $\text{easy } 1 \ n \ m == n + m$

**Proof:**

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

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

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

Proof:

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

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

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

**Proof:**
\[
\text{easy } k \ n \ m \quad \text{(left-hand side of equation)}
\]
\[
== k \ast (n + m) \quad \text{(by evaluating easy)}
\]
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:**

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

I'm not going to mention this from now on.
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:**

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

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

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

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

**Proof:**

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

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:

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

However, that is not how O’Caml evaluation works. O’Caml evaluates it’s arguments to a 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 ... typically we will take this for granted ...
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
```

Constant 

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

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

const ( n / 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)
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

**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,

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

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.

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

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

\[ \text{exp} \ (k+1) \]

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

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

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

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

(by unfolding \( \exp \))

(by evaluating case)

(by unfolding \( \exp \))

(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: \( n = k+1 \):
\[
\begin{align*}
\text{exp}(k+1) &= \text{match}(k+1) \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \times \text{exp}(n-1) \quad \text{(by unfolding exp)} \\
&= 2 \times \text{exp}(k+1-1) \quad \text{(by evaluating case)} \\
&= 2 \times (\text{match}(k+1-1) \text{ of } 0 \rightarrow 1 \mid n \rightarrow 2 \times \text{exp}(n-1)) \quad \text{(by unfolding exp)} \\
&= 2 \times (2 \times \text{exp}((k+1)-1-1)) \quad \text{(by evaluating case)} \\
&= \ldots \text{we aren’t making progress \ldots just unrolling the loop forever \ldots}
\end{align*}
\]
When proving theorems about recursive functions, we usually need to use \textit{induction}.  

- In inductive proofs, in a case for object \(X\), we assume that the theorem holds \textit{for all objects smaller than} \(X\)  
  - this assumption is called the \textit{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: \( n = k+1 \):
\[
\exp(k+1)
= \text{match } (k+1) \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \times \exp(n-1) \quad \text{(by unfolding } \exp) \\
= 2 \times \exp(k+1 - 1) \quad \text{(by evaluating case)}
\]
**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:** \( n = k+1 \):
\[
\begin{align*}
\text{exp} (k+1) \\
= & \text{match} (k+1) \text{ with } 0 \rightarrow 1 \mid n \rightarrow 2 \times \text{exp} (n-1) \quad \text{(by unfolding exp)} \\
= & 2 \times \text{exp} (k+1-1) \quad \text{(by evaluating case)} \\
= & 2 \times \text{exp} (k) \quad \text{(by math)}
\end{align*}
\]
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: \( n = k+1 \):
\[
\begin{align*}
\text{exp}(k+1) &= \text{match } (k+1) \text{ with } 0 \to 1 \mid n \to 2 \times \text{exp}(n-1) \\
&= 2 \times \text{exp}(k+1-1) \\
&= 2 \times \text{exp}(k) \\
&= 2 \times 2^k \\
&= 2^{k+1}
\end{align*}
\]
(by unfolding \( \text{exp} \))
(by evaluating case)
(by math)
(by IH!)
**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:** \( n = k+1: \)

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

(by unfolding \( \text{exp} \))

(by evaluating case)

(by math)

(by IH!)

(by math)

QED!
**Theorem:** For all natural numbers $n$, \( \text{even}(2\times n) == true \).

**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)
**Theorem:** For all natural numbers \( n \), even(\( 2 \times n \)) == 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) 
==
Another example

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

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

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

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

**Case:** \( n == k+1: \)
- \( \text{even } (2*(k+1)) \)

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

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

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

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

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

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

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

**Case:** \( n == k+1 \):
\[
\begin{align*}
even(2*(k+1)) & == even(2*k+2) \\
& == match 2*k+2 \text{ of (0 -> true | 1 -> false | n -> even (n-2))} \\
& == even((2*k+2)-2) \\
& == even(2*k)
\end{align*}
\]
(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)
```
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:** \( n == k+1: \)
\[
even(2*(k+1))
even(2*k+2)
match 2*k+2 of (0 -> true | 1 -> false | n -> even (n-2))
even ((2*k+2)-2)
even (2*k)
true
\]
QED.

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

(by math)
(by def of even)
(by evaluation)
(by math)
(by IH)
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:
... [write this down]

Cases must cover all natural numbers

Justifications to use:
• simple math
• evaluation, reverse evaluation
• IH

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 $k+2$
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
```
Proofs About Lists

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

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

**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

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

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

a clue: pattern matching on first argument.
In the theorem: \(\text{cat } xs \text{ } ys\)
Hence induction on \(xs\). Case split the same way as the program
Proofs About Lists

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

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

Proof: By induction on \(xs\).

```
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 \(xs2\)
```
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 \( xs = [ ] \):
\[
\text{length} \, (\text{cat} \, [ ] \, ys) \quad \text{(LHS of theorem)}
\]
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)

```plaintext
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)
- $= 0 + (\text{length} \; ys)$ (arithmetic)

let rec $\text{length} \; xs =$
  \text{match} \; xs \; \text{with}$
  | [] -> 0
  | x::xs -> 1 + \text{length} \; xs

let rec $\text{cat} \; xs1 \; xs2 =$
  \text{match} \; xs1 \; \text{with}$
  | [] -> xs2
  | hd::tl -> hd :: \text{cat} \; tl \; xs2
Theorem: For all lists \( xs \) and \( ys \),
\[
\text{length}(\text{cat} \: xs \: ys) = \text{length} \: xs + \text{length} \: ys
\]

Proof: By induction on \( xs \).

\[
\text{case} \: xs = [ ]:
\]
\[
\text{length} \: (\text{cat} \: [ ] \: ys) \quad \text{(LHS of theorem)}
\]
\[
= \text{length} \: ys \quad \text{(evaluate cat)}
\]
\[
= 0 + (\text{length} \: ys) \quad \text{(arithmetic)}
\]
\[
= (\text{length} \: [ ]) + (\text{length} \: ys) \quad \text{(fold length)}
\]
\[
\text{case done!}
\]

```
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
```
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 \( xs \).

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

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

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

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

$$===$$

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

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

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

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

- $\text{length} \, (\text{cat} \, (\text{hd} :: \text{tl}) \, ys)$
- $= \text{length} \, (\text{hd} :: (\text{cat} \, \text{tl} \, ys))$ (evaluate $\text{cat}$, take 2nd branch)
- $= 1 + \text{length} \, (\text{cat} \, \text{tl} \, ys)$ (evaluate $\text{length}$, take 2nd branch)
- $= \text{LHS of theorem}$

$$\text{let rec length xs} =$$
$$\text{match xs with}$$
$$\mid [] \rightarrow 0$$
$$\mid x :: xs \rightarrow 1 + \text{length xs}$$

$$\text{let rec cat xs1 xs2} =$$
$$\text{match xs1 with}$$
$$\mid [] \rightarrow \text{xs2}$$
$$\mid \text{hd} :: \text{tl} \rightarrow \text{hd} :: \text{cat} \, \text{tl} \, \text{xs2}$$
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 \(xs\).

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

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

\[
\text{let } \text{rec } \text{length } xs = \\
\text{match } xs \text{ with } \\
\text{| } [] \to 0 \\
\text{| } x::xs \to 1 + \text{length } xs
\]

\[
\text{let } \text{rec } \text{cat } xs1 \ xs2 = \\
\text{match } xs1 \text{ with } \\
\text{| } [] \to xs2 \\
\text{| } \text{hd}::\text{tl} \to \text{hd} :: \text{cat } \text{tl } xs2
\]
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 \( xs \).

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

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

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

case done!

| let rec length \( xs \) =          | let rec cat \( xs1 \) \( xs2 \) =                           |
|     match \( xs \) with            |     match \( xs1 \) with                                    |
|       | [] -> 0                        |       | [] -> \( xs2 \)                                    |
|       | x::xs -> 1 + length \( xs \)   |       | hd::tl -> hd :: cat tl \( xs2 \)                   |
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 = [ ]:

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

```ml
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 $xs$.

\[
\text{case}\ xs\ =\ [\ ]:\ \\
\text{add\_all}\ (\text{add\_all}\ [\ ]\ a)\ b\ \quad \text{(LHS of theorem)}\\
==\ \text{add\_all}\ [\ ]\ b\ \quad \text{(by evaluation of add\_all)}\\
\]
Another List example

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

Proof: By induction on \(xs\).

case \(xs = [ ]\):

\[
\begin{align*}
\text{add\_all } (\text{add\_all } [ ] \ a) \ b & \quad \text{(LHS of theorem)} \\
== \ \text{add\_all } [ ] \ b & \quad \text{(by evaluation of \ add\_all)} \\
== [ ] & \quad \text{(by evaluation of \ add\_all)} \\
==
\end{align*}
\]
Theorem: For all lists \( xs \),

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

Proof: By induction on \( xs \).

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

- \[
\text{add\_all}\ (\text{add\_all}\ [ ]\ a)\ b\quad \text{(LHS of theorem)}
\]
- \[
==\ \text{add\_all}\ [ ]\ b\quad \text{(by evaluation of add\_all)}
\]
- \[
==\ [ ]\quad \text{(by evaluation of add\_all)}
\]
- \[
==\ \text{add\_all}\ [ ]\ (a + b)\quad \text{(by evaluation of add\_all)}
\]

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 \( xs \).

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

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

Proof: By induction on $xs$.

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

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

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

Proof: By induction on $xs$.

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

$$\text{add\_all\ (add\_all\ (hd\ ::\ tl)\ a)\ b}$$  \hspace{1cm} \text{(LHS of theorem)}

$$==\ \text{add\_all\ ((hd+a) :: add\_all\ \text{tl}\ a)\ b}$$  \hspace{1cm} \text{(by eval inner add\_all)}

$$==\ (hd+a+b) :: (add\_all\ (add\_all\ \text{tl}\ a)\ b)$$  \hspace{1cm} \text{(by eval outer add\_all)}

$$==\$$

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

**Theorem:** For all lists xs,

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

**Proof:** By induction on xs.

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

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

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 \( xs \).

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

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

let rec add_all xs c =

match xs with
| [] -> []
| hd::tl -> (hd+c)::add_all tl c
**Theorem:** For all lists $xs$, property of $xs$.

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

Case: $xs == [ ]$:
...

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

Note there are other ways to cover all lists:
• eg: case for $[ ]$, case for $x1::[]$, case for $x1::x2::tl$
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 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 proof by induction to prove correctness of recursive functions