
VIBE
CONSTRUCTION


VIBE
CARPENTRY

Did I get it right?

COS 326
David Walker
Princeton University

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

slides copyright 2026 David Walker
permission granted to reuse these slides for non-commercial educational purposes

Did I get it right?

"Did I get it right?"

- Most fundamental question you can ask about a computer program

Techniques for answering:

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

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

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

Program proving

We have two goals during these lectures:

1. Teach techniques for proving properties of programs rigorously

2. Teach the use a tool (Lean) that will check those proofs for you

When teaching the use of a tool like Lean, or when teaching a new programming language, it is easy to get caught up in the details:

- for a programming language:
  - the details of the syntax
  - the details of the libraries
- for a theorem proving environment:
  - the details of the syntax
  - the details of the tactics (ie: the libraries)

It is necessary to learn a bunch of details, but my goal is to make sure you take away the broader concepts and techniques as those concepts and techniques are more reuseable across platforms and domains

Proofs about Functional Programs

Focus: proofs about *pure expressions and total functions*

- a pure expression does nothing but return a value
- given an argument, a total function does nothing but return a value
  - and the same value every time you supply the same argument
- Pure expressions and total functions don't:
  - print to the console
  - mutate global variables
  - raise exceptions
  - crash or fail to terminate
- another word for "pure expression" is "valuable expression"

In OCaml, one uses immutable data by default, but you don't have to.
In OCaml, most functions terminate, but they don't have to.

Lean adds mechanisms to *guarantee* these properties.
Many of the reasoning principles we will use fail otherwise.

Example Theorems

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

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

Theorem: easy 1 20 30 == 50

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

Theorem:

for all natural numbers n,

exp n == 2^n

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

Theorem:

for all lists xs, ys,

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

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

Things to Watch For

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

Things to Watch For

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

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

Things to Watch For

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

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

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

Things to Watch For

More structure:

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

Things to Watch For

More structure:

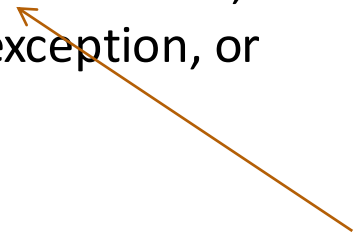
- when *programming* with lists:
  - [ ] is often easy
  - hd :: tl often requires a *recursive function call* on tl
    - we *assume* our recursive function behaves correctly on tl
- when *proving* with lists:
  - [ ] is often easy
  - hd :: tl often requires appeal to an *induction hypothesis* for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

Key Ideas

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

two expressions are equal if and only if:

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



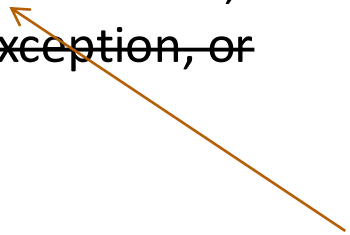
we will use what we learned about OCaml evaluation

Key Ideas

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

two expressions are equal if and only if:

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



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

Idea 2: A fundamental proof principle.

this is the principle of "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)

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]

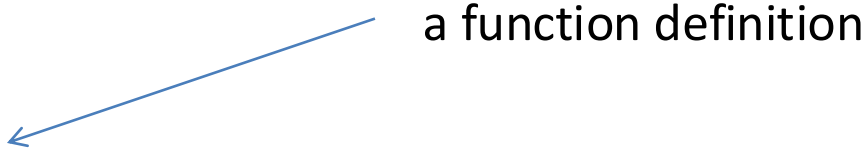
EASY EXAMPLES

Easy Examples

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

Given:

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



Easy Examples

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

Given:

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

Theorem: easy 1 20 30 == 50

Easy Examples

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

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

Theorem: easy 1 20 30 == 50

Proof:

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

Easy Examples

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

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

Theorem: easy 1 20 30 == 50

Proof:

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

== 1 * (20 + 30) (by evaluating easy 1 step) actually 3 steps

Easy Examples

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

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

Theorem: easy 1 20 30 == 50

Proof:

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

QED.

Easy Examples

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

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

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.

facts go on the left

justifications on the right

notice the 2-column proof style

Easy Examples

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

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

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

Proof:

When asked to prove something "for all n : t", one way to do that is to consider an *arbitrary* element 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.

Think back to our definition of the semantics of $\mathbf{M}, \rho \vDash \forall v.\ F$ to get some intuition for why: Your proof has to work for all possibilities

Easy Examples

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

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

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

Proof:

Consider arbitrary n, m:

easy 1 n m (LHS)

Easy Examples

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

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

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

Proof:

Consider arbitrary n, m:

easy 1 n m (LHS)

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

Easy Examples

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

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

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

Proof:

Consider arbitrary n, m:

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

QED.

Easy Examples

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

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

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

Proof:

~~Consider arbitrary n, m:~~

because this becomes repetitive, I'm going to stop saying this but this is how you should be thinking about what is happening

easy 1 n m (LHS)

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

== n + m (by math)

QED.

Easy Examples

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

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

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

Proof:

~~Consider arbitrary n, m:~~

because this becomes repetitive, I'm going to stop saying this but this is how you should be thinking about what is happening

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

QED.

Omitting comments like this is one way our proofs on paper become less formal than what we will do in Lean. Humans omit details when they are confident other experienced humans will be able to follow their argument anyway.

Another Example

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

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

Proof:

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

Another Example

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

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

Proof:

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

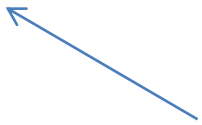
Another Example

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

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

Proof:

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



I'm not going to mention this from now on

Easy Examples

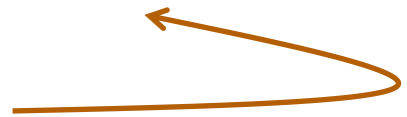
Given: `let easy x y z = x * (y + z)`

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

Proof:

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

QED.



if e1 --> e2 then: e1 == e2

and due to symmetry, e2 == e1

Easy Examples

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

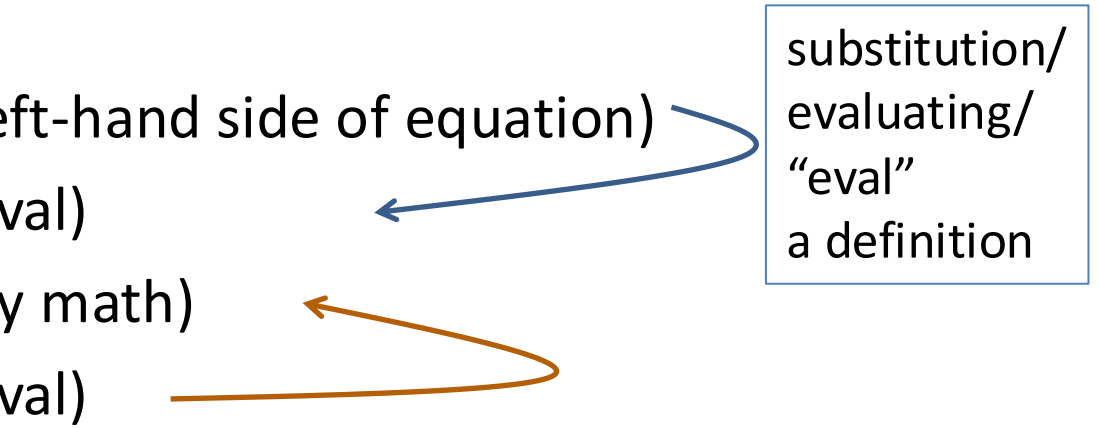
Given: `let easy x y z = x * (y + z)`

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

Proof:

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

QED.



An Aside: Symbolic Evaluation

Consider the following:

```
    easy x y (k+1)
== x * (y + (k+1))          (by evaluation of easy?)
```

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

Suppose we know: let f x = body

When is f (exp) == body[exp/x] ??

An Aside: Symbolic Evaluation

An interesting example:

```
let const x = 7
```

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



does this work for any expression?

An Aside: Symbolic Evaluation

An interesting example:

```
let const x = 7
```

const (n / 0) == 7 *Nope!*

An Aside: Symbolic Evaluation

An interesting example:

```
let const x = 7
```

const (n / 0) == 7 *Nope!*



- 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



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

An Aside: Symbolic Evaluation

An interesting example:

```
let const x = 7
```

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

Nope!

equations:

(1) (fun x -> e1) e2 == e1[e2/x]
(2) (f e2) == e1[e2/x] when let rec f x = e1

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

An Aside: Symbolic Evaluation

An interesting example:

```
let const x = 7
```

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

equations:

(1) (fun x -> e1) e2 == e1[e2/x]
(2) (f e2) == e1[e2/x] when let rec f x = e1

and when e2 evaluates to a value
***without side effects**, raising an exception, or infinite loops*

Summary so far: Proof by simple calculation

Some proofs are very easy and can be done by:

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

Eg:

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

given this

we do this proof

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

Proof:

easy a b c

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

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

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