Did I get it right?

Part 1: Simple Proofs

Speaker: David Walker
COS 326
Princeton University

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

Did I get it right?

"Did I get it right?"

- Most fundamental question you can ask about a computer program

Techniques for answering:

Grading

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

Testing

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

Proving

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

Program proving

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

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

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

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

Total Functions

A *total function* with type t1 -> t2 is

- a function that terminates on *all* args : t1, producing a value of type t2

A *partial function* with type t1 -> t2 is

- a function that terminates on *some* (but not necessarily all) of its arguments

Unless told otherwise, *when carrying out a proof*, you can assume all functions are total and all expressions are pure/valuable.

- Such facts can be proven by induction, but the proofs are usually rather boring so we typically won't make you do it.

Example Theorems

Theorem: easy 1 20 30 == 50

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

Theorem:

for all natural numbers n,

exp n == 2^n

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

Theorem:

for all lists xs, ys,

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

```
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 2
```

Key Ideas

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

two expressions are equal if and only if:

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

we will use what we learned about OCaml evaluation

Key Ideas

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

two expressions are equal if and only if:

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

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

Idea 2: A fundamental proof principle.

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

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

The Workhorse: Substitution of Equals for Equals

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

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

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

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

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

Important Properties of Expression Equality

(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

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

facts go on the left

Theorem: easy 1 20 30 == 50

justifications on the right

Proof:

easy 1 20 30	(left-hand side of equation)	notice the 2-column proof style
== 1 * (20 + 30)	(by evaluating easy 1 step)	
== 50	(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:

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

Easy Examples

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

Given:

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

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

Proof:

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

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

Easy Examples

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

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

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

Proof:

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

Easy Examples

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

Given:

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

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

Proof:

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

QED.

Easy Examples

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

Given:

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

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

Proof:

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

Easy Examples

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

Given:

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

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

Proof:

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

Easy Examples

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

Given:

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

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

Proof:

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

I'm not going to mention this from now on

Easy Examples

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

Given:

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

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

Proof:

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

QED.

Easy Examples

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

Given:

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

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

Proof:

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

substitution/
evaluating/
“eval”
a definition

the reverse:
“folding” a definition
back up

An Aside: Symbolic Evaluation

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

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

However, that is not how OCaml evaluation works. OCaml evaluates 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:

let const x = 7

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

does this work for any expression?

An Aside: Symbolic Evaluation

An interesting example:

```
let const x = 7
```

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

```
let const x = 7
```

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

more careless, wrong evaluation ...

equations:

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

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

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

An Aside: Symbolic Evaluation

An interesting example:

```
let const x = 7
```

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

equations:

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

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

Summary so far: Proof by simple calculation

Some proofs are very easy and can be done by:

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

Eg:

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

given this

we do this proof

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

Proof:

easy a b c

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

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

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

Exercise

Definition: A function f : t -> t -> t is *commutative* iff

for all x, y : t, f x y == f y x

Definition: A function f : t -> t -> t is *associative* iff

for all x, y, z : t, f x (f y z) == f (f x y) z

Theorem:
for all associative and commutative functions f : t -> t -> t, and
for all a, b : t,
foo a b = bar a b

Tip: As a justification, write "by associativity of f" or
"by commutativity of f" when you want to use those properties.

```
let foo (x:t) (y:t) : t = f (f x y) (f y x)
let bar (x:t) (y:t) : t = f x (f y (f x y))
```