Lecture 17: Proofs

Sebastiaan Joosten

October 16th

Outline

Proofs and proof objects

- Proofs are a form of communication between *humans*.
- Mathematical proofs are proofs about mathematics for mathematicians.
- Mathematicians have long-established traditions on what you can and cannot put in a proof, along with rigorous checks.
- Late 19th and early 20th century logicians like Hilbert and Gentzen came up with proof structures such that:
  - there is a proof object that proves the statement for every provable statement
  - the existence of a proof object for a statement would constitute a valid proof

Our proof objects

- Our main building block for our proof objects is going to be a *derivation*.
- A derivation takes this form:

  expression_1
  = { assumption_1 }
  expression_2
  = …
  …
  = { assumption_(n-1) }
  expression_n
- A derivation of this form proves that expression_1 = expression_n, under our assumptions.
- Our assumptions are each of the form expression_L = expression_R.

Example proof

Using the definition:

```
let rec ( ** ) a b =
      if b > 0
      then ( * ) a (a ** (b - 1))
      else (if b = 0 then 1
            else raise Negative_power)
```

```
 a ** 0
= { definition of ** }
 if 0 > 0
 then ( * ) a (a ** (0 - 1))
 else (if 0 = 0 then 1
       else raise Negative_power)
= { (a > a) = false }
 if false
 then ( * ) a (a ** (0 - 1))
 else (if 0 = 0 then 1
       else raise Negative_power)
= { (if false then a else b) = b }
 if 0 = 0 then 1
 else raise Negative_power
= { (a = a) = true }
 if true then 1
 else raise Negative_power
= { (if true then a else b) = a }
 1
```

Example proof

Using the definition:

```
let rec ( ** ) a b =
      if b > 0
      then ( * ) a (a ** (b - 1))
      else (if b = 0 then 1
            else raise Negative_power)
```

```
 a ** 0
= { definition of ** }
 if 0 > 0
 then ( * ) a (a ** (0 - 1))
 else (if 0 = 0 then 1
       else raise Negative_power)
= { (a > a) = false }
 if false
 then ( * ) a (a ** (0 - 1))
 else (if 0 = 0 then 1
       else raise Negative_power)
= { (if false then a else b) = b }
 if 0 = 0 then 1
 else raise Negative_power
= { (a = a) = true }
 if true then 1
 else raise Negative_power
= { (if true then a else b) = a }
 1
```

This proves:
a ** 0 = 1

Under the assumptions:
- definition of **
- (a > a) = false
- (a = a) = true
- two if-then-else assumptions

Different kinds of assumptions

- We saw two different kinds of assumptions:
  - assumptions on Ocaml's built-in functions (if-then-else, =, and >)
  - an assumption following our own definition (this one can be dangerous!)
- There are two other steps we can take, which we'll cover this lecture:
  - assumptions that stem from the proof structure
  - using another proof

An erroneous proof

- Let's define:
  let rec f x = if x > 0 then 1 + f x else 0

```
      0
= { a - a = 0 }
      f 1 - f 1
= { definition of f }
      (if 1 > 0 then 1 + f 1 else 0) - f 1
= { (1 > 0) = true }
      (if true then 1 + f 1 else 0) - f 1
= { (if true then a else b) = a }
      (1 + f 1) - f 1
= { (a + b) - b = a }
      1
```

Ways of explaining the contradiction…

- We can argue that any of these assumptions is invalid:
  - a - a = 0 (when filling in ‘f 1’ for a)
  - definition of f
  - (a + b) - b = a (when filling in ‘f 1’ for b)
- For the first assumption, we argue that ‘non-termination’ is a value, and that ‘a - a = 0’ is not true if we plug in non-termination as a value for ‘a’.
- For the second assumption, we argue that ‘f 1’ cannot be equal to its definition or ‘1 + f 1’, saying that ‘f’ is not properly defined.
- (the assumptions ‘1 > 0 = true’ and ‘if true then a else b = a’ are true in any reasonable proof system)

Let's make our lives easier

- Rather than dealing with non-termination properly, we'll avoid it in this course:
  - You get to use 'a - a = 0' (if you mention the assumption), without considering that 'a' might not terminate.
    (the Isabelle proof assistant allows this, but does not accept most non-terminating definitions)
  - You get to use definitional equality for definitions I give you.
    (system F semantics are defined this way, and therefore impose conditions to use 'a - a = 0')

Strategy for Rewrite proofs: evaluation

- Many proofs follow the evaluation of the ocaml code:
- (match true with
  | true -> 5 + 5
  | false -> 3) + 1
  = { match fits pattern }
  (5 + 5) + 1
  = { 5 + 5 = 10 }
  10 + 1
  = { 10 + 1 = 11}
  11

Strategy for Rewrite proofs: evaluation

- Some proofs follow the evaluation of the ocaml code bottom to top:
- 11
  = { 10 + 1 = 11}
   10 + 1
  = { 5 + 5 = 10 }
   (5 + 5) + 1
  = { match fits pattern }
   (match true with
     | true -> (5 + 5) + 1
     | false -> 3 + 1)

Combining a top-to-bottom with a bottom-to-top proof

- (match true with
  | true -> 5 + 5
  | false -> 3) + 1
  = { match fits pattern }
  (5 + 5) + 1
  = { 5 + 5 = 10 }
  10 + 1
  = { 10 + 1 = 11}
  11

11
= { 10 + 1 = 11}
10 + 1
= { 5 + 5 = 10 }
(5 + 5) + 1
= { match fits pattern }
(match true with
| true -> (5 + 5) + 1
| false -> 3 + 1)

These are two proofs, one that ‘M’ = 11 and one that 11 = ‘M2’

Combining a top-to-bottom with a bottom-to-top proof

- (match true with
  | true -> 5 + 5
  | false -> 3) + 1
  = { match fits pattern }
  (5 + 5) + 1
  = { 5 + 5 = 10 }
  10 + 1
  = { 10 + 1 = 11}
  11

= { 10 + 1 = 11}
10 + 1
= { 5 + 5 = 10 }
(5 + 5) + 1
= { match fits pattern }
(match true with
  | true -> (5 + 5) + 1
  | false -> 3 + 1)

This proof is valid, but there are some unnecessary steps

Combining a top-to-bottom with a bottom-to-top proof

- (match true with
  | true -> 5 + 5
  | false -> 3) + 1
  = { match fits pattern }
  (5 + 5) + 1
  = { match fits pattern }
  (match true with
  | true -> (5 + 5) + 1
  | false -> 3 + 1)

Just two steps!

Generalizing a proof:

- Generalizing a proof is a lot like generalizing an implementation:
- We know from the previous proof that:

```
(match true with
   | true -> 5 + 5
   | false -> 3) + 1
= { previous proof }
 (match true with
   | true -> (5 + 5) + 1
   | false -> 3 + 1)
```

Generalizing a proof:

- Generalizing a proof is a lot like generalizing an implementation:
- We can prove the ‘push function inwards’ rule for ‘match true’ statements:

```
f (match true with
      | true -> c
      | false -> d)
= { match fits pattern }
 f c
= { match fits pattern }
 (match true with
   | true -> f c
   | false -> f d)
```

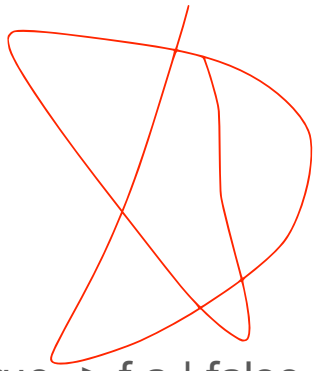
Any suggestions how to generalize further?

Proof format: proof by cases

We prove
f (match x with | true -> a | false -> b) = (match x with | true -> f a | false -> f b)
by …

Note that there is no ‘evaluation strategy’ step to take.

Proof format: proof by cases



We prove
f (match x with | true -> a | false -> b) = (match x with | true -> f a | false -> f b)
by cases on x.

Case x = true:

f (match x with | true -> a | false -> b)
= {case}
f (match true with | true -> a | false -> b)
= {match fits pattern}
f a
= {match fits pattern}
(match true with | true -> f a | false -> f b)
= {case}
(match x with | true -> f a | false -> f b)

Case x = false:

f (match x with | true -> a | false -> b)
= {case}
f (match false with | true -> a | false -> b)
= {match fits pattern}
f b
= {match fits pattern}
(match false with | true -> f a | false -> f b)
= {case}
(match x with | true -> f a | false -> f b)

This proves the statement in all cases for x : bool

Proof by cases…

- It looks nice to put two proofs by cases side by side.
- However, this is for slides only!
- If you write these as homework, please put the cases under one another.
- (You'll waste too much time on the layout, and it can make things confusing for the grader if they have word-wrap on)

So what about ‘match’ stuff?

- The rule for a match statement is a lot like applying a function definition. For example:
- match 1::(2::tl) with
  | [] -> 3
  | (a::b) -> a + foo b
  = { match fits pattern }
  1 + foo (2::tl)
- This is like applying the definition ‘a + foo b’ with the argument ‘a’ and ‘b’ filled in according to:
  ‘1::(2::tl) = a::b’

A match caveat!

- Here's a misleading problem
- match a::b with
  | [1] -> 1
  | [x] -> 2
  | 2::tl -> 3
  | _ -> 4
  = ..?

- What value does this code return?
- We can try it in utop if we're unsure!

A match caveat!

- Here's a misleading problem
- match a::b with
  | [1] -> 1
  | [x] -> 2
  | 2::tl -> 3
  | _ -> 4
  = ..?

```
utop # match a::b with | [1]
Error: Unbound value a
```

- What value does this code return?
- We can try it in utop if we're unsure!

A match caveat!

- Here's a misleading problem
- match a::b with
  | [1] -> 1
  | [x] -> 2
  | 2::tl -> 3
  | _ -> 4
  = ..?

- What value does this code return?
- We can try it in utop if we're unsure!

```
utop # match a::b with | [1]
Error: Unbound value a
utop # let a = 1;;
val a : int = 1
utop # match a::b with | [1]
Error: Unbound value b
utop # let b = [];;
val b : 'a list = []
utop # match a::b with | [1]
- : int = 1
```

A match caveat!

- Here's a misleading problem
- match a::b with
  | [1] -> 1
  | [x] -> 2
  | 2::tl -> 3
  | _ -> 4
  = ..?

```
utop # let a = 2;;
val a : int = 2
utop # let b = [];;
val b : 'a list = []
utop # match a::b with | [1]
- : int = 2
utop # let b = [2];;
val b : int list = [2]
utop # match a::b with | [1]
- : int = 3
utop # let a = 3;;
val a : int = 3
utop # match a::b with | [1]
- : int = 4
```

- What value does this code return?
- All of the above!!!

Match statements

- In most code, match statements are straightforward
- Regardless, don’t apply a match-rule unless all the previous rules can be ruled out
- If you cannot apply a match-rule, or are unsure, typically you need more case distinctions

If statement

- The if statement is just a match statement. Here's a proof by cases (on x : bool):
- match x with true -> a | false -> b
  = {case}
  match true with true -> a | false -> b
  = {match pattern true}
  a
  = {if true then a else b = a}
  if true then a else b
  = {case}
  if x then a else b

… the other case is left as an exercise in using copy-paste

Let statements

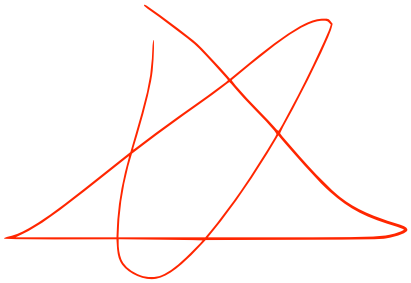
- The let statements are definitions, we covered them
- The let .. in statements can sometimes be seen as fun statements:
  let v = e1 in e2
  = { let as fun }
  (fun v -> e1) e2
- The let rec .. in statements need some careful consideration, we'll work around these in proofs by turning them into separate helper functions (using 'let rec' instead of 'let rec .. in').

Proof reuse

- It is often necessary to refer to separate proofs.
- Give your statements a name!

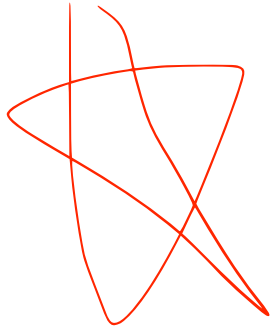
- For example, we can call the statement:
  ‘match x with true -> a | false -> b = if x then a else b’
  if then else is match
- We can call the statement:
  ‘f (match x with true -> a | false -> b) = (match x with true -> f a | false -> f b)’
  push function into a match

Proof reuse



- Here's a proof that reuses those statements:
- f (if x then a else b)
  = { if then else is match }
  f (match x with true -> a | false -> b)
  = { push function into a match }
  match x with true -> f a | false -> f b
  = { if then else is match }
  if x then f a else f b
- We were able to use 'if then else is match' twice, and we could avoid a case distinction in our proof by referring to 'push function into a match'

Proof reuse



- As you reuse more proofs, your proofs start to look more creative. Since you have copy-paste and nearly unlimited disk-space, you don't have to be creative!
- Here's a proof of the same by case distinction on x.

Case x = true:

f (if x then a else b)
= {case}
f (if true then a else b)
= {if true then e1 else e2 = e1}
f a
= {if true then e1 else e2 = e1}
if true then f a else f b
= {case}
if x then f a else f b

Case x = false:

f (if x then a else b)
= {case}
f (if false then a else b)
= {if false then e1 else e2 = e2}
f b
= {if false then e1 else e2 = e2}
if false then f a else f b
= {case}
if x then f a else f b

This proves
f (if x then a else b)
= if x then f a else f b

in all cases for
x : bool

Deep dive into a single proof step…

- Consider this step, where we apply ‘ L = R ’ from left to right

  e1
  = { L = R }
  e2

  - Example:

    (1 + 1) + 1
    = { x + x = 2 * x }
    (2 * 1) + 1

- Then there is a substitution of variables in L, such that L occurs in e1.
  e2 is then equal to e1 with the occurrence of L in e1 replaced by R under the same substitution.

Debugging proofs…

- Where's the error in this proof?
- ((x + y) * x) * y
  = { associativity of * }
  (x + y) * (x * y)
  = { right-distribute * over + }
  x * (x * y) + y * (x * y)
  = { definition of square }
  square (x * y) + y * x * y
  = { a * b = b * a }
  square (x * y) + x * y * y
  = { definition of square }
  square (x * y) + x * square y

Debugging proofs…

- Where's the error in this proof?
- ((x + y) * x) * y
  = { associativity of * }
  (x + y) * (x * y)
  = { right-distribute * over + }
  x * (x * y) + y * (x * y)
  = { definition of square }
  square (x * y) + y * x * y
  = { a * b = b * a }
  square (x * y) + x * y * y
  = { definition of square }
  square (x * y) + x * square y

- All your expressions should be valid ocaml!!

let x = 2;; let y = 3;;

first expression: 30

last expression: 54

… good, this choice of values allows us to find the error!

Debugging proofs…

- Where's the error in this proof?
- ((x + y) * x) * y
  = { associativity of * }
  (x + y) * (x * y)
  = { right-distribute * over + }
  x * (x * y) + y * (x * y)
  = { definition of square }
  square (x * y) + y * x * y
  = { a * b = b * a }
  square (x * y) + x * y * y
  = { definition of square }
  square (x * y) + x * square y

- 30

54

Debugging proofs…

- Where's the error in this proof?
- ((x + y) * x) * y
  = { associativity of * }
  (x + y) * (x * y)
  = { right-distribute * over + }
  x * (x * y) + y * (x * y)
  = { definition of square }
  square (x * y) + y * x * y
  = { a * b = b * a }
  square (x * y) + x * y * y
  = { definition of square }
  square (x * y) + x * square y

- 30

30

54

54

Debugging proofs…

- Where's the error in this proof?
- ((x + y) * x) * y
  = { associativity of * }
  (x + y) * (x * y)
  = { right-distribute * over + }
  x * (x * y) + y * (x * y)
  = { definition of square }
  square (x * y) + y * x * y
  = { a * b = b * a }
  square (x * y) + x * y * y
  = { definition of square }
  square (x * y) + x * square y

- 30

30

<— found the error!

54

54

Big overview of proof steps

- If ‘let rec e1 = e2’ or ‘let e1 = e2’ is a (terminating) definition, we can use ‘e1 = e2’ as a rule.
- Apply match statements that can be applied.
- Introduce case rules for variant types (this lecture used ‘bool’)
- Apply ‘case’ rules.
- Anything else we typically will let you know about.
- If you find an error, utop can help debug your proof!