Did I get it right?

Part 2: Induction for Naturals

Speaker: David Walker
COS 326
Princeton University

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

Last Time --> This Time

Last time, we saw some proofs 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)

But you might have noticed that none of the proofs we did last involved reasoning about recursive functions...

When you have a mix of recursive functions and symbolic values, you usually need more sophisticated proof techniques.

A problem

Theorem: For all natural numbers n,

exp(n) == 2^n.

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

A problem

Theorem: For all natural numbers n,

exp(n) == 2^n.

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

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.

A problem


```
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

A problem


```
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
== match 0 with 0 -> 1 | n -> 2 * exp (n -1)    (by eval exp)
```

A problem


```
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
== match 0 with 0 -> 1 | n -> 2 * exp (n -1)    (by eval exp)
== 1                                            (by evaluating match)
== 2^0                                          (by math)
```

A problem


```
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 == k+1:

exp (k+1)

A problem


```
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 == k+1:

```
   exp (k+1)
== match (k+1) with 0 -> 1 | n -> 2 * exp (n -1)        (by eval exp)
```

A problem


```
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 == 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 match)
```

A problem


```
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 == 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 match)
== 2 * (match (k+1-1) with 0 -> 1 | n -> 2 * exp (n -1)) (by eval exp)
```

A problem


```
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 == 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 match)
== 2 * (match (k+1-1) with 0 -> 1 | n -> 2 * exp (n -1))    (by eval exp)
== 2 * (2 * exp ((k+1) - 1 - 1))                            (by assuming (!) k > 0, eval)
```

A problem


```
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 == 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 match)
== 2 * (match (k+1-1) with 0 -> 1 | n -> 2 * exp (n -1)) (by eval exp)
== 2 * 2 * exp ((k+1) - 1 - 1)                            (by assuming (!) k > 0,  eval)
== 2 * 2 *  2 * exp ((k+1) - 1 - 1 - 1))                  (by assuming (!) k > 1,  eval)
== 2 * 2 * 2 * 2 * exp ((k+1) - 1 - 1 - 1 - 1))           (by assuming (!) k > 2,  eval)
== 2 * 2 * 2 * 2 * 2 * ... ...                            (by assuming (!) k > ... )
== ... we aren't making progress ... just unrolling the loop forever ...
```

Induction

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)

Back to the Proof


```
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 == 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)
```

Back to the Proof


```
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 == 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)

Back to the Proof

```
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 == 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!)
```

Back to the Proof


```
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+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!

Another example

Theorem: For all natural numbers n,

even(2*n) == true.

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

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:

...

Another example

Theorem: For all natural numbers n,

even(2*n) == true.

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

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, even(2*n) == true.

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

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)   (by math)
==

Another example


```
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 == 0:

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

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))

==

```
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.

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

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) == true.

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

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)
== match 2*k+2 with (0 -> true | 1 -> false | n -> even (n-2))   (by eval even)
```

Another example

Theorem: For all natural numbers n,

even(2*n) == true.

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

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)
== match 2*k+2 with (0 -> true | 1 -> false | n -> even (n-2))	(by eval even)
== even ((2*k+2)-2)	(by evaluation)

Another example

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

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

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

Another example


```
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)
== match 2*k+2 with (0 -> true | 1 -> false | n -> even (n-2))  (by eval even)
== even ((2*k+2)-2)                                           (by evaluation)
== even (2*k)                                                 (by math)
== true                                                       (by IH)
QED.
```

Template for Inductive Proofs on Natural Numbers

Theorem: For all natural numbers n, property of n.

Proof: By induction on natural numbers n.

proof methodology.
write this down.

Case: n == 0:

...

Case: n == k+1:

...

justifications to use:

- simple math
- eval, reverse eval, "by def"
- IH

cases must cover all natural numbers

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:

...

cases must cover all natural numbers

Note there are other ways to cover all natural numbers:

- eg: case for 0, case for 1, case for k+2

Exercise

Prove that add implements addition!

```
let rec add n m =
  match n with
    | 0 -> m
    | n -> add (n-1) (m+1)
```

Theorem: For all natural numbers n, m,

add n m = n + m

Note:
There are 2 parameters to this theorem – n and m.
You could do your proof by (a) "induction on n" or (b) "by induction on m"

If you choose (a) then you will consider the cases:

- n = 0 and m is an arbitrary number
- n = k+1 and m is an arbitrary number

You can use your inductive hypothesis whenever add is called and the first parameter of the function (n) is smaller.

If you choose (b) then you will consider the cases:

- m = 0 and n is an arbitrary number
- m = k+1 and n is an arbitrary number

You can use your inductive hypothesis whenever add is called and the second parameter of the function (m) is smaller.

Which should you choose? Why?