Programming Languages
COS 441

Denotational Semantics II

Last Time

- The denotational modus operandi:
  1. Define the syntax of the language
     - How do you write the programs down?
     - Use BNF notation (BNF = Bachus Naur Form)
  2. Define the denotation (aka meaning) of the language
     - Use a function from syntax to mathematical objects
     - Make sure the function is inductive and (usually) total

This Time

- The denotational modus operandi:
  1. Define the syntax of the language
     - How do you write the programs down?
     - Use BNF notation (BNF = Bachus Naur Form)
  2. Define the denotation (aka meaning) of the language
     - Use a function from syntax to mathematical objects
     - Make sure the function is inductive and (usually) total
  3. Prove something about the language
     - Most of our proofs about denotational definitions will be by induction on the structure of the syntax of the language

PROOFS BY STRUCTURAL INDUCTION

Proofs by induction

- Often, we want to know something about all objects of a certain type:
  - for all binary numbers b, there exists a larger binary number.
  - for all binary numbers b, either even(b) or odd(b) is true
  - for all arithmetic expressions e, if expsem(e) = 0 then e contains a subexpression of the form num(n) and mixsem(n) = 0
  - for all well-typed programs p, p never dereferences a null pointer
  - for all well-typed programs p, p never releases high-security information to a low-security client
  - for all programs p, semantics(p) = semantics(compile(p))
- We typically prove these properties by induction.
  - one kind of induction is structural induction or induction on syntax

Structure of inductive proofs for binary syntax

b ::= # | b0 | b1

Theorem: For all binary numbers b, property(b).

Proof: ?

Structure of inductive proofs for binary syntax

b ::= # | b0 | b1

Theorem: For all binary numbers b, property(b).

Proof: ?

for all
clues you in
to the fact
that you
need to do
induction

your goal
is to prove
the property
for all b.

Structure of inductive proofs for binary syntax

b ::= # | b0 | b1

Theorem: For all binary numbers b, property(b).

Proof strategy:

- tackle each case (#, b0, b1) separately. Be sure to tackle all cases (missing a case means your proof is incomplete) -- proofs must be total, like semantic functions were total in the last lecture.

- for base cases like #, prove the property directly

- for inductive cases like b0 and b1, use the inductive hypothesis. In other words, when proving case b0, assume that property(b) is true and use that information to conclude that property(b0) is true. (Likewise when proving b1.) In general, you get to assume your property is true for all smaller binary numbers.

Structure of inductive proofs for binary syntax

b ::= # | b0 | b1

Theorem: For all binary numbers b, property(b).

Proof: By induction on the structure of b.

case #:

....

must prove: property (#) is true

case b0:

IH: property(b) is true

...

must prove: property(b0) is true

case b0:

IH: property(b) is true

...

must prove: property(b0) is true

Structure of inductive proofs for binary syntax

b ::= # | b0 | b1

when I say always
I mean always

Theorem: For all binary numbers b, property(b).

Proof: By induction on the structure of b.

always write
proof method first

one
case
per
syntax
alternative

case #:

....

must prove: property (#) is true

proof of a case concludes
when you establish the property
for this specific piece of syntax

case b0:

IH: property(b) is true

...

must prove: property(b0) is true

case b0:

IH: property(b) is true

...

must prove: property(b0) is true

always state
the specific
induction hypothesis
you can use in
your proof case

BINARY SYNTAX:
AN EXAMPLE PROOF

Structure of inductive proofs for binary syntax

Theorem: For all binary numbers b,
if binsem(b) > 0 then b contains a 1.

Proof: By induction on the structure of b.

Definitions:

b ::= # | b0 | b1

binsem (#) = 0
binsem (b0) = 2*(binsem(b))
binsem (b1) = 2*(binsem(b)) + 1

Structure of inductive proofs for binary syntax

Theorem: For all binary numbers b,
if binsem(b) > 0 then b contains a 1.

Proof: By induction on the structure of b.

case #:

Definitions:

b ::= # | b0 | b1

binsem (#) = 0
binsem (b0) = 2*(binsem(b))
binsem (b1) = 2*(binsem(b)) + 1

Structure of inductive proofs for binary syntax

Theorem: For all binary numbers b,
if binsem(b) > 0 then b contains a 1.

Proof: By induction on the structure of b.

case #:

1: binsem (#) = 0 (by binsem def)

2: binsem(#) $\not>$ 0 (by 1)

case done (2 implies the theorem if statement is trivially satisfied)

Definitions:

b ::= # | b0 | b1

binsem (#) = 0
binsem (b0) = 2*(binsem(b))
binsem (b1) = 2*(binsem(b)) + 1

Structure of inductive proofs for binary syntax

Theorem: For all binary numbers b,
if binsem(b) > 0 then b contains a 1.

Proof: By induction on the structure of b.

case b’0:

Definitions:

b ::= # | b0 | b1

binsem (#) = 0
binsem (b0) = 2*(binsem(b))
binsem (b1) = 2*(binsem(b)) + 1

Structure of inductive proofs for binary syntax

Theorem: For all binary numbers b,
if binsem(b) > 0 then b contains a 1.

Proof: By induction on the structure of b.

case b'0:
IH: if binsem(b') > 0 then b' contains a 1

Definitions:

b ::= # | b0 | b1

binsem (#) = 0
binsem (b0) = 2*(binsem(b))
binsem (b1) = 2*(binsem(b)) + 1

Structure of inductive proofs for binary syntax

Theorem: For all binary numbers b,
if binsem(b) > 0 then b contains a 1.

Proof: By induction on the structure of b.

case b'0:

IH: if binsem(b') > 0 then b' contains a 1

1: binsem (b'0) = 2 * (binsem(b'))	(by binsem def)
2: if binsem(b'0) > 0 then binsem(b') > 0	(by 1)
3: if binsem(b'0) > 0 then b' contains a 1	(by 2 and IH)
4: if binsem(b'0) > 0 then b'0 contains a 1	(by 3 and meaning of "contains")

case done.

Definitions:

b ::= # | b0 | b1

binsem (#) = 0
binsem (b0) = 2*(binsem(b))
binsem (b1) = 2*(binsem(b)) + 1

Structure of inductive proofs for binary syntax

Theorem: For all binary numbers b,
if binsem(b) > 0 then b contains a 1.

Proof: By induction on the structure of b.

case b'1:

Definitions:

b ::= # | b0 | b1

binsem (#) = 0
binsem (b0) = 2*(binsem(b))
binsem (b1) = 2*(binsem(b)) + 1

Structure of inductive proofs for binary syntax

Theorem: For all binary numbers b,
if binsem(b) > 0 then b contains a 1.

Proof: By induction on the structure of b.

case b'1:
IH: if binsem(b') > 0 then b' contains a 1

not needed this time,
but write it down anyway

Definitions:

b ::= # | b0 | b1

binsem (#) = 0
binsem (b0) = 2*(binsem(b))
binsem (b1) = 2*(binsem(b)) + 1

Structure of inductive proofs for binary syntax

Theorem: For all binary numbers b,
if binsem(b) > 0 then b contains a 1.

Proof: By induction on the structure of b.

case b'1:

IH: if binsem(b') > 0 then b' contains a 1

1: binsem (b'1) = 2 * (binsem(b')) + 1 (by binsem def)

2: binsem (b'1) > 0 and b'1 contains a 1 (by 1 and meaning of contains)

case done (2 implies the required conclusion).

Definitions:

b ::= # | b0 | b1

binsem (#) = 0
binsem (b0) = 2*(binsem(b))
binsem (b1) = 2*(binsem(b)) + 1

Recap: structure of inductive proofs for binary syntax

Theorem: For all binary numbers b, property(b).

Proof: By induction on the structure of b.

case #:

...

property (#) is true

case done.

case b0:

IH: property(b)

...

property(b0) is true

case done.

case b0:

IH: property(b) is true

...

property(b0) is true

case done.

Definitions:

b ::= # | b0 | b1

binsem (#) = 0
binsem (b0) = 2*(binsem(b))
binsem (b1) = 2*(binsem(b)) + 1

A PROOF ABOUT ARITHMETIC EXPRESSIONS

Last time

- Arithmetic expression syntax:

e ::= num n | add(e,e) | mult(e, e)

- Arithmetic expression semantics:

depends on semantics
for number syntax;
(computes a natural number)

expsem (num (n)) = mixsem (n)

expsem (add (e_1,e_2)) = expsem (e_1) + expsem (e_2)

expsem (mult (e_1,e_2)) = expsem (e_1) * expsem (e_2)

Arithmetic Expressions

- Another definition: “contains a zero”

cz (num (n)) = if mixsem (n) = 0 then true else false
cz (add (e_1,e_2)) = cz (e_1) or cz (e_2)
cz(mult (e_1,e_2)) = cz (e_1) or cz (e_2)

- Goal Theorem:
  - for all e, if expsem(e) = 0 then cz(e)

Proving properties of expressions

Theorem: For all expressions e, property(e).

Proof: By induction on the structure of e.

case num n:

...

property (num n)

case done.

both e1 and e2 are smaller so we can use IH on both

case add(e_1, e_2):

IH1: property(e_1)

IH2: property(e_2)

...

property(add(e_1, e2))

case done.

case mult(e_1, e_2):

IH1: property(e_1) is true

IH2: property(e_2) is true

...

property(mult(e_1, e_2))

case done.

Definitions:

e ::= num n | add(e,e) | mult(e, e)

expsem (num (n)) = mixsem (n)
expsem (add (e_1,e_2)) = expsem (e_1) + expsem (e_2)
expsem (mult (e_1,e_2)) = expsem (e_1) * expsem (e_2)

cz (num (n)) = if mixsem (n) = 0 then true else false
cz(add (e_1,e_2)) = cz (e_1) or cz (e_2)
cz(mult (e_1,e_2)) = cz (e_1) or cz (e_2)

Proving properties of expressions

Theorem: For all e, if expsem(e) = 0 then cz(e).

Proof: By induction on the structure of e.

case num n:

1. expsem (num n) = mixsem (n) (by expsem def)

expsem (num (n)) = mixsem (n)
expsem (...) = ...

cz (num (n)) = if mixsem (n) = 0 then true else false
cz (...) = ...

Proving properties of expressions

Theorem: For all e, if expsem(e) = 0 then cz(e).

Proof: By induction on the structure of e.

case num n:

1. expsem (num n) = mixsem (n)   (by expsem def)

subcase expsem (num n) = 0:

subcase expsem (num n) not= 0

expsem (num (n)) = mixsem (n)
expsem (...) = ...

cz (num (n)) = if mixsem (n) = 0 then true else false
cz (...) = ...

Proving properties of expressions

Theorem: For all e, if expsem(e) = 0 then cz(e).

Proof: By induction on the structure of e.

case num n:

1. expsem (num n) = mixsem (n) (by expsem def)

subcase expsem (num n) = 0:

2. mixsem (n) = 0 (by 1 and subcase)
3. cz (num n) is true (by 2 and def of cz)

we have proven the theorem!

subcase expsem (num n) not= 0

expsem (num (n)) = mixsem (n)
expsem (...) = ...

cz (num (n)) = if mixsem (n) = 0 then true else false
cz (...) = ...

Proving properties of expressions

Theorem: For all e, if expsem(e) = 0 then cz(e).

Proof: By induction on the structure of e.

case num n:

1. expsem (num n) = mixsem (n) (by expsem def)

subcase expsem (num n) = 0:

2. mixsem (n) = 0 (by 1 and subcase)
3. cz (num n) is true (by 2 and def of cz)

we have proven the theorem!

subcase expsem (num n) not= 0

we have trivially proven the theorem!

case done.

```
expsem ( num (n) ) = mixsem (n)
expsem (...) = ...

cz ( num (n) ) = if mixsem (n) = 0 then true else false
cz (...) = ...
```

Theorem: For all e, if expsem(e) = 0 then cz(e).

Proving properties of expressions

Proof: By induction on the structure of e.

case add(e_1, e_2):

expsem (add (e_1,e_2)) = expsem (e_1) + expsem (e_2)
cz(add (e_1,e_2)) = cz (e_1) or cz (e_2)

Proving properties of expressions

Theorem: For all e, if expsem(e) = 0 then cz(e).

Proof: By induction on the structure of e.

case add(e_1, e_2):

IH1: if expsem(e_1) = 0 then cz(e_1).

IH2: if expsem(e_2) = 0 then cz(e_2).

expsem (add (e_1,e_2)) = expsem (e_1) + expsem (e_2)

cz(add (e_1,e_2)) = cz (e_1) or cz (e_2)

Proving properties of expressions

Theorem: For all e, if expsem(e) = 0 then cz(e).

Proof: By induction on the structure of e.

case add(e_1, e_2):

IH1: if expsem(e_1) = 0 then cz(e_1).

IH2: if expsem(e_2) = 0 then cz(e_2).

1. expsem (add(e_1, e_2))= expsem (e_1) + expsem (e_2)	(by expsem def)
2. if expsem (add(e_1, e_2)) = 0 then expsem (e_1) = 0 and expsem (e_2) = 0	(by 1)
3. if expsem (add(e_1, e_2)) = 0 then expsem (e_1) = 0	(by 2)

expsem (add (e_1,e_2)) = expsem (e_1) + expsem (e_2)

cz(add (e_1,e_2)) = cz (e_1) or cz (e_2)

Theorem: For all e, if expsem(e) = 0 then cz(e).

Proving properties of expressions

Proof: By induction on the structure of e.

case add(e_1, e_2):

IH1: if expsem(e_1) = 0 then cz(e_1).

IH2: if expsem(e_2) = 0 then cz(e_2).

1. expsem (add(e_1, e_2))= expsem (e_1) + expsem (e_2) (by expsem def)
2. if expsem (add(e_1, e_2)) = 0 then expsem (e_1) = 0 and expsem (e_2) = 0 (by 1)
3. if expsem (add(e_1, e_2)) = 0 then expsem (e_1) = 0 (by 2)
4. if expsem (add(e_1, e_2)) = 0 then cz (e_1) (by 3, IH1)

expsem (add (e_1,e_2)) = expsem (e_1) + expsem (e_2)

cz(add (e_1,e_2)) = cz (e_1) or cz (e_2)

Proving properties of expressions

Theorem: For all e, if expsem(e) = 0 then cz(e).

Proof: By induction on the structure of e.

case $add(e_1, e_2)$:

IH1: if $expsem(e_1) = 0$ then $cz(e_1)$.

IH2: if $expsem(e_2) = 0$ then $cz(e_2)$.

1. expsem ($add(e_1, e_2)$)= expsem (e_1) + expsem (e_2)	(by expsem def)
2. if expsem ($add(e_1, e_2)$) = 0 then expsem (e_1) = 0 and expsem (e_2) = 0	(by 1)
3. if expsem ($add(e_1, e_2)$) = 0 then expsem (e_1) = 0	(by 2)
4. if expsem ($add(e_1, e_2)$) = 0 then cz (e_1)	(by 3, IH1)
5. if expsem ($add(e_1, e_2)$) = 0 then cz ($add(e_1, e_2)$)	(by 4, cz def)

case done.

expsem (add (e_1,e_2)) = expsem (e_1) + expsem (e_2)
cz(add (e_1,e_2)) = cz (e_1) or cz (e_2)

Theorem: For all e, if expsem(e) = 0 then cz(e).

Proof: By induction on the structure of e.

Proving properties of expressions

case mult(e_1, e_2):

expsem (mult (e_1,e_2)) = expsem (e_1) * expsem (e_2)

cz(mult (e_1,e_2)) = cz (e_1) or cz (e_2)

Proving properties of expressions

Theorem: For all e, if $\text{expsem}(e) = 0$ then $\text{cz}(e)$.

Proof: By induction on the structure of e.

case $\text{mult}(e_1, e_2)$:

IH1: if $\text{expsem}(e_1) = 0$ then $\text{cz}(e_1)$.

IH2: if $\text{expsem}(e_2) = 0$ then $\text{cz}(e_2)$.

$\text{expsem}(\text{mult}(e_1,e_2)) = \text{expsem}(e_1) * \text{expsem}(e_2)$
$\text{cz}(\text{mult}(e_1,e_2)) = \text{cz}(e_1) \text{ or } \text{cz}(e_2)$

Theorem: For all e, if expsem(e) = 0 then cz(e).

Proof: By induction on the structure of e.

Proving properties of expressions

case mult(e_1, e_2):

IH1: if expsem(e_1) = 0 then cz(e_1).

IH2: if expsem(e_2) = 0 then cz(e_2).

1. expsem (mult(e_1, e_2)) = expsem (e_1) * expsem (e_2) (by expsem def)
2. if expsem (mult(e_1, e_2)) = 0 then expsem (e_1) = 0 or expsem (e_2) = 0 (by 1)

expsem (mult (e_1,e_2)) = expsem (e_1) * expsem (e_2)
cz(mult (e_1,e_2)) = cz (e_1) or cz (e_2)

Proving properties of expressions

Theorem: For all e, if expsem(e) = 0 then cz(e).

Proof: By induction on the structure of e.

case mult(e_1, e_2):

IH1: if expsem(e_1) = 0 then cz(e_1).

IH2: if expsem(e_2) = 0 then cz(e_2).

1. expsem (mult(e_1, e_2)) = expsem (e_1) * expsem (e_2) (by expsem def)
2. if expsem (mult(e_1, e_2)) = 0 then expsem (e_1) = 0 or expsem (e_2) = 0 (by 1)
3. if expsem (mult(e_1, e_2)) = 0 then cz (e_1) or cz (e_2) (by 2, IH1, IH2)

expsem (mult (e_1,e_2)) = expsem (e_1) * expsem (e_2)
cz(mult (e_1,e_2)) = cz (e_1) or cz (e_2)

Theorem: For all e, if expsem(e) = 0 then cz(e).

Proof: By induction on the structure of e.

Proving properties of expressions

case mult(e_1, e_2):

IH1: if expsem(e_1) = 0 then cz(e_1).

IH2: if expsem(e_2) = 0 then cz(e_2).

1. expsem (mult(e_1, e_2)) = expsem (e_1) * expsem (e_2) (by expsem def)
2. if expsem (mult(e_1, e_2)) = 0 then expsem (e_1) = 0 or expsem (e_2) = 0 (by 1)
3. if expsem (mult(e_1, e_2)) = 0 then cz (e_1) or cz (e_2) (by 2, IH1, IH2)
4. if expsem (mult(e_1, e_2)) = 0 then cz (mult(e_1, e_2)) (by 3, cz def)

case done.

expsem (mult (e_1,e_2)) = expsem (e_1) * expsem (e_2)

cz(mult (e_1,e_2)) = cz (e_1) or cz (e_2)

A NOTE ON TYPES FOR FUNCTIONS

Types for functions

- So far, function types have been implicit.
- When things start getting more complicated, it is useful to be able to write them down to remind ourselves what kinds of denotation functions we are dealing with:

```
x    ::    T1 -> T2
```

Types for functions

- So far, function types have been implicit.
- When things start getting more complicated, it is useful to be able to write them down to remind ourselves what kinds of denotation functions we are dealing with:

function with argument type T1
and result type T2

x :: T1 -> T2

variable name

"::" is read "has type"

- Examples:

```
binsem :: BinarySyntax -> Natural
even   :: BinarySyntax -> Bool
usem   :: UnarySyntax -> Natural
```

(we'll see more examples and more types shortly; you will pick it up as we go)

THE MATHEMATICAL STRUCTURE OF LISTS

Lists

- Natural numbers, integers, booleans, sets are well-known mathematical objects; so are lists

- A natural number j is either
  - 0, or
  - j'+1 (the successor of some natural number j')

- Analogously list of natural numbers l is either
  - [ ] (empty), or
  - j : l' (a list with at least one element j followed by a list l')

- In BNF:

  l ::= [ ] | j : l

Lists

- Lists have inductive structure like natural numbers
  - [ ] is the smallest list
  - the list l is smaller than the list with an extra element tacked on the front: (j : l)
- Some useful inductive functions over lists:
  - (check they total and inductive)

length ([ ]) = 0
length (j : l_1) = 1 + length (l_1)

concatenate ([ ], l_2) = l_2
concatenate (j : l_1, l_2) = j : (concatenate(l_1, l_2))

inductive because we define "smaller" for pairs here to be when the first element of the pair is smaller

(there are other ways to define "smaller" for pairs)

- Notation:
  - l_1 ++ l_2 means "concatenate (l_1, l_2)"
  - [1, 2, 3, 4] means "1 : 2 : 3 : 4 : [ ]"

Proofs over Lists

Theorem: For all l_1 and for all l_2, length (l_1 ++ l_2) = length (l_1) + length (l_2)

Proof: By induction on the structure of ??

$$
\begin{aligned}
&l \quad ::= \quad [\,] \quad | \quad j : l \\
\\
&\text{length}\ (\ [\,]\ ) = 0 \\
&\text{length}\ (\ j : l_1\ ) = 1 + \text{length}\ (\ l_1\ ) \\
\\
&[\,] \mathbin{++} l_2 = l_2 \\
&(j : l_1) \mathbin{++} l_2 = j : (l_1 \mathbin{++} l_2)
\end{aligned}
$$

Proofs over Lists

Theorem: For all l_1 and for all l_2, length (l_1 ++ l_2) = length (l_1) + length (l_2)

Proof: By induction on the structure of l_1.

why not l2?
It's because of the fact that length and ++ operate at the front of the list. However, this is not a rule. Often you just have to try induction on one thing or the other and see if it works.

l ::= [ ] | j : l

length ([ ]) = 0
length (j : l_1) = 1 + length (l_1)

[ ] ++ l_2 = l_2
(j : l_1) ++ l_2 = j : (l_1 ++ l_2)

Proofs over Lists

Theorem: For all l_1 and for all l_2, length (l_1 ++ l_2) = length (l_1) + length (l_2)

Proof: By induction on the structure of l_1.

case l_1 = [ ]:

l ::= [ ] | j : l		
length ([ ])	= 0	
length (j : l_1)	= 1 + length (l_1)	
[ ] ++ l_2	= l_2	
(j : l_1) ++ l_2	= j : (l_1 ++ l_2)	

Proofs over Lists

Theorem: For all l_1 and for all l_2, length (l_1 ++ l_2) = length (l_1) + length (l_2)

Proof: By induction on the structure of l_1.

case l_1 = [ ]:

length ([ ] ++ l_2)
= ?

l	::=	[ ]	|	j : l

length ([ ]) = 0
length (j : l_1) = 1 + length (l_1)

[ ] ++ l_2 = l_2
(j : l_1) ++ l_2 = j : (l_1 ++ l_2)

Proofs over Lists

Theorem: For all l_1 and for all l_2, length (l_1 ++ l_2) = length (l_1) + length (l_2)

Proof: By induction on the structure of l_1.

case l_1 = [ ]:

length ([ ] ++ l_2)	
= length (l_2)	(by def of ++)
= 0 + length (l_2)	(by ordinary arithmetic)
= length ([ ]) + length (l_2)	(by def of length, in reverse)

case done.

l ::= [ ] | j : l

length ([ ]) = 0
length (j : l_1) = 1 + length (l_1)

[ ] ++ l_2 = l_2
(j : l_1) ++ l_2 = j : (l_1 ++ l_2)

Proofs over Lists

Theorem: For all l_1 and for all l_2, length (l_1 ++ l_2) = length (l_1) + length (l_2)

Proof: By induction on the structure of l_1.

case $l_1 = j : l_1'$:

l ::= [ ] | j : l		
length ([ ])	= 0	
length (j : l_1)	= 1 + length (l_1)	
[ ] ++ l_2	= l_2	
(j : l_1) ++ l_2	= j : (l_1 ++ l_2)	

Proofs over Lists

Theorem: For all l_1 and for all l_2, length (l_1 ++ l_2) = length (l_1) + length (l_2)

Proof: By induction on the structure of l_1.

case $l_1 = j : l_1'$:
IH: length (l_1' ++ l_2) = length (l_1') + length (l_2)

l ::= [ ] | j : l

length ([ ]) = 0
length (j : l_1) = 1 + length (l_1)

[ ] ++ l_2 = l_2
(j : l_1) ++ l_2 = j : (l_1 ++ l_2)

Proofs over Lists

Theorem: For all l_1 and for all l_2, length (l_1 ++ l_2) = length (l_1) + length (l_2)

Proof: By induction on the structure of l_1.

case $l_1 = j : l_1'$:

IH: length (l_1' ++ l_2) = length (l_1') + length (l_2)

length (($j : l_1'$) ++ l_2)

=

l	::=	[ ]	|	j : l

length ([ ]) = 0

length ($j : l_1$) = 1 + length (l_1)

[ ] ++ l_2 = l_2

($j : l_1$) ++ l_2 = j : (l_1 ++ l_2)

Proofs over Lists

Theorem: For all l_1 and for all l_2, length (l_1 ++ l_2) = length (l_1) + length (l_2)

Proof: By induction on the structure of l_1.

case $l_1 = j : l_1'$:

IH: length (l_1' ++ l_2) = length (l_1') + length (l_2)

length (($j : l_1'$) ++ l_2)
= length (j : (l_1' ++ l_2)) (by def of ++)
= 1 + length (l_1' ++ l_2) (by def of length)
= 1 + length (l_1') + length (l_2) (by IH)
= length ($j : l_1'$) + length (l_2) (by def of length)

case done.

l ::= [ ] | j : l		
length ([ ])	= 0	
length (j : l_1)	= 1 + length (l_1)	
[ ] ++ l_2	= l_2	
(j : l_1) ++ l_2	= j : (l_1 ++ l_2)	

Typical Structure of Proofs About Lists

Theorem: For all l. ... property of l ...

Proof: By induction on the structure of l.

case l = [ ]

... 2-column proof of property of [ ] ...
... justifications use definitions given and basic mathematical facts

case done.

case l = j : l':
IH: property of l'

... 2-column proof of property of j : l'
... justifications use IH, definitions, basic mathematical facts

case done.

Exercises

theorem 1:
for all l_1, for all l_2,
length (l_1 ++ (j_2 : l_2)) = 1 + length (l_1 ++ l_2)

proof: ?

theorem 2:
for all l,
length (l ++ l) = 2 * length (l)

proof: ? (hint: use theorem 1 as one of your justifications)

theorem 3:
for all l, l ++ [ ] = l

proof: ?

Note: You don't have to do them,
but exercises given out in class might show up on exams!

A LIST-PROCESSING LANGUAGE

A list processing language

natural numbers

j ::= 0 | 1 | 2 | ...

list language syntax

```
s ::=
    empty             -- empty list
  | single j          -- singleton list containing j
  | cons (j, s)       -- prepend j onto s
  | concat (s1, s2)   -- concatentate s1 and s2
  | take (j, s)       -- the first j elements of s
  | rem (j, s)        -- everything but the first j elements of s
```

A list processing language

natural numbers

j ::= 0 | 1 | 2 | ...

list language syntax

```
s ::=
    empty             -- empty list
  | single j          -- singleton list containing j
  | cons (j, s)       -- prepend j onto s
  | concat (s1, s2)   -- concatentate s1 and s2
  | take (j, s)       -- the first j elements of s
  | rem (j, s)        -- everything but the first j elements of s
```

- Examples (all equal to the list [5, 3, 2]):
  - cons (5, cons (3, cons (2, empty)))
  - concat (cons (5, cons (3, empty)), single 2)
  - take (3,
cons (5, cons (3, cons (2, cons (6, cons (6, cons (7, empty)))))))
  - rem (2,
cons (9, cons (11, cons (5, cons (3, single 2)))))
  - concat (single 5, concat (single 2, single 3))

A list processing language

natural numbers
$j ::= 0 \mid 1 \mid 2 \mid \ldots$

list language syntax
$s ::=$ empty | single j | cons (j, s) | concat (s_1, s_2) | take (j, s) | rem (j, s)

- The denotational semantics will explain how to convert list syntax into concrete lists

A list processing language

natural numbers

j ::= 0 | 1 | 2 | ...

list language syntax

s ::= empty | single j | cons (j, s) | concat (s_1 , s_2) | take (j, s) | rem (j, s)

listsem :: ListSyntax -> List

listsem (empty)	= [ ]
listsem (single j)	= [j]
listsem (cons (j, s))	= j : (listsem(s))
listsem (concat (s_1 , s_2))	= listsem (s_1) ++ listsem (s_2)
listsem (take (j, s))	= ???
listsem (rem (j,s))	= ???

A list processing language

natural numbers

$j ::= 0 \mid 1 \mid 2 \mid \ldots$

list language syntax

$s ::=$ empty | single j | cons (j, s) | concat (s_1 , s_2) | take (j, s) | rem (j, s)

listsem :: ListSyntax -> List

listsem (empty) = [ ]
listsem (single j) = [j]
listsem (cons (j, s)) = j : (listsem(s))
listsem (concat (s_1 , s_2)) = listsem (s_1) ++ listsem (s_2)
listsem (take (j, s)) = takeaux (j, listsem (s))
listsem (rem (j,s)) = ???

takeaux :: (Natural, List) -> List

takeaux (0, list) = [ ]
takeaux (j+1, [ ]) = [ ]
takeaux (j+1, j' : list) = j' : (takeaux (j, list))

lexicographic ordering for inductive definition:
(x1,y1) is smaller than (x2, y2) if x1 smaller than x2
or x1 = x2 and y1 smaller than y2

A list processing language

natural numbers

j ::= 0 | 1 | 2 | ...

list language syntax

s ::= empty | single j | cons (j, s) | concat (s_1 , s_2) | take (j, s) | rem (j, s)

listsem :: ListSyntax -> List

listsem (empty)	= [ ]
listsem (single j)	= [j]
listsem (cons (j, s))	= j : (listsem(s))
listsem (concat (s_1 , s_2))	= listsem (s_1) ++ listsem (s_2)
listsem (take (j, s))	= takeaux (j, listsem (s))
listsem (rem (j,s))	= remaux (j, listsem(s))

takeaux :: (Natural, List) -> List

takeaux (0, list)	= [ ]
takeaux (j+1, [ ])	= [ ]
takeaux (j+1, j' : list)	= j : takeaux (j, list)

remaux :: (Natural , List) -> List

remaux (0, list)	= list
remaux (j+1, [ ])	= [ ]
remaux (j+1, j' : list)	= remaux (j, list)

Exercise

- Consider these additional definitions:

```
                      result ::= Yes | Maybe

isempty :: ListSyntax -> Result

isempty (empty)             = Yes
isempty (single j)          = Maybe
isempty (cons (j, s))       = Maybe
isempty (concat (s1 , s2))  = if (isempty (s1) = Yes) and isempty (s2) = Yes
                                 then Yes
                                 else Maybe
isempty (take (j, s))       = Maybe
isempty (rem (j,s))         = Maybe
```

- Prove this theorem:
  - for all s, if isempty(s) = Yes then listsem(s) = [ ]

Summary: Inductive proof structure

- Proofs by induction on syntax:
  - start with a statement of the methodology used:
    - eg: “By induction on the syntax of binary numbers”
  - must be total
    - they must have proof cases for all syntactic alternatives
  - have an induction hypothesis that can be applied to smaller subexpressions
  - should be done in a 2-column format and have cases that look like this:

case syntactic alterative:
  IH: ... statement of inductive hypothesis on subexpression ...
  1. fact   (justification)
  2. fact   (justification)
  3. fact   (justification)
case done.

justifications use:

- IH,
- previous facts established (1, 2),
- definitions like binsem or ++ given,
- simple mathematical reasoning

Summary: kinds of induction

- induction on natural numbers
  - case for 0
  - case for j+1 with IH used on j
- induction on lists
  - case for [ ]
  - case j : l with IH used on j
- induction on syntax: s ::= alt1 | alt2 | alt3 | ...
  - case for each of alt1, alt2, alt3, ... with IH used on subexpressions s
- mutual induction on syntax: s ::= alt1 | alt2 and t ::= alt3 | alt4
  - case for each of alt1, alt2, alt3, ... with IH used on subexpressions s or t
- induction on pairs (first, second)
  - sometimes: by induction on the first element
  - sometimes: by induction on the second element
  - sometimes: by lexicographic ordering of first and second (or second and first)
- in all of the above, sometimes you break down the basic cases further:
  - natural numbers: 0/j+1 broken down further to 0/1/j+1 or 0/1/j+2 etc.
  - whatever the breakdown, cover all cases & use IH on smaller subexpressions