Last Time

• The denotational modus operandi:
  1. Define the syntax of the language
     • How do you write the programs down?
     • Use BNF notation (BNF = Backus 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 $\text{expsem}(e) = 0$ then $e$ contains a subexpression of the form $\text{num}(n)$ and $\text{mixsem}(n) = 0$
  – 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 $p$, $\text{semantics}(p) = \text{semantics}(\text{compile}(p))$

• 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

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

Proof: ?

$b ::= \# | b0 | b1$

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

your goal is to prove the property for all $b$. 
Structure of inductive proofs for binary syntax

\[ b ::= \# \mid b_0 \mid b_1 \]

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

Proof strategy:

- tackle each case (\#, \( b_0 \), \( b_1 \)) 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 \( b_0 \) and \( b_1 \), use the induction hypothesis. In other words, when proving case \( b_0 \), assume that property(\( b \)) is true and use that information to conclude that property(\( b_0 \)) is true. (Likewise when proving \( b_1 \).) In general, you get to assume your property is true for all smaller binary numbers.
Theorem: For all binary numbers $b$, $\text{property}(b)$.

Proof: By induction on the structure of $b$.

case $#$:
    ....
    must prove: $\text{property}(\#)$ is true

case $b0$:
    IH: $\text{property}(b)$ is true
    ...
    must prove: $\text{property}(b0)$ is true

case $b1$:
    IH: $\text{property}(b)$ is true
    ...
    must prove: $\text{property}(b1)$ is true
Structure of inductive proofs for binary syntax

\[ b ::= \# | b0 | b1 \]

Theorem: For all binary numbers \( b \), \( \text{property}(b) \).

Proof: By induction on the structure of \( b \).

- **case \#:**
  
  ....
  
  must prove: \( \text{property}(\#) \) is true

- **case \( b0 \):**
  
  \( \text{IH: property}(b) \) is true
  
  ...
  
  must prove: \( \text{property}(b0) \) is true

- **case \( b1 \):**
  
  \( \text{IH: property}(b) \) is true
  
  ...
  
  must prove: \( \text{property}(b1) \) is true

when I say always I mean always

always write proof method first

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

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 $\text{binsem}(b) > 0$ then $b$ contains a 1.

Proof: By induction on the structure of $b$.

Definitions:

\[
\begin{align*}
    b & ::= \# \mid b0 \mid b1 \\
    \text{binsem} (\#) & = 0 \\
    \text{binsem} (b0) & = 2 \cdot (\text{binsem}(b)) \\
    \text{binsem} (b1) & = 2 \cdot (\text{binsem}(b)) + 1
\end{align*}
\]
Structure of inductive proofs for binary syntax

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

Proof: By induction on the structure of \( b \).

case \#:

Definitions:

\[
\begin{align*}
b & ::= \# \mid b0 \mid b1 \\
\text{binsem}(\#) & = 0 \\
\text{binsem}(b0) & = 2 \times \text{binsem}(b) \\
\text{binsem}(b1) & = 2 \times \text{binsem}(b) + 1
\end{align*}
\]
Theorem: For all binary numbers $b$, if $\text{binsem}(b) > 0$ then $b$ contains a 1.

Proof: By induction on the structure of $b$.

case $\#$:
   1: $\text{binsem}(\#) = 0$ (by binsem def)
   2: $\text{binsem}(\#) \neq 0$ (by 1)

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

Definitions:

\[
b ::= \# \mid b0 \mid b1
\]

\[
\text{binsem}(\#) = 0
\]

\[
\text{binsem}(b0) = 2*(\text{binsem}(b))
\]

\[
\text{binsem}(b1) = 2*(\text{binsem}(b)) + 1
\]
Structure of inductive proofs for binary syntax

Theorem: For all binary numbers $b$, if $\text{binsem}(b) > 0$ then $b$ contains a 1.

Proof: By induction on the structure of $b$.

case $b'0$:

Definitions:

$\text{b ::= # | b0 | b1}$

$\text{binsem}(\#) = 0$

$\text{binsem}(b0) = 2*(\text{binsem}(b))$

$\text{binsem}(b1) = 2*(\text{binsem}(b)) + 1$
Theorem: For all binary numbers $b$, if $\text{binsem}(b) > 0$ then $b$ contains a 1.

Proof: By induction on the structure of $b$.

case $b'0$:

IH: if $\text{binsem}(b') > 0$ then $b'$ contains a 1

Definitions:

$b ::= \# | b0 | b1$

$\text{binsem}(\#) = 0$

$\text{binsem}(b0) = 2*(\text{binsem}(b))$

$\text{binsem}(b1) = 2*(\text{binsem}(b)) + 1$
Structure of inductive proofs for binary syntax

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

Proof: By induction on the structure of \( b \).

case \( b'0 \):
   IH: if \( \text{binsem}(b') > 0 \) then \( b' \) contains a 1
   1: \( \text{binsem}(b'0) = 2 \times (\text{binsem}(b')) \) (by binsem def)
   2: if \( \text{binsem}(b'0) > 0 \) then \( \text{binsem}(b') > 0 \) (by 1)
   3: if \( \text{binsem}(b'0) > 0 \) then \( b' \) contains a 1 (by 2 and IH)
   4: if \( \text{binsem}(b'0) > 0 \) then \( b'0 \) contains a 1 (by 3 and meaning of “contains”)

case done.

Definitions:

\[
\begin{align*}
b & ::= \# \mid b0 \mid b1 \\
\text{binsem}(\#) &= 0 \\
\text{binsem}(b0) &= 2 \times (\text{binsem}(b)) \\
\text{binsem}(b1) &= 2 \times (\text{binsem}(b)) + 1
\end{align*}
\]
Structure of inductive proofs for binary syntax

Theorem: For all binary numbers $b$, if $\text{binsem}(b) > 0$ then $b$ contains a 1.

Proof: By induction on the structure of $b$.

case $b'1$:

<table>
<thead>
<tr>
<th>Definitions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b ::= #</td>
</tr>
<tr>
<td>$\text{binsem}(#) = 0$</td>
</tr>
<tr>
<td>$\text{binsem}(b0) = 2*(\text{binsem}(b))$</td>
</tr>
<tr>
<td>$\text{binsem}(b1) = 2*(\text{binsem}(b)) + 1$</td>
</tr>
</tbody>
</table>
Structure of inductive proofs for binary syntax

Theorem: For all binary numbers $b$, if $\text{binsem}(b) > 0$ then $b$ contains a 1.

Proof: By induction on the structure of $b$.

case $b'1$:
   IH: if $\text{binsem}(b') > 0$ then $b'$ contains a 1

Definitions:

$b ::= \# | b0 | b1$

$\text{binsem}(\#) = 0$
$\text{binsem}(b0) = 2*(\text{binsem}(b))$
$\text{binsem}(b1) = 2*(\text{binsem}(b)) + 1$
Theorem: For all binary numbers \( b \),
if \( \text{binsem}(b) > 0 \) then \( b \) contains a 1.

Proof: By induction on the structure of \( b \).

case \( b'1 \):

IH: if \( \text{binsem}(b') > 0 \) then \( b' \) contains a 1

1: \( \text{binsem}(b'1) = 2 \times (\text{binsem}(b')) + 1 \) (by binsem def)
2: \( \text{binsem}(b'1) > 0 \) and \( b'1 \) contains a 1 (by 1 and meaning of contains)

case done (2 implies the required conclusion).

Definitions:

\[
\begin{align*}
b & ::= \# \mid b0 \mid b1 \\
\text{binsem}(\#) & = 0 \\
\text{binsem}(b0) & = 2 \times (\text{binsem}(b)) \\
\text{binsem}(b1) & = 2 \times (\text{binsem}(b)) + 1
\end{align*}
\]
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 $b_0$:
    IH: property($b$)
    ...
    property($b_0$) is true
    case done.

case $b_1$:
    IH: property($b$) is true
    ...
    property($b_1$) is true
    case done.

Definitions:

$b ::= \# \mid b_0 \mid b_1$

$\text{binsem } (\#) = 0$
$\text{binsem } (b_0) = 2 \times (\text{binsem } (b))$
$\text{binsem } (b_1) = 2 \times (\text{binsem } (b)) + 1$
A PROOF ABOUT ARITHMETIC EXPRESSIONS
Last time

• Arithmetic expression syntax:

\[ e ::= \text{num } n \mid \text{add}(e, e) \mid \text{mult}(e, e) \]

• Arithmetic expression semantics:

\[
\begin{align*}
\text{expsem} ( \text{num } (n) ) &= \text{mixsem } (n) \\
\text{expsem} ( \text{add } (e_1, e_2) ) &= \text{expsem } (e_1) + \text{expsem } (e_2) \\
\text{expsem} ( \text{mult } (e_1, e_2) ) &= \text{expsem } (e_1) \times \text{expsem } (e_2)
\end{align*}
\]

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

• Another definition: “contains a zero”

\[
\begin{align*}
    cz(\text{num}(n)) &= \text{if } \text{mixsem}(n) = 0 \text{ then true else false} \\
    cz(\text{add}(e_1,e_2)) &= cz(e_1) \text{ or } cz(e_2) \\
    cz(\text{mult}(e_1,e_2)) &= cz(e_1) \text{ or } cz(e_2)
\end{align*}
\]

• Goal Theorem:
  
  – for all e, if expsem(e) = 0 then cz(e)
Theorem: For all expressions $e$, $\text{property}(e)$.

Proof: By induction on the structure of $e$.

case $\text{num} \ n$:
  ...
  $\text{property} (\text{num} \ n)$
case done.

case $\text{add}(e_1, e_2)$:
  IH1: $\text{property}(e_1)$
  IH2: $\text{property}(e_2)$
  ...
  $\text{property}(\text{add}(e_1, e_2))$
case done.

case $\text{mult}(e_1, e_2)$:
  IH1: $\text{property}(e_1)$ is true
  IH2: $\text{property}(e_2)$ is true
  ...
  $\text{property}(\text{mult}(e_1, e_2))$
case done.

Definitions:

e ::= \text{num} \ n \ | \ \text{add}(e,e) \ | \ \text{mult}(e,e)

\[
\begin{align*}
\text{expsem} ( \text{num} (n) ) &= \text{mixsem} (n) \\
\text{expsem} ( \text{add} (e_1,e_2) ) &= \text{expsem} (e_1) + \text{expsem} (e_2) \\
\text{expsem} ( \text{mult} (e_1,e_2) ) &= \text{expsem} (e_1) \times \text{expsem} (e_2) \\
\text{cz} ( \text{num} (n) ) &= \text{if} \ \text{mixsem} (n) = 0 \ \text{then} \ \text{true} \ \text{else} \ \text{false} \\
\text{cz} ( \text{add} (e_1,e_2) ) &= \text{cz} (e_1) \ \text{or} \ \text{cz} (e_2) \\
\text{cz} ( \text{mult} (e_1,e_2) ) &= \text{cz} (e_1) \ \text{or} \ \text{cz} (e_2)
\end{align*}
\]
Theorem: For all $e$, if $\text{expsem}(e) = 0$ then $\text{cz}(e)$.

Proof: By induction on the structure of $e$.

case $\text{num } n$:
  1. $\text{expsem}(\text{num } n) = \text{mixsem}(n)$ \hfill (by $\text{expsem}$ def)

$\text{expsem}(\text{num } n) = \text{mixsem}(n)$
$\text{expsem}(\ldots) = \ldots$

$\text{cz}(\text{num } n) = \text{if } \text{mixsem}(n) = 0 \text{ then true else false}$
$\text{cz}(\ldots) = \ldots$
Theorem: For all $e$, if $\text{expsem}(e) = 0$ then $\text{cz}(e)$.

Proof: By induction on the structure of $e$.

case $\text{num } n$:
1. $\text{expsem}(\text{num } n) = \text{mixsem}(n)$ (by expsem def)
   
   subcase $\text{expsem}(\text{num } n) = 0$:
   
   subcase $\text{expsem}(\text{num } n) \neq 0$

```
expsem(\text{num } n) = \text{mixsem}(n) 
expsem(...) = ...

\text{cz}(\text{num } n) = \text{if mixsem}(n) = 0 \text{ then true else false}
\text{cz}(...) = ...
```
Theorem: For all $e$, if $\text{expsem}(e) = 0$ then $\text{cz}(e)$.

Proof: By induction on the structure of $e$.

case $\text{num } n$:
1. $\text{expsem}(\text{num } n) = \text{mixsem}(n)$ (by $\text{expsem}$ def)

   subcase $\text{expsem}(\text{num } n) = 0$:
   2. $\text{mixsem}(n) = 0$ (by 1 and subcase)
   3. $\text{cz}(\text{num } n)$ is true (by 2 and def of $\text{cz}$)
   we have proven the theorem!

   subcase $\text{expsem}(\text{num } n) \neq 0$

\[
\begin{align*}
\text{expsem}(\text{num } n) &= \text{mixsem}(n) \\
\text{expsem}(\ldots) &= \\
\text{cz}(\text{num } n) &= \text{if } \text{mixsem}(n) = 0 \text{ then } \text{true} \text{ else } \text{false} \\
\text{cz}(\ldots) &= 
\end{align*}
\]
Theorem: For all $e$, if $\text{expsem}(e) = 0$ then $\text{cz}(e)$.

Proof: By induction on the structure of $e$.

case \text{num \, n}:
  1. $\text{expsem}(\text{num \, n}) = \text{mixsem}(n)$ \quad (by \, \text{expsem def})

    \text{subcase} \, \text{expsem}(\text{num \, n}) = 0:
      2. $\text{mixsem}(n) = 0$ \quad (by \, 1 \, \text{and subcase})
      3. $\text{cz}(\text{num \, n})$ is true \quad (by \, 2 \, \text{and def \, of \, cz})

    we have proven the theorem!

    \text{subcase} \, \text{expsem}(\text{num \, n}) \neq 0
      we have trivially proven the theorem!

case done.

\begin{align*}
\text{expsem} (\text{num \,(n)}) &= \text{mixsem} (n) \\
\text{expsem} (...) &= ... \\
\text{cz} (\text{num \,(n)}) &= \text{if mixsem} (n) = 0 \text{ then true else false} \\
\text{cz} (...) &= ...
\end{align*}
Theorem: For all $e$, if $\text{expsem}(e) = 0$ then $\text{cz}(e)$.

Proof: By induction on the structure of $e$.

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

<table>
<thead>
<tr>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{expsem}(\text{add}(e_1, e_2)) = \text{expsem}(e_1) + \text{expsem}(e_2)$</td>
</tr>
<tr>
<td>$\text{cz}(\text{add}(e_1, e_2)) = \text{cz}(e_1) \text{ or } \text{cz}(e_2)$</td>
</tr>
</tbody>
</table>
Theorem: For all \( e \), if \( \text{expsem}(e) = 0 \) then \( \text{cz}(e) \).

Proof: By induction on the structure of \( e \).

case \( \text{add}(e_1, e_2) \):

IH1: if \( \text{expsem}(e_1) = 0 \) then \( \text{cz}(e_1) \).
IH2: if \( \text{expsem}(e_2) = 0 \) then \( \text{cz}(e_2) \).

Proving properties of expressions

- \( \text{expsem} \left( \text{add} \left( e_1, e_2 \right) \right) = \text{expsem} \left( e_1 \right) + \text{expsem} \left( e_2 \right) \)
- \( \text{cz} \left( \text{add} \left( e_1, e_2 \right) \right) = \text{cz} \left( e_1 \right) \text{ or } \text{cz} \left( e_2 \right) \)
Theorem: For all $e$, if $\text{expsem}(e) = 0$ then $\text{cz}(e)$.

Proof: By induction on the structure of $e$.

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

IH1: if $\text{expsem}(e_1) = 0$ then $\text{cz}(e_1)$.
IH2: if $\text{expsem}(e_2) = 0$ then $\text{cz}(e_2)$.

1. $\text{expsem}\left(\text{add}(e_1, e_2)\right) = \text{expsem}(e_1) + \text{expsem}(e_2)$  \hspace{1cm} (by $\text{expsem}$ def)

1b. iff $\text{expsem}(\text{add}(e_1, e_2)) = 0$ then $\text{expsem}(e_1) + \text{expsem}(e_2) = 0$

2. if $\text{expsem}(\text{add}(e_1, e_2)) = 0$ then $\text{expsem}(e_1) = 0$ and $\text{expsem}(e_2) = 0$  \hspace{1cm} (by 1)

3. if $\text{expsem}(\text{add}(e_1, e_2)) = 0$ then $\text{expsem}(e_1) = 0$  \hspace{1cm} (by 2)
Theorem: For all $e$, if $\text{expsem}(e) = 0$ then $\text{cz}(e)$.

Proof: By induction on the structure of $e$.

Case $\text{add}(e_1, e_2)$:

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

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

1. $\text{expsem}(\text{add}(e_1, e_2)) = \text{expsem}(e_1) + \text{expsem}(e_2)$ \hspace{1cm} (by $\text{expsem}$ def)

2. if $\text{expsem}(\text{add}(e_1, e_2)) = 0$ then $\text{expsem}(e_1) = 0$ and $\text{expsem}(e_2) = 0$ \hspace{1cm} (by 1)

3. if $\text{expsem}(\text{add}(e_1, e_2)) = 0$ then $\text{expsem}(e_1) = 0$ \hspace{1cm} (by 2)

4. if $\text{expsem}(\text{add}(e_1, e_2)) = 0$ then $\text{cz}(e_1)$ \hspace{1cm} (by 3, IH1)

Proving properties of expressions

$$
\text{expsem} \left( \text{add} \left( e_1, e_2 \right) \right) = \text{expsem} \left( e_1 \right) + \text{expsem} \left( e_2 \right)
$$

$$
\text{cz} \left( \text{add} \left( e_1, e_2 \right) \right) = \text{cz} \left( e_1 \right) \text{ or } \text{cz} \left( e_2 \right)
$$
Theorem: For all $e$, if $\text{expsem}(e) = 0$ then $\text{cz}(e)$.

Proof: By induction on the structure of $e$.

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

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

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

1. $\text{expsem}(\text{add}(e_1, e_2)) = \text{expsem}(e_1) + \text{expsem}(e_2)$ \hspace{1cm} (by $\text{expsem}$ def)

2. if $\text{expsem}(\text{add}(e_1, e_2)) = 0$ then $\text{expsem}(e_1) = 0$ and $\text{expsem}(e_2) = 0$ \hspace{1cm} (by 1)

3. if $\text{expsem}(\text{add}(e_1, e_2)) = 0$ then $\text{expsem}(e_1) = 0$ \hspace{1cm} (by 2)

4. if $\text{expsem}(\text{add}(e_1, e_2)) = 0$ then $\text{cz}(e_1)$ \hspace{1cm} (by 3, IH1)

5. if $\text{expsem}(\text{add}(e_1, e_2)) = 0$ then $\text{cz}(\text{add}(e_1, e_2))$ \hspace{1cm} (by 4, $\text{cz}$ def)

case done.

$\text{expsem}\left(\text{add}\left(e_1, e_2\right)\right) = \text{expsem}\left(e_1\right) + \text{expsem}\left(e_2\right)$

$\text{cz}\left(\text{add}\left(e_1, e_2\right)\right) = \text{cz}\left(e_1\right)\text{ or } \text{cz}\left(e_2\right)$
Theorem: For all $e$, if $\text{expsem}(e) = 0$ then $\text{cz}(e)$.

Proof: By induction on the structure of $e$.

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

$$\text{expsem}(\text{mult}(e_1, e_2)) = \text{expsem}(e_1) \times \text{expsem}(e_2)$$

$$\text{cz}(\text{mult}(e_1, e_2)) = \text{cz}(e_1) \text{ or } \text{cz}(e_2)$$
Theorem: For all \( e \), if \( \text{expsem}(e) = 0 \) 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 \( \text{cz}(e_2) \).

Proving properties of expressions

\[
\text{expsem}(\text{mult}(e_1, e_2)) = \text{expsem}(e_1) \ast \text{expsem}(e_2)
\]

\[
\text{cz}(\text{mult}(e_1, e_2)) = \text{cz}(e_1) \text{ or } \text{cz}(e_2)
\]
Theorem: For all $e$, if $\text{expsem}(e) = 0$ 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 $\text{cz}(e_2)$.

1. $\text{expsem}(\text{mult}(e_1, e_2)) = \text{expsem}(e_1) \times \text{expsem}(e_2)$ (by $\text{expsem}$ def)
2. if $\text{expsem}(\text{mult}(e_1, e_2)) = 0$ then $\text{expsem}(e_1) = 0$ or $\text{expsem}(e_2) = 0$ (by 1)

$$\begin{align*}
\text{expsem}(\text{mult}(e_1, e_2)) &= \text{expsem}(e_1) \times \text{expsem}(e_2) \\
\text{cz}(\text{mult}(e_1, e_2)) &= \text{cz}(e_1) \text{ or } \text{cz}(e_2)
\end{align*}$$
Theorem: For all $e$, if $\text{expsem}(e) = 0$ then $\text{cz}(e)$.

Proof: By induction on the structure of $e$.

\textbf{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 $\text{cz}(e_2)$.

1. $\text{expsem}(\text{mult}(e_1, e_2)) = \text{expsem}(e_1) \times \text{expsem}(e_2)$ \hspace{1cm} (by expsem def)
2. if $\text{expsem}(\text{mult}(e_1, e_2)) = 0$ then $\text{expsem}(e_1) = 0$ or $\text{expsem}(e_2) = 0$ \hspace{1cm} (by 1)
3. if $\text{expsem}(\text{mult}(e_1, e_2)) = 0$ then $\text{cz}(e_1)$ or $\text{cz}(e_2)$ \hspace{1cm} (by 2, IH1, IH2)
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 $\text{cz}(e_2)$.

1. $\text{expsem}(\text{mult}(e_1, e_2)) = \text{expsem}(e_1) * \text{expsem}(e_2)$ (by expsem def)

2. if $\text{expsem}(\text{mult}(e_1, e_2)) = 0$ then $\text{expsem}(e_1) = 0$ or $\text{expsem}(e_2) = 0$ (by 1)

3. if $\text{expsem}(\text{mult}(e_1, e_2)) = 0$ then $\text{cz}(e_1)$ or $\text{cz}(e_2)$ (by 2, IH1, IH2)

4. if $\text{expsem}(\text{mult}(e_1, e_2)) = 0$ then $\text{cz}(\text{mult}(e_1, e_2))$ (by 3, cz def)

case done.

Proving properties of expressions

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

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

\[ x :: T1 \rightarrow T2 \]
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:

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)

\[
\begin{align*}
\text{length } ([ ]) & = 0 \\
\text{length } (j : l_1) & = 1 + \text{length } (l_1)
\end{align*}
\]

\[
\begin{align*}
\text{concatenate } ([], l_2) & = l_2 \\
\text{concatenate } (j : l_1, l_2) & = j : (\text{concatenate}(l_1, l_2))
\end{align*}
\]

• 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 \( l_2 \), \( \text{length} \ ( l_1 ++ l_2 ) = \text{length} \ ( l_1 ) + \text{length} \ ( l_2 ) \)

Proof: By induction on the structure of \( ?? \)
Theorem: For all $l_1$ and for all $l_2$, $\text{length} ( l_1 ++ l_2 ) = \text{length} ( l_1 ) + \text{length} ( l_2 )$

Proof: By induction on the structure of $l_1$.

why not $l_2$?
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 \( l_2 \), \( \text{length} \ ( l_1 ++ l_2 ) = \text{length} \ ( l_1 ) + \text{length} \ ( l_2 ) \)

Proof: By induction on the structure of \( l_1 \).

case \( l_1 = [ ] \):
Proofs over Lists

Theorem: For all $l_1$ and for all $l_2$, $\text{length}(l_1 ++ l_2) = \text{length}(l_1) + \text{length}(l_2)$

Proof: By induction on the structure of $l_1$.

Case $l_1 = []$:

\[
\text{length}([[]] ++ l_2) = \ ?
\]

\[
l ::= [ ] | j : l
\]

\[
\text{length}([ ]) = 0
\]

\[
\text{length}(j : l_1) = 1 + \text{length}(l_1)
\]

\[
[ ] ++ l_2 = l_2
\]

\[
(j : l_1) ++ l_2 = j : (l_1 ++ l_2)
\]
Theorem: For all \( l_1 \) and for all \( l_2 \), \( \text{length} ( l_1 ++ l_2 ) = \text{length} ( l_1 ) + \text{length} ( l_2 ) \)

Proof: By induction on the structure of \( l_1 \).

**case** \( l_1 = [] \):

\[
\begin{align*}
\text{length} ( [] ++ l_2 ) &= \text{length} ( l_2 ) \quad \text{(by def of ++)} \\
&= 0 + \text{length} ( l_2 ) \quad \text{(by ordinary arithmetic)} \\
&= \text{length} ( [] ) + \text{length} ( l_2 ) \quad \text{(by def of length, in reverse)}
\end{align*}
\]

**case done.**

\[
\begin{array}{l}
l ::= [] \mid j : l \\
length ( [] ) = 0 \\
length ( j : l_1 ) = 1 + \text{length} ( l_1 ) \\
[ ] ++ l_2 = l_2 \\
(j : l_1) ++ l_2 = j : (l_1 ++ l_2)
\end{array}
\]
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 \( l_1 = j : l_1' \):

\[
\begin{align*}
l & ::= \ [ \ ] \mid j : l \\
\text{length} ( \ [ \ ] ) &= 0 \\
\text{length} ( j : l_1 ) &= 1 + \text{length} ( l_1 ) \\
[ \ ] ++ l_2 &= l_2 \\
(j : l_1) ++ l_2 &= j : (l_1 ++ l_2)
\end{align*}
\]
Proofs over Lists

Theorem: For all \( l_1 \) and for all \( l_2 \), \( \text{length} ( l_1 ++ l_2 ) = \text{length} ( l_1 ) + \text{length} ( l_2 ) \)

Proof: By induction on the structure of \( l_1 \).

case \( l_1 = j : l_1' \):
  IH: \( \text{length} ( l_1' ++ l_2 ) = \text{length} ( l_1' ) + \text{length} ( l_2 ) \)

<table>
<thead>
<tr>
<th>l   ::=</th>
<th>[]</th>
<th>j : l</th>
</tr>
</thead>
<tbody>
<tr>
<td>length ( [ ] ) = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>length ( j : l_1 ) = 1 + length ( l_1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ ] ++ l_2 = l_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(j : l_1) ++ l_2 = j : (l_1 ++ l_2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Proofs over Lists

Theorem: For all $l_1$ and for all $l_2$, $\text{length } ( l_1 ++ l_2 ) = \text{length } ( l_1 ) + \text{length } ( l_2 )$

Proof: By induction on the structure of $l_1$.

case $l_1 = j : l_1'$:
IH: $\text{length } ( l_1' ++ l_2 ) = \text{length } ( l_1' ) + \text{length } ( l_2 )$

\[
\text{length } ( (j : l_1') ++ l_2 ) =
\]

\[
[ ] ::= [ ] | j : l
\]

\[
\text{length } ( [ ] ) = 0
\]

\[
\text{length } ( j : l_1 ) = 1 + \text{length } ( l_1 )
\]

\[
[ ] ++ l_2 = l_2
\]

\[
(j : l_1) ++ l_2 = j : (l_1 ++ l_2)
\]
Proofs over Lists

Theorem: For all \( l_1 \) and for all \( l_2 \), \( \text{length} ( l_1 ++ l_2 ) = \text{length} ( l_1 ) + \text{length} ( l_2 ) \)

Proof: By induction on the structure of \( l_1 \).

case \( l_1 = j : l_1' \):

IH: \( \text{length} ( l_1' ++ l_2 ) = \text{length} ( l_1' ) + \text{length} ( l_2 ) \)

\[
\begin{align*}
\text{length} ( (j : l_1') ++ l_2 ) &= \text{length} ( j : (l_1' ++ l_2) ) & \text{(by def of ++)} \\
&= 1 + \text{length} (l_1' ++ l_2) & \text{(by def of length)} \\
&= 1 + \text{length} ( l_1' ) + \text{length} ( l_2 ) & \text{(by IH)} \\
&= \text{length} (j : l_1') + \text{length} ( l_2 ) & \text{(by def of length)}
\end{align*}
\]

case done.

\[
\begin{align*}
\text{l} &::= \; [ ] \; | \; j : l \\
\text{length} ( [ ] ) &= 0 \\
\text{length} ( j : l_1 ) &= 1 + \text{length} ( l_1 ) \\
[ ] ++ l_2 &= l_2 \\
(j : l_1) ++ l_2 &= j : (l_1 ++ l_2)
\end{align*}
\]
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.

\[ \text{case } l = j : l' : \]
\[ \text{IH: property of } l' \]

... 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 \( l_2 \),
\[
\text{length (} l_1 ++ (j_2 : l_2) \text{)} = 1 + \text{length (} l_1 ++ l_2 \text{)}
\]
proof: ?

theorem 2:
for all \( l \),
\[
\text{length (} l ++ l \text{)} = 2 \times \text{length (} l \text{)}
\]
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) -- concatenate 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

\[ j ::= 0 \mid 1 \mid 2 \mid \ldots \]

list language syntax

\[ s ::= \]

- `empty` -- empty list
- `single j` -- singleton list containing \( j \)
- `cons (j, s)` -- prepend \( j \) onto \( s \)
- `concat (s_1, s_2)` -- concatenate \( s_1 \) and \( s_2 \)
- `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)))))), single 2)`
- `rem (2, cons (9, cons (11, cons (5, cons (3, single 2))))))`
- `concat (single 5, concat (single 2, single 3))`
### A list processing language

<table>
<thead>
<tr>
<th>natural numbers</th>
<th>list language syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>( j ::= 0</td>
<td>1</td>
</tr>
</tbody>
</table>

- The denotational semantics will explain how to convert list syntax into concrete lists.
## A list processing language

<table>
<thead>
<tr>
<th>Natural numbers</th>
<th>List language syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>( j ::= 0 \mid 1 \mid 2 \mid ... )</td>
<td>( s ::= \text{empty} \mid \text{single } j \mid \text{cons } (j, s) \mid \text{concat } (s_1, s_2) \mid \text{take } (j, s) \mid \text{rem } (j, s) )</td>
</tr>
</tbody>
</table>

\[
\text{listsem :: ListSyntax} \rightarrow \text{List}
\]

\[
\begin{align*}
\text{listsem } (\text{empty}) & = [ \ ] \\
\text{listsem } (\text{single } j) & = [ j ] \\
\text{listsem } (\text{cons } (j, s)) & = j : (\text{listsem } (s)) \\
\text{listsem } (\text{concat } (s_1, s_2)) & = \text{listsem } (s_1) ++ \text{listsem } (s_2) \\
\text{listsem } (\text{take } (j, s)) & = ??? \\
\text{listsem } (\text{rem } (j,s)) & = ???
\end{align*}
\]
A list processing language

natural numbers

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

list language syntax

s ::=  empty | single j | cons (j, s) | concat (s₁, s₂) | take (j, s) | rem (j, s)

listsem :: ListSyntax -> List

listsem (empty) = [ ]
listsem (single j) = [ j ]
listsem (cons (j, s)) = j : (listsem(s))
listsem (concat (s₁, s₂)) = listsem (s₁) ++ listsem (s₂)
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

<table>
<thead>
<tr>
<th>Natural numbers</th>
<th>List language syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>j ::= 0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### List language syntax

- `listsem :: ListSyntax -> List`

<table>
<thead>
<tr>
<th><code>listsem</code></th>
<th>=</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listsem (empty)</code></td>
<td>=</td>
<td><code>[ ]</code></td>
</tr>
<tr>
<td><code>listsem (single j)</code></td>
<td>=</td>
<td><code>[ j ]</code></td>
</tr>
<tr>
<td><code>listsem (cons (j, s))</code></td>
<td>=</td>
<td><code>j : (listsem(s))</code></td>
</tr>
<tr>
<td><code>listsem (concat (s₁, s₂))</code></td>
<td>=</td>
<td><code>listsem (s₁) ++ listsem (s₂)</code></td>
</tr>
<tr>
<td><code>listsem (take (j, s))</code></td>
<td>=</td>
<td><code>takeaux (j, listsem (s))</code></td>
</tr>
<tr>
<td><code>listsem (rem (j,s))</code></td>
<td>=</td>
<td><code>remaux (j, listsem(s))</code></td>
</tr>
</tbody>
</table>

#### Takeaux

- `takeaux :: (Natural, List) -> List`

<table>
<thead>
<tr>
<th><code>takeaux</code></th>
<th>=</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>takeaux (0, list)</code></td>
<td>=</td>
<td><code>[ ]</code></td>
</tr>
<tr>
<td><code>takeaux (j+1, [ ])</code></td>
<td>=</td>
<td><code>[ ]</code></td>
</tr>
<tr>
<td><code>takeaux (j+1, j’ : list)</code></td>
<td>=</td>
<td><code>j : takeaux (j, list)</code></td>
</tr>
</tbody>
</table>

#### Remaux

- `remaux :: (Natural, List) -> List`

<table>
<thead>
<tr>
<th><code>remaux</code></th>
<th>=</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>remaux (0, list)</code></td>
<td>=</td>
<td><code>list</code></td>
</tr>
<tr>
<td><code>remaux (j+1, [ ])</code></td>
<td>=</td>
<td><code>[ ]</code></td>
</tr>
<tr>
<td><code>remaux (j+1, j’ : list)</code></td>
<td>=</td>
<td><code>remaux (j, list)</code></td>
</tr>
</tbody>
</table>
Exercise

• Consider these additional definitions:

\[
\text{result ::= Yes | Maybe}
\]

\[
\text{isempty :: ListSyntax -> Result}
\]

\[
\begin{align*}
isempty (\text{empty}) & = \text{Yes} \\
isempty (\text{single j}) & = \text{Maybe} \\
isempty (\text{cons (j, s)}) & = \text{Maybe} \\
isempty (\text{concat (s}_1, s_2)) & = \text{if (isempty (s}_1) = Yes) \text{ and isempty (s}_2) = \text{Yes} \text{ then Yes} \text{ else Maybe} \\
isempty (\text{take (j, s)}) & = \text{Maybe} \\
isempty (\text{rem (j,s)}) & = \text{Maybe}
\end{align*}
\]

• 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 alternative:
  
  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