CSE 331
Floyd Logic
Katherine Murphy

Credits: Profs. Kevin Zatloukal and James Wilcox
Reasoning So Far

• Code so far made up of three elements
  – straight-line code
  – conditionals
  – recursion

• Know how to reason (think) about these already
  – saw the first two already
  – we reasoned about recursion in math, but this can be done in code also
    our code is direct translation of math, so easy to switch between
Consider this code

```javascript
// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: bigint, b: bigint): bigint => {
  if (a >= 0n && b >= 0n) {
    const L: List = cons(a, cons(b, nil));
    return sum(L);
  }
  ...
}
```

- Known facts include “a ≥ 0”, “b ≥ 0”, and “L = cons(...)”
- Prove that postcondition holds: “sum(L) ≥ 0”
Reasoning About Recursion

// @param n a natural number
// @returns n*n
cssquare = (n: bigint): bigint => {
  if (n === 0n) {
    return 0n;
  } else {
    return square(n - 1n) + n + n - 1n;
  }
};

• How do we check correctness?

• Option 1: translate this to math

\[
\begin{align*}
\text{func square(0) } &= 0 \\
\text{square(n+1) } &= \text{square(n) + 2(n+1) - 1} \quad \text{for any } n \in \mathbb{N}
\end{align*}
\]
Reasoning About Recursion

// @param n a natural number
// @returns n*n
c
const square = (n: bigint): bigint => {
  ...
};

func square(0) := 0
square(n+1) := square(n) + 2(n+1) - 1 for any n : ℤ

• Prove that square(n) = n^2 for any n : ℤ

• Structural induction requires proving two implications
  – base case: prove square(0) = 0^2
  – inductive step: prove square(n+1) = (n+1)^2
    can use the fact that square(n) = n^2
Reasoning About Recursion

// @param n a natural number
// @returns n*n
const square = (n: bigint): bigint => {
    if (n === 0n) {
        return 0n;
    } else {
        return square(n - 1n) + n + n - 1n;
    }
};

• Option 2: reason directly about the code

• Known fact at top return: n = 0

    square(0) = 0
    = 0^2 (code)
Reasoning About Recursion

// @param n a natural number
// @returns n*n  
const square = (n: bigint): bigint => {
  if (n === 0n) {
    return 0n;
  } else {
    return square(n - 1n) + n + n - 1n;
  }
};

• Known fact at bottom return: n > 0

  square(n)       = square(n - 1) + 2n - 1
  = (n - 1)^2 + 2n - 1
  = n^2 - 2n + 1 + 2n + 1
  = n^2

why is it okay to assume square is correct when we’re checking it?

Inductive Hypothesis
Reasoning So Far

• Code so far made up of three elements
  – straight-line code
  – conditionals
  – structural recursion

• Any\(^1\) program can be written with just these
  – we could stop the course right here!

• For performance reasons, we often use more
  – this week: mutation of local variables
  – later: mutation of arrays and heap data

\(^1\) only exception is code with infinite loops
Brief History of Software

• Computers used to be very slow
  Kevin’s first computer had 64k of memory

• Software had to be extremely efficient
  – loops, mutation all over the place
  – very hard to write correctly, so it did very little
Brief History of Software

• Computers used to be very slow
  – software had to be extremely efficient

• Today, **programmers** are the scarcest resource
  – we have enormous computing resources

• **Anti-pattern**: favoring efficiency over correctness
  – programmers overestimate importance of efficiency
    “programmers are notoriously bad” at guessing what is slow — B. Liskov
    “premature optimization is the root of all evil” — D. Knuth
  – programmers are overconfident about correctness
    routinely takes 3x as long as expected to get it right
“Programmers overestimate the importance of efficiency and underestimate the difficulty of correctness.”

— Class slogan #3
## Correctness Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Testing</th>
<th>Tools</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>small # of inputs</td>
<td>exhaustive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>straight from spec</td>
<td>heuristics</td>
<td>type checking</td>
<td>code reviews</td>
</tr>
<tr>
<td>2</td>
<td>no mutation</td>
<td>“</td>
<td>libraries</td>
<td>calculation induction</td>
</tr>
<tr>
<td>3</td>
<td>local variable mutation</td>
<td>“</td>
<td>“</td>
<td>Floyd logic</td>
</tr>
<tr>
<td>4</td>
<td>array mutation</td>
<td>“</td>
<td>“</td>
<td>for-any facts</td>
</tr>
<tr>
<td>5</td>
<td>heap state mutation</td>
<td>“</td>
<td>“</td>
<td>rep invariants</td>
</tr>
</tbody>
</table>
Recall: Finding Facts at a Return Statement

• Consider this code

```javascript
// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: bigint, b: bigint): bigint => {
  if (a >= 0n && b >= 0n) {
    a = a - 1n;
    const L: List = cons(a, cons(b, nil));
    return sum(L);
  }
  ...
}
```

• Facts no longer hold throughout the function

• When we state a fact, we have to say **where** it holds
Recall: Finding Facts at a Return Statement

// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: bigint, b: bigint): bigint => {
  if (a >= 0n && b >= 0n) {
    a = a - 1n;
    const L: List = cons(a, cons(b, nil));
    return sum(L);
  }
}

• When we state a fact, we have to say where it holds

• \{\{ .. \}\} notation indicates facts true at that point
  – cannot assume those are true anywhere else
Recall: Finding Facts at a Return Statement

// Inputs a and b must be integers.
// Returns a non-negative integer.

const f = (a: bigint, b: bigint): bigint => {
  if (a >= 0n && b >= 0n) {
    a = a - 1n;
    const L: List = cons(a, cons(b, nil));
    return sum(L);
  }
}

- There are mechanical tools for moving facts around
  - “forward reasoning” says how they change as we move down
  - “backward reasoning” says how they change as we move up
Recall: Finding Facts at a Return Statement

// Inputs a and b must be integers.
// Returns a non-negative integer.

const f = (a: bigint, b: bigint): bigint => {
    if (a >= 0n && b >= 0n) {
        a = a - 1n;  
        const L: List = cons(a, cons(b, nil));
        return sum(L);
    }
}

- Professionals are insanely good at forward reasoning
  - “programmers are the Olympic athletes of forward reasoning” - James
  - you’ll have an edge by learning backward reasoning too
Floyd Logic
Floyd Logic

- Invented by Robert Floyd and Sir Anthony Hoare
  - Floyd won the Turing award in 1978
  - Hoare won the Turing award in 1980
The program state is the values of the variables

An assertion (in \{\ .. \}) is a T/F claim about the state
  - an assertion “holds” if the claim is true
  - assertions are math not code
    (we do our reasoning in math)

Most important assertions:
  - precondition: claim about the state when the function starts
  - postcondition: claim about the state when the function ends
Hoare Triples

• A **Hoare triple** has two assertions and some code

\[
\{\{ P \}\}\quad S \quad \{\{ Q \}\}
\]

– \( P \) is the precondition, \( Q \) is the postcondition
– \( S \) is the code

• **Triple is “valid”** if the code is correct:
  – \( S \) takes *any* state satisfying \( P \) into a state satisfying \( Q \)
    does not matter what the code does if \( P \) does not hold initially
  – otherwise, the triple is invalid
Correctness Example

```javascript
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
const f = (n: bigint): bigint => {
    n = n + 3n;
    return n * n;
};
```

- Check that value returned, \( m = n^2 \), satisfies \( m \geq 10 \)
Correctness Example

```javascript
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
const f = (n: bigint): bigint => {
    {{ n >= 1 }}
    n = n + 3n;
    {{ n^2 >= 10 }}
    return n * n;
};
```

- Precondition and postcondition come from spec
- Remains to check that the triple is valid
Hoare Triples with No Code

• Code could be empty:

\[
\begin{align*}
&\{\{ P \}\} \\
&\{\{ Q \}\}
\end{align*}
\]

• When is such a triple valid?
  – valid iff P implies Q
  – we already know how to check validity in this case:
    prove each fact in Q by calculation, using facts from P
Hoare Triples with No Code

• Code could be empty:

\[
\begin{align*}
{\{ a \geq 0, \ b \geq 0, \ L = \text{cons}(a, \text{cons}(b, \text{nil})) \}} \\
{\{ \text{sum}(L) \geq 0 \}}
\end{align*}
\]

• Check that P implies Q by calculation

\[
\begin{align*}
\text{sum}(L) & = \text{sum}(\text{cons}(a, \text{cons}(b, \text{nil}))) \\
& = a + \text{sum}(\text{cons}(b, \text{nil})) \\
& = a + b + \text{sum}(\text{nil}) \\
& = a + b \\
& \geq 0 + b \\
& \geq 0 + 0 \\
& = 0
\end{align*}
\]

since L = …
def of sum
def of sum
def of sum
since a ≥ 0
since b ≥ 0
Stronger Assertions vs Specifications

- **Assertion** is stronger iff it holds in a subset of states

- **Stronger** assertion implies the weaker one
  - stronger is a synonym for “implies”
  - weaker is a synonym for “is implied by”
Stronger Assertions vs Specifications

- **Assertion** is stronger iff it holds in a subset of states.
- Weakest possible assertion is “true” (all states)
  - an empty assertion (“”) also means “true”
- Strongest possible assertion is “false” (no states!)
Hoare Triples with Multiple Lines of Code

• Code with multiple lines:

\[
\begin{align*}
\{\{ P \}\} & \quad \{\{ P \}\} \\
S & \quad S \\
T & \quad R \\
\{\{ Q \}\} & \quad \{\{ Q \}\}
\end{align*}
\]

• Valid iff there exists an \( R \) making both triples valid
  – i.e., \( \{\{ P \}\} S \{\{ R \}\} \) is valid and \( \{\{ R \}\} T \{\{ Q \}\} \) is valid

• Will see next how to put these to good use...
Mechanical Reasoning Tools

• Forward / backward reasoning fill in assertions
  – mechanically create valid triples

• **Forward** reasoning fills in postcondition
  
  \[ \{\{ P \}\} \ S \ \{\{ \_ \}\} \]
  
  – gives *strongest* postcondition making the triple valid

• **Backward** reasoning fills in precondition
  
  \[ \{\{ \_ \}\} \ S \ \{\{ Q \}\} \]
  
  – gives *weakest* precondition making the triple valid
Correctness via Forward Reasoning

• Apply forward reasoning

\[
\begin{align*}
\{\{ P \}\} & \quad \{\{ P \}\} \\
S & \quad S \\
\{\{ Q \}\} & \quad \{\{ R \}\} \\
\{\{ Q \}\} & \quad \{\{ Q \}\}
\end{align*}
\]

– first triple is always valid
– only need to check second triple
  just requires proving an implication (since no code is present)

• If second triple is invalid, the code is **incorrect**
  – true because \( R \) is the strongest assertion possible here
Correctness via Backward Reasoning

• Apply backward reasoning

\[
\begin{align*}
\{\{P\}\} & \quad \{\{P\}\} \\
S & \quad \{\{R\}\} \\
\{\{Q\}\} & \quad S \\
\{\{Q\}\} & \quad \{\{Q\}\}
\end{align*}
\]

1. \[\text{second triple is always valid}\]
   \[\text{only need to check first triple}\]
   \[\text{just requires proving an implication (since no code is present)}\]

• If first triple is invalid, the code is **incorrect**
   \[\text{true because } R \text{ is the weakest assertion possible here}\]
Mechanical Reasoning Tools

• Forward / backward reasoning fill in assertions
  – mechanically create valid triples

• Reduce correctness to proving implications
  – this was already true for functional code
  – will soon have the same for imperative code

• Implication will be false if the code is incorrect
  – reasoning can verify correct code
  – reasoning will never accept incorrect code
Correctness via Forward & Backward

• Can use both types of reasoning on longer code

\[
\begin{align*}
\{\{ P \}\} \\
S \\
\{\{ R_1 \}\} \\
\{\{ R_2 \}\} \\
T \\
\{\{ Q \}\}
\end{align*}
\]

– first and third triples is always valid
– only need to check second triple
  verify that \( R_1 \) implies \( R_2 \)
Forward & Backward Reasoning
Forward and Backward Reasoning

- Imperative code made up of
  - assignments (mutation)
  - conditionals
  - loops

- Anything can be rewritten with just these

- We will learn forward / backward rules to handle them
  - will also learn a rule for function calls
  - once we have those, we are done
**Example Forward Reasoning through Assignments**

```latex
{\{ w > 0 \}}
\begin{align*}
x &= 17n; \\
{\{ \rule{10cm}{0.1pt} \}} \\
y &= 42n; \\
{\{ \rule{10cm}{0.1pt} \}} \\
z &= w + x + y; \\
{\{ \rule{10cm}{0.1pt} \}}
\end{align*}
```

- What do we know is true after \( x = 17 \)?
  - want the strongest postcondition (most precise)
Example Forward Reasoning through Assignments

```plaintext
{{ w > 0 }}
x = 17n;
{{ w > 0 and x = 17 }}
y = 42n;
{{ _________________________ }}
z = w + x + y;
{{ _________________________ }}
```

• What do we know is true after $x = 17$?
  – $w$ was not changed, so $w > 0$ is still true
  – $x$ is now 17

• What do we know is true after $y = 42$?
Example Forward Reasoning through Assignments

\[
\begin{align*}
\{\ w > 0 \} \\
    & \quad x = 17n; \\
\{\ w > 0 \ \text{and} \ x = 17 \} \\
    & \quad y = 42n; \\
\{\ w > 0 \ \text{and} \ x = 17 \ \text{and} \ y = 42 \} \\
    & \quad z = w + x + y; \\
\end{align*}
\]

• What do we know is true after \( y = 42 \)?
  – \( w \) and \( x \) were not changed, so previous facts still true
  – \( y \) is now 42

• What do we know is true after \( z = w + x + y \)?
Example Forward Reasoning through Assignments

\[
\begin{align*}
\{ & w > 0 \} \\
x & = 17n; \\
\{ & w > 0 \text{ and } x = 17 \} \\
y & = 42n; \\
\{ & w > 0 \text{ and } x = 17 \text{ and } y = 42 \} \\
z & = w + x + y; \\
\{ & w > 0 \text{ and } x = 17 \text{ and } y = 42 \text{ and } z = w + x + y \}
\end{align*}
\]

- What do we know is true after \( z = w + x + y \)?
  - \( w, x, \) and \( y \) were not changed, so previous facts still true
  - \( z \) is now \( w + x + y \)

- Could also write \( z = w + 59 \) (since \( x = 17 \) and \( y = 42 \))
Example Forward Reasoning through Assignments

\[
\begin{align*}
\{ \{ w > 0 \}\} \\
x &= 17n; \\
\{ \{ w > 0 \text{ and } x = 17 \}\} \\
y &= 42n; \\
\{ \{ w > 0 \text{ and } x = 17 \text{ and } y = 42 \}\} \\
z &= w + x + y; \\
\{ \{ w > 0 \text{ and } x = 17 \text{ and } y = 42 \text{ and } z = w + x + y \}\}
\end{align*}
\]

- Could write \( z = w + 59 \), but do not write \( z > 59 \)!
  - that is true since \( w > 0 \), but...
Could write $z = w + 59$, but do not write $z > 59$!
   – that is true since $w > 0$, but...
Example Forward Reasoning through Assignments

\{\{ w > 0 \}\}\n \hspace{1em} x = 17n; \\
\{\{ w > 0 \text{ and } x = 17 \}\}\n \hspace{1em} y = 42n; \\
\{\{ w > 0 \text{ and } x = 17 \text{ and } y = 42 \}\}\n \hspace{1em} z = w + x + y; \\
\{\{ w > 0 \text{ and } x = 17 \text{ and } y = 42 \text{ and } z = w + x + y \}\}

- **Could write** $z = w + 59$, **but do not write** $z > 59$! 
  - that is true since $w > 0$, but...
  - that is not the **strongest** postcondition
    correctness check could now fail even if the code is right
Code Example of Forward Reasoning

```javascript
const f = (w: bigint): bigint => {
    const x = 17n;
    const y = 42n;
    const z = w + x + y;
    return z;
};
```

• Let’s check correctness using Floyd logic…
Code Example of Forward Reasoning

// @param w an integer > 0
// @returns an integer z > 59
const f = (w: bigint): bigint => {
  {\{ w > 0 \}\}
  const x = 17n;
  const y = 42n;
  const z = w + x + y;
  {\{ z > 59 \}\}
  return z;
};

• Reason forward...
Code Example of Forward Reasoning

```javascript
// @param w an integer > 0
// @returns an integer z > 59
const f = (w: bigint): bigint => {
    const x = 17n;
    const y = 42n;
    const z = w + x + y;
    return z;
};
```

- Check implication:
  
  \[
  z = w + x + y \\
  = w + 17 + y \\
  = w + 59 \\
  > 59
  \]
  
  since \( x = 17 \)
  since \( y = 42 \)
  since \( w > 0 \)
Code Example of Forward Reasoning

```javascript
// @param w an integer > 0
// @returns an integer z > 59
const f = (w: bigint): bigint => {
  const x = 17n;
  const y = 42n;
  const z = w + x + y;
  return z;
};
```

- How about if we use our old approach?

- Known facts: $w > 0$, $x = 17$, $y = 42$, and $z = w + x + y$

- Prove that postcondition holds: $z > 59$
Code Example of Forward Reasoning

// @param w an integer > 0
// @returns an integer z > 59
const f = (w: bigint): bigint => {
    const x = 17n;
    const y = 42n;
    const z = w + x + y;
    return z;
};

• We’ve been doing forward reasoning all quarter!
  – forward reasoning is (only) “and” with no mutation

• Line-by-line facts are for “let” (not “const”)

Forward Reasoning through Assignments

• Forward reasoning is trickier with mutation
  – gets harder if we mutate a variable

\[
\begin{align*}
w &= x + y; \\
\{\{ w = x + y \}\} \\
x &= 4n; \\
\{\{ w = x + y \text{ and } x = 4 \}\} \\
y &= 3n; \\
\{\{ w = x + y \text{ and } x = 4 \text{ and } y = 3 \}\}
\end{align*}
\]

• Final assertion is not necessarily true
  – \( w = x + y \) is true with their old values, not the new ones
  – changing the value of “x” can invalidate facts about \( x \)
    facts refer to the old value, not the new value
  – avoid this by using recognizing old versus new values
Forward Reasoning through Assignments

• Fix this by denoting original variable values
  – can use “x” and “y” to refer to current values
  – can use “x₀” and “y₀” to refer to original values
    rewrite existing facts in terms of the original values

\[
\begin{align*}
\{& \{ w = x + y \} \\
& x = 4n; \\
& \{& \{ w = x₀ + y \text{ and } x = 4 \} \\
& y = 3n; \\
& \{& \{ w = x₀ + y₀ \text{ and } x = 4 \text{ and } y = 3 \} \}
\end{align*}
\]

• Final assertion is now accurate
  – \( w \) is equal to the sum of the original values of \( x \) and \( y \)

WARNING:
Cannot use original values notation when values maintain a relationship to their previous value (i.e. incrementing, etc.)
Forward Reasoning through Assignments

• For assignments, general forward reasoning rule is

\[
\begin{align*}
\{ \{ P \} \} \\
x &= y; \\
\{ \{ P[x \mapsto x_0] \text{ and } x = y \} \}
\end{align*}
\]

– replace all “x”s in P with “x_0”s

• This process should be simplified in many cases

– do not use x_0 if we can write it in terms of new value
  we refer to this as the “OLD/NEW variable value technique”

– assertions will be easier to read without original values
  (Technically, this is weakening, but it’s usually fine
  Postconditions usually do not refer to old values of variables.)
Forward Reasoning through Assignments

- For assignments, general forward reasoning rule is

\[
\begin{align*}
\{ P \} \\
x = y; \\
\{ P[x \mapsto x_0] \text{ and } x = y \}
\end{align*}
\]

- If \( x_0 = f(x) \), then we can simplify this

\[
\begin{align*}
\{ P \} \\
x = \ldots x \ldots; \\
\{ P[x \mapsto f(x)] \}
\end{align*}
\]

- If assignment is “\( x = x_0 + 1 \)”, then “\( x_0 = x - 1 \)”
- If assignment is “\( x = 2x_0 \)”, then “\( x_0 = x/2 \)”
- Does not work for integer division (an un-invertible operation)
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */

const f = (n: bigint): bigint => {

  {{ n ≥ 1 }}
  n = n + 3n;
  {{ ______  }}
  {{ n² ≥ 10  }}

  return n * n;
};
Correctness Example by Forward Reasoning

/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */

cost f = (n: bigint): bigint => {
  {{ nOLD ≥ 1 }}
  n = n + 3n; // SCRATCH WORK: nNEW = nOLD + 3 → nOLD = nNEW - 3
  {{ _______ }}
  {{ n² ≥ 10 }}
  return n * n;
};
/**
 * @param n an integer with n \( \geq 1 \)
 * @returns an integer m with m \( \geq 10 \)
 */

const f = (n: bigint): bigint => {
    n = n + 3n; // SCRATCH WORK: \( n_{\text{NEW}} = n_{\text{OLD}} + 3 \Rightarrow n_{\text{OLD}} = n_{\text{NEW}} - 3 \)
    return n * n;
};
/**
* @param n an integer with n \( \geq 1 \)
* @returns an integer m with m \( \geq 10 \)
*/
const f = (n: bigint): bigint => {
  {{ n \( \geq 1 \) }}
  n = n + 3n; // SCRATCH WORK: \( n_{\text{NEW}} = n_{\text{OLD}} + 3 \rightarrow n_{\text{OLD}} = n_{\text{NEW}} - 3 \)
  {{ n - 3 \( \geq 1 \) }}
  {{ n^2 \geq 10 }}
  return n * n;
};
Correctness Example by Forward Reasoning

/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
const f = (n: bigint): bigint => {
  n = n + 3n;
  return n * n;
};

\[ n^2 \geq 4^2 \quad \text{since } n - 3 \geq 1 \text{ (i.e., } n \geq 4) \]
\[ = 16 \]
\[ > 10 \]
Example Backward Reasoning with Assignments

\[
\begin{align*}
&\{\quad\} \\
x &= 17n; \\
&\{\quad\} \\
y &= 42n; \\
&\{\quad\} \\
z &= w + x + y; \\
&\{ z < 0 \}
\end{align*}
\]

- What must be true before \( z = w + x + y \) so \( z < 0 \)?
  - want the weakest postcondition (most allowed states)
Example Backward Reasoning with Assignments

{{ _________________ }}
\[ x = 17n; \]
{{ _________________ }}
\[ y = 42n; \]
{{ \[ w + x + y < 0 \] }}
\[ z = w + x + y; \]
{{ \[ z < 0 \] }}

- What must be true before \( z = w + x + y \) so \( z < 0 \)?
  - must have \( w + x + y < 0 \) beforehand

- What must be true before \( y = 42 \) for \( w + x + y < 0 \)?
Example Backward Reasoning with Assignments

{x = 17n;}
{{ w + x + 42 < 0 }}
{y = 42n;}
{{ w + x + y < 0 }}
{z = w + x + y;}
{{ z < 0 }}

- What must be true before $y = 42$ for $w + x + y < 0$?
  - must have $w + x + 42 < 0$ beforehand

- What must be true before $x = 17$ for $w + x + 42 < 0$?
Example Backward Reasoning with Assignments

\[
\begin{align*}
\{ & w + 17 + 42 < 0 \} \\
& x = 17n; \\
\{ & w + x + 42 < 0 \} \\
& y = 42n; \\
\{ & w + x + y < 0 \} \\
& z = w + x + y; \\
\{ & z < 0 \}
\end{align*}
\]

- What must be true before \( x = 17 \) for \( w + x + 42 < 0 \)?
  - must have \( w + 59 < 0 \) beforehand

- All we did was substitute right side for the left side
  - e.g., substitute \( \text{“} w + x + y \text{”} \) for \( \text{“} z \text{”} \) in \( \text{“} z < 0 \text{”} \)
  - e.g., substitute \( \text{“} 42 \text{”} \) for \( \text{“} y \text{”} \) in \( \text{“} w + x + y < 0 \text{”} \)
  - e.g., substitute \( \text{“} 17 \text{”} \) for \( \text{“} x \text{”} \) in \( \text{“} w + x + 42 < 0 \text{”} \)
Backward Reasoning through Assignments

• For assignments, backward reasoning is substitution

\[
\{ Q[x \mapsto y] \}
\]
\[
\begin{array}{l}
  x = y; \\
  \{ Q \}
\end{array}
\]

– just replace all the “x”s with “y”s
– we will denote this substitution by \( Q[x \mapsto y] \)

• Mechanically simpler than forward reasoning
  – no need for subscripts **EVER!!! 😊**
Correctness Example by Backward Reasoning

```typescript
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
const f = (n: bigint): bigint => {
    // {{ n ≥ 1 }}
    n = n + 3n;
    // {{ n² ≥ 10 }}
    return n * n;
};
```

- Code is correct if this triple is valid...
Correctness Example by Backward Reasoning

```javascript
/**
 * @param n an integer with \( n \geq 1 \)
 * @returns an integer \( m \) with \( m \geq 10 \)
 */
const f = (n: bigint): bigint => {
    {{ n \geq 1 }}
    {{ (n + 3)^2 \geq 10 }}
    n = n + 3n;
    {{ n^2 \geq 10 }}
    return n * n;
};

(n+3)^2 \geq (1 + 3)^2 \quad \text{since } n \geq 1
= 16
> 10
```
Conditionals
Conditionals in Functional Programming

// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: bigint, b: bigint): bigint => {
    if (a >= 0n && b >= 0n) {
        const L: List = cons(a, cons(b, nil));
        return sum(L);
    }

    ...

• Prior reasoning also included conditionals
  – what does that look like in Floyd logic?
// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: bigint, b: bigint): bigint => {
    if (a >= 0n && b >= 0n) {
        const L: List = cons(a, cons(b, nil));
        return sum(L);
    }
    ...
}

• Conditionals introduce extra facts in forward reasoning
  – simple “and” case since nothing is mutated
Conditionals in Floyd Logic

```javascript
// Returns an integer m with m > n
const g = (n: bigint): bigint => {
    let m;
    if (n >= 0n) {
        m = 2n * n + 1n;
    } else {
        m = 0n;
    }
    return m;
}
```

- Code like this was impossible without mutation
  - cannot write to a “const” after its declaration

- How do we handle it now?
Conditionals in Floyd Logic

// Returns an integer m with m > n
const g = (n: bigint): bigint => {
  let m;
  if (n >= 0n) {
    m = 2n * n + 1n;
  } else {
    m = 0n;
  }
  return m;
}

• Reason separately about each path to a return
  – handle each path the same as before
  – but now there can be multiple paths to one return
Conditionals in Floyd Logic

// Returns an integer m with m > n
const g = (n: bigint): bigint => {
    let m;
    if (n >= 0n) {
        m = 2n * n + 1n;
    } else {
        m = 0n;
    }

    return m;
}

- Check correctness path through “then” branch
// Returns an integer m with m > n
const g = (n: bigint): bigint => {

  let m;
  if (n >= 0n) {
    m = 2n * n + 1n;
  } else {
    m = 0n;
  }

  return m;
}
// Returns an integer m with m > n
const g = (n: bigint): bigint => {
    let m;
    if (n >= 0n) {
        m = 2n * n + 1n;
    } else {
        m = 0n;
    }
    return m;
}
```
// Returns an integer m with m > n
const g = (n: bigint): bigint => {
    let m;
    if (n >= 0n) {
        m = 2n * n + 1n;
    } else {
        m = 0n;
    }
    return m;
}
```

\[
m = 2n + 1 > 2n \quad \text{since} \quad 1 > 0
\]

\[
m \geq n \quad \text{since} \quad n \geq 0
\]
// Returns an integer m with m > n
const g = (n: bigint): bigint => {

let m;
if (n >= 0n) {
    m = 2n * n + 1n;
} else {
    m = 0n;
}

{{ n ≥ 0 and m = 2n + 1 }}
{{ m > n }}
return m;

• Note: no mutation, so we can do this in our head
  – read along the path, and collect all the facts
// Returns an integer m with m > n
const g = (n: bigint): bigint => {
  let m;
  if (n >= 0n) {
    m = 2n * n + 1n;
  } else {
    m = 0n;
  }
  // n < 0 and m = 0
  // m > n
  return m;
}

- Check correctness path through “else” branch
  - note: no mutation, so we can do this in our head
// Returns an integer m with m > n
const g = (n: bigint): bigint => {

let m;
if (n >= 0n) {
    m = 2n * n + 1n;
} else {
    m = 0n;
}

{{m > n}}
return m;
}

• What is true after the either branches?
// Returns an integer m with m > n
const g = (n: bigint): bigint => {
  let m;
  if (n >= 0n) {
    m = 2n * n + 1n;
  } else {
    m = 0n;
  }

  {{{ (n ≥ 0 and m = 2n + 1) or (n < 0 and m = 0) }}}
  {{{ m > n }}}
  return m;
}

• What is true after the either branches?
  – the “or” means we have to reason by cases anyway!
Conditionals in Floyd Logic

// Returns an integer m with m > n
const g = (n: bigint): bigint => {
    let m;
    if (n >= 0n) {
        m = 2n * n + 1n;
    } else {
        return 0n;
    }
    return m;
}

• What is the state after a “return”? 
// Returns an integer m with m > n
const g = (n: bigint): bigint => {
    let m;
    if (n >= 0n) {
        m = 2n * n + 1n;
    } else {
        return 0n;
    }
    return m;
}

{{ (n ≥ 0 and m = 2n + 1) or (n < 0 and false) }}
{{ m > n }} simplifies to just n ≥ 0 and m = 2n + 1

• State after a “return” is false (no states)
Function Calls
Reasoning about Function Calls

```typescript
const f = (a: bigint, b: bigint): bigint => { .. }
```

- **Forward reasoning rule is**

  ```
  \{\{ P \}\} 
  x = f(a, b); 
  \{\{ P[x \mapsto x_0] \text{ and } R \}\} 
  ```

  **Must also check that** $P$ **implies** $P_2$

- **Backward reasoning rule is**

  ```
  \{\{ Q_1 \text{ and } P_2 \}\} 
  x = f(a, b); 
  \{\{ Q_1 \text{ and } Q_2 \}\} 
  ```

  **Must also check that** $R$ **implies** $Q_2$

  $Q_2$ **is the part of postcondition using “x”**
Loops
Correctness of Loops

• Assignment and condition reasoning is mechanical

• Loop reasoning **cannot** be made mechanical
  – no way around this
  (311 alert: this follows from Rice’s Theorem)

• Thankfully, one extra bit of information fixes this
  – need to provide a “loop invariant”
  – with the invariant, reasoning is again mechanical
Loop Invariants

• Loop invariant is true every time at the top of the loop

```{\texttt{Inv: I}}
\textbf{while} (\texttt{cond}) {
  S
}
```

  – must be true when we get to the top the first time
  – must remain true each time execute $S$ and loop back up

• Use “Inv:” to indicate a loop invariant

  otherwise, this assertion only claims to be true the first time at the loop
Loop Invariants

- Loop invariant is true every time at the top of the loop

```
{{ Inv: I }}
while (cond) {
  S
}
```

- must be true 0 times through the loop (at top the first time)
- if true \( n \) times through, must be true \( n+1 \) times through

- Why do these imply it is always true?
  - follows by structural induction (on \( \mathbb{N} \))
Checking Correctness with Loop Invariants

How do we check validity with a loop invariant?
  – intermediate assertion splits into three triples to check
Checking Correctness with Loop Invariants

{{ P }}

{{ Inv: I }}

while (cond) {
    _
}

{{ Q }}

Splits correctness into three parts

1. I holds initially
2. S preserves I
3. Q holds when loop exits
Checking Correctness with Loop Invariants

\[
\begin{align*}
\{\{ P \}\} \\
\{\{ \textbf{Inv: I}\}\} \\
\textbf{while} \ (\text{cond}) \ {\{ \{ I \ \text{and} \ \text{cond}\}\} \\
\quad S \\
\quad \{\{ I\}\} \\
\} \\
\{\{ Q\}\}
\end{align*}
\]

Splits correctness into three parts

1. I holds initially
2. S preserves I
3. Q holds when loop exits
Checking Correctness with Loop Invariants

\[
\begin{align*}
\{ \text{P} \} \\
\{ \text{Inv: I} \} \\
\textbf{while} (\text{cond}) \{ \\
\quad \{ \text{I and cond} \} \\
\quad S \\
\quad \{ \text{I} \} \\
\} \\
\{ \text{I and not cond} \} \\
\{ \text{Q} \}
\end{align*}
\]

1. I holds initially
2. S preserves I
3. Q holds when loop exits

Splits correctness into three parts

1. I holds initially
2. S preserves I
3. Q holds when loop exits
Checking Correctness with Loop Invariants

\[
\begin{align*}
\{\{ P \}\} \\
\{\{ \text{Inv: I} \}\} \\
\textbf{while} \ (\text{cond}) \ {\{ } \\
\quad \textbf{S} \\
\{\} \\
\{\{ Q \}\}
\end{align*}
\]

Formally, invariant split this into three Hoare triples:

1. \{\{ P \}\} \{\{ I \}\} \quad \text{I holds initially}
2. \{\{ I \text{ and cond} \}\} \textbf{S} \{\{ I \}\} \quad \text{S preserves I}
3. \{\{ I \text{ and not cond} \}\} \{\{ Q \}\} \quad \text{Q holds when loop exits}
Example Loop Correctness

• Recursive function to calculate $1 + 2 + ... + n$

\[
\text{func} \; \text{sum-to}(0) := 0 \\
\text{sum-to}(n+1) := (n+1) + \text{sum-to}(n) \quad \text{for any } n : \mathbb{N}
\]

• This loop claims to calculate it as well

\[
\{\{\}\} \\
\text{let } i : \text{bigint} = 0; \\
\text{let } s : \text{bigint} = 0; \\
\{\{ \text{Inv: } s = \text{sum-to}(i) \} \} \\
\text{while} \ (i \neq n) \{ \\
\quad i = i + 1n; \\
\quad s = s + i; \\
\} \\
\{\{ s = \text{sum-to}(n) \} \}
\]

Easy to get this wrong!
- might be initializing “i” wrong ($i = 1$?)
- might be exiting at the wrong time ($i \neq n-1$?)
- might have the assignments in wrong order
- ...

Fact that we need to check 3 implications is a strong indication that more bugs are possible.
Example Loop Correctness

• Recursive function to calculate $1 + 2 + ... + n$

```
func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n) for any n : N
```

• This loop claims to calculate it as well

```
{{
let i: number = 0n;
let s: number = 0n;
{{ i = 0 and s = 0 }}
{{ Inv: s = sum-to(i) }}
while (i != n) {
    ...
}

sum-to(i) = sum-to(0) since i = 0
    = 0 def of sum-to
    = s
```
Example Loop Correctness

- **Recursive function to calculate** $1 + 2 + ... + n$

  \[
  \text{func } \text{sum-to}(0) := 0 \\
  \text{sum-to}(n+1) := (n+1) + \text{sum-to}(n) \quad \text{for any } n : \mathbb{N}
  \]

- **This loop claims to calculate it as well**

  \[
  \{\{ \text{Inv: } s = \text{sum-to}(i) \} \}
  \text{while } (i \neq n) \{ \\
  \{\{ s = \text{sum-to}(i) \text{ and } i \neq n \} \}
  i = i + 1; \\
  s = s + i; \\
  \{\{ s = \text{sum-to}(i) \} \}
  \}
Example Loop Correctness

• Recursive function to calculate $1 + 2 + ... + n$

```plaintext
func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n) for any n : ℕ
```

• This loop claims to calculate it as well

```plaintext
{{ Inv: s = sum-to(i) }}
while (i != n) {
  {{ s = sum-to(i) and i ≠ n }}
  i = i + 1
  {{ s = sum-to(i-1) and i-1 ≠ n }}
  s = s + i;
  {{ s = sum-to(i) }}
}
```
Example Loop Correctness

- **Recursive function to calculate** $1 + 2 + \ldots + n$

  ```
  func sum-to(0) := 0
  sum-to(n+1):= (n+1) + sum-to(n) for any n : \mathbb{N}
  ```

- **This loop claims to calculate it as well**

  ```
  {{ Inv: s = sum-to(i) }}
  while (i != n) {
    {{ s = sum-to(i) and i \neq n }}
    i = i + 1n;
    {{ s = sum-to(i-1) and i-1 \neq n }}
    s = s + i;
    {{ s - i = sum-to(i-1) and i-1 \neq n }}
  }
  {{ s = sum-to(i) }}
  ```

  \[ s = i + \text{sum-to}(i-1) \quad \text{since } s - i = \text{sum-to}(i-1) \]

  def of sum-to
Example Loop Correctness

• **Recursive function to calculate** $1 + 2 + ... + n$

$$\text{func } \text{sum-to(0)} : = 0$$
$$\text{sum-to(n+1)} : = (n+1) + \text{sum-to(n)} \quad \text{for any } n : \mathbb{N}$$

• **This loop claims to calculate it as well**

$$\{\{ \textbf{Inv: } s = \text{sum-to(i) } \}\}$$

$$\text{while } (i \neq n) \{$$
$$\quad i = i + 1;$$
$$\quad s = s + i;$$
$$\}\$$

$$\{\{ s = \text{sum-to(i) and } i = n \}\}$$

$$\{\{ s = \text{sum-to(n) } \}\}$$

$$\text{sum-to(n)} \quad = \text{sum-to(i)} \quad \text{since } i = n$$
$$\quad = s \quad \text{since } s = \text{sum-to(i)}$$
Termination

• This analysis does not check that the code terminates
  – it shows that the postcondition holds if the loop exits
  – but we never showed that the loop does exit

• Termination follows from the running time analysis
  – e.g., if the code runs in $O(n^2)$ time, then it terminates
  – an infinite loop would be $O(\infty)$
  – any finite bound on the running time proves it terminates

• Normal to also analyze the running time of our code, and we get termination already from that analysis
Loops & Recursion
Loops and Recursion

• To check a loop, we need a loop invariant

• Where does this come from?
  – part of the algorithm idea / design
    see 421 for more discussion
  – Inv and the progress step **formalize** the algorithm idea
    most programmers can easily formalize an English description
    (very tricky loops are the exception to this)

• Today, we’ll focus on converting *recursion* into a loop
  – HW Weave will fit these patterns
  – (more loops later)
Example Loop Correctness

• Recursive function to calculate $n^2$ without multiplying

\[
\text{func square}(0) := 0 \\
\text{square}(n+1) := \text{square}(n) + 2n + 1 \quad \text{for any } n : \mathbb{N}
\]

• We already proved that this calculates $n^2$
  – we can implement it directly with recursion

• Let’s try writing it with a loop instead...
Example Loop Correctness

\[
\text{func square}(0) := 0 \\
\text{square}(n+1) := \text{square}(n) + 2n + 1 \quad \text{for any } n : \mathbb{N}
\]

- **Loop idea for calculating** \(\text{square}(n)\):
  - calculate \(i = 0, 1, 2, ..., n\)
  - keep track of \(\text{square}(i)\) in “s” as we go along

\[
i = \begin{array}{cccccc}
0 & 1 & 2 & ... & n \\
\end{array}
\]

\[
s = \begin{array}{cccccc}
0 & 1 & 4 & ... & n^2 \\
\end{array}
\]

- **Formalize that idea in the loop invariant**
  along with the fact that we make **progress** by advancing \(i\) to \(i+1\) each step
Example Loop Correctness

```r
func square(0) := 0
square(n+1) := square(n) + 2n + 1  for any n : \mathbb{N}
```

- Loop implementation
  ```r
  let i: bigint = 0n;
  let s: bigint = 0n;
  {{ Inv: s = square(i) }}
  while (i != n) {
    s = s + i + i + 1n;
    i = i + 1n;
  }
  return s;
  ```

  Loop invariant says how i and s relate
  s holds square(i), for whatever i
  i starts at 0 and increases to n

Now we can check correctness...
Example Loop Correctness

\[
\begin{align*}
\text{func} & \quad \text{square}(0) \quad := 0 \\
& \quad \text{square}(n+1) := \text{square}(n) + 2n + 1 \quad \text{for any } n : \mathbb{N}
\end{align*}
\]

- **Loop implementation**

```plaintext
{\{ \}
let i : bigint = 0n;
let s : bigint = 0n;
{\{ i = 0 \text{ and } s = 0 \}}
{\{ \text{Inv: } s = \text{square}(i) \}}
while (i != n) {
    s = s + i + i + 1n;
    i = i + 1n;
} return s;
```

\[\text{square}(i) = \text{square}(0) \quad \text{since } i = 0
\[
= 0 \quad \text{def of } \text{square}
\]

\[= s \quad \text{since } s = 0\]
Example Loop Correctness

\[
\text{func square(0) := 0} \\
\text{square(n+1) := square(n) + 2n + 1 \quad \text{for any } n : \mathbb{N}}
\]

• Loop implementation

\[
\text{let } i : \text{bigint} = 0n; \\
\text{let } s : \text{bigint} = 0n; \\
\text{\{\{ Inv: } s = \text{square}(i) \text{ \}\}} \\
\text{while } (i != n) \{ \\
\quad s = s + i + i + 1n; \\
\quad i = i + 1n; \\
\text{\}\} \\
\text{\{\{ s = \text{square}(i) \text{ and } i = n \}\}} \\
\text{\{\{ s = \text{square}(n) \}\}} \\
\text{return } s;
\]

\[
\text{square}(n) = \text{square}(i) \quad \text{since } i = n \\
= s \quad \text{since } s = \text{square}(i)
\]
Example Loop Correctness

```plaintext
func square(0) := 0
    square(n+1) := square(n) + 2n + 1 for any n : \mathbb{N}

• Loop implementation

{{ Inv: s = square(i) }}
while (i != n) {
    {{ s = square(i) and i \neq n }}
    s = s + i + i + 1n;
    i = i + 1n;
    {{ s = square(i) }}
}
return s;
```
Example Loop Correctness

```
func square(0) := 0
    square(n+1):= square(n) + 2n + 1  for any n : \mathbb{N}

• Loop implementation

{{ Inv: s = square(i) }}
while (i != n) {
    {{ s = square(i) and i \neq n }}
    s = s + i + i + 1n;
    {{ s = square(i+1) }}
    i = i + 1n;
    {{ s = square(i) }}
}
return s;
```
Example Loop Correctness

\[\text{func square}(0) := 0\]
\[\text{square}(n+1) := \text{square}(n) + 2n + 1 \quad \text{for any } n \in \mathbb{N}\]

- **Loop implementation**

\[
\begin{align*}
\{\{ \textbf{Inv: } s = \text{square}(i) \}\} \\
\textbf{while } (i \neq n) \{ \\
\quad \{\{ s = \text{square}(i) \text{ and } i \neq n \}\} \\
\quad \{\{ s + 2i + 1 = \text{square}(i+1) \}\} \\
\quad s &= s + i + i + 1n; \\
\quad \{\{ s = \text{square}(i+1) \}\} \\
\quad i &= i + 1n; \\
\quad \{\{ s = \text{square}(i) \}\} \\
\} \\
\textbf{return } s;
\end{align*}
\]
Example Loop Correctness

```
func square(0) := 0
square(n+1) := square(n) + 2n + 1 for any n ∈ ℕ
```

• Loop implementation

```
{{ Inv: s = square(i) }}
while (i ≠ n) {
  {{ s = square(i) and i ≠ n }}
  {{ s + 2i + 1 = square(i+1) }}
  s = s + i + i + 1n;
  {{ s = square(i+1) }}
  i = i + 1n;
  {{ s = square(i) }}
  s + 2i + 1 = square(i) + 2i + 1 = square(i+1)
  since s = square(i)
  def of square
}
```

return s;
“Bottom Up” Loops on Natural Numbers

• Previous examples store function value in a variable

\[
\{
\text{Inv: } s = \text{sum-to}(i) \}
\]
\[
\{
\text{Inv: } s = \text{square}(i) \}
\]

• Start with \( i = 0 \) and work up to \( i = n \)

• Call this a “bottom up” implementation
  – evaluates in the same order as recursion
  – from the base case up to the full input
“Bottom Up” Loops on the Natural Numbers

func f(0) := …
    f(n+1) := …f(n)… for any n : \mathbb{N}

• Can be implemented with a loop like this

const f = (n: bigint): bigint => {
    let i: bigint = 0n;
    let s: bigint = “…”; // = f(0)
    {{ Inv: s = f(i) }}
    while (i != n) {
        s = “…f(i)…”[f(i) \mapsto s] // = f(i+1)
        i = i + 1n;
    }
    return s;
};
“Bottom Up” Loops on Lists

• Works nicely on \( \mathbb{N} \)
  – numbers are built up from 0 using \( \text{succ} (+1) \)
  – e.g., build \( n = 3 \) up from 0

\[
  n = \begin{array}{cccc}
    & 3 & \rightarrow & 2 & \rightarrow & 1 & \rightarrow & 0 \\
  \end{array}
\]

• What about List?
  – lists are built up from nil using \( \text{cons} \)
  – e.g., build \( L = \text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil}))) \) from nil:

\[
  L = \begin{array}{cccc}
    & 1 & \rightarrow & 2 & \rightarrow & 3 & \rightarrow & \text{nil} \\
  \end{array}
\]
“Bottom Up” Loops on Lists?

• What about List?
  – lists are built up from nil using cons
  – e.g., build \( L = \text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil}))) \) from nil:

\[
\begin{array}{c}
L = \text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil}))) \\
\end{array}
\]

• First step to build \( L \) is to build \( \text{cons}(3, \text{nil}) \) from nil
  – how do we know what number to put in front of nil?
    3 is all the way at the end of the list!
  – how can we fix this?
  – reverse the list!
Example “Bottom Up” List Loop

```
f func twice(nil) := nil
t twice(cons(x, L)) := cons(2x, twice(L))  for any x : \mathbb{Z} and L : List
```

• **Loop idea for calculating** `twice(L)`:  
  – **store** `rev(L)` in “R”

```
L = 1 → 2 → 3 → nil
R = 3 → 2 → 1 → nil
```

  – watch what happens as we move R forward...
func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : ℤ and L : List

• Loop idea for calculating twice(L):
  – store rev(L) in “R”
  – moving forward in R is moving backward in L...

\[
\begin{align*}
L &= \begin{array}{c}
1 \\
2 \\
3 \\
nil
\end{array} \\
R &= \begin{array}{c}
3 \\
2 \\
1 \\
nil
\end{array} \\
R.tl &= \begin{array}{c}
2 \\
1 \\
nil
\end{array}
\]

– as R moves forward, rev(R) remains a prefix of L
Example “Bottom Up” List Loop

\[
\text{func } \text{twice(nil)} := \text{nil} \\
\text{twice(cons(x, L))} := \text{cons}(2x, \text{twice(L)}) \quad \text{for any } x : \mathbb{Z} \text{ and } L : \text{List}
\]

• **Loop idea for calculating** twice(L):  
  – store rev(L) in “R”  
  – moving forward in R is moving backward in L...

```
L = 1 → 2 → 3 → nil  
R = 3 → 2 → 1 → nil  
R.tl = 2 → 1 → nil
```

  – **value dropped from** R was last(L) = 3  
    can use it to build cons(3, nil)
Example “Bottom Up” List Loop

\[
\text{func twice(nil) := nil} \\
twice(\text{cons}(x, L)) := \text{cons}(2x, \text{twice}(L)) \quad \text{for any } x : \mathbb{Z} \text{ and } L : \text{List}
\]

- **Loop idea for calculating** \(\text{twice}(L)\):
  - store \(\text{rev}(L)\) in “R” initially. move forward to \(R.tl\), etc.
  - add items skipped over by \(R\) to the front of “S”

- \(L = 1 \rightarrow 2 \rightarrow 3 \rightarrow \text{nil}\)
- \(R = 2 \rightarrow 1 \rightarrow \text{nil}\)
- \(S = 3 \rightarrow \text{nil}\)

- as \(R\) moves forward, \(S\) stores a **suffix** of \(L\)
Example “Bottom Up” List Loop

\[
\text{func twice(nil) := nil}
\]
\[
\text{twice(cons(x, L)) := cons(2x, twice(L)) for any } x : \mathbb{Z} \text{ and } L : \text{List}
\]

- **Loop idea for calculating** twice(L):
  
  – store rev(L) in “R” initially. move forward to R.tl, etc.
  
  – add items skipped over by R to the front of “S”

L = \[\begin{array}{c}
1 & \rightarrow & 2 & \rightarrow & 3 & \rightarrow & \text{nil}
\end{array}\]

\(\text{rev}(R)\)

S

R = \[\begin{array}{c}
2 & \rightarrow & 1 & \rightarrow & \text{nil}
\end{array}\]

S = \[\begin{array}{c}
3 \rightarrow \text{nil}
\end{array}\]
Example “Bottom Up” List Loop

```plaintext
func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))  for any x : ℤ and L : List
```

- **Loop idea for calculating** twice(L):
  - store rev(L) in “R” initially. move forward to R.tl, etc.
  - add items skipped over by R to the front of “S”

```
L = 1 ➔ 2 ➔ 3 ➔ nil
    ➔ rev(R)
            ➔ S

R = 1 ➔ nil

S = 2 ➔ 3 ➔ nil
```
Example “Bottom Up” List Loop

L = concat(rev(R), S)
Example “Bottom Up” List Loop

\[ L = \begin{array}{c}
1 \\
2 \\
3 \\
nil
\end{array} \]

\[ R \]
1 \ 
3 \rightarrow 2 \rightarrow 1 \rightarrow nil
2 \ 
2 \rightarrow 1 \rightarrow nil
3 \ 
1 \rightarrow nil
4 \ 
nil

\[ S \]
3 \rightarrow nil
2 \rightarrow 3 \rightarrow nil
1 \rightarrow 2 \rightarrow 3 \rightarrow nil

S rebuilds the list L “bottom up” calculate twice(L) “bottom up” as we go
Example “Bottom Up” List Loop

\[
\begin{align*}
\text{func} \text{ twice}(\text{nil}) & := \text{nil} \\
\text{twice}(\text{cons}(x, L)) & := \text{cons}(2x, \text{twice}(L)) \quad \text{for any } x : \mathbb{Z} \text{ and } L : \text{List}
\end{align*}
\]

• Loop idea for calculating \( \text{twice}(L) \):
  – store \( \text{rev}(L) \) in “R” initially. move forward to \( R.tl \), etc.
  – add items skipped over by \( R \) to the front of “S”
    \( S \) rebuilds the list \( L \) “bottom up”
  – calculate \( \text{twice}(S) \), as we go, in “T”

• Formalize that idea in the loop invariant

\[
L = \text{concat}(\text{rev}(R), S) \quad \text{and} \quad T = \text{twice}(S)
\]
Example “Bottom Up” List Loop

```plaintext
func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : ℤ and L : List
```

- This loop claims to calculate twice(L)...

```plaintext
let R: List = rev(L);
let S: List = nil;
let T: List = nil;
{{ Inv: L = concat(rev(R), S) and T = twice(S) }}
while (R.kind !== "nil") {
    T = cons(2n * R.hd, T); // Still need to check this.
    S = cons(R.hd, S);      // Hopefully obvious that it could be wrong.
    R = R.tl;
}
return T; // = twice(L)
```

(Testing length 0, 1, 2, 3 is not enough!)
Example “Bottom Up” List Loop

```
func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))  for any x : ℤ and L : List
```

- This loop claims to calculate twice(L)

```
...{ Inv: L = concat(rev(R), S) and T = twice(S) }
while (R.kind !== "nil") {
    T = cons(2n * R.hd, T);
    S = cons(R.hd, S);
    R = R.tl;
}
{ L = concat(rev(R), S) and T = twice(S) and R = nil }
{ T = twice(L) }
return T;  // = twice(L)
```
Example “Bottom Up” List Loop

\[
\text{func } \text{twice}(\text{nil}) := \text{nil} \\
\text{twice}(\text{cons}(x, L)) := \text{cons}(2x, \text{twice}(L)) \quad \text{for any } x : \mathbb{Z} \text{ and } L : \text{List}
\]

- **Check that \text{Inv} is implies the postcondition:**

\[
\begin{align*}
\{ & \{ L = \text{concat}(\text{rev}(R), S) \text{ and } T = \text{twice}(S) \text{ and } R = \text{nil} \} \\
& \{ T = \text{twice}(L) \} \\
L &= \text{concat}(\text{rev}(R), S) \\
    &= \text{concat}(\text{rev}(\text{nil}), S) \quad \text{since } R = \text{nil} \\
    &= \text{concat}(\text{nil}, S) \quad \text{def of } \text{rev} \\
    &= S \quad \text{def of } \text{concat} \\
T &= \text{twice}(S) \\
    &= \text{twice}(L) \quad \text{since } L = S
\end{align*}
\]
Example “Bottom Up” List Loop

```plaintext
func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))  for any x : ℤ and L : List
```

- This loop claims to calculate twice(L)

```plaintext
let R: List = rev(L);
let S: List = nil;
let T: List = nil;

{{ R = rev(L) and S = nil and T = nil }}

{{ Inv: L = concat(rev(R), S) and T = twice(S) }}

while (R.kind !== “nil”) {
    T = cons(2n * R.hd, T);
    S = cons(R.hd, S);
    R = R.tl;
}
```
Example “Bottom Up” List Loop

```
func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L))  for any x : Z and L : List
```

- Check that Inv is true initially:

  ```
  {{ R = rev(L) and S = nil and T = nil }}
  {{ Inv: L = concat(rev(R), S) and T = twice(S) }}
  ```

  `concat(rev(R), S)`
  
  = `concat(rev(rev(L)), S)`  since `R = rev(L)`
  = `concat(L, S)`  Lemma 3
  = `concat(L, nil)`  since `S = nil`
  = `L`  Lemma 2

  `twice(S)`
  
  = `twice(nil)`  since `S = nil`
  = `nil`  def of twice
  = `T`  since `T = nil`
**Example “Bottom Up” List Loop**

```
func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : ℤ and L : List
```

- **This loop claims to calculate** twice(L)

```
{ { Inv: L = concat(rev(R), S) and T = twice(S) } }
while (R.kind !== "nil") {
    { { L = concat(rev(R), S) and T = twice(S) and R ≠ nil } }
    T = cons(2n * R.hd, T);
    S = cons(R.hd, S);
    R = R.tl;
    { { L = concat(rev(R), S) and T = twice(S) } }
}
```
Example “Bottom Up” List Loop

```haskell
func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : ℤ and L : List
```

- **This loop claims to calculate** `twice(L)`

```haskell
{{ Inv: L = concat(rev(R), S) and T = twice(S) }}
while (R.kind !== “nil”) {
    {{ L = concat(rev(R), S) and T = twice(S) and R ≠ nil }}
    T = cons(2n * R.hd, T);
    S = cons(R.hd, S);
    {{ L = concat(rev(R.tl), S) and T = twice(S) }}
    R = R.tl;
    {{ L = concat(rev(R), S) and T = twice(S) }}
}
```
Example “Bottom Up” List Loop

```plaintext
func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : ℤ and L : List
```

- **This loop claims to calculate** `twice(L)`

```
{{ Inv: L = concat(rev(R), S) and T = twice(S) }}
while (R.kind !== “nil”) {
    {{ L = concat(rev(R), S) and T = twice(S) and R ≠ nil }}
    T = cons(2n * R.hd, T);
    {{ L = concat(rev(R.tl), cons(R.hd, S)) and T = twice(cons(R.hd, S)) }}
    S = cons(R.hd, S);
    {{ L = concat(rev(R.tl), S) and T = twice(S) }}
    R = R.tl;
    {{ L = concat(rev(R), S) and T = twice(S) }}
}
```
Example “Bottom Up” List Loop

\[
\begin{align*}
\textbf{func} & \quad \text{twice(nil)} \quad := \text{nil} \\
& \quad \text{twice}(\text{cons}(x, L)) \quad := \text{cons}(2x, \text{twice}(L)) \quad \text{for any } x : \mathbb{Z} \text{ and } L : \text{List}
\end{align*}
\]

- This loop claims to calculate \text{twice}(L)

\[
\begin{align*}
\{\{ \textbf{Inv: } L = \text{concat}(\text{rev}(R), S) \text{ and } T = \text{twice}(S) \}\} \\
\textbf{while } (R.\text{kind } \neq \text{ "nil"}) \{ \\
\quad \{\{ L = \text{concat}(\text{rev}(R), S) \text{ and } T = \text{twice}(S) \text{ and } R \neq \text{nil} \}\} \\
\quad \{\{ L = \text{concat}(\text{rev}(R.tl), \text{cons}(R.hd, S)) \text{ and } \text{cons}(2 \cdot R.hd, T) = \text{twice}(\text{cons}(R.hd, S)) \}\} \\
\quad T = \text{cons}(2n \cdot R.hd, T); \\
\quad \{\{ L = \text{concat}(\text{rev}(R.tl), \text{cons}(R.hd, S)) \text{ and } T = \text{twice}(\text{cons}(R.hd, S)) \}\} \\
\quad S = \text{cons}(R.hd, S); \\
\quad \{\{ L = \text{concat}(\text{rev}(R.tl), S) \text{ and } T = \text{twice}(S) \}\} \\
\quad R = R.tl; \\
\quad \{\{ L = \text{concat}(\text{rev}(R), S) \text{ and } T = \text{twice}(S) \}\}
\}
\]
Example “Bottom Up” List Loop

\[
\begin{align*}
\text{func } \text{twice}(\text{nil}) & := \text{nil} \\
\text{twice}(\text{cons}(x, L)) & := \text{cons}(2x, \text{twice}(L)) \quad \text{for any } x : \mathbb{Z} \text{ and } L : \text{List}
\end{align*}
\]

- Check that Inv is preserved by the loop body:

\[
\begin{align*}
\{ & \text{ L } = \text{concat}(\text{rev}(R), S) \text{ and } T = \text{twice}(S) \text{ and } R \neq \text{nil} \} \\
\{ & \text{ L } = \text{concat}(\text{rev}(R.\text{tl}), \text{cons}(R.\text{hd}, S)) \text{ and } \text{cons}(2 \cdot R.\text{hd}, T) = \text{twice}(\text{cons}(R.\text{hd}, S)) \} \\
\text{twice}(\text{cons}(R.\text{hd}, S)) & = \text{cons}(2 \cdot R.\text{hd}, \text{twice}(S)) \quad \text{def of twice} \\
& = \text{cons}(2 \cdot R.\text{hd}, T) \quad \text{since } T = \text{twice}(S)
\end{align*}
\]

Note that \( R \neq \text{nil} \) means \( R = \text{cons}(R.\text{hd}, R.\text{tl}) \)
Example “Bottom Up” List Loop

```
func twice(nil) := nil
twice(cons(x, L)) := cons(2x, twice(L)) for any x : Z and L : List
```

• Check that Inv is preserved by the loop body:

\[
\begin{align*}
\{ \{ L = \text{concat}(\text{rev}(R), S) \text{ and } T = \text{twice}(S) \text{ and } R \neq \text{nil} \} \} \\
\{ \{ L = \text{concat}(\text{rev}(R.tl), \text{cons}(R.hd, S)) \text{ and } \text{cons}(2 \cdot R.hd, T) = \text{twice}(\text{cons}(R.hd, S)) \} \} \\
L &= \text{concat}(\text{rev}(R), S) \\
&= \text{concat}(\text{rev}(\text{cons}(R.hd, R.tl)), S) \quad \text{since } R \neq \text{nil} \\
&= \text{concat}(\text{concat}(\text{rev}(R.tl), \text{cons}(R.hd, nil)), S) \quad \text{def of rev} \\
&= \text{concat}(\text{rev}(R.tl), \text{concat}(\text{cons}(R.hd, nil), S)) \quad \text{Lemma 4} \\
&= \text{concat}(\text{rev}(R.tl), \text{cons}(R.hd, \text{concat}(\text{nil}, S))) \quad \text{def of concat} \\
&= \text{concat}(\text{rev}(R.tl), \text{cons}(R.hd, S)) \quad \text{def of concat}
\end{align*}
\]
Example “Bottom Up” List Loop

\[
\text{func } \text{twice}(\text{nil}) := \text{nil} \\
\text{twice}(\text{cons}(x, L)) := \text{cons}(2x, \text{twice}(L)) \text{ for any } x : \mathbb{Z} \text{ and } L : \text{List}
\]

- This loop claims to calculate \(\text{twice}(L)\)

\[
\begin{align*}
\text{let } R : \text{List} &= \text{rev}(L); \\
\text{let } S : \text{List} &= \text{nil}; \\
\text{let } T : \text{List} &= \text{nil}; \\
\{\{ \text{Inv: } L = \text{concat}(\text{rev}(R), S) \text{ and } T = \text{twice}(S) \}\} \\
\text{while } (R.\text{kind} \neq \text{“nil”}) \{ \\
& \quad T = \text{cons}(2n \times R.\text{hd}, T); \\
& \quad S = \text{cons}(R.\text{hd}, S); \\
& \quad R = R.\text{tl}; \\
\}
\]

\[
\text{return } T; \quad // = \text{twice}(L)
\]

“\(S\)” is unused! We could remove it.

“\(S\)” is useful for proving correctness but it is not needed at run-time.
(Example of a “ghost” variable.)
“Bottom Up” Loops on Lists

\[
\begin{align*}
\text{func } f(\text{nil}) & := \ldots \\
f(\text{cons}(x, L)) & := \ldots f(L) \ldots \\
\text{for any } x : \mathbb{Z} \text{ and } L : \text{List}
\end{align*}
\]

- Can be implemented with a loop like this

```javascript
const f = (L: List): List => {
  let R: List = rev(L);
  let S: List = nil;
  let T: List = ...;  // = f(nil)
  {{ Inv: L = concat(rev(R), S) and T = f(S) }}
  while (R.kind !== "nil") {
    T = "... f(L) ..." [f(L) ↦ T]
    S = cons(R.hd, S);
    R = R.tl;
  }
  return T;  // = f(L)
};
```
Recursion versus Loops

\[
\begin{align*}
\text{func} \ \text{sum}(\text{nil}) & := 0 \\
\text{sum}(\text{cons}(x, L)) & := x + \text{sum}(L) \quad \text{for any } x : \mathbb{Z} \text{ and } L : \text{List}
\end{align*}
\]

- **This is bottom-up: to calculate** \(\text{sum}(\text{cons}(x, L))\)
  - computation order is back-to-front
  - recursively calculate \(n = \text{sum}(L)\)
  - when that returns, compute \(x + n\)

\[
\begin{array}{c}
\text{sum(cons(1,cons(2,nil)))} \\
\downarrow \\
\text{sum(cons(2,nil))} \\
\downarrow \\
\text{sum(nil)}
\end{array}
\]
Recursion versus Loops

```haskell
func sum(nil) := 0
    sum(cons(x, L)) := x + sum(L) for any x : ℤ and L : List
```

- This is bottom-up: to calculate \( \text{sum(cons(x, L))} \)
  - computation order is back-to-front
  - recursively calculate \( n = \text{sum(L)} \)
  - when that returns, compute \( x + n \)

- The natural loop is front-to-back.

- This is a fundamental tension!
Recursion versus Loops

• There is a fundamental tension between:
  – Natural recursive order (bottom-up, aka back-to-front)
  – Natural loop order (front-to-back)

• Three ways to bridge this gap:
  – Make the loop serve the recursion
    We just saw this with the bottom-up list loop template calling \( \text{rev}(L) \)
  – Make the recursion serve the loop
    Tail recursion, up next
  – Change the data structure
    Arrays
Tail Recursion

\[\begin{align*}
\text{func} \; \text{twice}(\text{nil}) & := \text{nil} \\
\text{twice}(\text{cons}(x, L)) & := \text{cons}(2x, \text{twice}(L)) \quad \text{for any } x : \mathbb{Z} \text{ and } L : \text{List}
\end{align*}\]

• **To calculate** \(\text{twice}(\text{cons}(x, L))\):
  – **recursively calculate** \(S = \text{twice}(L)\)
  – when that returns, **construct and return** \(\text{cons}(2x, S)\)

• **Not all functions require work after recursion:**

\[\begin{align*}
\text{func} \; \text{rev-acc}(\text{nil}, R) & := R \\
\text{rev-acc}(\text{cons}(x, L), R) & := \text{rev-acc}(L, \text{cons}(x, R)) \quad \text{for any } x : \mathbb{Z} \text{ and any } L, R : \text{List}
\end{align*}\]

  – such functions are called “tail recursive”
“Top Down” List Loop

• We can write a top-down sum function:

```haskell
func sum-acc(nil, acc) := acc
sum-acc(cons(x, L), acc) := sum-acc(L, x + acc)
```

• Translate to code without reversing the list:

```javascript
let s: bigint = 0n;
{{ Inv: sum-acc(L₀, 0) = sum-acc(L, s) }}
while (L.kind !== “nil”) {
  s = L.hd + s;
  L = L.tl;
}
return s;  // sum-acc(L₀, 0)
```

It’s immediate that the invariant is initially true

\[ \text{sum-acc}(L₀, 0) = \text{sum-acc}(L, s) \quad \text{since} \quad s = 0 \]
“Top Down” List Loop

• Check the body preserves invariant

```
let s: bigint = 0n;
{{ Inv: sum-acc(L_0, 0) = sum-acc(L, s) }}
while (L.kind !== "nil") {
    {{ sum-acc(L_0, 0) = sum-acc(L, s) and L ≠ nil }}
    s = L.hd + s;
    L = L.tl;
    {{ sum-acc(L_0, 0) = sum-acc(L, s) }}
}
return s;
```
“Top Down” List Loop

- Check the body preserves invariant

```plaintext
let s: bigint = 0n;
{{ Inv: sum-acc(L₀, 0) = sum-acc(L, s) }}
while (L.kind !== "nil") {
    {{ sum-acc(L₀, 0) = sum-acc(L, s) and L ≠ nil }}
    s = L.hd + s;
    {{ sum-acc(L₀, 0) = sum-acc(L.tl, s) }}
    L = L.tl;
    {{ sum-acc(L₀, 0) = sum-acc(L, s) }}
}
return s;
```
“Top Down” List Loop

- Check the body preserves invariant

```
let s: bigint = 0n;
{{ Inv: \text{sum-acc}(L_0, 0) = \text{sum-acc}(L, s) }}
while (L.kind !== "nil") {
    {{ \text{sum-acc}(L_0, 0) = \text{sum-acc}(L, s) \text{ and } L \neq \text{nil} }}
    {{ \text{sum-acc}(L_0, 0) = \text{sum-acc}(L.tl, L.hd + s) }}
    s = L.hd + s;
    {{ \text{sum-acc}(L_0, 0) = \text{sum-acc}(L.tl, s) }}
    L = L.tl;
    {{ \text{sum-acc}(L_0, 0) = \text{sum-acc}(L, s) }}
}
return s;
```

```
\text{sum-acc}(L_0, 0) = \text{sum-acc}(L, s) \\
= \text{sum-acc}(\text{cons}(L.hd, L.tl), s) \quad \text{since } L \neq \text{nil} \\
= \text{sum-acc}(L.tl, L.hd + s) \quad \text{def } \text{sum-acc}
```
“Top down” Loops on Lists

\[
\begin{align*}
\text{func } & f(\text{nil, acc}) := \text{acc} \\
& f(\text{cons}(x, L), \text{acc}) := f(L, \ldots x \ldots \text{acc} \ldots)
\end{align*}
\]

- Can be implemented with a loop like this

\[
\begin{align*}
\text{const } & f = (L: \text{List}, \text{acc}: \text{bigint}): \text{List} \Rightarrow \{ \\
\{\text{Inv: } f(L_0, \text{acc}_0) = f(L, \text{acc}) \}\} \\
& \text{while } (L.\text{kind } != \text{“nil”}) \{ \\
& \quad \text{acc} = \text{“...x...acc...”} \\
& \quad L = L.\text{tl}; \\
& \} \\
& \text{return } \text{acc}; \quad // = f(L_0, \text{acc}_0)
\end{align*}
\]
Tail Recursion Elimination

• Most functional languages eliminate tail recursion
  – acts like a loop at run-time
    Fast and no extra space usage
  – true of JavaScript as well

• Alternatives implementing recursion:
  1. Find a loop that implements it
     check correctness with Floyd logic
  2. Find an equivalent tail-recursive function
     check equivalence with structural induction
Recursion versus Loops

• There is a fundamental tension between:
  – Natural recursive order (bottom-up, aka back-to-front)
  – Natural loop order (front-to-back)

• Three ways to bridge this gap:
  – Make the loop serve the recursion
    Bottom-up list loop template calling \texttt{rev(L)}
  – Make the recursion serve the loop
    Tail recursion
  – Change the data structure
    Arrays, up next