CSE 331

Loops & Recursion

Kevin Zatloukal

Checking Correctness with Loop Invariants

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

Formally, invariant split this into three Hoare triples:

1.	{{ P }} {{ I }}	I **holds initially**
2.	{{ I and cond }} **S** {{ I }}	**S preserves** I
3.	{{ I and not cond }} {{ Q }}	Q **holds when loop exits**

Example Loop Correctness

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

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

- This loop claims to calculate it as well

```
{{ }}
let i: number = 0;
let s: number = 0;
{{ Inv: s = sum-to(i) }}
while (i != n) {
  i = i + 1;
  s = s + i;
}
{{ s = 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 = 0;
let s: number = 0;
{{ 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   **since** s = 0

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 : ℕ
```

- This loop claims to calculate it as well

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

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

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

sum-to(i+1) = (i+1) + sum-to(i)   **def of** sum-to
            = (i+1) + s   **since** s = sum-to(i)

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**

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

sum-to(n)
= sum-to(i)   **since** i = n
= s   **since** s = 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(infinity)
  - 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 and Recursion

Loops and Recursion

- In order 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

- Today, we'll focus on converting *recursion* into a loop
  - HW5 will fit these patterns
  - (more loops later)

Example Loop Correctness

- Recursive function to calculate n^2 without multiply

$$\begin{aligned} &\textbf{func}\ \mathrm{square}(0) && := 0 \\ &\phantom{\textbf{func}}\ \mathrm{square}(\mathrm{n}+1) && := \mathrm{square}(\mathrm{n}) + 2\mathrm{n} + 1 \qquad \text{for any } \mathrm{n} : \mathbb{N} \end{aligned}$$

- 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

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

- Loop implementation

```
let i: number = 0;
let s: number = 0;
while (i != n) {
  s = s + i + i + 1;
  i = i + 1;
}
return s;  // = square(n)
```

Needs a loop invariant!

Example Loop Correctness

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

- Loop implementation

```
let i: number = 0;
let s: number = 0;
{{ Inv: s = square(i) }}
while (i != n) {
  s = s + i + i + 1;
  i = i + 1;
}
return s;
```

Loop invariant says how i and s relate
s holds square(i), whatever i

i starts at 0 and increases to n

Now we can check correctness…

Example Loop Correctness

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

- Loop implementation

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

square(i)
= square(0) **since** i = 0
= 0 **def of** square
= s **since** s = 0

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 = s + i + i + 1;
  i = i + 1;
  {{ s = square(i) }}
}
return s;
```

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 + 1;
  {{ s = square(i+1) }}
  i = i + 1;
  {{ s = square(i) }}
}
return s;
```

s + 2i + 1 = square(i) + 2i + 1 **since** s = square(i)
= square(i+1) **def of** square

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 = s + i + i + 1;
   {{ s – 2i – 1 = square(i) and i ≠ n }}
   i = i + 1;
   {{ s – 2(i – 1) – 1 = square(i – 1) and i – 1 ≠ n }}
   {{ s = square(i) }}
}
return s;
```

s – 2(i – 1) – 1 = square(i – 1)
or equiv s = square(i – 1) + 2(i – 1) + 1

square(i)
= square(i – 1) + 2(i – 1) + 1 **def of square**
= s **since** s = square(i – 1) + ...

Example Loop Correctness

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

- **Loop implementation**

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

square(n)
= square(i) **since** i = n
= s **since** s = square(i)

“Bottom Up” Loops

- Previous examples store function value in a variable

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

  {{ **Inv**: $s = \text{square}(i)$ }}

- Start with $i = 0$ and work up to $i = n$

- Call this a “bottom up” implementation
  - calculates from the base case up to the full input
  - evaluates in the same order as recursion

“Bottom Up” Loops on the Natural Numbers

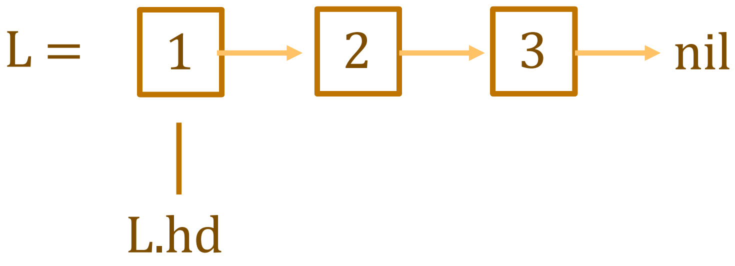
```
func f(0)      := …
     f(n+1)    := … f(n) …              for any n : ℕ
```

- Can be implemented with a loop like this

```
function f(n: number): number {
  let i: number = 0;
  let s: number = “…”;      // = f(0)
  {{ Inv: s = f(i) }}
  while (i != n) {
    s = “… f(i) …”[f(i) ↦ s]     // = f(i+1)
    i = i + 1;
  }
  return s;
}
```

"Bottom Up" Loops

- Works nicely on $\mathbb{N}$
  - start at 0 and work up to $n : \mathbb{N}$
- Does not work nicely on Lists
  - build $\text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil})))$ from nil then $3, 2, 1$
  - don't have easy access to 3 at the start (only 1 is easy to get)



“Top Down” Loops

- **Can try to work “top down” instead**
  - **start at n and work down to 0**
  - **start at full list and work down to nil**

- **Next two examples:**
  - **we will do this first for $\mathbb{N}$ (as a demonstration only)**

    bottom-up is the preferred way to write loops on $\mathbb{N}$
  - **then we will do this on Lists**

“Bottom Up” Computation

```
let i: number = 0;
let s: number = 0;
while (i != n) {
  s = s + i + i + 1;
  i = i + 1;
}
```

- **Operates as follows:**

i	s	
0	0	$= \text{square}(0)$
1	$0 + (2 \cdot 0 + 1)$	$= \text{square}(1)$
2	$0 + (2 \cdot 0 + 1) + (2 \cdot 1 + 1)$	$= \text{square}(2)$
3	$0 + (2 \cdot 0 + 1) + (2 \cdot 1 + 1) + (2 \cdot 2 + 1)$	$= \text{square}(3)$

$$\textbf{square}(n) = 0 + (2 \cdot 0 + 1) + (2 \cdot 1 + 1) + \ldots + (2 \cdot n + 1)$$

“Top Down” Computation

- “Top down” starts by adding large values
  - it operates as follows:

i	s
n	0
n – 1	(2·n + 1)
n – 2	(2·n + 1) + (2·(n-1) + 1)
n – 3	(2·n + 1) + (2·(n-1) + 1) + (2·(n-2) + 1)

≠ square(k)
for any k

  - how do we describe what we have calculated?

$$\mathbf{square}(n) = 0 + (2\cdot 0 + 1) + (2\cdot 1 + 1) + \ldots + (2\cdot n + 1)$$

$$= \underbrace{[0 + (2\cdot 0 + 1) + \ldots + (2\cdot(n - k - 1) + 1)]}_{\text{square}(n - k - 1)} + \underbrace{[(2\cdot(n - k) + 1) + \ldots + (2\cdot n + 1)]}_{s}$$

“Top Down” Computation

- “Top down” starts by adding large values
  - it operates as follows:

i	s
n	0
n – 1	$(2 \cdot n + 1)$
n – 2	$(2 \cdot n + 1) + (2 \cdot (n-1) + 1)$
n – 3	$(2 \cdot n + 1) + (2 \cdot (n-1) + 1) + (2 \cdot (n-2) + 1)$

  - how do we describe what we have calculated?

$$\text{square}(n) = \text{square}(i) + s$$

  - with “top down”, part missing is described by function call

still need to add square(i) to s to get the desired answer

Top Down on Natural Numbers

func square(0) := 0

square(n+1) := square(n) + 2n + 1   for any n : ℕ

- “Top down” loop implementation

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

Still need to check this.

(Why $s + 2i - 1$ here?)

Top Down on Natural Numbers

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

- “Top down” loop implementation

```
let i: number = n;
let s: number = 0;
{{ i = n and s = 0 }}
{{ Inv: square(n) = square(i) + s }}
while (i != 0) {
  s = s + i + i - 1;
  i = i - 1;
}
{{ s = square(n) }}
return s;
```

square(i) + s = square(i) **since** s = 0
= square(n) **since** i = n

Top Down on Natural Numbers

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

- “Top down” loop implementation

```
let i: number = n;
let s: number = 0;
{{ Inv: square(n) = square(i) + s }}
while (i != 0) {
  s = s + i + i - 1;
  i = i - 1;
}
{{ square(n) = square(i) + s and i = 0 }}
{{ s = square(n) }}
return s;
```

square(n) = square(i) + s
= square(0) + s **since** i = 0
= s **def of** square

Top Down on Natural Numbers

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

- “Top down” loop implementation

```
{{ Inv: square(n) = square(i) + s }}
while (i != 0) {
  {{ square(n) = square(i) + s and i ≠ 0 }}
  s = s + i + i - 1;
  i = i - 1;
  {{ square(n) = square(i) + s }}
}
```

Top Down on Natural Numbers

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

- “Top down” loop implementation

```
{{ Inv: square(n) = square(i) + s }}
while (i != 0) {
  {{ square(n) = square(i) + s and i ≠ 0 }}
  {{ square(n) = square(i – 1) + s + 2i – 1 }}
  s = s + i + i - 1;
  {{ square(n) = square(i – 1) + s }}
  i = i - 1;
  {{ square(n) = square(i) + s }}
}
```

$$\begin{aligned} \text{square}(n) &= \text{square}(i) + s \\ &= \text{square}(i-1) + 2(i-1) + 1 + s && \textbf{def of } \text{square} \\ &= \text{square}\,(i-1) + 2i - 1 + s && \textbf{(since } i \neq 0) \end{aligned}$$

“Top Down” Loops

- Invariant describes what is missing as a function call
- Not the best approach for ℕ
  - bottom-up is easier to understand and get correct
  - do that instead
- Often the best approach for Lists
  - can’t easily access the end of the list

    (unless we reverse it)
  - top-down approach possible, but is more complicated

    invariant will be like we just saw

    but there is one more tricky issue here…

Top Down on Lists

func twice(nil) := nil

twice(cons(x, L)) := cons(2x, twice(L)) for any x : $\mathbb{Z}$ and L : List

- Does this calculate $S = \text{twice}(L)$?

```
let R: List = L;
let S: List = nil;
while (R !== nil) {
  S = cons(2 * R.hd, S);
  R = R.tl;
}
return S;
```

Top Down on Lists

```
let R: List = L;
let S: List = nil;
while (R !== nil) {
  S = cons(2 * R.hd, S);
  R = R.tl;
}
return S;
```

- **Operates as follows on** cons(1, cons(2, cons(3, nil))):

Iter	S	R
0	nil	cons(1, cons(2, cons(3, nil)))
1	cons(2·1, nil)	cons(2, cons(3, nil))
…		

Top Down on Lists

- Operates as follows on $\text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil})))$:

Iter	S	R
0	nil	$\text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil})))$
1	$\text{cons}(2 \cdot 1, \text{nil})$	$\text{cons}(2, \text{cons}(3, \text{nil}))$

- Looks good so far:

$\text{twice}(\text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil}))))$

$= \text{cons}(2 \cdot 1, \text{twice}(\text{cons}(2, \text{cons}(3, \text{nil}))))$ def of twice

$= \text{concat}(\text{cons}(2 \cdot 1, \text{nil}), \text{twice}(\text{cons}(2, \text{cons}(3, \text{nil}))))$ def of concat

Answer is what we have so far (S) + twice of what is left (R)

We've only looked at length 0 and 1… Maybe we should do more?

Top Down on Lists

```
let R: List = L;
let S: List = nil;
while (R !== nil) {
  S = cons(2 * R.hd, S);
  R = R.tl;
}
return S;
```

- **Operates as follows on** $\mathrm{cons}(1, \mathrm{cons}(2, \mathrm{cons}(3, \mathrm{nil})))$**:**

Iter	S	R
0	nil	$\mathrm{cons}(1, \mathrm{cons}(2, \mathrm{cons}(3, \mathrm{nil})))$
1	$\mathrm{cons}(2 \cdot 1, \mathrm{nil})$	$\mathrm{cons}(2, \mathrm{cons}(3, \mathrm{nil}))$
2	$\mathrm{cons}(2 \cdot 2, \mathrm{cons}(2 \cdot 1, \mathrm{nil}))$	$\mathrm{cons}(3, \mathrm{nil})$
3	$\mathrm{cons}(2 \cdot 3, \mathrm{cons}(2 \cdot 1, \mathrm{cons}(2 \cdot 1, \mathrm{nil})))$	nil

Top Down on Lists

- **Operates as follows on** cons(1, cons(2, cons(3, nil))):

Iter	S	R
0	nil	cons(1, cons(2, cons(3, nil)))
1	cons(2·1, nil)	cons(2, cons(3, nil))
2	cons(2·2, cons(2·1, nil))	cons(3, nil)
3	cons(2·3, cons(2·1, cons(2·1, nil)))	nil

reversal of the answer

twice(cons(1, cons(2, cons(3, nil))))
= cons(2·1, twice(cons(2, cons(3, nil)))) **def of** twice
= cons(2·1, cons(2·2, twice(cons(3, nil)))) **def of** twice
≠ concat(cons(2·2, cons(2·1, nil)), twice(cons(3, nil)))

Top Down on Lists

- **Operates as follows on** cons(1, cons(2, cons(3, nil))):

Iter	S	R
0	nil	cons(1, cons(2, cons(3, nil)))
1	cons(2·1, nil)	cons(2, cons(3, nil))
2	cons(2·2, cons(2·1, nil))	cons(3, nil)
3	cons(2·3, cons(2·1, cons(2·1, nil)))	nil

reversal of the answer

twice(cons(1, cons(2, cons(3, nil))))
= cons(2·1, twice(cons(2, cons(3, nil)))) — **def of** twice
= cons(2·1, cons(2·2, twice(cons(3, nil)))) — **def of** twice
= concat(cons(2·1, cons(2·2, nil)), twice(cons(3, nil))) — **def of** concat
= concat(rev(S), twice(R)) — **since** S, R = ...

What is going on here?

- Doesn't matter what order we add up numbers:

$$1 + 2 + 3 \quad = \quad 3 + 2 + 1$$

- Does matter what order you build the list

$$\mathrm{cons}(1, \mathrm{cons}(2, \mathrm{cons}(3, \mathrm{nil}))) \quad \neq \quad \mathrm{cons}(3, \mathrm{cons}(2, \mathrm{cons}(1, \mathrm{nil})))$$

- Top-down produces the answer in reverse order
  - doesn't matter for integers; does matter for lists

- Possible to make tricky interview questions like this
  - wouldn't think a loop over a list would be tricky

Top Down on Lists

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

- "Top down" loop to calculate twice(L)

```
let R: List = L;
let S: List = nil;
{{ Inv: twice(L) = concat(rev(S), twice(R)) }}
while (R !== nil) {
  S = cons(2 * R.hd, S);
  R = R.tl;
}
{{ twice(L) = rev(S) }}
return rev(S);  // = twice(L)
```

Still need to check this.

Hopefully obvious that it could be wrong.
(Testing length 0, 1, 2, 3 is not enough!)

Top Down on Lists

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

- **“Top down” loop to calculate** twice(L)

```
let R: List = L;
let S: List = nil;
{{ R = L and S = nil }}
{{ Inv: twice(L) = concat(rev(S), twice(R)) }}
while (R !== nil) {
  S = cons(2 * R.hd, S);
  R = R.tl;
}
{{ twice(L) = rev(S) }}
```

concat(rev(S), twice(R))
= concat(rev(nil), twice(R))   **since** S = nil
= concat(nil, twice(R))   **def of** rev
= twice(R)   **def of** concat
= twice(L)   **since** R = L

Top Down on Lists

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

- "**Top down" loop to calculate** twice(L)

```
{{ Inv: twice(L) = concat(rev(S), twice(R)) }}
while (R !== nil) {
  S = cons(2 * R.hd, S);
  R = R.tl;
}
{{ twice(L) = concat(rev(S), twice(R)) and R = nil }}
{{ twice(L) = rev(S) }}
```

twice(L)	= concat(rev(S), twice(R))	
	= concat(rev(S), twice(nil))	**since** R = nil
	= concat(rev(S), nil)	**def of** twice
	= rev(S)	**by Lemma 2**

Top Down on Lists

func twice(nil) := nil

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

- "Top down" loop to calculate twice(L)

```
{{ Inv: twice(L) = concat(rev(S), twice(R)) }}
while (R !== nil) {
  {{ twice(L) = concat(rev(S), twice(R)) and R ≠ nil }}
  S = cons(2 * R.hd, S);
  R = R.tl;
  {{ twice(L) = concat(rev(S), twice(R)) }}
}
```

Top Down on Lists

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

- **“Top down” loop to calculate** twice(L)

```
{{ Inv: twice(L) = concat(rev(S), twice(R)) }}
while (R !== nil) {
  {{ twice(L) = concat(rev(S), twice(R)) and R ≠ nil }}
  {{ twice(L) = concat(rev(cons(2 · R.hd, S)), twice(R.tl)) }}
  S = cons(2 * R.hd, S);
  {{ twice(L) = concat(rev(S), twice(R.tl)) }}
  R = R.tl;
  {{ twice(L) = concat(rev(S), twice(R)) }}
}
```

Top Down on Lists

func $\text{twice}(\text{nil}) := \text{nil}$

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

- "Top down" loop to calculate $\text{twice}(L)$

$\{\{\ \text{twice}(L) = \text{concat}(\text{rev}(S), \text{twice}(R))$ and $R \neq \text{nil}\ \}\}$

$\{\{\ \text{twice}(L) = \text{concat}(\text{rev}(\text{cons}(2 \cdot R.\text{hd}, S)), \text{twice}(R.\text{tl}))\ \}\}$

Calculate a bit to see if this looks right (note: not a correct proof!)...

$\text{rev}(\text{cons}(2 \cdot R.\text{hd}, S)) = \text{concat}(\text{rev}(S), \text{cons}(2 \cdot R.\text{hd}, \text{nil}))$

so $\text{concat}(\text{rev}(\text{cons}(2 \cdot R.\text{hd}, S)), \text{twice}(R.\text{tl}))$

$= \text{concat}(\text{concat}(\text{rev}(S), \text{cons}(2 \cdot R.\text{hd}, \text{nil})), \text{twice}(R.\text{tl}))$

$= \text{concat}(\text{rev}(S), \text{concat}(\text{cons}(2 \cdot R.\text{hd}, \text{nil}), \text{twice}(R.\text{tl})))$

$= \text{concat}(\text{rev}(S), \text{cons}(2 \cdot R.\text{hd}, \text{twice}(R.\text{tl})))$

$= \text{concat}(\text{rev}(S), \text{twice}(R))$

Top Down on Lists

$$\begin{aligned}
&\textbf{func}\ \text{twice}(\text{nil}) && := \text{nil} \\
&\phantom{\textbf{func}}\ \text{twice}(\text{cons}(x, L)) && := \text{cons}(2x, \text{twice}(L)) \quad \text{for any } x : \mathbb{Z} \text{ and } L : \text{List}
\end{aligned}$$

- **“Top down” loop to calculate** twice(L)

$\{\{\ \text{twice}(L) = \text{concat}(\text{rev}(S), \text{twice}(R)) \text{ and } R \neq \text{nil}\ \}\}$

$\{\{\ \text{twice}(L) = \text{concat}(\text{rev}(\text{cons}(2 \cdot R.\text{hd}, S)), \text{twice}(R.\text{tl}))\ \}\}$

$$\begin{aligned}
\text{twice}(L) &= \text{concat}(\text{rev}(S), \text{twice}(R)) && \\
&= \text{concat}(\text{rev}(S), \text{twice}(\text{cons}(R.\text{hd}, R.\text{tl}))) && \textbf{since } R \neq \text{nil} \\
&= \text{concat}(\text{rev}(S), \text{cons}(2 \cdot R.\text{hd}, \text{twice}(R.\text{tl}))) && \textbf{def of } \text{twice} \\
&= \text{concat}(\text{rev}(S), \text{concat}(\text{nil}, \text{cons}(2 \cdot R.\text{hd}, \text{twice}(R.\text{tl})))) && \textbf{def of } \text{twice} \\
&= \text{concat}(\text{rev}(S), \text{concat}(\text{cons}(2 \cdot R.\text{hd}, \text{nil}), \text{twice}(R.\text{tl}))) && \textbf{def of } \text{concat} \\
&= \text{concat}(\text{concat}(\text{rev}(S), \text{cons}(2 \cdot R.\text{hd}, \text{nil})), \text{twice}(R.\text{tl})) && \textbf{assoc. of } \text{concat} \\
&= \text{concat}(\text{rev}(\text{cons}(2 \cdot R.\text{hd}, S)), \text{twice}(R.\text{tl})) && \textbf{def of } \text{rev}
\end{aligned}$$

“Top Down” Loops on Lists

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

- Can be implemented with a loop like this

```
function f(L: List): List {
  let R: List = L;
  let S: List = nil;    // = f(nil)
  {{ Inv: f(L) = concat(rev(S), f(R)) }}
  while (R !== nil) {
    S = cons(g(R.hd), S);
    R = R.tl;
  }
  return rev(S);        // = f(L)
}
```