CSE 331

Structural Induction

Katherine Murphy

Credits: Profs. Kevin Zatloukal and James Wilcox
Recall: Reasoning

• In an intro class, you might be asked:

  *what does this code do on this input?*

• In this class, we are often interested in:

  *what does this code do on all inputs?*

• This is a very different question!
Recall: Reasoning

• “Thinking through” what the code does on all inputs
  – neither testing nor type checking can do this

• Required in principle and in practice
  – you are expected to know what your code does
  – in practice, “reasoning is not optional: either reason up front or debug and then reason”

• Very different problem from analyzing specific inputs
  – will require new tools
Recall: Proving Correctness by Calculation

```javascript
const f = (a: bigint, b: bigint): bigint => {
  const L: List = cons(a, cons(b, nil));
  const s: bigint = sum(L);  // = a + b
  ...
};
```

- Can prove the claim in the comments by calculation

  
  \[
  \text{sum}(L) = \text{sum}(\text{cons}(a, \text{cons}(b, \text{nil}))) \quad \text{since } L = \ldots \\
  = a + \text{sum}(\text{cons}(b, \text{nil})) \quad \text{def of sum} \\
  = a + b + \text{sum}(\text{nil}) \quad \text{def of sum} \\
  = a + b \quad \text{def of sum}
  \]

- We checked correctness for infinitely many inputs!
Proof by Calculation

• Our proofs so far have used fixed-length lists
  – e.g., \( \text{len(twice(cons(a, cons(b, nil))))} = \text{len(cons(a, cons(b, nil)))} \)
  – problems in HW Quilt restrict to this case

• Would like to prove correctness on any length list \( L \)

• Need more tools for this...
  – structural recursion *calculates* on inductive types
  – structural induction *reasons* about structural recursion or more generally, to prove facts containing variables of an inductive type
  – both tools are specific to inductive types
Code Without Mutation

• Code without mutation can be written with just...
  – straight-line code
  – conditionals
  – recursion

• Already saw how to reason about...
  – straight-line code
  – conditionals

• Recursion is all that is left
Structural Induction

Let $P(S)$ be the claim “$\text{len(twice}(S)) = \text{len}(S)$”

To prove $P(S)$ holds for any list $S$, prove two implications

**Base Case:** prove $P(\text{nil})$
- use any known facts and definitions

**Inductive Step:** prove $P(\text{cons}(x, L))$ for any $x : \mathbb{Z}$, $L : \text{List}$
- $x$ and $L$ are variables
- use any known facts and definitions plus one more fact...
- make use of the fact that $L$ is also a List
To prove $P(S)$ holds for any list $S$, prove two implications

**Base Case:** prove $P(\text{nil})$
- use any known facts and definitions

**Inductive Hypothesis:** assume $P(L)$ is true
- use this in the inductive step, but not anywhere else

**Inductive Step:** prove $P(\text{cons}(x, L))$ for any $x : \mathbb{Z}$, $L : \text{List}$
- use known facts and definitions and **Inductive Hypothesis**
Why This Works

With Structural Induction, we prove two facts

\[
\begin{align*}
P(\text{nil}) & : \text{len}(\text{twice}(\text{nil})) = \text{len}(\text{nil}) \\
P(\text{cons}(x, L)) & : \text{len}(\text{twice}(\text{cons}(x, L)))) = \text{len}(\text{cons}(x, L)) \\
\text{(second assuming } & \text{len}(\text{twice}(L)) = \text{len}(L))
\end{align*}
\]

Why is this enough to prove \( P(S) \) for any \( S : \text{List} \)?
Why This Works

Build up an object using constructors:

nil  
cons(2, nil)  
cons(1, cons(2, nil))

first constructor
second constructor
second constructor

1 ➔ 2 ➔ nil

nil already exists when building cons(2, nil)

cons(2, nil) already exists when building cons(1, cons(2, nil))
Why This Works

Build up a proof the same way we built up the object

\[
\begin{align*}
P(\text{nil}) & \quad \text{len(twice(\text{nil}))} = \text{len(\text{nil})} \\
P(\text{cons}(x, L)) & \quad \text{len(twice(cons(x, L)))} = \text{len(cons(x, L))} \\
& \quad \text{(second assuming len(twice(L))} = \text{len(L))}
\end{align*}
\]
“We go together”

structural induction

inductive types
Structural Induction in General

• General case: assume \( P \) holds for constructor arguments

\[
\text{type } T := A \mid B(x : \mathbb{Z}) \mid C(y : \mathbb{Z}, t : T) \mid D(z : \mathbb{Z}, u : T, v : T)
\]

• To prove \( P(t) \) for any \( t \), we need to prove:
  - \( P(A) \)
  - \( P(B(x)) \) for any \( x : \mathbb{Z} \)
  - \( P(C(y, t)) \) for any \( y : \mathbb{Z} \) and \( t : T \) assuming \( P(t) \) is true
  - \( P(D(z, u, v)) \) for any \( z : \mathbb{Z} \) and \( u, v : T \) assuming \( P(u) \) and \( P(v) \)

• These four facts are enough to prove \( P(t) \) for any \( t \)
  - for each constructor, have proof that it produces an object satisfying \( P \)
Structural Induction in General

• General case: assume $P$ holds for constructor arguments

\[
\text{type}\ T := A \mid \text{B}(x : \mathbb{Z}) \mid \text{C}(y : \mathbb{Z}, t : T) \mid \text{D}(z : \mathbb{Z}, u : T, v : T)
\]

• To prove $P(t)$ for any $t$, we need to prove:
  
  – $P(A)$
  
  – $P(B(x))$ for any $x : \mathbb{Z}$
  
  – $P(C(y, t))$ for any $y : \mathbb{Z}$ and $t : T$ assuming $P(t)$ is true
  
  – $P(D(z, u, v))$ for any $z : \mathbb{Z}$ and $u, v : T$ assuming $P(u)$ and $P(v)$

• Each inductive type has its own form of induction
  
  – special way to reason about that type
Example: Repeating List Elements

• Consider the following function:

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

• Produces a list where every element is repeated twice

```haskell
echo(cons(1, cons(2, nil)))
    = cons(1, cons(1, echo(cons(2, nil))))
    = cons(1, cons(1, cons(2, cons(2, echo(nil)))))
    = cons(1, cons(1, cons(2, cons(2, nil))))
    def of echo
```

Example: Repeating List Elements

```go
func echo(nil) := nil
echo(cons(x, L)) := cons(x, cons(x, echo(L))) for any x : ℤ, L : List
```

• Suppose we have the following code:

```go
const m: bigint = len(S); // S is some List
const R: List = echo(S);
...
return 2*m; // = len(echo(S)) not straight from the spec
```

– spec says to return \(\text{len}(\text{echo}(S))\) but code returns \(2 \times \text{len}(S)\)

• Need to prove that \(\text{len}(\text{echo}(S)) = 2 \times \text{len}(S)\)
Example: Repeating List Elements

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

- Prove that \( \text{len}(\text{echo}(S)) = 2 \text{len}(S) \) for any \( S \in \text{List} \)

**Base Case** (nil):

Need to prove that \( \text{len}(\text{echo}(\text{nil})) = 2 \text{len}(\text{nil}) \)

\[
\text{len}(\text{echo}(\text{nil})) =
\]

\[
\text{func } \text{len}(\text{nil}) \quad := 0
\]
\[
\text{len}(\text{cons}(x, L)) \quad := 1 + \text{len}(L) \quad \text{for any } x \in \mathbb{Z} \text{ and any } L \in \text{List}
\]
Example: Repeating List Elements

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

- **Prove that** \(\text{len}(\text{echo}(S)) = 2 \text{ len}(S)\) **for any** \(S : \text{List}\)

**Base Case** (nil):

\[
\begin{align*}
\text{len}(\text{echo}(\text{nil})) & = \text{len}(\text{nil}) \quad \text{def of } \text{echo} \\
& = 0 \quad \text{def of } \text{len} \\
& = 2 \cdot 0 \quad \text{def of } \text{len} \\
& = 2 \text{ len}(\text{nil}) \quad \text{def of } \text{len}
\end{align*}
\]
Example: Repeating List Elements

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

- **Prove that** \(\text{len}(\text{echo}(S)) = 2 \text{len}(S)\) **for any** \(S : \text{List}\)

  **Inductive Step** \((\text{cons}(x, L))\):
  
  **Need to prove that** \(\text{len}(\text{echo}(\text{cons}(x, L))) = 2 \text{len}(\text{cons}(x, L))\)

  **Get to assume claim holds for** \(L, \text{i.e., that} \) \(\text{len}(\text{echo}(L)) = 2 \text{len}(L)\)
Example: Repeating List Elements

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

- **Prove that** \(\text{len}(\text{echo}(S)) = 2 \text{len}(S)\) **for any** \(S : \text{List}\)

  **Inductive Hypothesis:** assume that \(\text{len}(\text{echo}(L)) = 2 \text{len}(L)\)

  **Inductive Step** (\(\text{cons}(x, L)\)):

  \[
  \text{len}(\text{echo}(\text{cons}(x, L)))
  \]

  \[
  = 2 \text{len}(\text{cons}(x, L))
  \]

\[
\text{func } \text{len}(\text{cons}(x, L)) \quad := \quad 1 + \text{len}(L) \quad \text{for any } x \in \mathbb{Z} \text{ and any } L \in \text{List}
\]
Example: Repeating List Elements

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

- **Prove that** \(\text{len}(\text{echo}(S)) = 2 \text{len}(S) \) **for any** \(S : \text{List}\)

**Inductive Hypothesis:** assume that \(\text{len}(\text{echo}(\text{L})) = 2 \text{len}(\text{L})\)

**Inductive Step** (\(\text{cons}(x, \text{L})\)):
\[
\text{len}(\text{echo}(\text{cons}(x, \text{L}))) = \text{len}(\text{cons}(x, \text{cons}(x, \text{echo}(\text{L}))))
= 1 + \text{len}(\text{cons}(x, \text{echo}(\text{L})))
= 2 + \text{len}(\text{echo}(\text{L}))
= 2 + 2 \text{len}(\text{L})
= 2(1 + \text{len}(\text{L}))
= 2 \text{len}(\text{cons}(x, \text{L}))
\]

**def of** \(\text{echo}\)

**def of** \(\text{len}\)

**def of** \(\text{len}\)

**Ind. Hyp.**
Example 2: Repeating List Elements

```go
def echo(nil) := nil
echo(cons(x, L)) := cons(x, cons(x, echo(L))) for any x : ℤ, L : List
```

- Suppose we have the following code:

```go
const y: bigint = sum(S); // S is some List
cost R: List = echo(S);
...
return 2*y; // = sum(echo(S)) not straight from the spec
```

- spec says to return sum(echo(S)) but code returns 2 * sum(S)

- Need to prove that sum(echo(S)) = 2 * sum(S)
Example 2: Repeating List Elements

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

• Prove that \(\text{sum}(\text{echo}(S)) = 2 \text{ sum}(S)\) for any \(S : \text{List}\)

Base Case (\text{nil}):

\[
\text{sum}(\text{echo}(\text{nil})) \quad = \\
= 2 \text{ sum(\text{nil})}
\]

\[
\textbf{func} \ \text{sum}(\text{nil}) \quad := \quad 0 \\
\text{sum}(\text{cons}(x, L)) \quad := \quad x + \text{sum}(L) \quad \text{for any } x \in \mathbb{Z} \text{ and any } L \in \text{List}
\]
Example 2: Repeating List Elements

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

• Prove that \( \text{sum(echo(S))} = 2 \text{ sum(S)} \) for any \( S : \text{List} \)

Base Case (nil):

\[
\text{sum(echo(nil)) } = \text{ sum(nil) def of echo} \\
= \text{ 0 def of sum} \\
= 2 \cdot \text{ 0} \\
= 2 \text{ sum(nil) def of sum}
\]

Inductive Step (cons(x, L)):

Need to prove that \( \text{sum(echo(cons(x, L)))} = 2 \text{ sum(cons(x, L))} \)

Get to assume claim holds for \( L \), i.e., that \( \text{sum(echo(L))} = 2 \text{ sum(L)} \)
Example 2: Repeating List Elements

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

• Prove that \( \text{sum(echo(S))} = 2 \text{ sum(S)} \) for any \( S : \text{List} \)

**Inductive Hypothesis:** assume that \( \text{sum(echo(L))} = 2 \text{ sum(L)} \)

**Inductive Step** (\( \text{cons(x, L)} \)):

\[
\text{sum(echo(cons(x, L)))} =
\]

\[
= 2 \text{ sum(cons(x, L))}
\]

\[
\text{func sum(nil)} := 0 \\
\text{sum(cons(x, L))} := x + \text{sum(L)} \quad \text{for any } x \in \mathbb{Z} \text{ and any } L \in \text{List}
\]
Example 2: Repeating List Elements

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

• Prove that \(\text{sum}(\text{echo}(S)) = 2 \times \text{sum}(S)\) for any \(S : \text{List}\)

**Inductive Hypothesis**: assume that \(\text{sum}(\text{echo}(L)) = 2 \times \text{sum}(L)\)

**Inductive Step** (\(\text{cons}(x, L)\)):

\[
\begin{align*}
\text{sum}(\text{echo}(\text{cons}(x, L)))) & = \text{sum}(\text{cons}(x, \text{cons}(x, \text{echo}(L)))) \\
& = x + \text{sum}(\text{cons}(x, \text{echo}(L))) \\
& = 2x + \text{sum}(\text{echo}(L)) \\
& = 2x + 2 \times \text{sum}(L) \\
& = 2(x + \text{sum}(L)) \\
& = 2 \times \text{sum}(\text{cons}(x, L))
\end{align*}
\]

\text{def of echo} \quad \text{def of sum} \quad \text{def of sum} \quad \text{Ind. Hyp.} \quad \text{def of sum}
Recall: Concatenating Two Lists

- **Mathematical definition of concat(S, R)**

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

- \text{concat}(S, R) defined by pattern matching on S (not R)
Example 3: Length of Concatenated Lists

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

• Suppose we have the following code:

```plaintext
const m: bigint = len(S); // S is some List
cost n: bigint = len(R); // R is some List
...
return m + n; // = len(concat(S, R))
```

– spec returns \( \text{len}(\text{concat}(S, R)) \) but code returns \( \text{len}(S) + \text{len}(R) \)

• Need to prove that \( \text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R) \)
Example 3: Length of Concatenated Lists

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

- **Prove that** $\text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R)$
  - prove by induction on $S$
  - prove the claim for any choice of $R$ (i.e., $R$ is a variable)

**Base Case** (nil):

\[
\text{len}(\text{concat}(\text{nil}, R)) = \text{len}(\text{nil}) + \text{len}(R)
\]
Example 3: Length of Concatenated Lists

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

- **Prove that** \( \text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R) \)
  - prove by induction on \( S \)
  - prove the claim for any choice of \( R \) (i.e., \( R \) is a variable)

**Base Case** (nil):

\[
\text{len}(\text{concat}(\text{nil}, \ R)) = \text{len}(R) \quad \text{def of concat} \\
= 0 + \text{len}(R) \\
= \text{len}(\text{nil}) + \text{len}(R) \quad \text{def of len}
\]
Example 3: Length of Concatenated Lists

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

• Prove that \( \text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R) \)

Inductive Step \((\text{cons}(x, L)):

Need to prove that

\[
\text{len}(\text{concat}(\text{cons}(x, L), R)) = \text{len}(\text{cons}(x, L)) + \text{len}(R)
\]

Get to assume claim holds for \(L\), i.e., that

\[
\text{len}(\text{concat}(L, R)) = \text{len}(L) + \text{len}(R)
\]
Example 3: Length of Concatenated Lists

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

- Prove that \( \text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R) \)

**Inductive Hypothesis:** assume that \( \text{len}(\text{concat}(L, R)) = \text{len}(L) + \text{len}(R) \)

**Inductive Step** (\( \text{cons}(x, L) \)):

\[
\text{len}(\text{concat}(\text{cons}(x, L), R)) =
\]

\[
= \text{len}(\text{cons}(x, L)) + \text{len}(R)
\]

\text{func } \text{len}(\text{nil}) := 0
\]
\[
\text{func } \text{len}(\text{cons}(x, L)) := 1 + \text{len}(L) \quad \text{for any } x \in \mathbb{Z} \text{ and any } L \in \text{List}
\]
Example 3: Length of Concatenated Lists

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

- **Prove that** \(\text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R)\)

  **Inductive Hypothesis**: assume that \(\text{len}(\text{concat}(L, R)) = \text{len}(L) + \text{len}(R)\)

  **Inductive Step** \((\text{cons}(x, L)):\)

  \[
  \begin{align*}
  \text{len}(\text{concat}(\text{cons}(x, L), R)) &= \text{len}(\text{cons}(x, \text{concat}(L, R))) \\
  &= 1 + \text{len}(\text{concat}(L, R)) \\
  &= 1 + \text{len}(L) + \text{len}(R) \\
  &= \text{len}(\text{cons}(x, L)) + \text{len}(R)
  \end{align*}
  \]

  def of \(\text{concat}\)

  def of \(\text{len}\)

  Ind. Hyp.

  def of \(\text{len}\)
Comparing Reasoning vs Testing

```javascript
const concat = (S: List, R: List): List => {
    if (S.kind === "nil") {
        return R;
    } else {
        return cons(S.hd, concat(S.tl, R));
    }
};
```

- **Testing:** 6 cases
  - 3 subdomains require 2 test cases each

- **Reasoning:** 2 calculations
Recall: Reversing a List

- **Mathematical definition of** \( \text{rev}(S) \)

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

- note that \( \text{rev} \) uses \( \text{concat} \) as a helper function

---

move 1 to end

reverse this too
Example 4: Length of Reversed List

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

- Suppose we have the following code:

\[
\begin{align*}
\text{const } m : \text{bigint} & = \text{len}(S); \quad \text{// } S \text{ is some List} \\
\text{const } R : \text{List} & = \text{rev}(S);
\end{align*}
\]

\[
\text{...}
\]

\[
\text{return } m; \quad \text{// } = \text{len}(\text{rev}(S))
\]

- spec returns \(\text{len}(\text{rev}(S))\) but code returns \(\text{len}(S)\)

- Need to prove that \(\text{len}(\text{rev}(S)) = \text{len}(S)\) for any \(S : \text{List}\)
Example 4: Length of Reversed List

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

• **Prove that** \(\text{len}(\text{rev}(S)) = \text{len}(S)\) **for any** \(S : \text{List}\)

**Base Case** (nil):

\[
\text{len}(\text{rev}(\text{nil})) = \text{len}(\text{nil}) \quad \text{def of rev}
\]

**Inductive Step** (cons(x, L)):

Need to prove that \(\text{len}(\text{rev}(\text{cons}(x, L))) = \text{len}(\text{cons}(x, L))\)

Get to assume that \(\text{len}(\text{rev}(L)) = \text{len}(L)\)
Example 4: Length of Reversed List

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

• **Prove that** \( \text{len}(\text{rev}(S)) = \text{len}(S) \) **for any** \( S : \text{List} \)

**Inductive Hypothesis:** assume that \( \text{len}(\text{rev}(L)) = \text{len}(L) \)

**Inductive Step** \((\text{cons}(x, L)):\)

\[
\text{len}(\text{rev}(\text{cons}(x, L))) = \text{len}(\text{cons}(x, L)) = \text{len}(\text{cons}(x, L))
\]
Example 4: Length of Reversed List

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

• Prove that \(\text{len}(\text{rev}(S)) = \text{len}(S)\) for any \(S : \text{List}\)

\textbf{Inductive Hypothesis:} assume that \(\text{len}(\text{rev}(L)) = \text{len}(L)\)

\textbf{Inductive Step} (\text{cons}(x, L)):

\[
\begin{align*}
    \text{len}(\text{rev}(\text{cons}(x, L))) & \\
    = \text{len}(\text{concat}(\text{rev}(L), \text{cons}(x, \text{nil}))) & \text{def of rev} \\
    = \text{len}(\text{rev}(L)) + \text{len}(\text{cons}(x, \text{nil})) & \text{by Example 3} \\
    = \text{len}(L) + \text{len}(\text{cons}(x, \text{nil})) & \text{Ind. Hyp.} \\
    = \text{len}(L) + 1 + \text{len}(\text{nil}) & \text{def of len} \\
    = \text{len}(L) + 1 & \text{def of len} \\
    = \text{len}(\text{cons}(x, L)) & \text{def of len}
\end{align*}
\]
Finer Points of Structural Induction

• Structural Induction is how we reason about recursion

• Reasoning also follows structure of code
  – code uses structural recursion, so reasoning uses structural induction

• Note that rev is defined in terms of concat
  – reasoning about \texttt{len(rev(...))} used fact about \texttt{len(concat(...))}
  – this is common
Example 5: Reversing a List

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

- This correctly reverses a list but is slow
  - concat takes $\Theta(n)$ time, where $n$ is length of $L$
  - $n$ calls to concat takes $\Theta(n^2)$ time

- Can we do this faster?
  - yes, but we need a helper function
Example 5: Reversing a List

• **Helper function** \( \text{rev-acc}(S, R) \) **for any** \( S, R : \text{List} \)

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

for any \( R : \text{List} \) and any \( x : \mathbb{Z} \) and any \( L, R : \text{List} \)

\[
\text{rev-acc} \left( \begin{array}{c}
\text{3} \rightarrow \text{4} \rightarrow \text{nil} \\
\text{2} \rightarrow \text{1} \rightarrow \text{nil}
\end{array} \right) = \text{rev-acc} \left( \begin{array}{c}
\text{4} \rightarrow \text{nil} \\
\text{3} \rightarrow \text{2} \rightarrow \text{1} \rightarrow \text{nil}
\end{array} \right) = \text{rev-acc} \left( \begin{array}{c}
\text{nil} \\
\text{4} \rightarrow \text{3} \rightarrow \text{2} \rightarrow \text{1} \rightarrow \text{nil}
\end{array} \right)
\]
Recall: Reversing a List

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

- **Can prove that** \( \text{rev-acc}(S, \text{R}) = \text{concat}(\text{rev}(S), \text{R}) \)  \text{(Lemma 1)}

- **Can prove that** \( \text{concat}(\text{L}, \text{nil}) = \text{L} \)  \text{(Lemma 2)}
  - structural induction like prior examples

- **Prove that** \( \text{rev}(S) = \text{rev-acc}(S, \text{nil}) \)

\[
\text{rev-acc}(S, \text{nil}) = \text{concat}(\text{rev}(S), \text{nil}) \quad \text{Lemma 1}
\]
\[
= \text{rev}(S) \quad \text{Lemma 2}
\]

doing our analysis in math so that it works for any programming language
**Example 5: Helper Lemma 2**

\[
\text{func } \text{concat}(\text{nil}, R) := R \\
\text{concat}(\text{cons}(x, L), R) := \text{cons}(x, \text{concat}(L, R))
\]

for any \( R : \text{List} \) and any \( x : \mathbb{Z} \) and any \( L, R : \text{List} \)

- **Prove that** \( \text{concat}(S, \text{nil}) = S \)

  **Base Case** (nil):

  \[
  \text{concat}(\text{nil}, \text{nil}) = \text{nil} \quad \text{def of concat}
  \]

  **Inductive Hypothesis**: assume that \( \text{concat}(L, \text{nil}) = \text{nil} \)

  **Inductive Step** (\( \text{cons}(x, L) \)): prove that \( \text{concat}(\text{cons}(x, L), \text{nil}) = \text{cons}(x, L) \)
Example 5: Helper Lemma 2

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

- Prove that \( \text{concat}(S, \text{nil}) = S \)

**Inductive Hypothesis:** assume that \( \text{concat}(L, \text{nil}) = L \)

**Inductive Step** (cons(x, L)):

\[
\text{concat}(\text{cons}(x, L), \text{nil}) = \\
= \text{cons}(x, L)  \quad \text{Ind. Hyp.}
\]
Example 5: Helper Lemma 2

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

- **Prove that** \(\text{concat}(S, \text{nil}) = S\)

  \textbf{Inductive Hypothesis:} assume that \(\text{concat}(L, \text{nil}) = L\)

  \textbf{Inductive Step} (\text{cons}(x, L)):

  \[
  \begin{align*}
  \text{concat(cons}(x, L), \text{nil}) & = \text{cons}(x, \text{concat}(L, \text{nil})) & \text{def of } \text{concat} \\
  & = \text{cons}(x, L) & \text{Ind. Hyp.}
  \end{align*}
  \]
Example 5: Helper Lemma 1

```
func rev-acc(nil, R) := R
      for any R : List
rev-acc(cons(x, L), R) := rev-acc(L, cons(x, R))
      for any x : ℤ and
      any L, R : List
```

• **Prove that** \( \text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R) \)
  – prove by structural induction

• **Need the following property of** \( \text{concat} \)
  
  \[ \text{concat}(A, \text{concat}(B, C)) = \text{concat}(\text{concat}(A, B), C) \]
  for any A, B, C : List
  – with strings, we know that “A + (B + C) = (A + B) + C”
  – this says the same thing for lists
Example 5: Helper Lemma 1

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

• **Prove that** \( \text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R) \)
  
  – prove by induction on \( S \) (so \( R \) is a variable)

  **Base Case** (nil):
  
  \[
  \text{rev-acc}(\text{nil}, R) = \text{concat}(\text{rev}(\text{nil}), R)
  \]

  \[
  = \text{concat}(\text{rev}(\text{nil}), R)
  \]
Example 5: Helper Lemma 1

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

- **Prove that** \( \text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R) \)
  - **prove by induction on** \( S \) (so \( R \) is a variable)

**Base Case** (\text{nil}):

\[
\begin{align*}
\text{rev-acc}(\text{nil}, R) &= R & \text{def of \text{rev-acc}} \\
&= \text{concat}(\text{nil}, R) & \text{def of \text{concat}} \\
&= \text{concat}(\text{rev}(\text{nil}), R) & \text{def of \text{rev}}
\end{align*}
\]

\[
\begin{align*}
\text{func} & \quad \text{concat}(\text{nil}, R) := R \\
\text{concat}(\text{cons}(x, L), R) := \text{cons}(x, \text{concat}(L, R))
\end{align*}
\]

\[
\begin{align*}
\text{func} & \quad \text{rev}(\text{nil}) := \text{nil} \\
\text{rev}(\text{cons}(x, L)) := \text{concat}(\text{rev}(L), \text{cons}(x, \text{nil}))
\end{align*}
\]
Example 5: Helper Lemma 1

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

- **Prove that** \(\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)\)

  \textbf{Inductive Hypothesis:} assume that \(\text{rev-acc}(L, R) = \text{concat}(\text{rev}(L), R)\) for any \(R\)

  \textbf{Inductive Step} (\text{cons}(x, L)):

  \[
  \text{rev-acc}(\text{cons}(x, L), R) = \text{concat}(\text{rev}(\text{cons}(x, L)), R)
  \]

\[
\begin{align*}
\text{func } \text{concat} & (\text{nil, } R) := R \\
\text{concat} & (\text{cons}(x, L), R) := \text{cons}(x, \text{concat}(L, R)) \\
\text{func } \text{rev} & (\text{nil}) := \text{nil} \\
\text{rev} & (\text{cons}(x, L)) := \text{concat}(\text{rev}(L), \text{cons}(x, \text{nil}))
\end{align*}
\]
**Example 5: Helper Lemma 1**

```
func rev-acc(nil, R) := R
    rev-acc(cons(x, L), R) := rev-acc(L, cons(x, R))
for any R : List
    for any x : ℤ and any L, R : List
```

- **Prove that** $\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)$

  **Inductive Hypothesis:** assume that $\text{rev-acc}(L, R) = \text{concat}(\text{rev}(L), R)$ for any $R$

  **Inductive Step** ($\text{cons}(x, L)$):

  $\text{rev-acc}(\text{cons}(x, L), R) = \text{rev-acc}(L, \text{cons}(x, R))$
  $\quad = \text{concat}(\text{rev}(L), \text{cons}(x, R))$
  $\quad \text{def of rev-acc}$
  $\quad \text{Ind. Hyp.}$

  $\quad = \text{concat}(\text{concat}(\text{rev}(L), \text{cons}(x, nil)), R)$
  $\quad ??$
  $\quad = \text{concat}(\text{rev}(\text{cons}(x, L)), R)$
  $\quad \text{def of rev}$

```
func concat(nil, R) := R
    concat(cons(x, L), R) := cons(x, concat(L, R))
```

```
func rev(nil) := nil
    rev(cons(x, L)) := concat(rev(L), cons(x, nil))
```
Example 5: Helper Lemma 1

\[
\text{func } \text{rev-acc}(\text{nil}, R) := R \\
\text{rev-acc}(\text{cons}(x, L), R) := \text{rev-acc}(L, \text{cons}(x, R))
\]

for any \( R : \text{List} \)

for any \( x : \mathbb{Z} \) and any \( L, R : \text{List} \)

- Prove that \( \text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R) \)

**Inductive Hypothesis**: assume that \( \text{rev-acc}(L, R) = \text{concat}(\text{rev}(L), R) \) for any \( R \)

**Inductive Step** (cons\((x, L)\)):

\[
\text{rev-acc}(\text{cons}(x, L), R) = \text{rev-acc}(L, \text{cons}(x, R)) \\
= \text{concat}(\text{rev}(L), \text{cons}(x, R)) \quad \text{def of \text{rev-acc}} \\
= \text{concat}(\text{rev}(L), \text{concat}(\text{cons}(x, \text{nil}), R)) \quad \text{Ind. Hyp.} \\
= \text{concat}(\text{concat}(\text{rev}(L), \text{cons}(x, \text{nil})), R) \quad \text{Prop of \text{concat}} \\
= \text{concat}(\text{rev}(\text{cons}(x, L)), R) \quad \text{def of \text{rev}}
\]

\[
\text{func } \text{concat}(\text{nil}, R) := R \\
\text{concat}(\text{cons}(x, L), R) := \text{cons}(x, \text{concat}(L, R))
\]

\[
\text{func } \text{rev}(\text{nil}) := \text{nil} \\
\text{rev}(\text{cons}(x, L)) := \text{concat}(\text{rev}(L), \text{cons}(x, \text{nil}))
\]
Example 5: Helper Lemma 1

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

- **Prove that** \( \text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R) \)**

  **Inductive Hypothesis:** assume that \( \text{rev-acc}(L, R) = \text{concat}(\text{rev}(L), R) \) **for any** \( R \)

  **Inductive Step** (\( \text{cons}(x, L) \)):

  \[
  \text{rev-acc}(\text{cons}(x, L), R) = \text{rev-acc}(L, \text{cons}(x, R)) \quad \text{def of } \text{rev-acc}
  \]
  \[
  = \text{concat}(\text{rev}(L), \text{cons}(x, R)) \quad \text{Ind. Hyp.}
  \]
  \[
  = \text{concat}(\text{rev}(L), \text{cons}(x, \text{concat}(\text{nil}, R))) \quad \text{def of } \text{concat}
  \]
  \[
  = \text{concat}(\text{rev}(L), \text{concat}(\text{cons}(x, \text{nil}), R)) \quad \text{def of } \text{concat}
  \]
  \[
  = \text{concat}(\text{concat}(\text{rev}(L), \text{cons}(x, \text{nil})), R) \quad \text{Prop of } \text{concat}
  \]
  \[
  = \text{concat}(\text{rev}(\text{cons}(x, L)), R) \quad \text{def of } \text{rev}
  \]

\[
\text{func } \text{concat}(\text{nil}, R) := R
\]
\[
\text{concat}(\text{cons}(x, L), R) := \text{cons}(x, \text{concat}(L, R))
\]

\[
\text{func } \text{rev}(\text{nil}) := \text{nil}
\]
\[
\text{rev}(\text{cons}(x, L)) := \text{concat}(\text{rev}(L), \text{cons}(x, \text{nil}))
\]
Proof By Cases
Defining Functions by Cases

• Usually combine pattern matching with recursion

• Can use pattern matching on its own

```plaintext
def empty(nil) := T
empty(cons(x, L)) := F for any x : \mathbb{Z}, L : \text{List}
```

– every list is either nil or cons(x, L) for some x and L
– rule can be applied to any list

• Pattern matching is one way to define by cases
  – we’ve seen another way to do this...
Defining Functions by Cases

- Pattern matching is one way to define by cases

- Side conditions also define by cases
  - e.g., define $f(m)$ where $m : \mathbb{Z}$
    
    \[
    \begin{align*}
    \text{func } f(m) & := 2m + 1 \quad \text{if } m \geq 0 \\
    f(m) & := 0 \quad \text{if } m < 0
    \end{align*}
    \]

  - to use the definition on $f(x)$, need to know if $x < 0$ or not

- Need ways to reason about these functions as well
Proof By Cases

• New code structure means new proof structure

• Can split a proof into cases
  – e.g., $x \geq 0$ and $x < 0$
  – need to be sure the cases are exhaustive
    (don’t need to be exclusive in this case)

• If we can prove both cases, it is true in general
Proof By Cases

\[
\text{func } f(m) := \begin{cases} 
2m + 1 & \text{if } m \geq 0 \\
0 & \text{if } m < 0 
\end{cases}
\]

• Prove that \( f(m) > m \) for any \( m : \mathbb{Z} \)

Case \( m \geq 0 \):

\[
f(m) = \\
> m
\]
Proof By Cases

\[
\text{func } f(m) := \begin{cases} 
2m + 1 & \text{if } m \geq 0 \\
0 & \text{if } m < 0
\end{cases}
\]

- **Prove that** \( f(m) > m \) **for any** \( m : \mathbb{Z} \)

**Case** \( m \geq 0 \):

\[
f(m) = 2m + 1 \quad \text{def of } f \ (\text{since } m \geq 0) \\
\geq m + 1 \quad \text{since } m \geq 0 \\
> m \quad \text{since } 1 > 0
\]
Proof By Cases

\[
\text{func } f(m) := \begin{cases} 
2m + 1 & \text{if } m \geq 0 \\
0 & \text{if } m < 0 
\end{cases}
\]

- Prove that \( f(m) > m \) for any \( m : \mathbb{Z} \)

Case \( m \geq 0 \):

\[
f(m) = \ldots > m
\]

Case \( m < 0 \):

\[
f(m) = \quad \text{def of } f \quad (\text{since } m < 0)
\]

\[
> m \quad \text{since } m < 0
\]

Since these two cases are exhaustive, \( f(m) > m \) holds in general.
Recall: Pattern Matching

- Define a function by an exhaustive set of patterns

```plaintext
type Steps := {n : ℕ, fwd : BOOL}

func change({n: n, fwd: T}) := n for any n : ℕ
change({n: n, fwd: F}) := -n for any n : ℕ
```

- Steps describes movement on the number line
- `change(s : Steps)` says how the position changes

- one of these two rules always applies

![Diagram showing movement on the number line with change({n: 12, fwd: F})]
More Proof By Cases

```plaintext
func change({n: n, fwd: T}) := n for any n : \mathbb{N}
change({n: n, fwd: F}) := -n for any n : \mathbb{N}
```

- **Prove that** $|\text{change}(s)| = n$ for any $s = \{n: n, \text{fwd}: f\}$
  - we need to know if $f = T$ or $f = F$ to apply the definition!

  **Case** $f = T$:

  $$
  \begin{align*}
  |\text{change}({n: n, \text{fwd}: f})| \\
  &= |\text{change}({n: n, \text{fwd}: T})| & \text{since } f = T \\
  &= |n| & \text{def of } \text{change} \\
  &= n & \text{since } n \geq 0
  \end{align*}
  $$
More Proof By Cases

\[
\text{func change}\{\{n: n, \text{fwd: } T\}\} := n \quad \text{for any } n : \mathbb{N}
\]
\[
\text{change}\{\{n: n, \text{fwd: } F\}\} := -n \quad \text{for any } n : \mathbb{N}
\]

- **Prove that** \(|\text{change}(s)| = n \text{ for any } s = \{\{n: n, \text{fwd: } f\}\}\)

  **Case** \(f = T\): \(|\text{change}(\{n: n, \text{fwd: } f\})| = \ldots = n\)

  **Case** \(f = F\):

  \[
  |\text{change}(\{n: n, \text{fwd: } f\})| = |\text{change}(\{n: n, \text{fwd: } F\})| \quad \text{since } f = F
  \]

  \[
  = |-n| \quad \text{def of change}
  \]

  \[
  = n \quad \text{since } n \geq 0
  \]

Since these two cases are exhaustive, the claim holds in general.
Exceptions
More List Functions

Functions to return the first or last element of a list

\[ \text{func } \text{first}(\text{nil}) \ := \ ? \]
\[ \text{func } \text{first}(\text{cons}(x, L)) \ := \ x \quad \text{for any } L : \text{List} \]

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

• Only makes sense for non-empty lists
  – there is no first or last element of an empty list

• What do we do when the input is \text{nil}?
Partial Functions in Math

Some functions do not have answers for some inputs

```
func first(nil) := undefined
    first(cons(x, L)) := x
        for any L : List
```

```
func last(nil) := undefined
    last(cons(x, nil)) := x
        for any x : ℤ
    last(cons(x, cons(y, L))) := last(cons(y, L))
        for any x, y : ℤ and any L : List
```

- In math, we want functions to always be defined, so I had it return “undefined” in this case
  - return type is ℤ ∪ {undefined}
**Partial Functions in Code**

- When programming, we also have invalid inputs, but we can handle them differently: disallow them

```javascript
// L must be a non-empty list
const last = (L: List): bigint => {
    if (L.kind === "nil") {
        throw new Error("empty list! Boooo");
    } else if (L.tl.kind === "nil") {
        return L.hd;
    } else {
        return last(L.tl);
    }
};
```
Partial Functions in Code

- When programming, we also have invalid inputs, but we can handle them differently: disallow them

```javascript
// L must be a non-empty list
const last = (L: List): bigint => {
  if (L.kind === "nil") {
    throw new Error("empty list! Boooo");
  }
  ...
};

- Specification says L will not be nil
  - we assume it is not nil when reasoning
  - do not assume it is not nil at run time
    an example of defensive programming
Partial Functions in Code

• When programming, we also have invalid inputs, but we can handle them differently: disallow them

```javascript
// L must be a non-empty list
const last = (L: List): bigint => {
    if (L.kind === "nil") {
        throw new Error("empty list! Boooo");
    }
    ...
};
```

• In this case, we don’t want to return undefined
  – better to “fail fast”...
  – debugging is easier if the crash is closer to the bug
Defensive Programming Rules

• Fine to disallow any inputs you don’t want to handle
  – spec can say which inputs are allowed
    (the type system cannot always express this)

• Should also check that the inputs are valid
  – throw an exception if not
  – skip this only if the check is too expensive:
    if checking would make the function asymptotically slower, then skip it
  – after you spend 4 hours debugging a problem like this, you’ll wish you had written the check
Generics
Lots of Lists of Things

We have now seen lists of

- integers
- blocks (Row in HW Quilt)
- rows (Quilt in HW Quilt)
- HTML elements (jsxList in HW Quilt)

These are all “the same” in some sense

- have nil and cons
- cons puts a new value at the front
Generic Types

We can describe this pattern with a “generic” list type

```typescript
type List<A> = {kind: "nil"}
| {kind: "cons", hd: A, tl: List<A>};
```

- We can pick any type for `A`
  - TypeScript replaces all the “A”s by the type we give
  - e.g., `List<bigint>` is this type:

```typescript
type List<bigint> =
| {kind: "nil"}
| {kind: "cons", hd: bigint, tl: List<bigint>};
```
Generic Types

We can describe this pattern with a “generic” list type

type List<A> = {kind: “nil”}
| {kind: “cons”, hd: A, tl: List<A>};

Can now have

• List<bigint> = List
• List<Block> = Row
• List<List<Block>> = Quilt
• List<JSX.Element> = JsxList
Generic Types

We can describe this pattern with a “generic” list type

type List<A> = {kind: “nil”}
  | {kind: “cons”, hd: A, tl: List<A>};

• “A” is called a type parameter

• List is a function that takes a type as an argument and returns a new type
  – argument is the type of elements, result is list type
    (this is an analogy in Java, but it’s literally true in TypeScript)

• Illegal to write “List” without its argument
We also need to update the `cons` helper function

```typescript
type List<A> = {kind: "nil"}
  | {kind: "cons", hd: A, tl: List<A>};

const cons = <A,>(x: A, L: List<A>): List<A> => {
  return {kind: "cons", hd: x, tl: L};
};
```

- This is now a “generic function”
  - it has its own type parameter `<A,>`
  - extra comma is weird but required
    compiler thinks `<A>` is an HTML tag
Generic Functions

We also need to update the `cons` helper function

```javascript
const cons = <A,>(x: A, L: List<A>): List<A> => {
  return {kind: "cons", hd: x, tl: L};
};
```

- Parameters to generic types must be provided
- Parameters to generic functions are usually *inferred*

```
cons(1n, cons(2n, nil))  // has type List<bigint>
```
Generic Types & Functions

• We won’t ask you to define generic types this quarter

• But you will need to use them
  – we will use `List<A>` in every assignment from now on
  – lists are the basic data structure of functional programming
Type Erasure
Type Checkers

• Type checkers eliminate large classes of bugs
  – e.g., cannot pass a string where an int is expected
  – critical part of ensuring **correctness**

• Sometimes give you ways to **opt out** of type checking
  – type casts says “just trust me”
  – “any” type
Run-Time Type Checking

- Java will double-check at **run-time** that you were right
  - type cast will fail with `ClassCastException`
  - however, there are cases where it **cannot** double-check

```java
Integer n = (Integer) obj;    // okay
List<Integer> L = (List<Integer>) obj;  // okay?
```

- Java can do some checks at run-time
  - can check if `obj` is an `Integer`
  - can check if `obj` is a `List<?>` *(list of something)*
  - **cannot** check if `obj` is a `List<Integer>`!
Run-Time Type Checking

• Java will double-check at run-time that you were right
  – type cast will fail with `ClassCastException`
  – however, there are cases where it cannot double-check

  ```java
  Integer n = (Integer) obj; // okay
  List<Integer> L = (List<Integer>) obj; // not okay
  ```

• Cannot check if `obj` is a `List<Integer>`
  – all type parameters are “erased”
  – all `Lists` are `List<Object>` at run-time
    if it is correct, it is a `List<Object>` that happens to hold `Integers`
Type Erasure in Java

```java
if (obj instanceof List<Integer>) {  // not okay

• Java will give you an error on this line
  – it can tell if \text{L} is a List
  – it cannot tell if \text{L} is a List<Integer> (vs List<String>)

  Integer n = (Integer) obj;  // okay
  List<Integer> L = (List<Integer>) obj;  // not okay

• Java only gives a warning about the second cast
  – should really be an error
  – programs with these warnings are unsafe
```
Type Erasure in TypeScript

• In TypeScript, all declared type information is erased!
  – no way to tell what type anything had in the source code

• Type casts are not double-checked at run-time
  – the only run-time type checks are ones you write

• If you use casts or “any” types, expect pain
  – variables will have values of types you didn’t expect
  – code will fail in bizarre ways
Handling Type Erasure

Options for avoiding painful debugging

1. Do not use (unchecked) type casts or “any” types
   – almost certainly the best option

2. Check the types yourself at run-time
   – lots of extra work
   – easy to make mistakes
   – (sometimes the only option)
Debugging
A Bug’s Life

- **Defect** (“the bug”): mistake made by a human
- **Error**: computation performed incorrectly
- **Failure**: mistake visible to the user

Debugging is the search from failure back to defect
Debugging

• Debugging is **different from coding**
  – only happens when states are not as expected
    
    variable has an unexpected type
    
    state does not satisfy the expected assertions

• Never know how long it will take
  – happens when you made a mistake in reasoning
    
    initially, you don’t understand what is going on (by definition!)
  – requires a full **understanding** of all code involved
    
    could be a lot of code...
  – important to start early!
Debugging

- **Debugging is different from coding**
  - only happens when states are not as expected
    - variable has an unexpected type
    - state does not satisfy the expected assertions

- **Arguably harder than coding...**

  “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

  - write code as simply as possible
    - if not straight from the spec, then no mutation
    - if not no mutation, then only local variable mutation

Brian Kernighan
Debugging Sucks

• Rule #1 for debugging: **avoid** it

• Tips for *avoiding* debugging:
  1. Write the code as **simply** as possible
     - save complication for complicated problems
  2. Apply rigorous testing and **reasoning**
     - get the code right the first time
  3. Practice **defensive** programming
     - catch errors as quickly as possible (reduce the search space)

• Fight the temptation to skip these steps...
Debugging Sucks

“Easy is the road that leads to destruction.”
— Ancient Wisdom (on Programming?)
Debugging Sucks

• Rule #1 for debugging: avoid it

• Tips for avoiding debugging:
  1. Write the code as simply as possible
     save complication for complicated problems
  2. Apply rigorous testing and reasoning
     get the code right the first time
  3. Practice defensive programming
     catch errors as quickly as possible (reduce the search space)

• Tips for doing (surviving) debugging...
  – concise notes published on website
Debugging Tip #1

• Check the easy stuff first
  – make sure all the files are saved
  – restart the server
  – restart your computer
  – make sure someone didn’t already fix it

• If it is one of the first 3, you will not find it debugging
  – every minute you spend until you hit save / restart is wasted
Debugging Tip #2

• Create a minimal example that demonstrates the bug
  – easier to look through everything in the debugger

• Shrink the input that fails:

  Find “very happy” in “Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all.”

  Find “very happy” in “I am very very happy to see you all.”
  
  Find “very happy” in “very very happy”
  
  Find “ab” in “aab”
How to Fix a Bug

• Start with a test that **fails**
  – make sure you see it fail!
  – can mistakenly write a test that worked already

• Understand why it fails
  – understand where your reasoning was wrong

• Fix the bug

• Make sure the all the tests now pass
  – new test and all previous tests
Debugging Tip #3

• Look for common silly mistakes
  – comparing records with `===`
  – misspelling the name of a method you were implementing in Java, implementing “equal” instead of “equals”
  – passing arguments in the wrong order

• Easy for these to slip past reasoning
  – better chance of finding them with tools or testing
    tools will miss wrong order if both arguments have the same type
  – but some will slip through
Debugging Tip #4

- Make sure it is a bug!
  - check the spec carefully
  - tricky specs can trick you

- These are the absolute worst
  - spend hours and then discover the code was right all along
Debugging Tip #5

• After 20+ min debugging, be **systematic**
  – don’t just try things you think might fix it

• Write down what you have tried
  – don’t try the same thing again and again

• Use the Scientific Method:
  - Formulate a **hypothesis**
  - Design an **experiment**
  - Perform an **experiment**
  - Interpret **results**
Debugging Tip #5

• **Use Binary Search to find the error**

  state is good when the object is created
  ...
  state is bad when user clicks “submit”

• **Find an event that happens somewhere in the middle**

  state is good when the object is created
  ...
  Is the state good when the user clicks on the dropdown?
  ...
  state is bad when user clicks “submit”

  – save an alias to the object when created
Debugging Tip #6

• Try explaining the problem to someone / something
  – can even be a rubber duck
    Pragmatic Programmer calls this “rubber ducking”

• Talking through the problem often helps you spot it
  – this happens all the time
Debugging Tip #7

• Get some sleep!
  – the later it gets, the dumber I get
  – often don’t realize it until 4–5am

• Common to wake up and instantly see the problem

• Important to start early!
  – can’t do this the night it is due
Debugging Tip #8

• Get some help!
  – easy for bugs to hide in your blind spots

• After some number of hours, continuing is not helpful
  – need new ideas about where to look

• Important to start early!
  – no office hours late at night
Defensive Programming Tip #4

- If you spent 30+ min debugging, make it a test case
  - solid evidence that it’s a tricky case

- Bugs that happen once often come back
  - code is changed in the future
  - good chance the same error will happen in the new version

- These are called “regression tests”
  - avoid the bug coming back (“regressing”)

Reasoning Is Not Optional

• Debugging happens after a reasoning mistake
  – you missed that case when reasoning

• Fixing one case can break another case
  – increasingly likely as the problems get harder
    fixing individual cases is unlikely to ever make it work on all inputs
  – eventually, you get the reasoning right...