Logic and Mechanized Reasoning

Introduction, Induction, and Invariants

Marijn J.H. Heule

Carnegie Mellon University

Introduction

Induction Examples

Structural Induction

Invariants

The Team

Marijn Heule
Instructor

Jeremy Avigad
Instructor

Joseph Reeves
TA

Josh Clune
TA

Tika Naik
TA

Alex Knox
TA

Material, Homework, and Grading

Homepage: `https://www.cs.cmu.edu/~mheule/15311-s24/`

Textbook: `https://avigad.github.io/lamr/`

Repository: `https://github.com/avigad/lamr`

Homework on gradescope

- Assignments each Wednesday and due a week later
- Obtain and submit homework on Gradescope
- One day late policy, penalty 3 points (10%)
- Email us if you didn't receive an invitation

Grading

- Homework 40% (10 times 30 points)
- Exams 60% (3 times 150 points)

Office Hours

Jeremy: Tuesdays at 10-11am

Joseph: Wednesdays at 10:30-11:30am

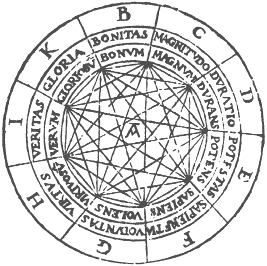
Josh: Tuesdays at 4:30-5:30pm

Tika: Wednesdays at 2-3pm

Alex: Mondays at 4-5pm

What time would you prefer?

Introduction: Ramon Lull, a 13th Century Monk



Three fundamental ideas in the work of Ramon Lull

1. Use symbols or tokens to stand for ideas or concepts
2. Compound ideas and concepts are formed by putting together simpler ones
3. Mechanical devices can serve as aids to reasoning

Introduction: Gottfried Leibniz

Leibniz about a calculus for reasoning:

> *If controversies were to arise, there would be no more need of disputation between two philosophers than between two calculators. For it would suffice for them to take their pencils in their hands and to sit down at the abacus, and say to each other (and if they so wish also to a friend called to help): Let us calculate.*

Calculemus! has become a motto of computer scientists and computationally-minded mathematicians today.

Introduction: Kurt Gödel

In 1931, Kurt Gödel wrote:

> *The development of mathematics towards greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules.*

"Mechanical" predates the modern computer by a decade

Today we have a million-line mathematical library in Lean

Introduction: Course Overview

- Theory We will teach you the syntax and semantics of propositional and first-order logic. The goal is to teach you to think about and talk about logic in a mathematically rigorous way.
- Implementation We will teach you how to implement logical syntax in a functional programming language called Lean. We will also teach you how to carry out fundamental operations and transformations on these objects.
- Application We will show you how to use logic-based automated reasoning tools to solve interesting and difficult problems.

Introduction

Induction Examples

Structural Induction

Invariants

Induction Examples: Sum of Natural Numbers

Theorem

For every natural number n, $\sum_{i \leq n} i = n(n+1)/2$

Proof by induction.

In the base case, we have $\sum_{i \leq 0} i = 0 = 0(0+1)/2$

In the inductive case, assuming $\sum_{i \leq n} i = n(n+1)/2$

$$\begin{aligned} \sum_{i \leq n+1} i &= \sum_{i \leq n} i + (n+1) \\ &= n(n+1)/2 + 2(n+1)/2 \\ &= (n+1)(n+2)/2 \end{aligned}$$

$\square$

Induction Examples: Recursion

A close companion to induction is the principle of recursion

$$
\begin{aligned}
g(0) &= 1 \\
g(n+1) &= (n+1)g(n)
\end{aligned}
$$

The function $g(n)$ is equivalent to factorial: $n!$

Induction Examples: Factorial Example

Theorem

$$\sum_{i \leq n} i \cdot i! = (n+1)! - 1$$

Proof by induction.

The base case is easy. Assuming the claim holds for n

$$\begin{aligned}
\sum_{i \leq n+1} i \cdot i! &= \sum_{i \leq n} i \cdot i! + (n+1) \cdot (n+1)! \\
&= (n+1)! + (n+1) \cdot (n+1)! - 1 \\
&= (n+1)! \cdot (1 + (n+1)) - 1 \\
&= (n+2)! - 1
\end{aligned}$$

$\square$

Induction Examples: General Recursion I

In general, we can define a function recursively as long as some well-founded measure on the arguments decreases.

Example (Greatest common divisor)

$$gcd(x,y) = \begin{cases} x & \text{if } y = 0 \\ gcd(y, mod(x,y)) & \text{otherwise} \end{cases}$$

$$gcd(21,15) \Rightarrow gcd(15,6) \Rightarrow gcd(6,3) \Rightarrow gcd(3,0) \Rightarrow 3$$

Question: What decreases in the recursive call?

Induction Examples: General Recursion II

Example

$$gcd(x,y) = \begin{cases} x & \text{if } y = 0 \\ gcd(y, mod(x,y)) & \text{otherwise} \end{cases}$$

Homework: using the above definition, show that for every nonnegative x and y, there are integers a and b such that $gcd(x,y) = ax + by$.

E.g. $gcd(21,15) = -2 \cdot 21 + 3 \cdot 15$

Hint: You can prove the claim as stated, assuming that it is true for any smaller value of y and any x at all.

Structural Induction: Beyond the natural numbers

The natural numbers are an example of an inductively defined structure:

- 0 is a natural number.
- If x is a natural number, so is $succ(x)$.

Structural Induction: Beyond the natural numbers

The natural numbers are an example of an inductively defined structure:

- 0 is a natural number.
- If x is a natural number, so is $succ(x)$.

Can we also define datastructures in a similar way?

Structural Induction: Lists

Let α be a data type.

Let $List(\alpha)$ be the set of all lists of type α:

- The element nil is an element of $List(\alpha)$.
- If a is an element of α and ℓ is an element of $List(\alpha)$, then the element $cons(a, \ell)$ is an element of $List(\alpha)$.

Structural Induction: Lists

Let α be a data type.

Let $List(\alpha)$ be the set of all lists of type α:

- The element nil is an element of $List(\alpha)$.
- If a is an element of α and ℓ is an element of $List(\alpha)$, then the element $cons(a, \ell)$ is an element of $List(\alpha)$.

Notation:

- nil denotes the empty list, also denote by $[]$.
- $cons(a, \ell)$ denotes adding a to the beginning of list ℓ, also written as $a :: \ell$

Structural Induction: Lists

Let α be a data type.

Let $List(\alpha)$ be the set of all lists of type α:

- The element nil is an element of $List(\alpha)$.
- If a is an element of α and ℓ is an element of $List(\alpha)$, then the element $cons(a, \ell)$ is an element of $List(\alpha)$.

Notation:

- nil denotes the empty list, also denote by $[]$.
- $cons(a, \ell)$ denotes adding a to the beginning of list ℓ, also written as $a :: \ell$

Example

The list of natural numbers $[1, 2, 3]$ would be written as $cons(1, cons(2, cons(3, nil)))$ or $1 :: (2 :: (3 :: []))$

Structural Induction: Append

Definition of *append*:

$$\begin{aligned} append(nil, m) &= m \\ append(cons(a, \ell), m) &= cons(a, append(\ell, m)) \end{aligned}$$

Structural Induction: Append

Definition of *append*:

$$\begin{aligned} append(nil, m) &= m \\ append(cons(a, \ell), m) &= cons(a, append(\ell, m)) \end{aligned}$$

Alternatively written as:

$$\begin{aligned} [] \mathbin{+\!\!+} m &= m \\ (a :: \ell) \mathbin{+\!\!+} m &= a :: (\ell \mathbin{+\!\!+} m) \end{aligned}$$

Structural Induction: *append* Lemma

Recall the definition of *append*:

$$\begin{aligned} [\,] \mathbin{+\!\!+} m &= m \\ (a :: \ell) \mathbin{+\!\!+} m &= a :: (\ell \mathbin{+\!\!+} m) \end{aligned}$$

Lemma

For every List ℓ, we have $\ell \mathbin{+\!\!+} [\,] = \ell$.

Structural Induction: *append* Lemma

Recall the definition of *append*:

$$\begin{aligned} [] \mathbin{+\!\!+} m &= m \\ (a :: \ell) \mathbin{+\!\!+} m &= a :: (\ell \mathbin{+\!\!+} m) \end{aligned}$$

Lemma
For every List ℓ, we have $\ell \mathbin{+\!\!+} [] = \ell$.

Proof.
Base case:

Structural Induction: *append* Lemma

Recall the definition of *append*:

$$\begin{aligned} [] +\!\!+\ m &= m \\ (a :: \ell) +\!\!+\ m &= a :: (\ell +\!\!+\ m) \end{aligned}$$

Lemma

For every List ℓ, we have $\ell +\!\!+ [] = \ell$.

Proof.

Base case: $[] +\!\!+ [] = []$

Inductive case:

Structural Induction: *append* Lemma

Recall the definition of *append*:

$$\begin{aligned} [\,] \mathbin{+\!\!+} m &= m \\ (a :: \ell) \mathbin{+\!\!+} m &= a :: (\ell \mathbin{+\!\!+} m) \end{aligned}$$

Lemma

For every List ℓ, we have $\ell \mathbin{+\!\!+} [\,] = \ell$.

Proof.

Base case: $[\,] \mathbin{+\!\!+} [\,] = [\,]$

Inductive case: Suppose we have $\ell \mathbin{+\!\!+} [\,] = \ell$

$$(a :: \ell) \mathbin{+\!\!+} [\,] =$$

Structural Induction: *append* Lemma

Recall the definition of *append*:

$$\begin{aligned} [] +\!\!+\, m &= m \\ (a :: \ell) +\!\!+\, m &= a :: (\ell +\!\!+\, m) \end{aligned}$$

Lemma

For every List ℓ, we have $\ell +\!\!+\, [] = \ell$.

Proof.

Base case: $[] +\!\!+\, [] = []$

Inductive case: Suppose we have $\ell +\!\!+\, [] = \ell$

$$(a :: \ell) +\!\!+\, [] = a :: (\ell +\!\!+\, [])$$

Structural Induction: *append* Lemma

Recall the definition of *append*:

$$\begin{aligned} [\,] \mathbin{+\!\!+} m &= m \\ (a :: \ell) \mathbin{+\!\!+} m &= a :: (\ell \mathbin{+\!\!+} m) \end{aligned}$$

Lemma

For every List ℓ, we have $\ell \mathbin{+\!\!+} [\,] = \ell$.

Proof.

Base case: $[\,] \mathbin{+\!\!+} [\,] = [\,]$

Inductive case: Suppose we have $\ell \mathbin{+\!\!+} [\,] = \ell$

$$\begin{aligned} (a :: \ell) \mathbin{+\!\!+} [\,] &= a :: (\ell \mathbin{+\!\!+} [\,]) \\ &= a :: \ell \end{aligned}$$

$\square$

Structural Induction: Associativity of *append*

Recall the definition of *append*:

$$
\begin{aligned}
[] +\!\!+\, m &= m \\
(a :: \ell) +\!\!+\, m &= a :: (\ell +\!\!+\, m)
\end{aligned}
$$

Lemma

For every List ℓ, m, n: $\ell +\!\!+\, (m +\!\!+\, n) = (\ell +\!\!+\, m) +\!\!+\, n$

Structural Induction: Associativity of *append*

Recall the definition of *append*:

$$\begin{aligned} [] +\!\!+\, m &= m \\ (a :: \ell) +\!\!+\, m &= a :: (\ell +\!\!+\, m) \end{aligned}$$

Lemma

For every List ℓ, m, n: $\ell +\!\!+\, (m +\!\!+\, n) = (\ell +\!\!+\, m) +\!\!+\, n$

Proof.

Base case:

Structural Induction: Associativity of *append*

Recall the definition of *append*:

$$\begin{aligned} [] \mathbin{+\!\!+} m &= m \\ (a :: \ell) \mathbin{+\!\!+} m &= a :: (\ell \mathbin{+\!\!+} m) \end{aligned}$$

Lemma

For every List ℓ, m, n*:* $\ell \mathbin{+\!\!+} (m \mathbin{+\!\!+} n) = (\ell \mathbin{+\!\!+} m) \mathbin{+\!\!+} n$

Proof.

Base case: $[] \mathbin{+\!\!+} (m \mathbin{+\!\!+} n) = m \mathbin{+\!\!+} n = ([] \mathbin{+\!\!+} m) \mathbin{+\!\!+} n$

Inductive case:

Structural Induction: Associativity of *append*

Recall the definition of *append*:

$$
\begin{aligned}
[] +\!\!+\; m &= m \\
(a :: \ell) +\!\!+\; m &= a :: (\ell +\!\!+\; m)
\end{aligned}
$$

Lemma

For every List ℓ, m, n*:* $\ell +\!\!+\; (m +\!\!+\; n) = (\ell +\!\!+\; m) +\!\!+\; n$

Proof.

Base case: $[] +\!\!+\; (m +\!\!+\; n) = m +\!\!+\; n = ([] +\!\!+\; m) +\!\!+\; n$

Inductive case:

Suppose we have $\ell +\!\!+\; (m +\!\!+\; n) = (\ell +\!\!+\; m) +\!\!+\; n$

$$
(a :: \ell) +\!\!+\; (m +\!\!+\; n) \;=
$$

Structural Induction: Associativity of *append*

Recall the definition of *append*:

$$
\begin{aligned}
[] +\!\!+\ m &= m \\
(a :: \ell) +\!\!+\ m &= a :: (\ell +\!\!+\ m)
\end{aligned}
$$

Lemma

For every List ℓ, m, n: $\ell +\!\!+\ (m +\!\!+\ n) = (\ell +\!\!+\ m) +\!\!+\ n$

Proof.

Base case: $[] +\!\!+\ (m +\!\!+\ n) = m +\!\!+\ n = ([] +\!\!+\ m) +\!\!+\ n$

Inductive case:

Suppose we have $\ell +\!\!+\ (m +\!\!+\ n) = (\ell +\!\!+\ m) +\!\!+\ n$

$$
\begin{aligned}
(a :: \ell) +\!\!+\ (m +\!\!+\ n) &= a :: (\ell +\!\!+\ (m +\!\!+\ n)) \\
&= a :: ((\ell +\!\!+\ m) +\!\!+\ n) \\
&= (a :: (\ell +\!\!+\ m)) +\!\!+\ n \\
&= ((a :: \ell) +\!\!+\ m) +\!\!+\ n
\end{aligned}
$$

□

Structural Induction: The function *append*1 (or *snoc*)

The function *append*1 adds an element to the end of a list:

$$
\begin{aligned}
append1(nil, a) &= cons(a, nil) \\
append1(cons(b, \ell), a) &= cons(b, append1(\ell, a))
\end{aligned}
$$

Structural Induction: The function *append*1 (or *snoc*)

The function *append*1 adds an element to the end of a list:

$$\begin{aligned} append1(nil, a) &= cons(a, nil) \\ append1(cons(b, \ell), a) &= cons(b, append1(\ell, a)) \end{aligned}$$

More compactly it can be written as:

$$\begin{aligned} append1([], a) &= [a] \\ append1(b :: \ell, a) &= b :: append1(\ell, a) \end{aligned}$$

Structural Induction: The function *append*1 (or *snoc*)

The function $append1$ adds an element to the end of a list:

$$\begin{aligned} append1(nil, a) &= cons(a, nil) \\ append1(cons(b, \ell), a) &= cons(b, append1(\ell, a)) \end{aligned}$$

More compactly it can be written as:

$$\begin{aligned} append1([], a) &= [a] \\ append1(b :: \ell, a) &= b :: append1(\ell, a) \end{aligned}$$

Observe that $append1(\ell, a)$ equals $\ell +\!\!+ [a]$

Structural Induction: *reverse* of Lists

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) \mathbin{+\!\!+} [a] \end{aligned}$$

Structural Induction: *reverse* of Lists

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) +\!\!+ [a] \end{aligned}$$

Lemma

For all List ℓ, m: $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

Structural Induction: *reverse* of Lists

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) +\!\!+ [a] \end{aligned}$$

Lemma

For all List ℓ, m*:* $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

Proof.

Base case:

Structural Induction: *reverse* of Lists

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) +\!\!+ [a] \end{aligned}$$

Lemma

For all List ℓ, m*:* $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

Proof.

Base case: $r([] +\!\!+ m) = r(m) = r(m) +\!\!+ [] = r(m) +\!\!+ r([])$

Induction:

Structural Induction: *reverse* of Lists

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) +\!\!+ [a] \end{aligned}$$

Lemma

For all List ℓ, m*:* $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

Proof.

Base case: $r([] +\!\!+ m) = r(m) = r(m) +\!\!+ [] = r(m) +\!\!+ r([])$

Induction:

Suppose we have $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

$$reverse((a :: \ell) +\!\!+ m) =$$

Structural Induction: *reverse* of Lists

$$\begin{array}{rcl} reverse([]) & = & [] \\ reverse(a :: \ell) & = & reverse(\ell) +\!\!+ [a] \end{array}$$

Lemma

For all List ℓ, m: $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

Proof.

Base case: $r([] +\!\!+ m) = r(m) = r(m) +\!\!+ [] = r(m) +\!\!+ r([])$
Induction:
Suppose we have $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

$$reverse((a :: \ell) +\!\!+ m) \;=\; reverse(a :: (\ell +\!\!+ m))$$

Structural Induction: *reverse* of Lists

$$
\begin{array}{rcl}
reverse([]) & = & [] \\
reverse(a :: \ell) & = & reverse(\ell) +\!\!+ [a]
\end{array}
$$

Lemma

For all List ℓ, m: $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

Proof.

Base case: $r([] +\!\!+ m) = r(m) = r(m) +\!\!+ [] = r(m) +\!\!+ r([])$

Induction:

Suppose we have $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

$$
\begin{array}{rcl}
reverse((a :: \ell) +\!\!+ m) & = & reverse(a :: (\ell +\!\!+ m)) \\
 & = & reverse(\ell +\!\!+ m) +\!\!+ [a]
\end{array}
$$

Structural Induction: *reverse* of Lists

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) +\!\!+ [a] \end{aligned}$$

Lemma

For all List ℓ, m: $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

Proof.

Base case: $r([] +\!\!+ m) = r(m) = r(m) +\!\!+ [] = r(m) +\!\!+ r([])$

Induction:

Suppose we have $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

$$\begin{aligned} reverse((a :: \ell) +\!\!+ m) &= reverse(a :: (\ell +\!\!+ m)) \\ &= reverse(\ell +\!\!+ m) +\!\!+ [a] \\ &= (reverse(m) +\!\!+ reverse(\ell)) +\!\!+ [a] \end{aligned}$$

Structural Induction: *reverse* of Lists

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) +\!\!+ [a] \end{aligned}$$

Lemma

For all List ℓ, m: $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

Proof.

Base case: $r([] +\!\!+ m) = r(m) = r(m) +\!\!+ [] = r(m) +\!\!+ r([])$

Induction:

Suppose we have $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

$$\begin{aligned} reverse((a :: \ell) +\!\!+ m) &= reverse(a :: (\ell +\!\!+ m)) \\ &= reverse(\ell +\!\!+ m) +\!\!+ [a] \\ &= (reverse(m) +\!\!+ reverse(\ell)) +\!\!+ [a] \\ &= reverse(m) +\!\!+ (reverse(\ell) +\!\!+ [a]) \end{aligned}$$

Structural Induction: *reverse* of Lists

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) +\!\!+ [a] \end{aligned}$$

Lemma

For all List ℓ, m*:* $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

Proof.

Base case: $r([] +\!\!+ m) = r(m) = r(m) +\!\!+ [] = r(m) +\!\!+ r([])$

Induction:

Suppose we have $reverse(\ell +\!\!+ m) = reverse(m) +\!\!+ reverse(\ell)$

$$\begin{aligned} reverse((a :: \ell) +\!\!+ m) &= reverse(a :: (\ell +\!\!+ m)) \\ &= reverse(\ell +\!\!+ m) +\!\!+ [a] \\ &= (reverse(m) +\!\!+ reverse(\ell)) +\!\!+ [a] \\ &= reverse(m) +\!\!+ (reverse(\ell) +\!\!+ [a]) \\ &= reverse(m) +\!\!+ reverse(a :: \ell) \qquad \square \end{aligned}$$

Structural Induction: *reverse* of *reverse*

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) ++ [a] \end{aligned}$$

Lemma

For every List ℓ holds that $reverse(reverse(\ell)) = \ell$

Structural Induction: *reverse* of *reverse*

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) \mathbin{+\!\!+} [a] \end{aligned}$$

Lemma

For every List ℓ holds that $reverse(reverse(\ell)) = \ell$

Proof.

Base case: $reverse(reverse([])) = reverse([]) = []$
Induction: Suppose we have $reverse(reverse(\ell)) = \ell$

$$reverse(reverse(a :: \ell)) =$$

Structural Induction: *reverse* of *reverse*

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) \mathbin{+\!\!+} [a] \end{aligned}$$

Lemma

For every List ℓ holds that $reverse(reverse(\ell)) = \ell$

Proof.

Base case: $reverse(reverse([])) = reverse([]) = []$
Induction: Suppose we have $reverse(reverse(\ell)) = \ell$

$$reverse(reverse(a :: \ell)) = reverse(reverse(\ell) \mathbin{+\!\!+} [a])$$

Structural Induction: *reverse* of *reverse*

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) +\!\!+ [a] \end{aligned}$$

Lemma

For every List ℓ holds that $reverse(reverse(\ell)) = \ell$

Proof.

Base case: $reverse(reverse([])) = reverse([]) = []$

Induction: Suppose we have $reverse(reverse(\ell)) = \ell$

$$\begin{aligned} reverse(reverse(a :: \ell)) &= reverse(reverse(\ell) +\!\!+ [a]) \\ &= reverse([a]) +\!\!+ reverse(reverse(\ell)) \end{aligned}$$

Structural Induction: *reverse* of *reverse*

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) \mathbin{+\!\!+} [a] \end{aligned}$$

Lemma

For every List ℓ holds that $reverse(reverse(\ell)) = \ell$

Proof.

Base case: $reverse(reverse([])) = reverse([]) = []$

Induction: Suppose we have $reverse(reverse(\ell)) = \ell$

$$\begin{aligned} reverse(reverse(a :: \ell)) &= reverse(reverse(\ell) \mathbin{+\!\!+} [a]) \\ &= reverse([a]) \mathbin{+\!\!+} reverse(reverse(\ell)) \\ &= [a] \mathbin{+\!\!+} reverse(reverse(\ell)) \end{aligned}$$

Structural Induction: *reverse* of *reverse*

$$
\begin{aligned}
reverse([]) &= [] \\
reverse(a :: \ell) &= reverse(\ell) +\!\!+ [a]
\end{aligned}
$$

Lemma

For every List ℓ holds that $reverse(reverse(\ell)) = \ell$

Proof.

Base case: $reverse(reverse([])) = reverse([]) = []$

Induction: Suppose we have $reverse(reverse(\ell)) = \ell$

$$
\begin{aligned}
reverse(reverse(a :: \ell)) &= reverse(reverse(\ell) +\!\!+ [a]) \\
&= reverse([a]) +\!\!+ reverse(reverse(\ell)) \\
&= [a] +\!\!+ reverse(reverse(\ell)) \\
&= [a] +\!\!+ \ell
\end{aligned}
$$

Structural Induction: *reverse* of *reverse*

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) +\!\!+ [a] \end{aligned}$$

Lemma

For every List ℓ holds that $reverse(reverse(\ell)) = \ell$

Proof.

Base case: $reverse(reverse([])) = reverse([]) = []$
Induction: Suppose we have $reverse(reverse(\ell)) = \ell$

$$\begin{aligned} reverse(reverse(a :: \ell)) &= reverse(reverse(\ell) +\!\!+ [a]) \\ &= reverse([a]) +\!\!+ reverse(reverse(\ell)) \\ &= [a] +\!\!+ reverse(reverse(\ell)) \\ &= [a] +\!\!+ \ell \\ &= a :: \ell \end{aligned}$$

□

Structural Induction: *reverse* of *reverse*

$$\begin{aligned} reverse([]) &= [] \\ reverse(a :: \ell) &= reverse(\ell) +\!\!+ [a] \end{aligned}$$

Lemma

For every List ℓ holds that $reverse(reverse(\ell)) = \ell$

Proof.

Base case: $reverse(reverse([])) = reverse([]) = []$

Induction: Suppose we have $reverse(reverse(\ell)) = \ell$

$$\begin{aligned} reverse(reverse(a :: \ell)) &= reverse(reverse(\ell) +\!\!+ [a]) \\ &= reverse([a]) +\!\!+ reverse(reverse(\ell)) \\ &= [a] +\!\!+ reverse(reverse(\ell)) \\ &= [a] +\!\!+ \ell \\ &= a :: \ell \end{aligned}$$

$\square$

Structural Induction: What is the complexity of *reverse*?

$$\begin{array}{rcl} \mathit{reverse}([]) & = & [] \\ \mathit{reverse}(a :: \ell) & = & \mathit{reverse}(\ell) +\!\!+ [a] \end{array}$$

Structural Induction: What is the complexity of *reverse*?

$$\begin{array}{rcl} reverse([]) & = & [] \\ reverse(a :: \ell) & = & reverse(\ell) +\!\!+ [a] \end{array}$$

Example

$$\begin{array}{rcl} reverse([1,2,3]) & = & (reverse([2,3])) +\!\!+ [1] \\ & = & ((reverse([3])) +\!\!+ [2]) +\!\!+ [1] \\ & = & (((reverse([])) +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] \\ & = & (([] +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] \\ & = & ([3] +\!\!+ [2]) +\!\!+ [1] \end{array}$$

Structural Induction: What is the complexity of *reverse*?

$$
\begin{aligned}
\mathit{reverse}([]) &= [] \\
\mathit{reverse}(a :: \ell) &= \mathit{reverse}(\ell) +\!\!+ [a]
\end{aligned}
$$

Example

$$
\begin{aligned}
\mathit{reverse}([1,2,3]) &= (\mathit{reverse}([2,3])) +\!\!+ [1] \\
&= ((\mathit{reverse}([3])) +\!\!+ [2]) +\!\!+ [1] \\
&= (((\mathit{reverse}([])) +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] \\
&= (([] +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] \\
&= ([3] +\!\!+ [2]) +\!\!+ [1] \\
&= ((3 :: []) +\!\!+ [2]) +\!\!+ [1] \\
&= (3 :: ([] +\!\!+ [2])) +\!\!+ [1] \\
&= (3 :: [2]) +\!\!+ [1] \\
&= 3 :: ([2] +\!\!+ [1]) \\
&= 3 :: ((2 :: []) +\!\!+ [1]) \\
&= 3 :: (2 :: ([] +\!\!+ [1]) = 3 :: (2 :: [1]) = [3,2,1]
\end{aligned}
$$

Structural Induction: Efficient Execution

Consider an alternative function to reverse a list:

$$\begin{aligned}
\mathit{reverseAux}([], m) &= m \\
\mathit{reverseAux}((a :: \ell), m) &= \mathit{reverseAux}(\ell, (a :: m)) \\
\mathit{reverse}'(\ell) &= \mathit{reverseAux}(\ell, [])
\end{aligned}$$

Structural Induction: Efficient Execution

Consider an alternative function to reverse a list:

$$\begin{aligned} reverseAux([], m) &= m \\ reverseAux((a :: \ell), m) &= reverseAux(\ell, (a :: m)) \\ reverse'(\ell) &= reverseAux(\ell, []) \end{aligned}$$

Lemma

For every List ℓ, m*:* $reverseAux(\ell, m) = reverse(\ell) +\!\!+\ m$

Structural Induction: Efficient Execution

Consider an alternative function to reverse a list:

$$\begin{array}{rcl} reverseAux([], m) & = & m \\ reverseAux((a :: \ell), m) & = & reverseAux(\ell, (a :: m)) \\ reverse'(\ell) & = & reverseAux(\ell, []) \end{array}$$

Lemma

For every List ℓ, m: $reverseAux(\ell, m) = reverse(\ell) \mathbin{+\!\!+} m$

Proof.

Base case: $reverseAux([], m) = m = [] \mathbin{+\!\!+} m = reverse([]) \mathbin{+\!\!+} m$
Induction: Assume $reverseAux(\ell, m) = reverse(\ell) \mathbin{+\!\!+} m$

$$reverseAux((a :: \ell), m) \quad =$$

Structural Induction: Efficient Execution

Consider an alternative function to reverse a list:

$$\begin{aligned} reverseAux([], m) &= m \\ reverseAux((a :: \ell), m) &= reverseAux(\ell, (a :: m)) \\ reverse'(\ell) &= reverseAux(\ell, []) \end{aligned}$$

Lemma

For every List ℓ, m: $reverseAux(\ell, m) = reverse(\ell) +\!\!+\ m$

Proof.

Base case: $reverseAux([], m) = m = [] +\!\!+\ m = reverse([]) +\!\!+\ m$
Induction: Assume $reverseAux(\ell, m) = reverse(\ell) +\!\!+\ m$

$$reverseAux((a :: \ell), m) = reverseAux(\ell, (a :: m))$$

Structural Induction: Efficient Execution

Consider an alternative function to reverse a list:

$$\begin{aligned}
reverseAux([], m) &= m \\
reverseAux((a :: \ell), m) &= reverseAux(\ell, (a :: m)) \\
reverse'(\ell) &= reverseAux(\ell, [])
\end{aligned}$$

Lemma

For every List ℓ, m: $reverseAux(\ell, m) = reverse(\ell) +\!\!+\ m$

Proof.

Base case: $reverseAux([], m) = m = [] +\!\!+\ m = reverse([]) +\!\!+\ m$

Induction: Assume $reverseAux(\ell, m) = reverse(\ell) +\!\!+\ m$

$$\begin{aligned}
reverseAux((a :: \ell), m) &= reverseAux(\ell, (a :: m)) \\
&= reverse(\ell) +\!\!+\ (a :: m)
\end{aligned}$$

Structural Induction: Efficient Execution

Consider an alternative function to reverse a list:

$$\begin{aligned} reverseAux([], m) &= m \\ reverseAux((a :: \ell), m) &= reverseAux(\ell, (a :: m)) \\ reverse'(\ell) &= reverseAux(\ell, []) \end{aligned}$$

Lemma

For every List ℓ, m: $reverseAux(\ell, m) = reverse(\ell) ++ m$

Proof.

Base case: $reverseAux([], m) = m = [] ++ m = reverse([]) ++ m$

Induction: Assume $reverseAux(\ell, m) = reverse(\ell) ++ m$

$$\begin{aligned} reverseAux((a :: \ell), m) &= reverseAux(\ell, (a :: m)) \\ &= reverse(\ell) ++ (a :: m) \\ &= reverse(\ell) ++ ([a] ++ m) \end{aligned}$$

Structural Induction: Efficient Execution

Consider an alternative function to reverse a list:

$$\begin{aligned} reverseAux([], m) &= m \\ reverseAux((a :: \ell), m) &= reverseAux(\ell, (a :: m)) \\ reverse'(\ell) &= reverseAux(\ell, []) \end{aligned}$$

Lemma

For every List ℓ, m: $reverseAux(\ell, m) = reverse(\ell) +\!\!+\ m$

Proof.

Base case: $reverseAux([], m) = m = [] +\!\!+\ m = reverse([]) +\!\!+\ m$

Induction: Assume $reverseAux(\ell, m) = reverse(\ell) +\!\!+\ m$

$$\begin{aligned} reverseAux((a :: \ell), m) &= reverseAux(\ell, (a :: m)) \\ &= reverse(\ell) +\!\!+\ (a :: m) \\ &= reverse(\ell) +\!\!+\ ([a] +\!\!+\ m) \\ &= (reverse(\ell) +\!\!+\ [a]) +\!\!+\ m \end{aligned}$$

Structural Induction: Efficient Execution

Consider an alternative function to reverse a list:

$$\begin{aligned} reverseAux([], m) &= m \\ reverseAux((a :: \ell), m) &= reverseAux(\ell, (a :: m)) \\ reverse'(\ell) &= reverseAux(\ell, []) \end{aligned}$$

Lemma

For every List ℓ, m: $reverseAux(\ell, m) = reverse(\ell) +\!\!+\ m$

Proof.

Base case: $reverseAux([], m) = m = [] +\!\!+\ m = reverse([]) +\!\!+\ m$
Induction: Assume $reverseAux(\ell, m) = reverse(\ell) +\!\!+\ m$

$$\begin{aligned} reverseAux((a :: \ell), m) &= reverseAux(\ell, (a :: m)) \\ &= reverse(\ell) +\!\!+\ (a :: m) \\ &= reverse(\ell) +\!\!+\ ([a] +\!\!+\ m) \\ &= (reverse(\ell) +\!\!+\ [a]) +\!\!+\ m \\ &= reverse(a :: \ell) +\!\!+\ m \end{aligned}$$

$\square$

Structural Induction: Complexity Measurements

We can assign any complexity measure to a data type, and do induction on complexity, as long as the measure is well founded.

$$\begin{aligned} \mathit{length}([]) &= 0 \\ \mathit{length}(a :: \ell) &= \mathit{length}(\ell) + 1 \end{aligned}$$

Structural Induction: Properties of Extended Binary Trees

- The element $empty$ is a binary tree.
- If s and t are finite binary trees, so is the $node(s,t)$.

Structural Induction: Properties of Extended Binary Trees

- The element *empty* is a binary tree.
- If s and t are finite binary trees, so is the $node(s,t)$.

Compute the size of an extended binary tree as follows:

$$\begin{aligned} size(empty) &= 0 \\ size(node(s,t)) &= 1 + size(s) + size(t) \end{aligned}$$

Structural Induction: Properties of Extended Binary Trees

- The element *empty* is a binary tree.
- If s and t are finite binary trees, so is the $node(s,t)$.

Compute the size of an extended binary tree as follows:

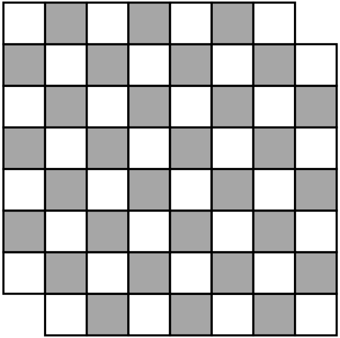
$$\begin{aligned} size(empty) &= 0 \\ size(node(s,t)) &= 1 + size(s) + size(t) \end{aligned}$$

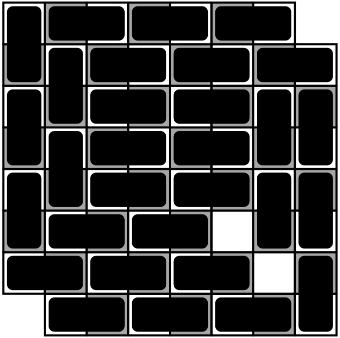
Compute the depth of an extended binary tree as follows:

$$\begin{aligned} depth(empty) &= 0 \\ depth(node(s,t)) &= 1 + \max(depth(s), depth(t)) \end{aligned}$$

Invariants: Mutilated Chessboard I

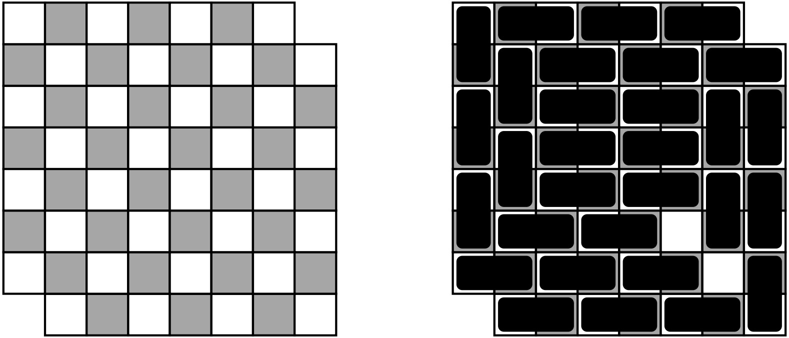
Can a chessboard be fully covered with dominos after removing two diagonally opposite corner squares?





Invariants: Mutilated Chessboard I

Can a chessboard be fully covered with dominos after removing two diagonally opposite corner squares?



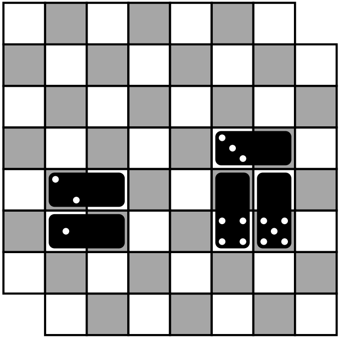
Easy to refute based on the following two observations:

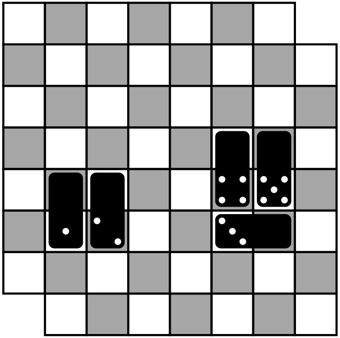
- There are more white squares than black squares; and
- A domino covers exactly one white and one black square.

Invariants: Mutilated Chessboard II

The chessboard pattern invariant is hard to find

Mechanized reasoning can find alternative invariants





Invariants: MU Puzzle by Douglas Hofstadter

Consider string with letters M, I, and U.

1. Replace xI by xIU: append any string ending in I with U.
2. Replace Mx by Mxx: double the string after the initial M.
3. Replace xIIIy by xUy: replace three consecutive Is by U.
4. Replace xUUy by xy: delete any consecutive pair of Us.

The starting with the string MI. Can we get to MU?

Invariants: MU Puzzle by Douglas Hofstadter

Consider string with letters M, I, and U.

1. Replace xI by xIU: append any string ending in I with U.
2. Replace Mx by Mxx: double the string after the initial M.
3. Replace xIIIy by xUy: replace three consecutive Is by U.
4. Replace xUUy by xy: delete any consecutive pair of Us.

The starting with the string MI. Can we get to MU?

What is the invariant?

Invariants: MU Puzzle Invariant

Invariant: The number of Is is $2^a \pmod 3$ for $a \in \mathbb{N}$

Base case: $a = 0$

Induction:

1. Replace xI by xIU: append any string ending in I with U.
2. Replace Mx by Mxx: double the string after the initial M.
3. Replace xIIIy by xUy: replace three consecutive Is by U.
4. Replace xUUy by xy: delete any consecutive pair of Us.

Invariants: MU Puzzle Invariant

Invariant: The number of Is is $2^a \pmod 3$ for $a \in \mathbb{N}$

Base case: $a = 0$

Induction:

1. Replace xI by xIU: append any string ending in I with U.
   - This doesn't change the number of Is
2. Replace Mx by Mxx: double the string after the initial M.
3. Replace xIIIy by xUy: replace three consecutive Is by U.
4. Replace xUUy by xy: delete any consecutive pair of Us.

Invariants: MU Puzzle Invariant

Invariant: The number of Is is $2^a \pmod 3$ for $a \in \mathbb{N}$

Base case: $a = 0$

Induction:

1. Replace xI by xIU: append any string ending in I with U.
   - This doesn't change the number of Is
2. Replace Mx by Mxx: double the string after the initial M.
   - This doubles the number of Is: increases a by 1
3. Replace xIIIy by xUy: replace three consecutive Is by U.
4. Replace xUUy by xy: delete any consecutive pair of Us.

Invariants: MU Puzzle Invariant

Invariant: The number of Is is $2^a \pmod 3$ for $a \in \mathbb{N}$

Base case: $a = 0$

Induction:

1. Replace xI by xIU: append any string ending in I with U.
   - This doesn't change the number of Is
2. Replace Mx by Mxx: double the string after the initial M.
   - This doubles the number of Is: increases a by 1
3. Replace xIIIy by xUy: replace three consecutive Is by U.
   - It reduces the number of Is by 3: no change $\pmod 3$
4. Replace xUUy by xy: delete any consecutive pair of Us.

Invariants: MU Puzzle Invariant

Invariant: The number of Is is $2^a \pmod 3$ for $a \in \mathbb{N}$

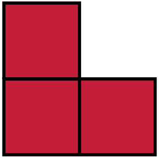
Base case: $a = 0$

Induction:

1. Replace xI by xIU: append any string ending in I with U.
   - This doesn't change the number of Is
2. Replace Mx by Mxx: double the string after the initial M.
   - This doubles the number of Is: increases a by 1
3. Replace xIIIy by xUy: replace three consecutive Is by U.
   - It reduces the number of Is by 3: no change $\pmod 3$
4. Replace xUUy by xy: delete any consecutive pair of Us.
   - This doesn't change the number of Is

Invariants: Golomb's Tromino Theorem

A tromino is an L-shaped configuration of three squares.



Theorem (Golomb's Trominoes Theorem)

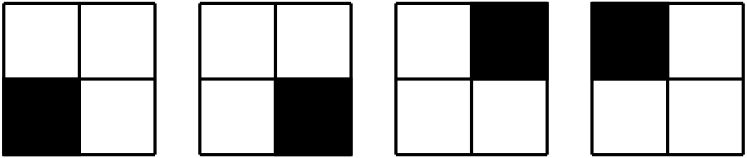
Any $2^n \times 2^n$ *chessboard with one square removed can be tiled with trominoes.*

Invariants: Trominoes 2×2 grid

Theorem (Golomb's Trominoes Theorem)
Any $2^n \times 2^n$ chessboard with one square removed can be tiled with trominoes.

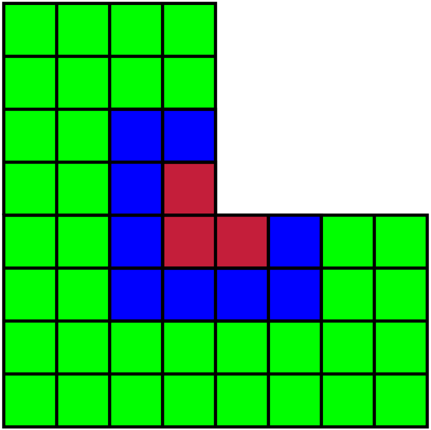
Let's first consider the $n = 1$ case.

All cases are isomorphic. A tromino covers the remaining grid.



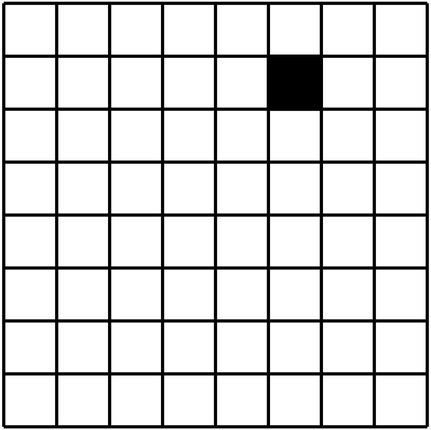
Invariants: Larger Trominoes

Use 4 trominoes of size n to make on of size $2n$



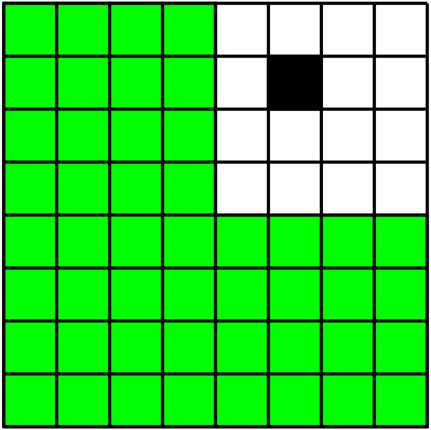
Invariants: Trominoes 8×8 grid

Cover the three quadrants that are not blocked by the square



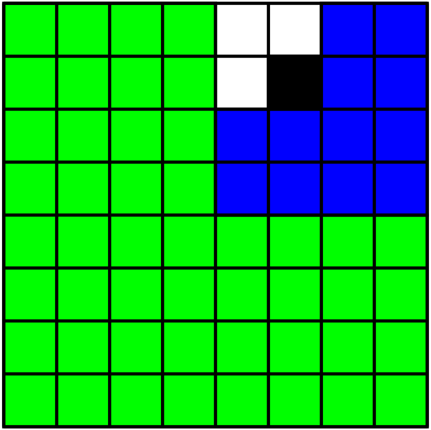
Invariants: Trominoes 8×8 grid

Cover the three quadrants that are not blocked by the square



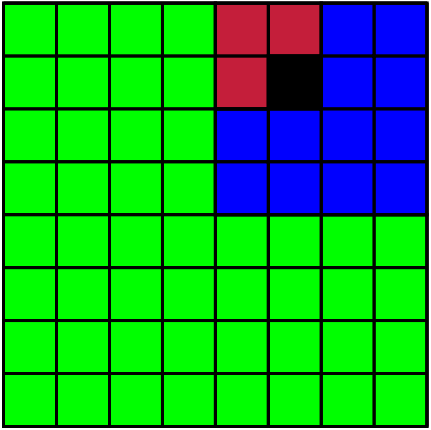
Invariants: Trominoes 8×8 grid

Cover the three quadrants that are not blocked by the square



Invariants: Trominoes 8×8 grid

Cover the three quadrants that are not blocked by the square



Invariants: Loop Invariants

Invariants are not restricted to recursive definitions. Imperative code frequently has invariants and the can be crucial to prove correctness.

Example (Loop invariant)

```
int j = 9;
for (int i=0; i<10; i++)
  j--;
```

The code above has the loop invariant `i + j == 9`