Logic and Mechanized Reasoning

Structural Induction and Invariants

Marijn J.H. Heule

Carnegie Mellon University

Structural Induction

Invariants

Structural Induction

Invariants

Structural Induction: Beyond the natural numbers

Recall the inductive definition of the natural numbers

- 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 $List(\alpha)$.

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*:

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

Alternatively written as:

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

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 $\ell \mathbin{+\!\!+} [] = \ell$.

Proof.

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

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

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

□

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:

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

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

□

Structural Induction: The function *append*1

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:

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

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

Structural Induction: *reverse* of Lists

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

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

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

Structural Induction: *reverse* of *reverse*

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

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: What is the complexity of *reverse*?

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

Example

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

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

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

□

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 $node(s,t)$.

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:

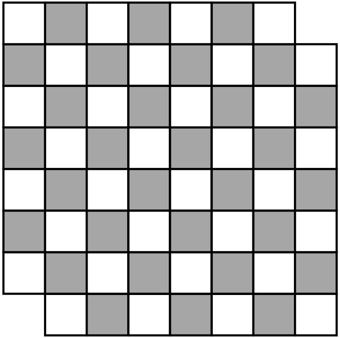
$$\begin{aligned} depth(empty) &= 0 \\ depth(node(s,t)) &= 1 + \max(depth(s), depth(t)) \end{aligned}$$

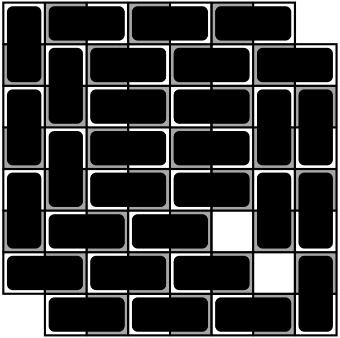
Structural Induction

Invariants

Invariants: Mutilated Chessboard I

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





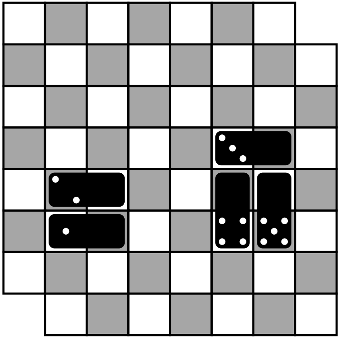
Easy to refute based on the following two observations:

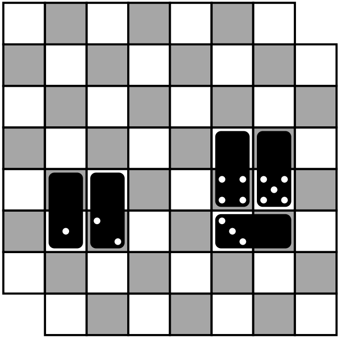
- There are more white squares than black squares; and
- A domino covers exactly one white and one black square.

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?

What is the invariant?

Invariants: MU Puzzle Invariant

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

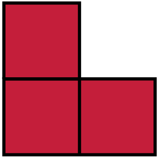
Base case: $a = 0$

Induction:

1. Replace xI by xIU: append any string ending in I with U.
   - 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.*

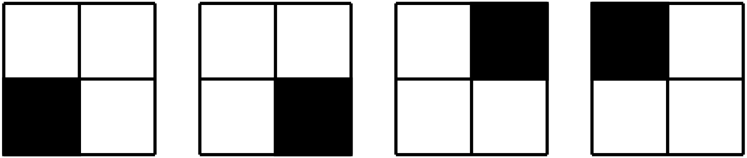
Invariants: Trominoes 2×2 grid

Theorem (Golomb's Trominoes Theorem)

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

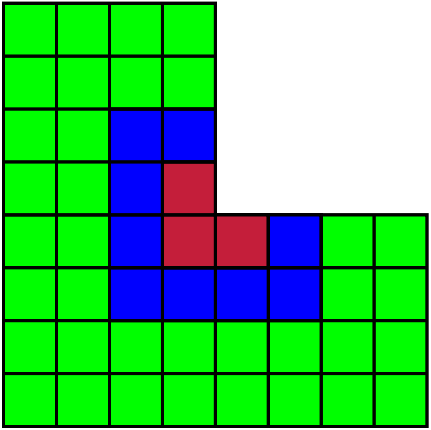
Let's first consider the $n = 1$ case.

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



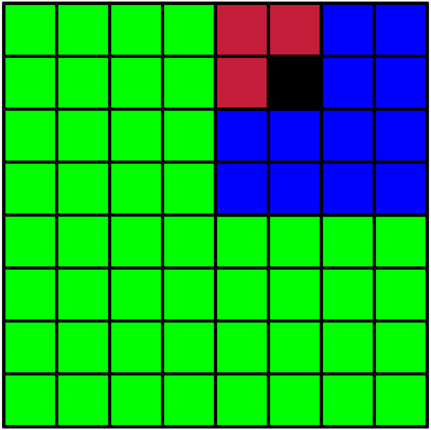
Invariants: Larger Trominoes

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: 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`

Structural Induction

Invariants