



Formal Verification of Rust with Stainless

Master Thesis

Yann Bolliger

August 18, 2021

Company supervisor:

Romain Rüetschi, Informal Systems

Academic co-supervisor:

Georg Stefan Schmid, Laboratory for Automated Reasoning and Analysis, EPFL

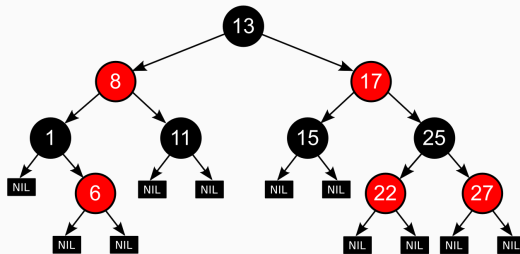
Academic supervisor:

Prof. Viktor Kunčák, Laboratory for Automated Reasoning and Analysis, EPFL

Presenting our tool

rust-stainless

- Created by Georg Schmid



Master Project

Over the course of this project I substantially extended `rust-stainless`.

- Support for mutability, shared and mutable references
- General translation algorithm from Rust mutability to Scala and theoretical argument for its correctness.
- Heap allocation
- `old` helper in postconditions
- Methods on data types
- Implementation blocks, traits and laws (algebraic properties),
- Stainless Map and Set implementation,
- `return` keyword,
- Pattern matching for tuples,
- and many notational improvements.

Master Project

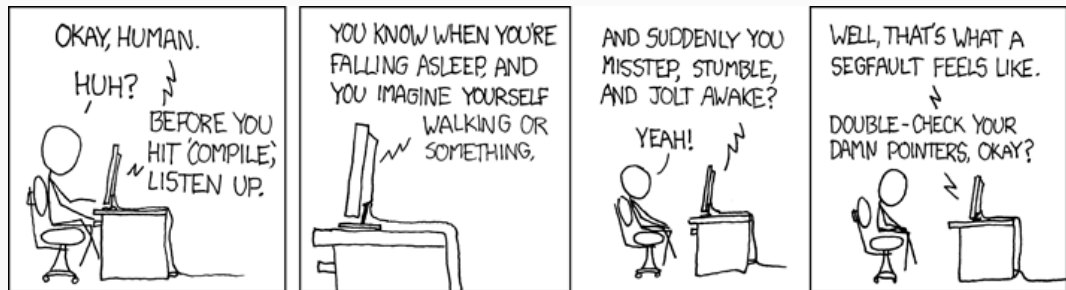
Over the course of this project I substantially extended `rust-stainless`.

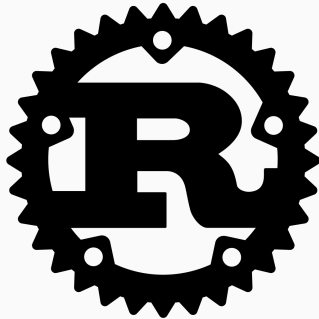
- Support for mutability, shared and mutable references
- General translation algorithm from Rust mutability to Scala and theoretical argument for its correctness.
- Heap allocation
- `old` helper in postconditions
- Methods on data types
- Implementation blocks, traits and laws (algebraic properties),
- Stainless Map and Set implementation,
- `return` keyword,
- Pattern matching for tuples,
- and many notational improvements.

1. Motivation
2. Tool Overview
3. Verifying a Red-Black Tree
4. Mutability Translation
5. Conclusion

Motivation

Motivation






```
/// Returns the square of `x`. Squares are non-negative.  
fn square(x: i32) -> i32 { -2 * x }
```

Motivation

```
/// Returns the square of `x`. Squares are non-negative.  
#[post(ret >= 0)]  
fn square(x: i32) -> i32 { -2 * x }
```

- Result for 'postcondition' VC for square @?:?:

$-2 * x \geq 0$

=> INVALID

Found counter-example:

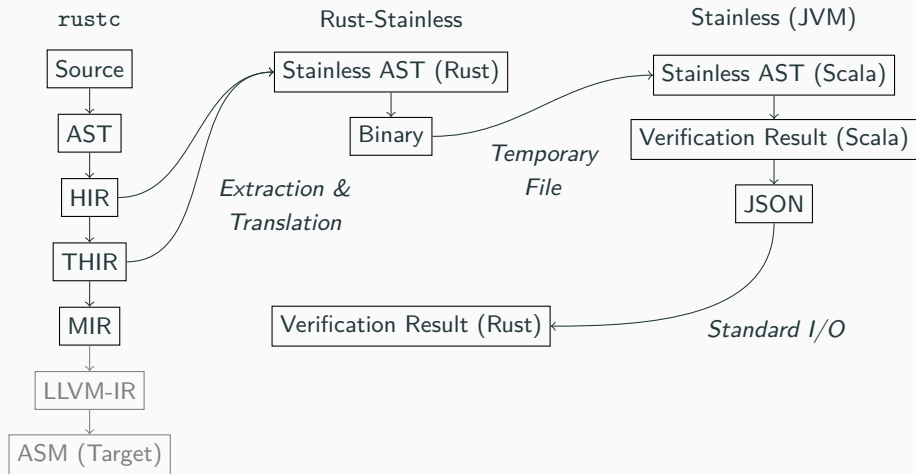
$x: \text{Int} \rightarrow 1$

Tool Overview

How it works

- Verification by Stainless in Scala.
- `rust-stainless` is a frontend to Stainless.
- Translate supported Rust features to Scala.
- Stainless verifies the generated Scala.
But it only supports a subset of Scala, in particular concerning mutability.
- Get verification result back.

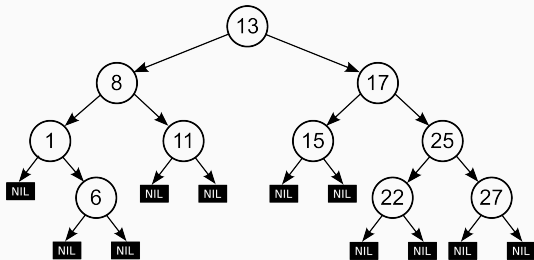
Pipeline



Verifying a Red-Black Tree

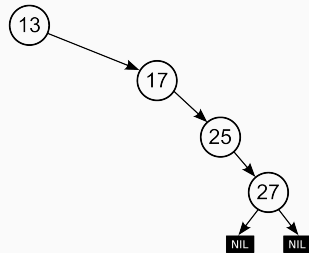
Red-Black Tree

- Binary search tree
- Lookup, insertion and deletion in $\mathcal{O}(\log n)$ time



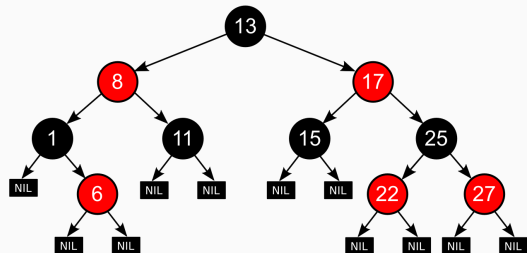
Red-Black Tree

- Binary search tree
- Lookup, insertion and deletion in $\mathcal{O}(\log n)$ time
- But only if well balanced



Red-Black Tree

- Binary search tree
- Lookup, insertion and deletion in $\mathcal{O}(\log n)$ time
- But only if well balanced
- Red-Black Tree uses colouring to automatically rebalance at insertion and deletion

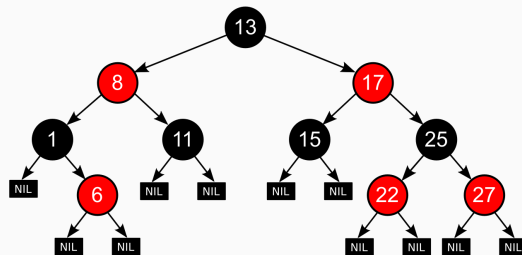


Why verify it?

Red-Black Tree

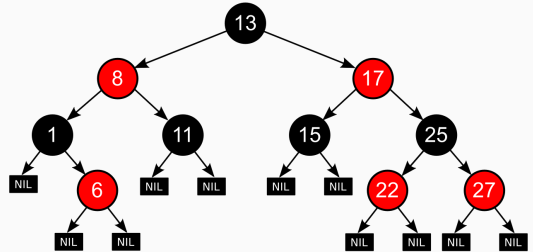
Properties [1]

1. Each node is either red or black.
2. All NIL (empty) nodes are considered black.
3. A red node does not have a red child.
4. Every path from a given node to any of its descendant NIL nodes goes through the same number of black nodes.



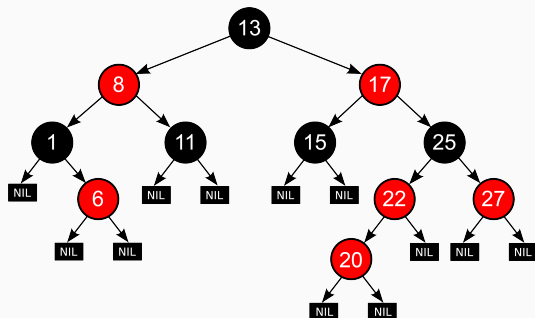
Algorithm

1. Recursively descend in tree to correct position

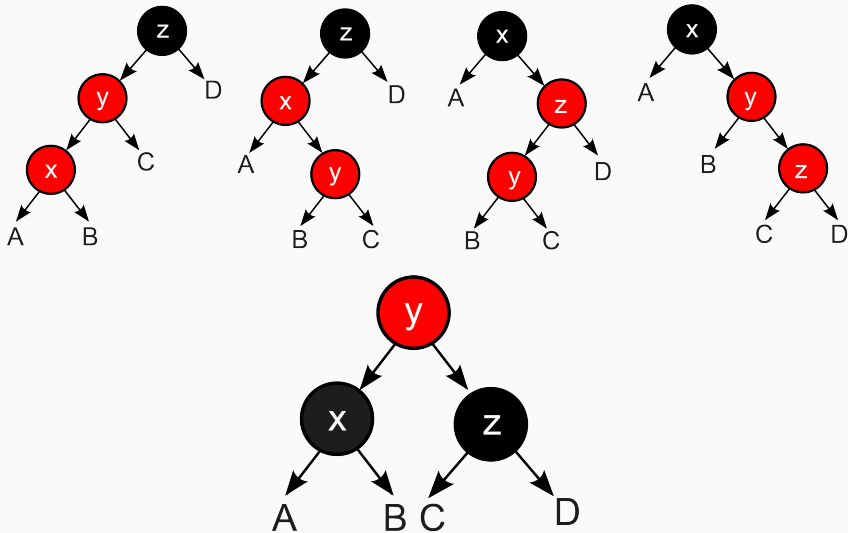


Algorithm

1. Recursively descend in tree to correct position
2. Insert a new red node
3. Recursively go back up and solve all property violations



Rebalancing



Why in Rust?

To insert the integers from 1 to 5000 into the implementation takes:

- Red-Black Tree in Scala, fully functional, no mutation: 185s

To insert the integers from 1 to 5000 into the implementation takes:

- Red-Black Tree in Scala, fully functional, no mutation: 185s
- Red-Black Tree in Rust, fully mutable, no allocation in rebalancing: **10ms**

Red-Black Tree in Rust

```
1 enum Color {  
2     Red,  
3     Black,  
4 }  
5  
6 enum RBTREE<T> {  
7     Empty,  
8     Node(Color, Box<RBTREE<T>>, T, Box<RBTREE<T>>),  
9 }
```

Listing 1: Data types with heap allocation

Red-Black Tree in Rust

```
1  impl RBTREE<i32> {  
2      #[pre(  
3          self.red_nodes_have_black_children()  
4          && self.black_balanced()  
5      )]  
6      #[post(set_equals(  
7          &self.content(), &old(&self).content().insert(&t)  
8          ) && (old(&self).size() == self.size())  
9          || old(&self).size() + 1 == self.size())  
10         && self.red_desc_have_black_children()  
11         && self.black_balanced()  
12     )]  
13     pub fn ins(&mut self, t: i32) { ... }  
14 }
```

Listing 2: Insert method with specification

Red-Black Tree in Rust

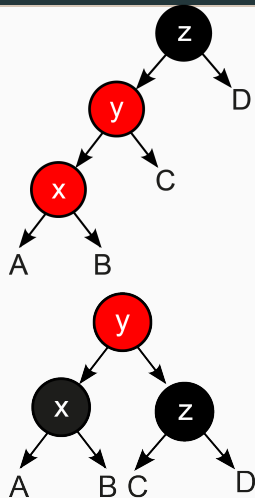
```
1 fn ins(&mut self, t: i32) {
2     match self {
3         Empty => {
4             *self = Node(Red, Box::new(Empty), t, Box::new(Empty));
5         }
6         Node(_, left, value, right) => {
7             if t < *value {
8                 left.ins(t); self.balance();
9             } else if t > *value {
10                 right.ins(t); self.balance();
11             }
12         }
13     }
14 }
```

Listing 3: Recursive insertion

Red-Black Tree in Rust

```
1 fn balance(&mut self) {
2     match self {
3         Node(Black, left, _, _) if left.is_red() => {
4             match &mut **left {
5                 Node(Red, ll, _, _) if ll.is_red() => {
6                     self.rotate_right();
7                     self.recolor();
8                 }
9                 Node(Red, _, _, lr) if lr.is_red() => { ... }
10                _ => {}
11            }
12        }
13        Node(Black, _, _, right) if right.is_red() =>
14            { ... }
15        _ => {}
16    }
17 }
```

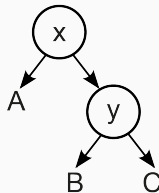
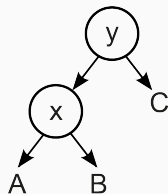
Listing 4: Balance imperatively



Red-Black Tree in Rust

```
1 fn rotate_right(&mut self) {  
2     let self_tree = std::mem::replace(self, Empty);  
3     if let Node(c1, mut left, y, c) = self_tree {  
4         let left_tree = std::mem::replace(&mut *left, Empty);  
5         if let Node(c2, a, x, b) = left_tree {  
6             *left = Node(c1, b, y, c);  
7             *self = Node(c2, a, x, left);  
8         } else {  
9             *left = left_tree;  
10            *self = Node(c1, left, y, c);  
11        }  
12    }  
13 }
```

Listing 5: Tree rotation without allocation, *in-place*.



Now, the tool developed in this project finally comes to action!

```
$ STAINLESS_FLAGS='--timeout=60' cargo stainless
```

- See whether the implementation is correct
- Tool translates Rust code to Scala
- Feeds that to Stainless

Now, the tool developed in this project finally comes to action!

```
$ STAINLESS_FLAGS='--timeout=60' cargo stainless
```

Stainless says: **INVALID**

```
- Result for 'postcondition' VC for balance @?:?:\\  
...\\  
=> INVALID\\
```

```
Found counter-example:\\  
self: MutCell[RBTree[Int]] -> ...
```

Red-Black Tree in Rust

```
1 fn balance(&mut self) {  
2     match self {  
3         Node(Black, left, _, _) if left.is_red() => {  
4             match &mut **left {  
5                 Node(Red, ll, _, _) if ll.is_red() => { ... }  
6                 ...  
7             }  
8         }  
9         Node(Black, _, _, right) if right.is_red() => { ... }  
10        _ => {}  
11    }  
12 }
```

Listing 6: Pattern match acts like an `else if`.

Red-Black Tree in Rust

```
1 fn balance(&mut self) {  
2     if let Node(Black, left, _, _) = self {  
3         if left.is_red() {  
4             match &mut **left {  
5                 Node(Red, ll, _, _) if ll.is_red() => { ... }  
6                 ...  
7             }  
8         }  
9     }  
10    if let Node(Black, _, _, right) = self {  
11        if right.is_red() { ... }  
12    }  
13 }
```

Listing 9: Solved bug in balance function.

That's why we need verification!

Mutability Translation

Translate Rust mutability and references to Scala while providing runtime equivalence in Scala and verification in Stainless.

- Rust has explicit memory management with references. Scala does not.
- Rust has data on the heap & stack. Scala has all objects on the heap.
- Stainless does not support all mutability.

Mutable Cells

To make references possible, a layer of indirection is needed. All mutability is modelled with the mutable cell object:

```
case class MutCell[T](var value: T)
```

Idea

Whenever something is (possibly) mutable, we wrap it into a mutable cell.

In Scala (on the JVM):

- all objects are on the heap,
- objects are handled by reference only.

Example

This allows modelling references by wrapping values in mutable cells:

```
1 struct A<T> {  
2     a: T, b: i32  
3 }  
4 let x = A {  
5     a: "foo", b: 123  
6 }  
7 let mut y = 123;  
8 assert!(y == x.b);
```

```
1 case class A[T](  
2     a: MutCell[T], b: MutCell[Int]  
3 )  
4 val x = MutCell(  
5     A(MutCell("foo"), MutCell(123))  
6 )  
7 val y = MutCell(123)  
8 assert(y.value == x.value.b.value)
```

A reference to y in Rust, e.g. `&mut y`, is now equivalent to the mutable cell y in Scala.

To model Rust's ownership type system, the translation uses the `freshCopy` operator of Stainless.

- Semantically, a deep copy
- Models move and copy semantics
- Helps avoiding problems with Stainless's support of mutability

Example

```
1 let x = (  
2     123,  
3     false  
4 );  
5 // copies `x`  
6 let mut y = x;  
7  
8 y.0 = 456;  
9 // holds:  
10 assert!(x.0 == 123)
```

```
1 val x = MutCell(Tuple2(  
2     MutCell(123),  
3     MutCell(false)  
4 ))  
5 // shares tuple object `x.value`  
6 val y = MutCell(x.value)  
7 // i.e. x.value == y.value  
8 y.value._0.value = 456  
9 // fails:  
10 assert(x.value._0.value == 123)
```

Example

```
1 let x = (  
2     123,  
3     false  
4 );  
5 // copies `x`  
6 let mut y = x;  
7  
8 y.0 = 456;  
9 // holds:  
10 assert!(x.0 == 123)
```

```
1 val x = MutCell(Tuple2(  
2     MutCell(123),  
3     MutCell(false)  
4 ))  
5 // copies tuple object `x.value`  
6 val y = MutCell(freshCopy(x.value))  
7 // i.e. x.value != y.value  
8 y.value._0.value = 456  
9 // holds:  
10 assert(x.value._0.value == 123)
```

Insert `freshCopy` wherever data is moved or copied.

In-place Update

```
1 fn f(a: &mut A) -> A {  
2     std::mem::replace(  
3         a,  
4         A { a: "bar", b: 0 }  
5     )  
6 }
```

```
1 def f(a: MutCell[A]): A = {  
2     val res = freshCopy(a.value)  
3     a.value = A(  
4         MutCell("bar"), MutCell(0)  
5     )  
6     freshCopy(res)  
7 }
```


Conclusion


- `rust-stainless` tool supports a large part of Rust
- Full support for mutable and shared references
- Frontend only accounts for 5 % of running time

- Missing Rust features: arrays, loops, iterators and closures
- Stainless's limitations of mutability
- One-crate-limitation, cannot use functions of the standard library
- No access to lifetimes
- Refinement lifting in the backend


- Using the new fully imperative phase of Stainless currently in development to overcome mutability problems.
- Solving the **one-crate-limitation** by allowing specifications for crate-external items.
- Providing Rust counter-examples.
- Uncover new use-cases, for example in the blockchain context.

That was


 **epfl-lara / rust-stainless**

 Watch ▾

5


 Unstar


5


 Fork


1


<> Code


 Issues 29

 Pull requests 5


 Discussions

 Actions

 Projects

 Security


...




 master ▾








Go to file

Add file ▾


Code ▾


About 

 **yannbolliger** Stainless flags should overwrite default config (...  26 days ago  208


 .github/workflows	Allow local modules and imports (#152)	4 months ago
 demo	Turn on Type Checker (#168)	27 days ago
 libstainless	Rename stainless collection API to match Rust s...	4 months ago
 libstainless_macros	Fix spec macros for mut self param (#148)	4 months ago
 scripts	Change path to rustc_to_stainless binary in scrip...	14 months ago
 stainless_backend	Stainless flags should overwrite default config (...	26 days ago
 stainless_data	Fix anti-aliasing problems by using fresh copy (#...	3 months ago


An experimental Rust frontend for Stainless


 Readme

3. Mai 2021, 16:44 MESZ  Apache-2.0 License

Contributors 4

 **gsps** Georg Stefan Schmid

 **yannbolliger** Yann Bolliger

 **romac** Romain Ruetschi

Thank you!

Questions?

-  T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

Red-Black Tree in Rust

```
1 impl RBTREE<i32> {  
2     pub fn insert(&mut self, t: i32) {  
3         // Insert and color the root black.  
4         self.ins(t);  
5         if let Node(ref mut c, _, _, _) = self {  
6             *c = Black;  
7         }  
8     }  
9 }
```

Listing 10: Insert method

Implementation blocks, traits and laws

```
1 trait Equals {  
2     fn equals(&self, x: &Self) -> bool;  
3     fn not_equals(&self, x: &Self) -> bool {  
4         !self.equals(x)  
5     }  
6     #[law]  
7     fn reflexive(x: &Self) -> bool {  
8         x.equals(x)  
9     }  
10    #[law]  
11    fn symmetric(x: &Self, y: &Self) -> bool {  
12        x.equals(y) == y.equals(x)  
13    }  
14    #[law]  
15    fn transitive(x: &Self, y: &Self, z: &Self) -> bool {  
16        !(x.equals(y) && y.equals(z)) || x.equals(z)  
17    }  
18 }
```

Implementation blocks, traits and laws

```
1  impl Equals for i32 {
2      fn equals(&self, y: &i32) -> bool {
3          *self == *y
4      }
5  }
6  enum List<T> {
7      Nil,
8      Cons(T, Box<List<T>>)
9  }
10 impl<T: Equals> Equals for List<T> {
11     fn equals(&self, other: &List<T>) -> bool {
12         match (self, other) {
13             (List::Nil, List::Nil) => true,
14             (List::Cons(x, xs), List::Cons(y, ys)) => x.equals(y) && xs.equals(ys),
15             _ => false,
16         }
17     }
18     ...
19 }
```


Implementation blocks, traits and laws

```
1 abstract class Equals[Self] {
2   def equals(self: Self, x: Self): Boolean
3   def notEquals(self: Self, x: Self): Boolean = !this.equals(self, x)
4   @law def reflexive(x: Self): Boolean = this.equals(x, x)
5   ...
6 }
7 case object i32asEquals extends Equals[Int] {
8   def equals(self: Int, y: Int): Boolean = self == y
9 }
10 case class ListasEquals[T](ev0: Equals[T]) extends Equals[List[T]] {
11   def equals(self: List[T], other: List[T]): Boolean =
12     (self, other) match {
13       case (Nil(), Nil()) => true
14       case (Cons(x,xs),Cons(y,ys)) => ev0.equals(x,y) && this.equals(xs,ys)
15       case _ => false
16     }
17   ...
18 }
19 val list = Cons(123, Nil())
20 ListasEquals[i32](i32asEquals).equals(list, Nil())
```

Heap Allocation

With mutable cells and memory safety, heap allocation is for free:

- Achieved with Boxes in Rust, otherwise data is stack-allocated.
- Distinction not needed in Scala because all objects are on the heap.
- Translation erases boxes because it wraps the variables in mutable cells anyway.

Aliasing Limitations

```
1 let mut a = 123;  
2 let mut b = 456;  
3 let mut x = &mut a;  
4  
5 x = &mut b;
```

```
1 val a: MutCell[Int] = MutCell[Int](123)  
2 val b: MutCell[Int] = MutCell[Int](456)  
3 val x: MutCell[MutCell[Int]] =  
4   MutCell[MutCell[Int]](a)  
5 x.value = b // Illegal aliasing
```