Leveraging Rust Types for Modular Specification and Verification

Vytautas Astrauskas,
Peter Müller, Federico Poli,
Alexander J. Summers

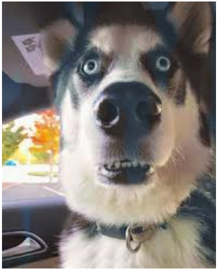
ETH Zurich

September 12, 2019

Plan

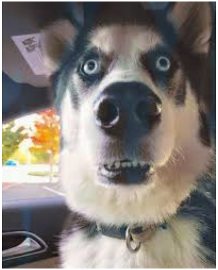
What is verification like, anyway?

Verification $\neq$ Absence of bugs



What is verification like, anyway?

Verification $\neq$ Absence of bugs



$$\text{Program} \xrightarrow{\textit{satisfies}} \text{Property}$$

or, in verification speak:

$$\text{Implementation} \xrightarrow{\textit{satisfies}} \text{Specification}$$

What is verification like, anyway?

Properties

☺

- no null pointer dereferences;
- no integer / buffer overflows;
- no data races;
- some property $P(\text{output}, \text{input})$ holds
  → *functional specifications*

What is verification like, anyway?

Properties

☺

- no null pointer dereferences;
- no integer / buffer overflows;
- no data races;
- some property $P(\text{output}, \text{input})$ holds
  → *functional specifications*

☹

- does what I want;
- no bugs;
- no vulnerabilities

A binary search function

```
method Binfind(a: seq<int>, v: int) returns (rv: int)
{
  rv := -1;
  var l, r := 0, |a|;
  if |a| = 0 { return; }
  while l + 1 < r {
    var mid := (l + r) / 2;
    if      v = a[mid]  { rv := mid; return; }
    else if v < a[mid]   { r := mid; }
    else                 { l := mid; }
  }
  if a[l] = v { rv := l; return; }
  rv := -1; return;
}
```

A binary search function cont.

```
method Binfind(a: seq<int>, v: int) returns (rv: int)
  requires // PRECONDITION:  sorted
    ∀ i, j :: 0 ≤ i ≤ j < |a| → a[i] ≤ a[j]
  ensures  // POSTCONDITION: not found
    rv = -1 → ∀ j :: 0 ≤ j < |a| → a[j] ≠ v
  ensures  // POSTCONDITION: found
    rv ≥ 0  → rv < |a| ∧ a[rv] = v
{
  rv := -1;
  var l, r := 0, |a|;
  if |a| = 0 { return; }
  while l + 1 < r {
    var mid := (l + r) / 2;
    if      v = a[mid]  { rv := mid; return; }
    else if v < a[mid]   { r := mid; }
    else                 { l := mid; }
  }
  if a[l] = v { rv := l; return; }
  rv := -1; return;
}
```

A binary search function cont.

```
method Binfind(a: seq<int>, v: int) returns (rv: int)
  // requires and ensures ...
{
  rv := -1;
  var l, r := 0, |a|;
  if |a| = 0 { return; }
  while l + 1 < r
    // LOOP INVARIANTS
    invariant 0 ≤ l < r ≤ |a|
    invariant ∀ i :: 0 ≤ i < l → a[i] < v
    invariant ∀ i :: r ≤ i < |a| → v < a[i]
  {
    var mid := (l + r) / 2;
    if      v = a[mid]  { rv := mid; return; }
    else if v < a[mid]   { r := mid; }
    else                 { l := mid; }
  }
  if a[l] = v { rv := l; return; }
  rv := -1; return;
}
```

A binary search function cont.

```
method Binfind(a: seq<int>, v: int) returns (rv: int)
  // requires and ensures ...
{
  rv := -1;
  var l, r := 0, |a|;
  if |a| = 0 { return; }
  while l + 1 < r
    // loop invariants
  {
    var mid := (l + r) / 2;
    if      v = a[mid]  { rv := mid; return; }
    else if v < a[mid]   { r := mid; }
    else                 { l := mid; }
  }
  if a[l] = v { rv := l; return; }
  assert l + 1 = r  // ASSERTION
  rv := -1; return;
}
```

The language

Elements

- Basic expressions / statements
- Preconditions, postconditions
- Loop invariants
- Assertions
- Propositions : FOF $(\wedge, \vee, \neg, \forall, \exists)$

Characteristics

- Imperative.
- No pointers. No dynamic memory allocation.

Bad for systems programming.

Verification Mechanism

Encode the conditions into a logical formula. Prove the formula is always true.

```
method NonzeroSquare(a: int) returns (rv: int)
  requires a ≠ 0
  ensures  rv > 0
{

  rv := a * a;

}
```

Verification Mechanism cont.

Automatically synthesis Hoare triples $\{P\}\ s\ \{Q\}$.
P and Q: propositions. s: statement.

```
method NonzeroSquare(a: int) returns (rv: int)
  requires a ≠ 0
  ensures  rv > 0
{
  // { a ≠ 0 ∧ a * a = a * a }
  rv := a * a;
  // { a ≠ 0 ∧ rv = a * a }
}
```

Hoare triple of assignment: $\{Q[x \mapsto e]\}\ x := e\ \{Q\}$

Verification Mechanism cont.

Automatically synthesis Hoare triples $\{P\}\ s\ \{Q\}$.
P and Q: propositions. s: statement.

```
method NonzeroSquare(a: int) returns (rv: int)
  requires a ≠ 0
  ensures  rv > 0
{
  // { a ≠ 0 ∧ a * a = a * a }
  rv := a * a;
  // { a ≠ 0 ∧ rv = a * a }
}
```

Hoare triple of assignment: $\{Q[x \mapsto e]\}\ x := e\ \{Q\}$
Prove the validity of

$$(a \neq 0 \land rv = a \times a) \to (rv > 0)$$

Verification Mechanism cont.

Prove the validity of

$$(a \neq 0 \land rv = a \times a) \rightarrow (rv > 0)$$

- Automatic: SMT solvers → z3
- Semi-automatic: theorem provers → coq
- Manual: 

Pointers aliasing is bad for verification

Problem with Hoare logic

No pointers. No heaps i.e. dynamically allocated memory.

We could extend Hoare logic with pointers in some way ...

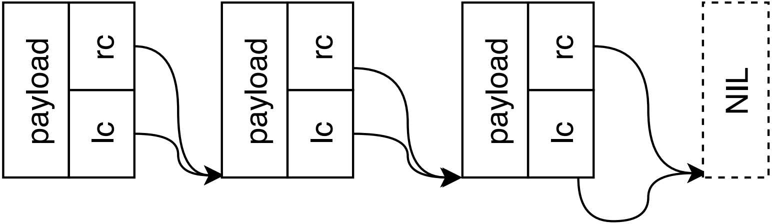
A binary tree example

```
method FreeBinTree(a: BinTree*) returns ()
{
  if a = NIL { return; }
  FreeBinTree(a->lc);
  FreeBinTree(a->rc);
  free(a);
}
```

A binary tree example

```
method FreeBinTree(a: BinTree*) returns ()
{
  if a = NIL { return; }
  FreeBinTree(a->lc);
  FreeBinTree(a->rc);
  free(a);
}
```

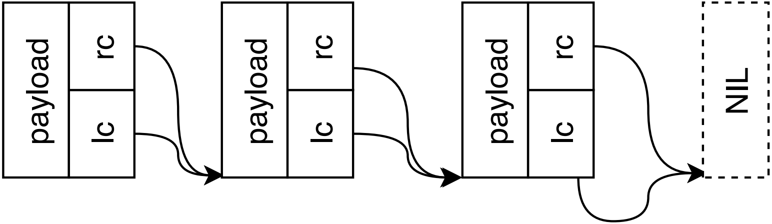
Seems ok. But what if ...



A binary tree example

```
method FreeBinTree(a: BinTree*) returns ()
{
  if a = NIL { return; }
  FreeBinTree(a->lc);
  FreeBinTree(a->rc);
  free(a);
}
```

Seems ok. But what if ...



→ Double free!

A binary tree example cont.

Add a precondition

```
method FreeBinTree(a: BinTree*) returns ()
  requires IsBinTree(a)
{
  if a = NIL { return; }
  FreeBinTree(a->lc);
  FreeBinTree(a->rc);
  free(a);
}
```

Naively, ∧ (logical and)

```
predicate IsBinTree(a: BinTree*)
  { a = NIL ∨ (IsBinTree(a->lc) ∧ IsBinTree(a->rc)) }
```

A binary tree example cont.

Add a precondition

```
method FreeBinTree(a: BinTree*) returns ()
  requires IsBinTree(a)
{
  if a = NIL { return; }
  FreeBinTree(a->lc);
  FreeBinTree(a->rc);
  free(a);
}
```

Separation logic: $*$ (and, separately)

```
predicate IsBinTree(a: BinTree*)
  { a = NIL ∨ (IsBinTree(a->lc) ∗ IsBinTree(a->rc)) }
```

`a->lc` and `a->rc` must not alias.

Separation Logic

Key insight

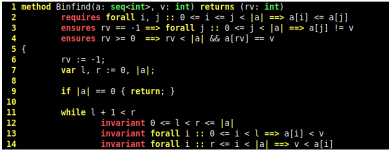
Aliasing needs special attention!

→ Rust's exclusive ownership

Plan

Status

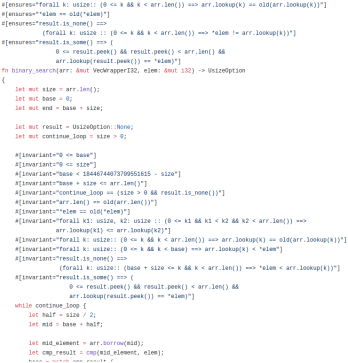
Still a prototype. ☹



```
method Binfind(a: seq<int>, v: int) returns (rv: int)
        requires forall i, j :: 0 <= i <= j < |a| ==> a[i] <= a[j]
        ensures rv == -1 ==> forall j :: 0 <= j < |a| ==> a[j] != v
        ensures rv >= 0  ==> rv < |a| && a[rv] == v
{
        rv := -1;
        var l, r := 0, |a|;

        if |a| == 0 { return; }

        while l + 1 < r
                invariant 0 <= l < r <= |a|
                invariant forall i :: 0 <= i < l ==> a[i] < v
                invariant forall i :: r <= i < |a| ==> v < a[i]
```



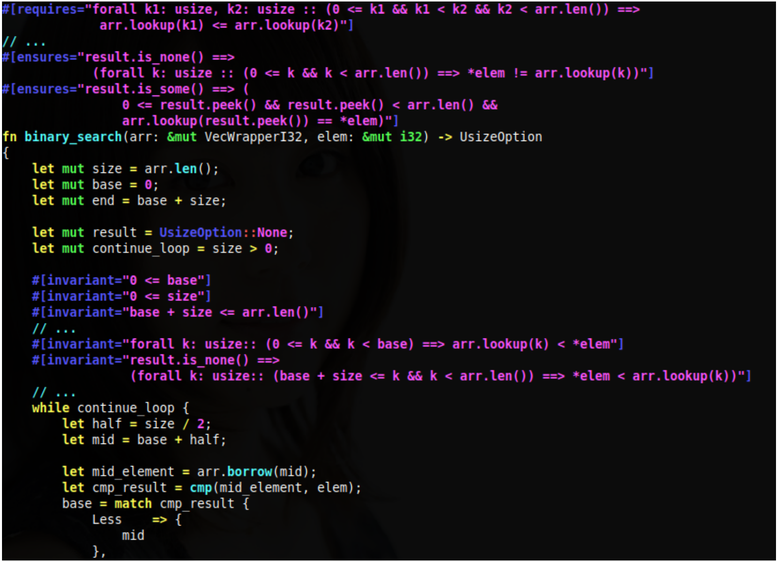
```
#[ensures="forall k: usize:: (0 <= k && k < arr.len()) ==> arr.lookup(k) == old(arr.lookup(k))"]
#[ensures="*elem == old(*elem)"]
#[ensures="result.is_none() ==>
            (forall k: usize :: (0 <= k && k < arr.len()) ==> *elem != arr.lookup(k))"]
#[ensures="result.is_some() ==> (
                0 <= result.peek() && result.peek() < arr.len() &&
                arr.lookup(result.peek()) == *elem)"]
fn binary_search(arr: &mut VecWrapperI32, elem: &mut i32) -> UsizeOption
{
    let mut size = arr.len();
    let mut base = 0;
    let mut end = base + size;

    let mut result = UsizeOption::None;
    let mut continue_loop = size > 0;

    #[invariant="0 <= base"]
    #[invariant="0 <= size"]
    #[invariant="base < 18446744073709551615 - size"]
    #[invariant="base + size <= arr.len()"]
    #[invariant="continue_loop == (size > 0 && result.is_none())"]
    #[invariant="arr.len() == old(arr.len())"]
    #[invariant="*elem == old(*elem)"]
    #[invariant="forall k1: usize, k2: usize :: (0 <= k1 && k1 < k2 && k2 < arr.len()) ==>
                arr.lookup(k1) <= arr.lookup(k2)"]
    #[invariant="forall k: usize:: (0 <= k && k < arr.len()) ==> arr.lookup(k) == old(arr.lookup(k))"]
    #[invariant="forall k: usize:: (0 <= k && k < base) ==> arr.lookup(k) < *elem"]
    #[invariant="result.is_none() ==>
                 (forall k: usize:: (base + size <= k && k < arr.len()) ==> *elem < arr.lookup(k))"]
    #[invariant="result.is_some() ==> (
                    0 <= result.peek() && result.peek() < arr.len() &&
                    arr.lookup(result.peek()) == *elem)"]
    while continue_loop {
        let half = size / 2;
        let mid = base + half;

        let mid_element = arr.borrow(mid);
        let cmp_result = cmp(mid_element, elem);
        base = match cmp_result {
```

Closer look



```
#[requires="forall k1: usize, k2: usize :: (0 <= k1 && k1 < k2 && k2 < arr.len()) ==>
             arr.lookup(k1) <= arr.lookup(k2)"]
// ...
#[ensures="result.is_none() ==>
            (forall k: usize :: (0 <= k && k < arr.len()) ==> *elem != arr.lookup(k))"]
#[ensures="result.is_some() ==> (
                0 <= result.peek() && result.peek() < arr.len() &&
                arr.lookup(result.peek()) == *elem)"]
fn binary_search(arr: &mut VecWrapperI32, elem: &mut i32) -> UsizeOption
{
    let mut size = arr.len();
    let mut base = 0;
    let mut end = base + size;

    let mut result = UsizeOption::None;
    let mut continue_loop = size > 0;

    #[invariant="0 <= base"]
    #[invariant="0 <= size"]
    #[invariant="base + size <= arr.len()"]
    // ...
    #[invariant="forall k: usize:: (0 <= k && k < base) ==> arr.lookup(k) < *elem"]
    #[invariant="result.is_none() ==>
                 (forall k: usize:: (base + size <= k && k < arr.len()) ==> *elem < arr.lookup(k))"]
    // ...
    while continue_loop {
        let half = size / 2;
        let mid = base + half;

        let mid_element = arr.borrow(mid);
        let cmp_result = cmp(mid_element, elem);
        base = match cmp_result {
            Less    => {
                mid
            },
```

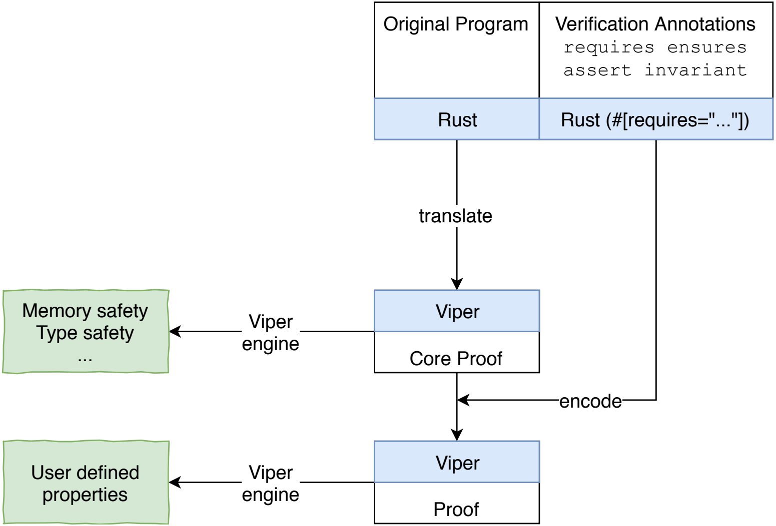
Research Question

Deductive verification in Rust.

Key Questions

- How automatic? → "fully automatic"
- Mechanism overview → program translation
- Infrastructures? → bottom-most: z3
- Why Rust? How about C? → controlled aliasing
- Dealing with libraries? → wrappers

Flow graph



Original Program
Verification Annotations
requires ensures
assert invariant
Rust
Rust (#[requires="..."])
translate
Viper
Core Proof
encode
Viper
Proof
Viper engine
Memory safety
Type safety
...
Viper engine
User defined properties

Infrastructure

Verification Language: Viper

- Implicit Dynamic Frames Logic $\rightarrow$ Separation Logic
- Pointers, heaps!
- Resource capability

The bottom-most

- Symbolic execution verifier
- z3

Why Rust?

Reason: what logic do we use for verification?

Why Rust?

Reason: what logic do we use for verification? Separation logic:

Why Rust?

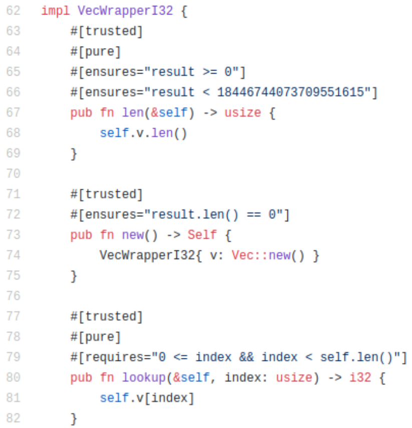
Reason: what logic do we use for verification? Separation logic:aliasing!

```
struct Node {
    val: i32,
    l: Box<Tree>,
    rc: Box<Tree>
}
```

`Node.lc` and `Node.rc` must not alias.

Libraries

- `#[trusted]`
- wrappers



```
impl VecWrapperI32 {
    #[trusted]
    #[pure]
    #[ensures="result >= 0"]
    #[ensures="result < 18446744073709551615"]
    pub fn len(&self) -> usize {
        self.v.len()
    }

    #[trusted]
    #[ensures="result.len() == 0"]
    pub fn new() -> Self {
        VecWrapperI32{ v: Vec::new() }
    }

    #[trusted]
    #[pure]
    #[requires="0 <= index && index < self.len()"]
    pub fn lookup(&self, index: usize) -> i32 {
        self.v[index]
    }
```

Evaluation questions

Core proof

- 500 popular crates
- automatic generation
- verification time: 90% < 2 sec

Evaluation questions

Core proof

- 500 popular crates
- automatic generation
- verification time: 90% < 2 sec

Overflow & panic freedom

- filtered, 519 functions
- automatic assertion generation
- error in 467 functions

Evaluation questions

Core proof

- 500 popular crates
- automatic generation
- verification time: 90% < 2 sec

Overflow & panic freedom

- filtered, 519 functions
- automatic assertion generation
- error in 467 functions

Functional specification

- Hand constructed data, 11 files
- e.g. binary search

Plan

Main takeaways

- Automatic verification: z3

Main takeaways

- Automatic verification: z3
- Throw work to infrastructure

Main takeaways

- Automatic verification: z3
- Throw work to infrastructure
- Aliasing with mutability is bad → Rust, SL