# Formal Verification in Rust

Carson Storm

University of Utah

November 13 2020

# Outline

# Why Rust?

- a low-level language that offers zero cost abstraction
- a multi-paradigm language
- guarantees memory safety at compile time

# Rust memory safety (moves)

```
fn first<S,T>((s,_):(S,T)) -> S {
    s
}
...

let p: (S,T) = ...;
let s = first(p);
// let t = second(p); // error[E0382]: use of moved value: `p`
```

# Rust memory safety (borrows I)

```
fn first<S,T>(&(ref s,_):&(S,T)) -> &S {
    s
}
...

let p: (S,T) = ...;
let s = first(&p);
let t = second(&p);
```

# Rust memory safety (borrows II)

```
fn first<S,T>(&(ref s,_):&(S,T)) -> &S {
    s
}
...

let p: (S,T) = ...;
let s = first(&p);     // immutable borrow occurs here
let t = second(&mut p); // mutable borrow occurs here

// immutable borrow ends here
// will result in compiler error
```

# Rust memory safety (lifetimes I)

```
fn first<S,T>(&(ref s,_):&(S,T)) -> &S {
    s
}
...
let s = {
    let p: (S,T) = ...;

    return first(&p);
    // p is freed here, but we are trying to return a reference to it
    // this will result in a compilation error
}
```

# Rust memory safety (lifetimes II)

```
fn first<'a, S,T>(&(ref s,_): &'a (S,T)) -> &'a S {
    s
}
...
let p: (S,T) + 'a = ...;
let s: &'a S = first(&p);
```

# Rust memory safety

- All references are guaranteed to point to valid memory
- Alias is allowed, but only for non mutable references

```
fn compute(input: &u32, output: &mut u32) {
    if *input > 10 { *output = 1; }
    if *input > 5 { *output *= 2;}
}
...
compute(&x, &mut x); // this will result in a compiler error
```

- Eliminates data races

# Rust vs. C

```c
void client(list * a, list *b) {
    int old_len = b->len;
    append(a, 100);
    assert(b->len == old_len);
}
```

- Could have memory errors in `b->len`.
- Could have aliasing (eg. if `a = b`).
- Could have data races if another thread mutates `a` or `b`.

# Rust vs. C

```
fn client(a: &mut List, b: &mut List) {
    let old_len = b.len();
    append(a, 100);
    assert!(b->len == old_len);
}
```

- Memory is always valid so `b.len()` will not result in a memory error.
- Mutable references cannot alias, so `a` and `b` are disjoint.
- Only one mutable reference is allowed at a time, so another thread cannot mutate `a` or `b`.

# Formal Verification in Rust with Lean

# What is Lean?

- Lean is a theorem prover and programming language developed at Microsoft Research
- Provides some automatic theorem proving, but focuses on the verification of theorem.
- Has been used to formalize abstract algebra, group theory, number theory, topology, computability, etc.
- Based on Dependent Type Theory

```
theorem halting_problem (n) : ¬ computable_pred (  c, (eval c
↪  n).dom)
| h := rice {f | (f n).dom} h nat.partrec.zero nat.partrec.none
↪  trivial
```

# Type Theory
## What is Type Theory?

- A logical system that can server as a replacement for set theory
- Judgments are made through type checking. A proof of a theorem is considered valid if it has the same type as the theorem.
- Does not depend on implementation details.

# Type Theory

## Constructive vs. Classical Logic

- Constructive logic must provide an example for proposition of the form

$$\exists x \in \mathbb{N}, x > 1$$

  where as, in classical logic, an example is not always known.
- Does not allow for statements such as $p \wedge \neg p$ or $\neg\neg p \implies p$.
  - This allows for type theory to handle undecidable problems
  - It's not always true that $p$ is either true of false.

# Lean Demo

# Functional Purification

Since mutability is localized in Rust we perform functional rewrites

```
p.x += 1
```

```
let p = Point {x = p.x + 1, ..p}
```

```
// where f(& mut Point) -> A
let x = f(&mut p)
```

```
// where f(Point) -> (A, Point)
let (x,p) = f(p)
```

# Specifications

```
theorem core.slice.binary_search.sem :  {T : Type} [Ord' T]
    (self : list T) (needle : T), sorted le self →
    option.any binary_search_res (binary_search self needle)
```

# Specifications

```
inductive binary_search_res : Result usize usize → Prop :=
-- if the value is found then Ok is returned, containing the index of
↪  the matching element
| found      : ∏i, nth self i = some needle → binary_search_res
↪  (Result.Ok i)
-- if the value is not found then Err is returned, containing the
↪  index where a matching
-- element could be inserted while maintaining sorted order.
| not_found : ∏i, needle ∉ self → sorted le (insert_at self i
↪  needle) →
    binary_search_res (Result.Err i)
```

# Translation

```
fn f(i: i32) -> i32 {
    if i == 42 {
        1
    }
    else {
        0
    }
}
```

```
definition test.f (iₐ : i32) : sem
↪   (i32) :=
let' «i$2» ← iₐ;
let' t4 ← «i$2»;
let' t3 ← t4 =ᵇ (42 : int);
if t3 = bool.tt then
let' ret ← (1 : int);
return (ret)
else
let' ret ← (1 : int);
return (ret)
```

# Translation

```
struct Point {x: i32, y: i32}

fn main() {
    let p = Point {x: 10, y: 11};
    let px: i32 = p.x;
}
```

```
structure test.Point := mk {} ::
(x : i32)
(y : i32)

definition test.main : sem (unit)
 ↪  :=
let' «p$1» ← test.Point.mk (10 :
 ↪  int) (11 : int);
let' t3 ← (test.Point.x «p$1»);
let' «px$2» ← t3;
let' ret ←  ;
return ( )
```

# Electrolysis Demo

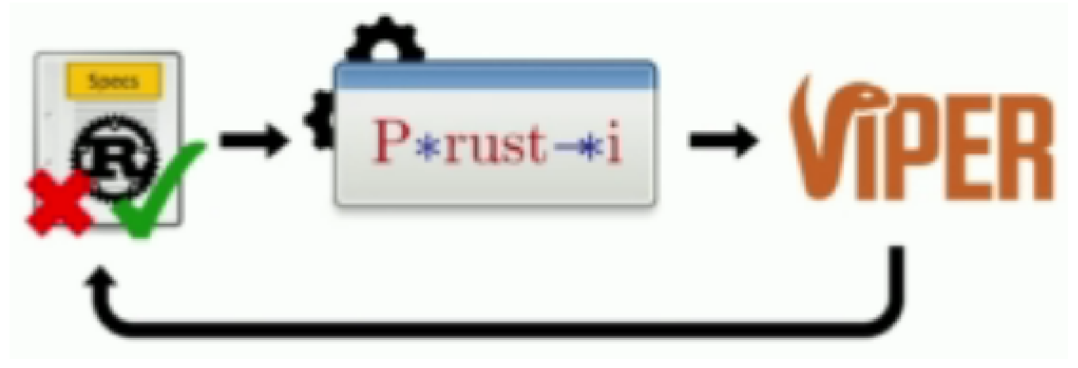
# Formal Verification in Rust with Prusti

# How is Prusti different?

- Relies on a SMT solver to verify program
- Embeds specification directly in source

```rust
#[ensures(result >= a && result >= b)]
#[ensures(result == a || result == b)]
fn max(a: i32, b: i32) -> i32 {
    if a < b {
        b
    } else {
        a
    }
}
```

# Prusti Design



# What is Viper?

- A language independent verification tool built on top of Z3
- Prusti transpiles the rust source to viper statements

# Viper Translation

```
struct List {
  val: i32;
  next: Option<Box<List>>
}
```

```
predicate List(self: Ref) {
    acc(self.val) *
    acc(self.next) *
    i32(self.val) *
    OptionBoxList(self.next)
}
```

```
fn client(a:&mut List, b:&mut List)
```

```
method client(a : Ref, b: Ref)
    requires List(a) * List(b)
    ensures List(a) * List(b)
```

# Prusti Demo

# Further Reading I

[1] Sebastian Ullrich, *A Formal Verification of Rust's Binary Search Implementation*, Blog. [Online]. Available: `https://kha.github.io/2016/07/22/formally-verifying-rusts-binary-search.html` (visited on 11/05/2020).

[2] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, 147:1–147:30, Oct. 2019. DOI: `10.1145/3360573`. [Online]. Available: `https://doi.org/10.1145/3360573` (visited on 11/05/2020).

[3] Sebastian Ullrich, "Simple Verification of RustPrograms via FunctionalPurification," English, Ph.D. dissertation, Karlsruhe Institute of Technology. [Online]. Available: `https://raw.githubusercontent.com/Kha/masters-thesis/master/main.pdf`.

# Further Reading II


[4] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The Lean Theorem Prover (System Description)," en, in *Automated Deduction - CADE-25*, A. P. Felty and A. Middeldorp, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2015, pp. 378–388, ISBN: 978-3-319-21401-6. DOI: `10.1007/978-3-319-21401-6_26`.

[5] T. Coquand and G. Huet, "The calculus of constructions," en, *Information and Computation*, vol. 76, no. 2, pp. 95–120, Feb. 1988, ISSN: 0890-5401. DOI: `10.1016/0890-5401(88)90005-3`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/0890540188900053` (visited on 11/06/2020).

[6] *Rust-Proof/rustproof*, original-date: 2016-05-03T02:23:56Z, Aug. 2020. [Online]. Available: `https://github.com/Rust-Proof/rustproof` (visited on 11/07/2020).

[7] S. Thompson, *Type theory and functional programming*. Addison Wesley, 1991.