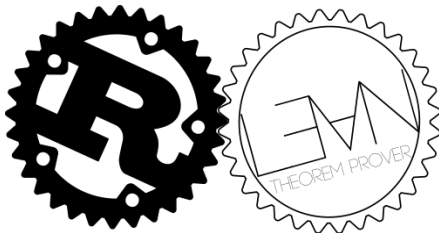


# Electrolysis

## Simple Verification of Rust Programs via Functional Purification



Sebastian Ullrich

Karlsruhe Institute of Technology, advisor Gregor Snelting  
Carnegie Mellon University, advisor Jeremy Avigad

OPLSS 2016

# Why Rust? (What is Rust?)

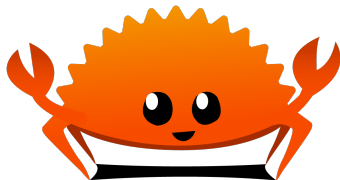
Rust is a new systems programming language sponsored by Mozilla Research

- ▶ multi-paradigm with an ML-like syntax
- ▶ pursues “the trifecta: safety, concurrency, and speed”
  - ▶ **speed** through zero-cost abstractions and manual memory management
  - ▶ **memory safety** through tracking reference lifetimes in the type system
  - ▶ **safe concurrency** through forbidding shared mutable references

# Why Rust? (What is Rust?)

Rust is a new systems programming language sponsored by Mozilla Research

- ▶ multi-paradigm with an ML-like syntax
- ▶ pursues “the trifecta: safety, concurrency, and speed”
  - ▶ **speed** through zero-cost abstractions and manual memory management
  - ▶ **memory safety** through tracking reference lifetimes in the type system
  - ▶ **safe concurrency** through forbidding shared mutable references



# Why Rust: Because It's Almost Pure Already

- ▶ turn destructive updates into functional ones

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

- ▶ references: save value instead of pointer, write back at end of lifetime

```
let x = f(&mut p); let (x, p) = f(p);
```

# Simple Verification via Purification

1. make Rust program purely functional
2. transpile it into expression language of a theorem prover (Lean)
3. prove correctness of the Lean definition

# Simple Verification via Purification

1. make Rust program purely functional
2. transpile it into expression language of a theorem prover (Lean)
  - ▶ run `rustc` up to CFG generation
  - ▶ sort definitions topologically by dependencies
  - ▶ extract loops from CFG and put them into loop combinator
  - ▶ resolve static/dynamic trait calls

Things Rust fortunately does not have:

- ▶ exceptions
  - ▶ subtyping
3. prove correctness of the Lean definition

# Verifying std::[T]::binary\_search: Input

```
fn binary_search_by<'a, F>(&'a self, mut f: F) -> Result<usize, usize>
where F: FnMut(&'a T) -> Ordering
{
    let mut base = 0usize;
    let mut s = self;

    loop {
        let (head, tail) = s.split_at(s.len() >> 1);
        if tail.is_empty() {
            return Err(base)
        }
        match f(&tail[0]) {
            Less => {
                base += head.len() + 1;
                s = &tail[1..];
            }
            Greater => s = head,
            Equal => return Ok(base + head.len()),
        }
    }
}

fn binary_search(&self, x: &T) -> Result<usize, usize> where T: Ord {
    self.binary_search_by(|p| p.cmp(x))
}
```

- ▶ high-level implementation working with subslices instead of explicit indexing
- ▶ transitively uses
  - 5 traits
  - 6 structs and enums
  - 7 functions

# Verifying std::[T]::binary\_search: Output

```
section
  parameters {F : Type1} {T : Type1}
  parameters [«ops.FnMut F (T)» : ops.FnMut F (T) (cmp.Ordering)]
  parameters (selfa : (slice T)) (fa : F)

  definition «[T] as core.slice.SliceExt».binary_search_by.loop_4 (state__ : F × usize × (slice T)) :
    ↪ sem (sum (F × usize × (slice T)) ((result.Result usize usize))) :=
    ...

  definition «[T] as core.slice.SliceExt».binary_search_by : sem ((result.Result usize usize)) :=
    let' self ← (selfa);
    let' f ← (fa);
    let' base ← ((0 : nat));
    let' t1 ← (self);
    let' s ← (t1);
    loop («[T] as core.slice.SliceExt».binary_search_by.loop_4) (f, base, s)
end

...

structure cmp.Ord [class] (Self : Type1) extends cmp.Eq Self, cmp.PartialOrd Self Self :=
  (cmp : Self → Self → sem ((cmp.Ordering)))

definition «[T] as core.slice.SliceExt».binary_search {T : Type1} [«cmp.Ord T» : cmp.Ord T] (selfa :
  ↪ (slice T)) (xa : T) : sem ((result.Result usize usize)) :=
  let' self ← (selfa);
  let' x ← (xa);
  let' t0 ← (self);
  let' t2 ← (x);
  let' t1 ← ((λ upvarsa pa, let' p ← (pa);
  let' t0 ← (p);
  let' t1 ← ((upvarsa)));
  dostep «$tmp» ← @cmp.Ord.cmp _ «cmp.Ord T» (t0) (t1);
  let' ret ← «$tmp»;
  return ret) (t2));
  dostep «$tmp» ← @«[T] as core.slice.SliceExt».binary_search_by _ _ fn (t0) (t1);
  let' ret ← «$tmp»;
  return (ret)
```



# Verifying `std::[T]::binary_search`: Proof

```
/- fn binary_search(&self, x: &T) -> Result<usize, usize> where T: Ord
```

```
Binary search a sorted slice for a given element.
```

```
If the value is found then Ok is returned, containing the index of the matching element;  
if the value is not found then Err is returned, containing the index where a matching element could  
be inserted while maintaining sorted order.-/
```

```
inductive binary_search_res : Result usize usize → Prop :=  
| found      : ∀ i, nth self i = some needle → binary_search_res (Result.Ok i)  
| not_found  : ∀ i, needle ∉ self → sorted le (insert_at self i needle) →  
  binary_search_res (Result.Err i)
```

```
...
```

```
theorem binary_search.spec :  
  ∃ f ∈  $\mathcal{O}(\lambda p, \log_2 p. 1 * p.2)$  [at  $\infty \times \infty$ ],  
  ∀ (self : slice T) (needle : T), sorted le self → sem.terminates_within  
    (binary_search_res self needle)  
    (f (length self, Ord'.cmp_max_cost needle self))  
    (binary_search self needle) :=
```

```
...
```

# Conclusion and Future Work

- ▶ a tool for verifying real-world Rust code
- ▶ correctness proof of a central stdlib algorithm
- ▶ next step: find a new algorithm to verify!
- ▶ possible enhancement: different monad stacks for e.g. complexity analysis, global side effects, ...
- ▶ maybe allow some restricted forms of unsafe code

[github.com/Kha/electrolysis](https://github.com/Kha/electrolysis)