Electrolysis

Verifying 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?)

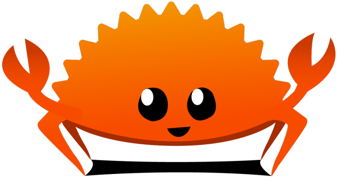
Rust is a new systems programming language sponsored by Mozilla Research

- 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, ..p };
```

- 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<F>(&self, mut f: F) -> Result<usize, usize> where
    F: FnMut(&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 indicing
- transitively uses
  - 5 traits
  - 6 structs and enums
  - 7 functions

Verifying std::[T]::binary_search: Output

```
section
  parameters {T : Type} {F : Type}
  parameters [ops_FnMut__T__F : ops.FnMut (T) F (cmp.Ordering)]
  parameters (self : (slice T)) (f : F)

  definition slice._T_.slice_SliceExt.binary_search_by.loop_4 state__ :=
  match state__ with (f, base, s) :=
  ...

  definition slice._T_.slice_SliceExt.binary_search_by :=
  let self ← self;
  let f ← f;
  let base ← (0 : nat);
  let t1 ← self;
  let s ← t1;
  loop' (slice._T_.slice_SliceExt.binary_search_by.loop_4) (f, base, s)
end

...

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

definition slice._T_.slice_SliceExt.binary_search {T : Type} [cmp_Ord_T : cmp.Ord T] (self : (slice T)) (x : T) :=
...
```

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

```
parameter {T : Type}
parameter [Ord' T]
parameter self : slice T
parameter needle : T

hypothesis Hsorted : sorted le self

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

section loop_4
  variable s : slice T
  variable base : usize

  structure loop_4_invar :=
  (s_in_self  : s ⊑ₚ dropn base self)
  (insert_pos : sorted.insert_pos self needle ∈ '[base, base + length s])
  (needle_mem : needle ∈ self → needle ∈ s)

  inductive loop_4_step : loop_4.state → Prop :=
  mk : Πbase' s', loop_4_invar s' base' → length s' < length s → loop_4_step (f, base', s')

  ...
end

...

theorem binary_search.sem : option.any binary_search_res (binary_search self needle) :=
begin
  rewrite [↑binary_search, bind_some_eq_id, funext (λx, bind_some_eq_id)],
  apply binary_search_by.sem,
end
```

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`