Karlsruher Institut für Technologie

Simple Verification of Rust Programs via Functional Purification

Sebastian Ullrich

Lehrstuhl Programmierparadigmen, IPD Snelting

Goals

Karlsruher Institut für Technologie

A general tool for formally verifying Rust programs

- via a *shallow* embedding into the theorem prover Lean
  - map Rust's semantics onto Lean's instead of explicitly formalizing them
- without being fundamentally more complex than verifying *Lean* programs
  - no Separation Logic etc.
- *no modifications or annotations* of the source necessary
- *extendable* via a shallow *monadic* embedding
  - so far: Maybe monad for partiality, Writer monad on nats for asymptotic function runtime

Why Rust? (What is Rust, anyway?)

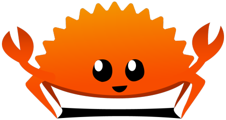
Karlsruher Institut für Technologie

Rust is a modern language for systems programming

manual memory management …but (type-)safe

functional abstractions …but zero-cost, where possible

package manager C interoperability



Rust: memory safety through static typing

```rust
fn index<T>(self: &[T], index: usize) -> &T
```

```rust
{
  let v = vec![1, 2, 3];
  let p = index(&v, 1);
  ...
}
```

Rust: memory safety through static typing

```
fn index<T>(self: &[T], index: usize) -> &T
```

```
fn index<'a, T>(self: &'a [T], index: usize) -> &'a T
```

⇒ static tracking of inter-procedural lifetime relations inside the type system

```
{
  let v = vec![1, 2, 3];
  let p = index(&v, 1);
  ...
}
```

Rust: memory safety through static typing

```
fn index<T>(self: &[T], index: usize) -> &T
```

```
fn index<'a, T>(self: &'a [T], index: usize) -> &'a T
```

⇒ static tracking of inter-procedural lifetime relations inside the type system

```
{
  let mut v = vec![1, 2, 3];
  let p = index(&v, 1);
  v.clear();
  *p // ???
}
```

Rust: memory safety through static typing

```
error[E0502]: cannot borrow `v` as mutable because it is also
 ↪  borrowed as immutable
|
|   let p = index(&v, 1);
|                      - immutable borrow occurs here
|   v.clear();
|   ^ mutable borrow occurs here
|   *p
| }
| - immutable borrow ends here
```

Aliasing XOR mutability

Karlsruher Institut für Technologie

Rust *has to* prevent mutable aliasing to guarantee safety!

Some nice benefits from the absence of aliasing

- no data races
- no iterator invalidation
- and also...

“Dealing with aliasing is one of the key challenges for the verification of imperative programs” [1]

[1] Dietl, W. & Müller, P. (2013). Object ownership in program verification.

Why Rust?

Karlsruher Institut für Technologie

no aliasing[2]
⇒ mutability always locally scoped
⇒ can be reduced to immutability...?

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

p may not be aliased, so the update can only be observed via p

[2] in safe[3] Rust

[3] in the safe part that doesn't use the unsafe part to reintroduce (dynamically checked) aliasing

Why Rust?

no aliasing[2]
⇒ mutability always locally scoped
⇒ can be reduced to immutability...?

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

⇒

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

p may not be aliased, so the update can only be observed via p

---

[2] in safe[3] Rust

[3] in the safe part that doesn't use the unsafe part to reintroduce (dynamically checked) aliasing

Simple Verification via Functional Purification

1. reduce Rust definition to purely functional code
2. generate Lean definition as shallow monadic embedding
3. prove the Lean definition correct

Simple Verification via Functional Purification

1. reduce Rust definition to purely functional code
2. generate Lean definition as shallow monadic embedding
   - create a dependency graph and output definitions in a topological ordering
   - obtain a control flow graph (MIR) of each definition
   - extract control flow SCCs and put them in a Lean loop combinator
   - replace definitions that could not be translated automatically (unsafe code, primitives)
3. prove the Lean definition correct

Simple Verification via Functional Purification

1. reduce Rust definition to purely functional code
2. generate Lean definition as shallow monadic embedding
   - create a dependency graph and output definitions in a topological ordering
   - obtain a control flow graph (MIR) of each definition
   - extract control flow SCCs and put them in a Lean loop combinator
   - replace definitions that could not be translated automatically (unsafe code, primitives)
3. prove the Lean definition correct

In practice, steps 1 and 2 are implemented as a single transformation by a Rust program interfacing with the Rust compiler.

Translation of references

```rust
fn index<'a, T>(self: &'a [T], index: usize) -> &'a T
```

```
definition index {T : Type} (self : list T) (index : nat) : sem T
```

Because of the absence of aliasing, passing by immutable reference is semantically equivalent with passing by value. `sem` is the semantics monad.

Translation of references

```
fn index<'a, T>(self: &'a [T], index: usize) -> &'a T
```

```
definition index {T : Type} (self : list T) (index : nat) : sem T
```

Because of the absence of aliasing, passing by immutable reference is semantically equivalent with passing by value. `sem` is the semantics monad.

A mutable input reference can be translated to an input and an output parameter.

```
fn index_mut<'a, T>(self: &mut 'a [T], index: usize) -> &mut 'a T
```

```
definition index_mut {T : Type} (self : list T) (index : nat) :
  sem (??? × list T)
```

Translation of references

We translate mutable output references via *lenses*, also known as *functional references*.

```
fn index_mut<'a, T>(self: &mut 'a [T], index: usize) -> &mut 'a T
```

```
structure lens (Outer Inner : Type) :=
(get : Outer → sem Inner)
(set : Outer → Inner → sem Outer)

...

definition index_mut {T : Type} (self : list T) (index : nat) :
  sem (lens (list T) T × list T)
```

Verifying [T]::binary_search

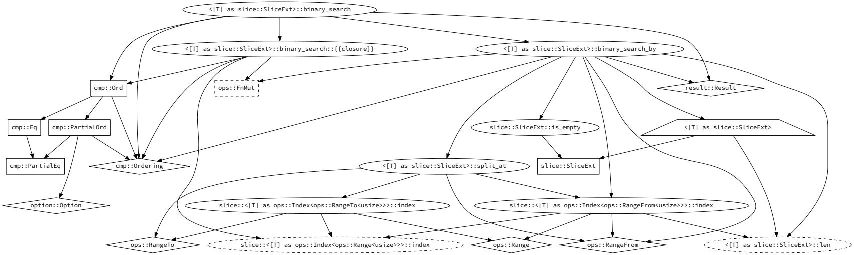
Not exactly low-level...

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

    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()),
            }
        }
    }
}
```

Verifying [T]::binary_search



Turned out to be a great first test case: a non-trivial algorithm perusing a good chunk of the language and quite a few dependencies

Verifying [T]::binary_search

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

```
definition binary_search {T : Type} [Ord T] (self : list T) (x : T)
  : sem (Result nat nat)
```

`[Ord T]` will be inferred by *typeclass inference*.
Bounded integral types are translated to unbounded ones with overflow-checking operators.

Correctness proof

```
parameter {T : Type}
parameter self : list T
parameter x : T
...

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

```
theorem binary_search.spec : sorted self → is_slice self →
  sem.terminates_with
    binary_search_res
    (binary_search self x) := ...
```

`is_slice` checks that a list is bounded by the memory size, which is a sufficient premise to prove the absence of any integer overflows.

Proof of asymptotic upper bound

```
definition sem (A : Type) := option (A × ℕ)
```

```
theorem binary_search.spec :
  ∃ f ∈ 𝒪(λ p, log₂ p.1 * p.2) [at ∞ × ∞],
  ∀ (self : list T) (x : T), is_slice self → sorted self →
    sem.terminates_with_in
      (binary_search_res self x)
      (f (length self, Ord'.cmp_max_cost x self))
      (binary_search self x) := ...
```

"The runtime of `binary_search` is bounded logarithmically by the size of the slice and linearly by the maximum comparison cost."

Evaluation

2556 lines of Rust

3199 lines of Lean

953	misc lemmas
838	verification
287	loop combinator
194	asymptotic analysis
192	semantic monad
...	

Evaluation: language reference coverage

kha.github.io/electrolysis

- Notation, Lexical structure, Syntax extensions ✔
- 5 Crates and source files ✔
- 6 Items And Attributes
  - 6.1 Items
    - 6.1.2 Modules ✔
    - 6.1.3 Functions
      - 6.1.3.1 Generic functions ✔
      - 6.1.3.2 Diverging functions ✔
      - Returning mutable reference to first argument ✔
      - Returning arbitrary mutable references ✗
    - 6.1.4 Type aliases ✔
    - 6.1.5 Structs ✔
    - 6.1.6 Enumerations ✔
      - Struct-like enum variants ✔
      - Enum discriminants ✔
    - 6.1.7 Constant items ✔
    - 6.1.8 Static items ✔
    - 6.1.9 Traits
      - Generic traits and trait methods ✔
      - Default methods ✔
      - Calling default methods from inside the trait ✗
      - Overriding default methods ✗
      - Trait bounds ✔
      - Associated types ✔
      - Trait objects ✗
      - Static trait methods ✔
    - 6.1.10 Implementations ✔
      - Trait implementations ✔
      - That other type of implementations ✔
  - 6.3 Attributes ✔
    - 6.3.8 Conditional compilation ✔
- 7 Statements and expressions
  - 7.1 Statements ✔

Evaluation: coverage of `core`

#definitions	**outcome** (reason)
6731	**succeeds and type checks**
2761	**succeeds, but some failed dependencies**
2649	**translation failed**
713	overriding default method
388	&mut nested in type
360	variadic function signature
280	float
243	raw pointer
209	cast from function pointer to usize
173	unimplemented intrinsic function
45	error from rustc API during translation
40	unimplemented rvalue kind
...	

Conclusion

- The first general tool for verifying safe Rust code
- successfully tested on real-world code
- including asymptotic runtime analysis
- already supports most language features

`github.com/Kha/electrolysis`

Specification of Ord

```
definition ordering {T : Type} [decidable_linear_order T] (x y : T)
  ↪  :
  cmp.Ordering :=
if x < y then Ordering.Less
else if x = y then Ordering.Equal
else Ordering.Greater

structure Ord' [class] (T) extends Ord T, decidable_linear_order T
  ↪  :=
(cmp_eq : ∀ x y : T, Σ k, Ord.cmp x y = some (ordering x y, k))
```

Loop Combinator

```
...

definition terminating (s : State) :=
∃ Hwf : well_founded R, loop.fix s ≠ mzero

noncomputable definition loop (s : State) : sem Res :=
if Hex : ∃ R, terminating R s then
  @loop.fix (classical.some Hex) _ (classical.some
↪  (classical.some_spec Hex)) s
```

Loop Combinator

```
theorem loop.terminates_with_in_ub
  {In State Res : Type}
  (body : In → State → sem (State ⊕ Res))
  (pre : In → State → Prop)
  (p : In → State → State → Prop)
  (q : In → State → Res → Prop)
  (citer aiter : ℕ → ℕ)
  (miter : State → ℕ)
  (cbody abody : ℕ → ℕ)
  (mbody : In → State → ℕ)
  (citer_aiter : citer ∈ 𝒪(aiter) [at ∞] ∩ Ω(1) [at ∞])
  (cbody_abody : cbody ∈ 𝒪(abody) [at ∞] ∩ Ω(1) [at ∞])
  (pre_p : ∀ args s, pre args s → p args s s)
  (step : ∀ args init s, pre args init → p args init s →
    sem.terminates_with_in (λ x, match x with
      | inl s' := p args init s' ∧ citer (miter s') < citer (miter s)
      | inr r  := q args init r
      end) (cbody (mbody args init)) (body args s)) :
  ∃ f ∈ 𝒪(λ p, aiter p.1 * abody p.2) [at ∞ × ∞], ∀ args s, pre
↪  args s →
    sem.terminates_with_in (q args s) (f (miter s, mbody args s))
      (loop (body args) s) := ...
```