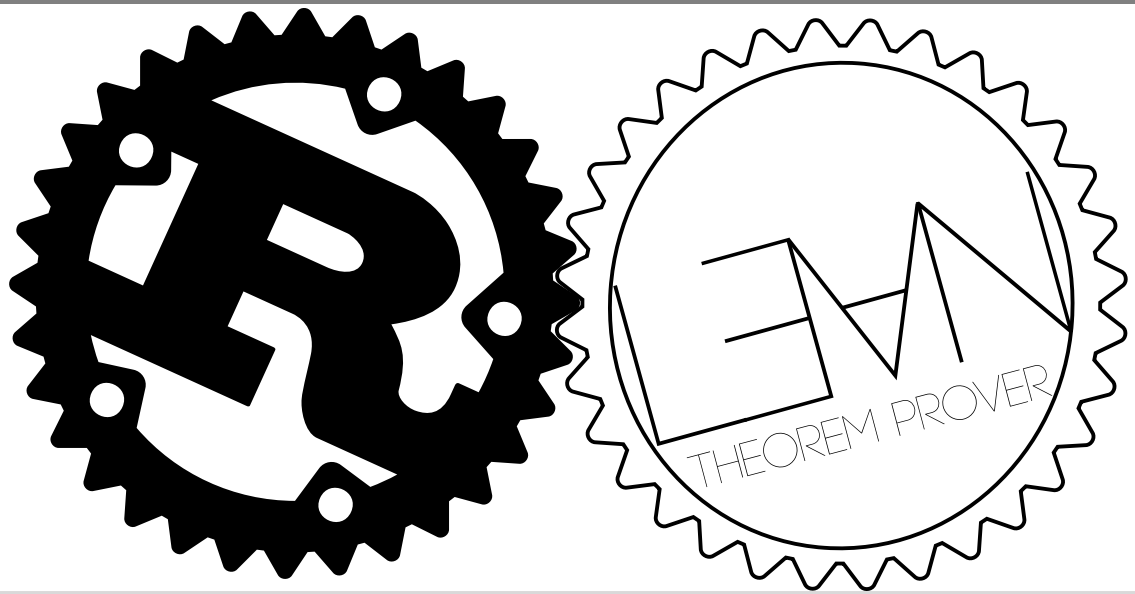
Karlsruher Institut für Technologie

Simple Verification of Rust Programs via Functional Purification

Sebastian Ullrich

Lehrstuhl Programmierparadigmen, IPD Snelting



Zielsetzung

Ein allgemeines Werkzeug zur Verifikation von Rust-Programmen

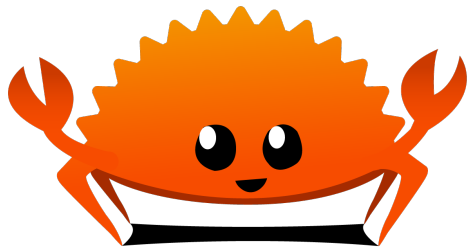
- durch *flache* Einbettung in Lean
- nicht wesentlich komplexer als von *Lean*-Programmen
  - keine Separation Logic o.Ä.
- *ohne* Notwendigkeit von Modifikationen oder Annotationen
- *erweiterbar* durch *monadische* Einbettung
  - bisher: Nichttermination, Funktionslaufzeit

Warum Rust? (Was ist Rust?)

Karlsruher Institut für Technologie

Rust ist eine moderne Sprache für Systemprogrammierung

Manuelles Speichermanagement	...aber (typ-)sicher
Funktionale Abstraktionen	...aber möglichst kostenlos
Paketmanager	C-Interoperabilität



Rust: Speichersicherheit durch Typisierung

Karlsruher Institut für Technologie

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

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

Rust: Speichersicherheit durch Typisierung

Karlsruher Institut für Technologie

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

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

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

Rust: Speichersicherheit durch Typisierung

Karlsruher Institut für Technologie

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

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

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

Rust: Speichersicherheit durch Typisierung

```
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 *muss* für Typsicherheit veränderbares Aliasing verbieten.

Schöne Nebenwirkungen:

- Data Races unmöglich
- Iterator Invalidation unmöglich
- außerdem...

"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.

Warum Rust?

Kein Aliasing
⇒ Veränderbarkeit lokal beschränkt
⇒ zurückführbar auf Unveränderbarkeit

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

Warum Rust?

Kein Aliasing
⇒ Veränderbarkeit lokal beschränkt
⇒ zurückführbar auf Unveränderbarkeit

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

⇒

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

Simple Verification via Functional Purification

Karlsruher Institut für Technologie

1. Führe Rust-Definition auf pur funktionalen Code zurück
2. Generiere Lean-Definition als flache monadische Einbettung
3. Beweise Korrektheit der Lean-Definition

Simple Verification via Functional Purification

Karlsruher Institut für Technologie

1. Führe Rust-Definition auf pur funktionalen Code zurück
2. Generiere Lean-Definition als flache monadische Einbettung
   - Führe den Rust-Compiler bis CFG-Generierung aus
   - Sortiere Definitionen topologisch nach Abhängigkeitsgraph
   - Extrahiere SCCs und wende Schleifenkombinator an
   - Ersetze nicht automatisch übersetzbare Definitionen
3. Beweise Korrektheit der Lean-Definition

Übersetzung von Referenzen

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

```
definition index {T : Type₁} (self : list T) (index : nat) : sem T
```

Übersetzung von Referenzen

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

```
definition index {T : Type₁} (self : list T) (index : nat) : sem T
```

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

```
definition index_mut {T : Type₁} (self : list T) (index : nat) :
  sem (??? × list T)
```

Übersetzung von Referenzen

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

```
structure lens (Outer Inner : Type₁) :=
(get : Outer → sem Inner)
(set : Outer → Inner → sem Outer)

definition index_mut {T : Type₁} (self : list T) (index : nat) :
  sem (lens (list T) T × list T)
```

Verifikation von `[T]::binary_search`

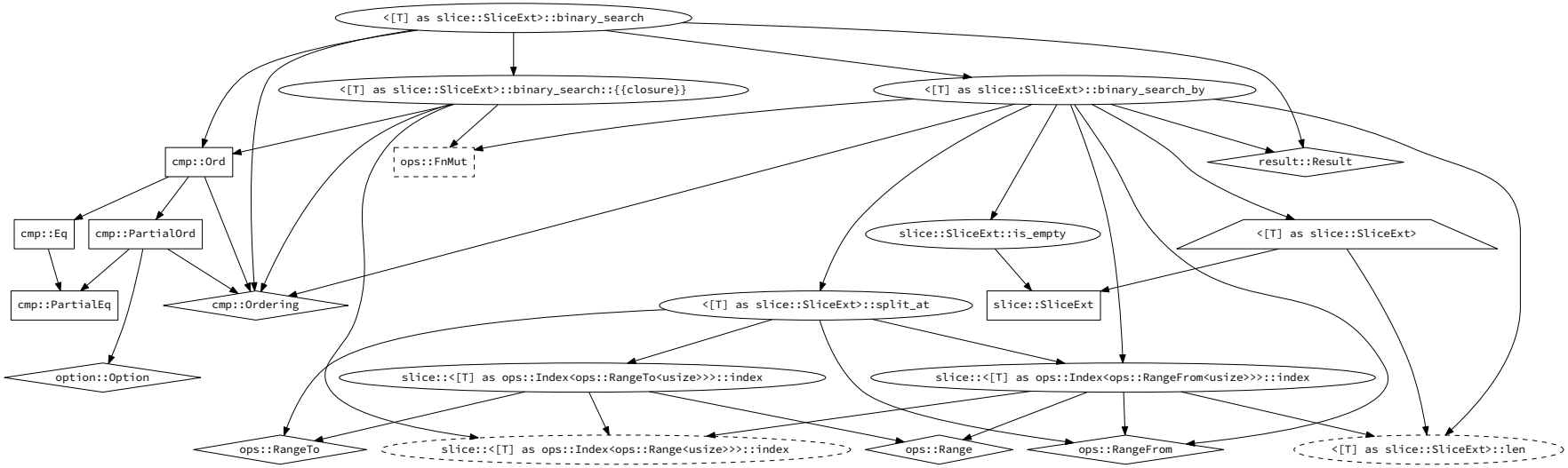
Karlsruher Institut für Technologie

```rust
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()),
            }
        }
    }
}
```

Verifikation von `[T]::binary_search`



Verifikation von `[T]::binary_search`

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

```
definition binary_search {T : Type₁} [Ord T] (self : list T) (x : T)
  : sem (Result nat nat)
```

Verifikation von `[T]::binary_search`

```
parameter {T : Type₁}
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) := ...
```

Verifikation von `[T]::binary_search`

```
definition sem (A : Type₁) := 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 le self →
    sem.terminates_with_in
      (binary_search_res self x)
      (f (length self, Ord'.cmp_max_cost x self))
      (binary_search self x) := ...
```

Evaluation

2556 Zeilen Rust

3199 Zeilen Lean

953	Sonstige Lemmata
838	Verifikation
287	Schleifenkombinator
194	Asymptotische Analyse
192	Semantikmonade
...	

Evaluation: Abdeckung der Sprachreferenz

Karlsruher Institut für Technologie

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
  - 71 Statements ✔

Evaluation: Abdeckung von `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
...	

Zusammenfassung

- Das erste allgemeine Werkzeug zur Verifikation von safe Rust
- erfolgreich angewendet auf realen Rust-Code
- inklusive asymptotischer Laufzeitanalyse
- Breite Unterstützung der Rust-Sprache

`github.com/Kha/electrolysis`

Ord-Spezifikation

```
definition ordering {T : Type₁} [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))
```

Schleifenkombinator

```
...

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
```

Schleifenkombinator

```
theorem loop.terminates_with_in_ub
  {In State Res : Type₁}
  (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) := ...
```