Leveraging Rust Types for Program Synthesis

Jonáš Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, Ilya Sergey

21.06.23

ETH zürich

TECHNION Israel Institute of Technology

UC San Diego

NUS National University of Singapore

Program Synthesis for Rust with Guarantees

Rust type

+

functional spec

RusSOL

code

- well-typed
- correct

This Talk

1. A taste of RusSOL
2. Synthetic Ownership Logic (SOL)
3. Evaluation

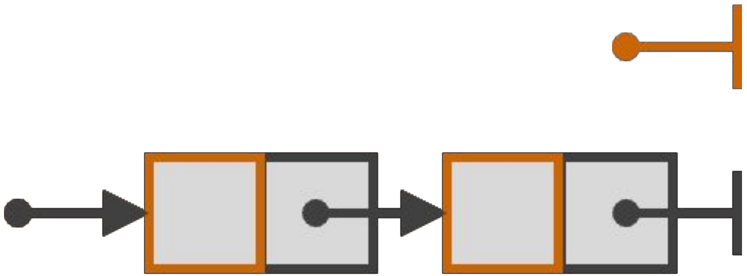
Demo

sum type

```
enum List<T> {
    Nil,
    Cons { elem: T, next: Box<List<T>> },
}
```

generic payload

pointer with ownership

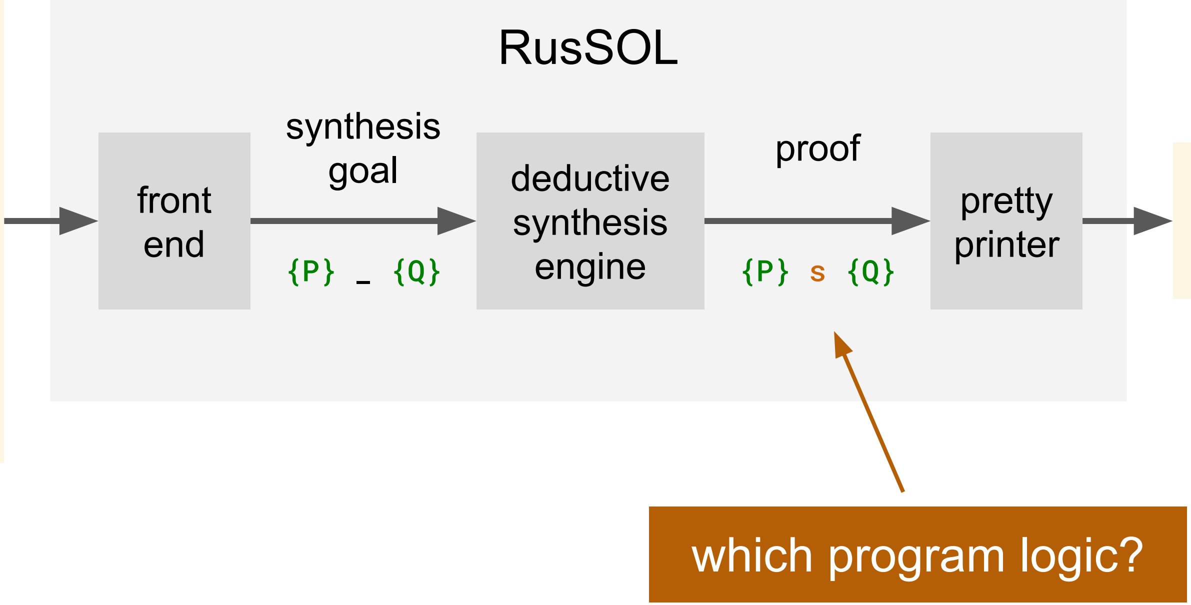


RusSOL Workflow

```
struct UserType
{ … }

#[requires(…)]
#[ensures(…)]
fn target(x: T1,…)
  -> T

#[pure]
fn f(x: T1,…) -> T
{ … }
```




```
fn target(x: T1,…)
  -> T
{ s' }
```

This Talk

1. A taste of RusSOL
2. Synthetic Ownership Logic (SOL)
3. Evaluation

Key Properties

ownership & borrowing

Aeneas [Ho & Protzenko’22]

Synthetic Ownership Logic (SOL)

program synthesis

SuSLik [Polikarpova & Sergey’19]

```
#[requires(…)]
#[ensures(…)]
fn target(x: T_1,…)
  -> T

#[pure]
fn f(x: T_1,…) -> T
{ … }
```

functional properties of safe Rust

Prusti [Astrauskas et al’19]

Creusot [Denis et al’21]

Example: Constr Rule

$$\frac{\textsc{Constr.Cons}}{\left\{\begin{array}{c} \mathsf{e}: \mathsf{T} * \\ \mathsf{n}: \mathsf{Box} \end{array}\right\} \ \textbf{let}\ \mathsf{x} = \mathsf{List::Cons}\ \{\ \mathsf{e}, \mathsf{n}\ \}\ \{\mathsf{x}: \mathsf{List}\}}$$

Example: Singleton

```
fn singleton<T>(elem: T) -> List<T> {
  // {elem: T}



  todo!();



  // {result: List}
  result
}
```

CONSTR.CONS

$$\frac{}{\left\{\begin{array}{l} \mathtt{e}\colon \mathtt{T} * \\ \mathtt{n}\colon \mathtt{Box} \end{array}\right\} \ \textbf{let}\ x = \mathtt{List::Cons}\ \{\ e, n\ \}\ \{\mathtt{x}\colon \mathtt{List}\}}$$

Example: Singleton

```
fn singleton<T>(elem: T) -> List<T> {
  // {elem: T}



  todo!();



  // {elem: T * next: Box}
  let result = List::Cons{elem, next}; // Constr.Cons
  // {result: List}
  result
}
```

CONSTR.CONS

$$\frac{}{\left\{\begin{array}{l} e\colon T \ast \\ n\colon Box \end{array}\right\} \ \textbf{let}\ x = List::Cons\,\{\, e, n \,\}\ \{x\colon List\}}$$

Example: Singleton

```
fn singleton<T>(elem: T) -> List<T> {
  // {elem: T}



  todo!();



  // {elem: T * next: Box}
  let result = List::Cons{elem, next}; // Constr.Cons
  // {result: List}
  result
}
```

CONSTR.BOX

$$\frac{}{\{l : \texttt{List}\}\ \textbf{let}\ x = \texttt{Box::new}(l)\ \{x : \texttt{Box}\}}$$

Example: Singleton

```
fn singleton<T>(elem: T) -> List<T> {
  // {elem: T}

  todo!();

  // {elem: T * list: List}
  let next = Box::new(list);            // Constr.Box
  // {elem: T * next: Box}
  let result = List::Cons{elem, next}; // Constr.Cons
  // {result: List}
  result
}
```

Constr.Box

$$\frac{}{\{\texttt{l}: \texttt{List}\}\ \textbf{let}\ \texttt{x} = \texttt{Box::new(l)}\ \{\texttt{x}: \texttt{Box}\}}$$

Example: Singleton

```
fn singleton<T>(elem: T) -> List<T> {
  // {elem: T}

  todo!();

  // {elem: T * list: List}
  let next = Box::new(list);              // Constr.Box
  // {elem: T * next: Box}
  let result = List::Cons{elem, next}; // Constr.Cons
  // {result: List}
  result
}
```

Constr.Nil

$$\frac{}{\{\mathrm{emp}\}\ \textbf{let}\ x = \texttt{List::Nil}\ \{x : \texttt{List}\}}$$

Example: Singleton

```
fn singleton<T>(elem: T) -> List<T> {
  // {elem: T}
  todo!();
  // {elem: T}
  let list = List::Nil;                 // Constr.Nil
  // {elem: T * list: List}
  let next = Box::new(list);            // Constr.Box
  // {elem: T * next: Box}
  let result = List::Cons{elem, next}; // Constr.Cons
  // {result: List}
  result
}
```

CONSTR.NIL

$$\frac{}{\{emp\}\ \textbf{let}\ x = \texttt{List::Nil}\ \{x : \texttt{List}\}}$$

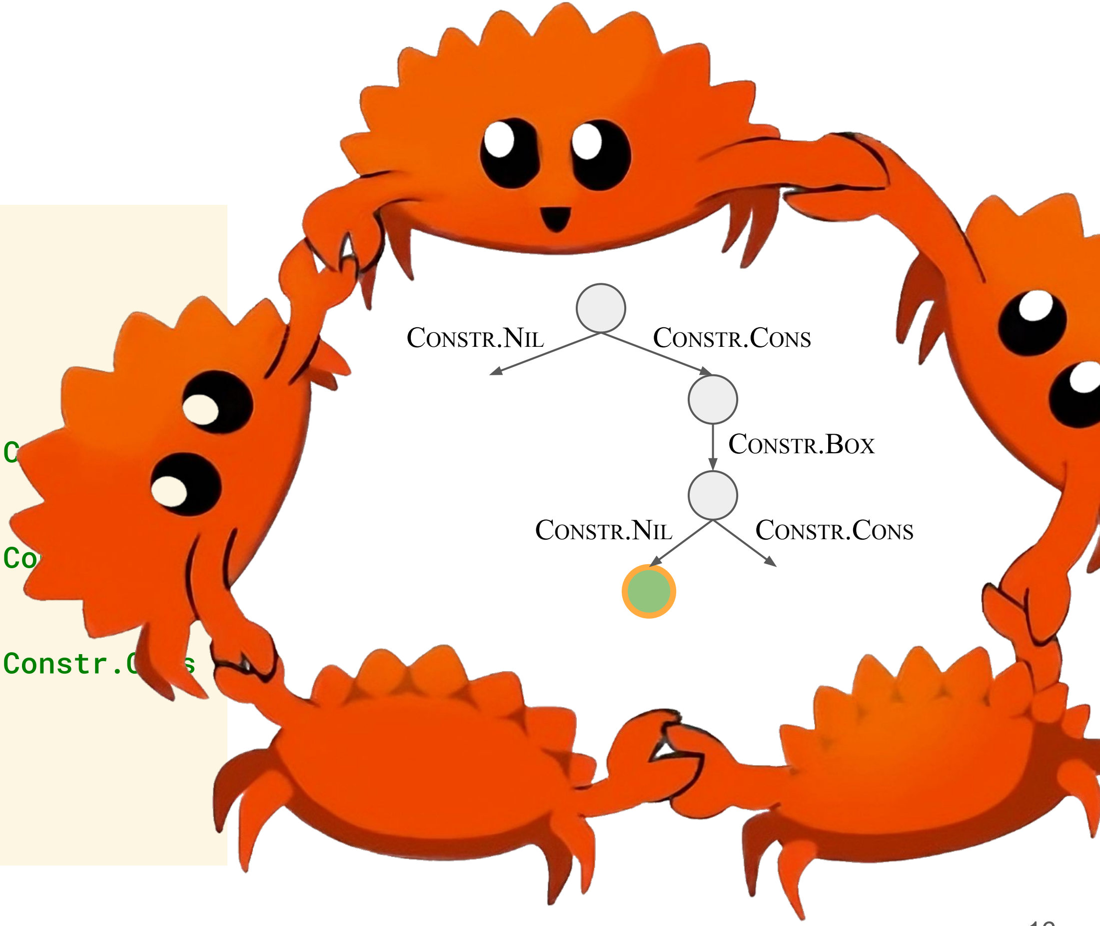
Example: Singleton

```
fn singleton<T>(elem: T) -> List<T> {
  // {elem: T}
  todo!();
  // {elem: T}
  let list = List::Nil;                     // Constr.Nil
  // {elem: T * list: List}
  let next = Box::new(list);                // Constr.Box
  // {elem: T * next: Box}
  let result = List::Cons{elem, next}; // Constr.Cons
  // {result: List}
  result
}
```

Example: Singleton

```
fn singleton<T>(elem: T) -> List<T> {
  // {elem: T}

  // {elem: T}
  let list = List::Nil;                 // C
  // {elem: T * list: List}
  let next = Box::new(list);            // Co
  // {elem: T * next: Box}
  let result = List::Cons{elem, next}; // Constr.C
  // {result: List}
  result
}
```



This Talk

1. A taste of RusSOL
2. Synthetic Ownership Logic (SOL)
3. Evaluation

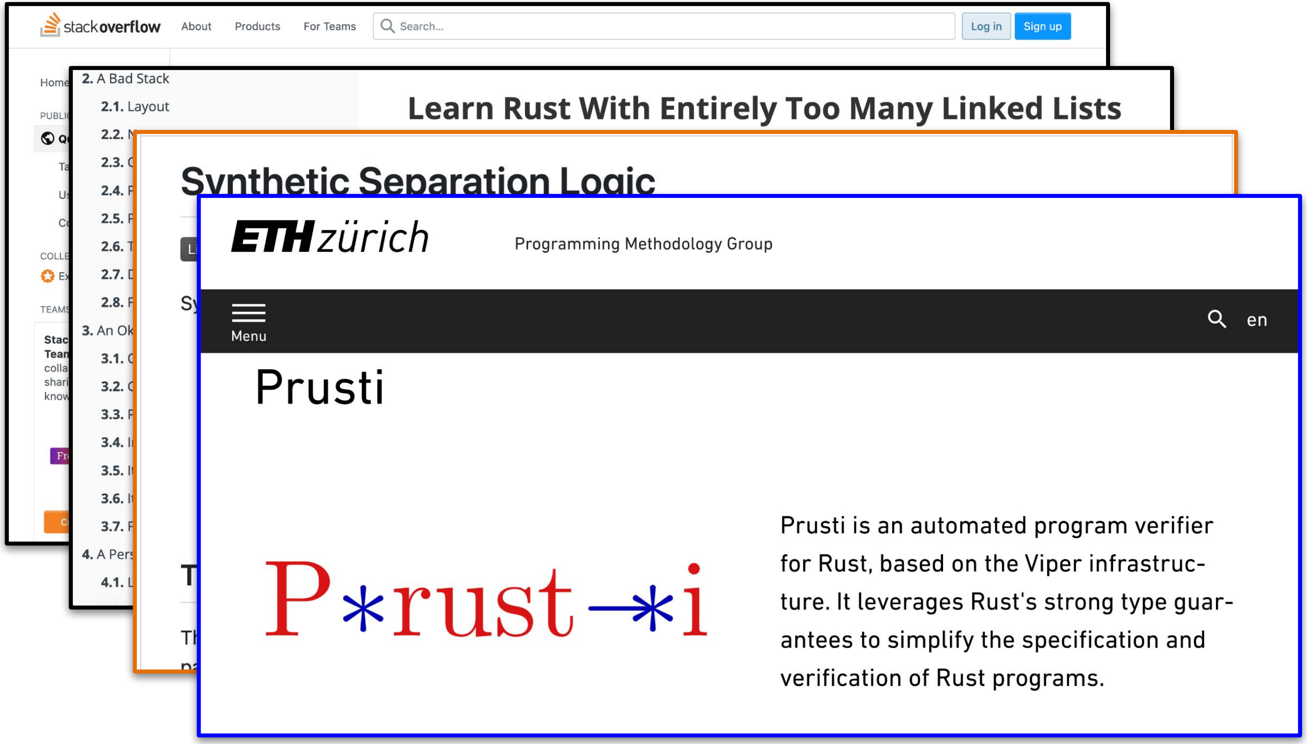
Annotated

3 sources

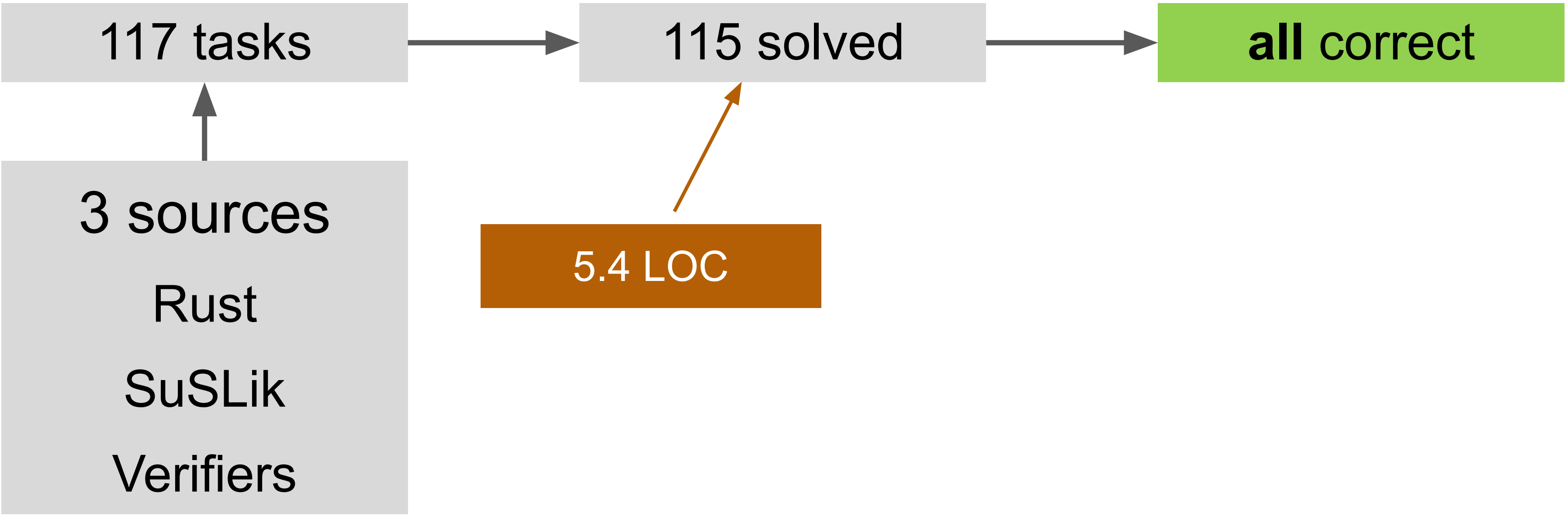
Rust

SuSLik

Verifiers



Annotated

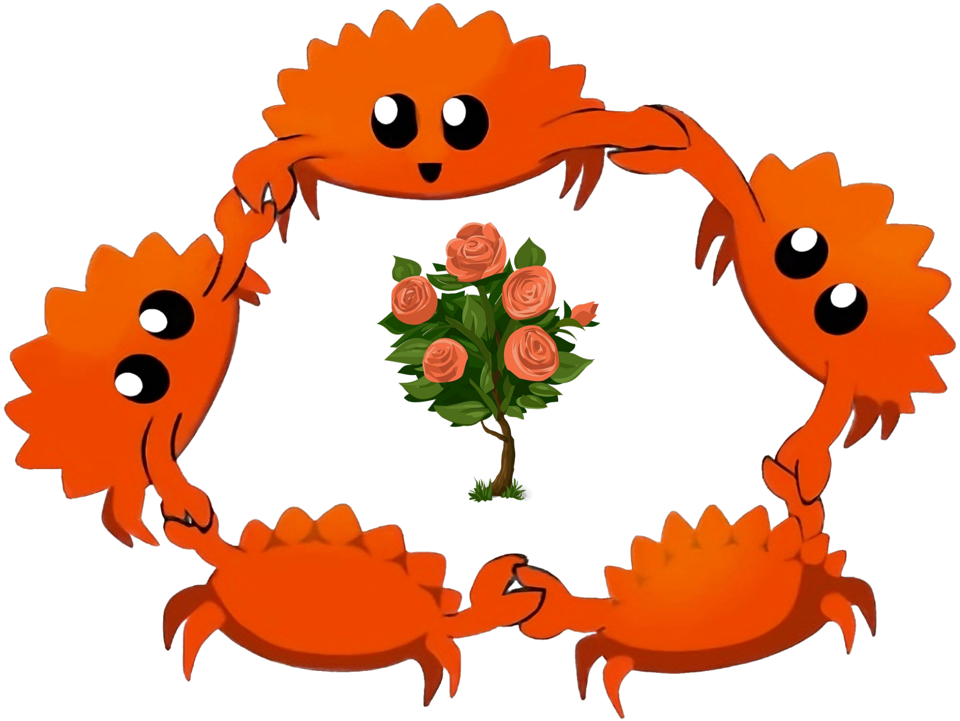

117 tasks
115 solved
all correct
3 sources
Rust
SuSLik
Verifiers
5.4 LOC

Annotated

```
enum Tree<T> { Nil, Cons { elem: T, next: List<Tree<T>> } }
enum List<T> { Nil, Cons(Box<(T, List<T>)>) }

#[ensures(result.len() == tree.size())]
fn flatten<T>(tree: Tree<T>) -> List<T> {
  match tree {
    Tree::Nil => List::Nil,
    Tree::Cons { elem, next } => flatten_3(elem, next),
  }
}
fn flatten_3<T>(elem: T, next: List<Tree<T>>) -> List<T> {
  match next {
    List::Nil => {
      let bx = (elem, List::Nil);
      let _0 = Box::new(bx);
      List::Cons(_0)
    }
    List::Cons(_0) => {
      let result = flatten(_0.0);
      flatten_14(elem, _0.1, result)
    }
  }
}
fn flatten_14<T>(elem: T, _1: List<Tree<T>>, result: List<T>) -> List<T> {
  match result {
    List::Nil => flatten_3(elem, _1),
    List::Cons(_0) => {
      let result = flatten_14(_0.0, _1, _0.1);
      let bx = (elem, result);
      let _0 = Box::new(bx);
      List::Cons(_0)
    }
  }
}
```

2 aux fns	24 LOC	10s



Unannotated

no spec

100 most downloaded packages (crates)

→

~3K LOC

all solutions well-typed

filter unsupported fns

```rust
// either/src/lib.rs:832
fn factor_first<T, L, R>
  (x: Either<(T, L), (T, R)>)
    -> (T, Either<L, R>) {
  match x {
    Either::Left(_0) => {
      let _1 = Either::Left(_0.1);
      (_0.0, _1)
    }
    Either::Right(_0) => {
      let _1 = Either::Right(_0.1);
      (_0.0, _1)
    }
  }
}
```

Conclusion

Online demo
t.ly/2-8A
Tools -> RusSOL

RusSOL

- first deductive synthesis tool for Rust
- leverages Rust types to reduce spec overhead

Synthetic Ownership Logic (SOL)

- verifies well-typed programs which satisfy spec
- leverages Rust types to prune search space
- friendly to synthesis

```rust
fn push<T>(x: &mut List<T>, elem: T) {
  let list: List<T>      =
                    std::mem::replace(x, List::Nil);
  let next: Box<List<T>> = Box::new(list);
  let extended: List<T>  = List::Cons{elem, next};
  *x = extended
}
```

&mut V → V
impossible in safe Rust

```rust
#[ensures(result == *idol)]
fn std::mem::replace<V>(idol: &mut V, bag: V) -> V {
  unsafe
}
```