Leveraging Rust Types for Program Synthesis

JONÁŠ FIALA, Department of Computer Science, ETH Zurich, Switzerland
SHACHAR ITZHAKY, Technion, Israel
PETER MÜLLER, Department of Computer Science, ETH Zurich, Switzerland
NADIA POLIKARPOVA, University of California, San Diego, USA
ILYA SERGEY, National University of Singapore, Singapore

The Rust type system guarantees memory safety and data-race freedom. However, to satisfy Rust's type rules, many familiar implementation patterns must be adapted substantially. These necessary adaptations complicate programming and might hinder language adoption. In this paper, we demonstrate that, in contrast to manual programming, automatic synthesis is *not* complicated by Rust's type system, but rather benefits in two major ways. First, a Rust synthesizer can get away with significantly simpler *specifications*. While in more traditional imperative languages, synthesizers often require lengthy annotations in a complex logic to describe the shape of data structures, aliasing, and potential side effects, in Rust, all this information can be inferred from the types, letting the user focus on specifying functional properties using a slight extension of Rust expressions. Second, the Rust type system reduces the *search space* for synthesis, which improves performance.

In this work, we present the first approach to automatically synthesizing correct-by-construction programs in safe Rust. The key ingredient of our synthesis procedure is Synthetic Ownership Logic, a new program logic for deriving programs that are guaranteed to satisfy both a user-provided functional specification and, importantly, Rust's intricate type system. We implement this logic in a new tool called RuSSOL. Our evaluation shows the effectiveness of RuSSOL, both in terms of annotation burden and performance, in synthesizing provably correct solutions to common problems faced by new Rust developers.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: program synthesis, program logic, Rust, type systems

ACM Reference Format:
Jonáš Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. 2023. Leveraging Rust Types for Program Synthesis. *Proc. ACM Program. Lang.* 7, PLDI, Article 164 (June 2023), 24 pages. https://doi.org/10.1145/3591278

1 INTRODUCTION

Rust is a systems programming language designed for writing low-level code that combines strong correctness guarantees with the runtime performance of traditional "unsafe" systems languages, such as C/C++ (Matsakis and Klock II 2014; Rust Team 2017). A distinct feature of Rust is its *ownership* type system, which ensures memory safety and data-race freedom, and enables the compiler to manage memory automatically, without requiring a garbage collector.

The type system of Rust, which is so effective at preventing bugs statically, is also what makes Rust code hard to write for developers who are new to the language. As an example, consider the

Authors' addresses: Jonáš Fiala, Department of Computer Science, ETH Zurich, Switzerland, jonas.fiala@inf.ethz.ch; Shachar Itzhaky, Technion, Israel, shachari@cs.technion.ac.il; Peter Müller, Department of Computer Science, ETH Zurich, Switzerland, peter.mueller@inf.ethz.ch; Nadia Polikarpova, University of California, San Diego, USA, nadia.polikarpova@ucsd.edu; Ilya Sergey, National University of Singapore, Singapore, ilya@nus.edu.sg.

2475-1421/2023/6-ART164
https://doi.org/10.1145/3591278

code in Fig. 1, adopted from StackOverflow.[1] Function `borrow` attempts to return a *reference* to the contents of the `data` component of a value of type `Bar`. If `data` contains some value, a reference to it is wrapped in the constructor `Ok` of the `Result` type,[2] otherwise, the `Err` result is returned.

```
1  struct Bar<T> { data: Option<T> }
2  impl<T> Bar<T> {
3    fn borrow(&self) -> Result<&T, ()> {
4      match self.data {
5        Some(d) => Ok(&d),
6        None => Err(()),
7      }
8    }
9  }
```

Fig. 1. An ill-typed Rust example

This code fragment is rejected by the Rust type system because it violates a fundamental invariant of safe Rust: in each state, a memory location is either mutable or shared (aliased), but never both. This invariant prevents data-races by excluding concurrent write accesses and it allows safe de-allocation once the owner of a location goes out of scope. In our example, the invariant is violated after pattern-matching on the value of the `self.data` field and assigning its content to `d` in the first branch. At this point, the memory location is mutable through `d` and potentially shared (because additional immutable references may exist, for instance, in `borrow`'s caller). To fix this problem, the top answer on StackOverflow suggests changing line 4 to match a read-only reference `&self.data`, such that its value is immutably shared between `&d` and any other references that might exist.

Program Synthesis for Rust. Since navigating the intricacies of the Rust type system can be difficult for programmers, it is appealing to apply program synthesis to automatically generate a well-typed implementation of a function from its signature. Of course, in the case of `borrow`, synthesizing *a* well-typed body is trivial—by always returning `Err()`. In order to capture the programmer's intent more precisely, type signatures can be annotated with boolean expressions that describe the desired behavior of the function. For instance, the following specification for the `borrow` function annotates a type signature with a *postcondition*, which states that `borrow`'s result should be of the form `Ok` if and only if `self.data` is of the form `Some`:

```
#[ensures(self.data.is_some() == result.is_ok())]
fn borrow(&self) -> Result<&T, ()> { todo!() }
```

This specification is sufficient to *automatically synthesize* the correct implementation of `borrow` suggested on StackOverflow. Three key observations make a successful search for such a program possible: (a) the return type informs that the body should make use of constructors of the `Result` type; (b) the postcondition ensures that the result will not be a vacuous `Err`; (c) most interestingly, thanks to the Rust's strong type system, *the only way* to obtain a value of type `&T` is by pattern matching on `&self.data`. In particular, the implementation of `borrow` cannot construct a new memory location and return a reference to it: this is outlawed in Rust, since the new location would be de-allocated at the end of the function, leaving the caller with a dangling reference. Therefore, our postcondition needs to provide only a very partial specification of the intended behavior, saying nothing about *which* `&T` reference to return inside the `Ok` result. Nevertheless, together with the type signature and Rust's type rules, it is sufficient to synthesize the intended implementation.

We present a novel technique for synthesizing programs in *safe* Rust that satisfy a type signature and a set of functional annotations, expressed as (mostly) plain boolean Rust expressions. Our key insight is that the type rules of safe Rust restrict the synthesis search space, which both improves performance and allows us to capture the programmer's intent with only partial annotations.

Approach and Challenges. In this work, we follow the *deductive* approach to program synthesis, which is based on formal logical specifications, and in which the search for a program is phrased

[1] https://stackoverflow.com/q/22282117

[2] The type `&T` denotes an immutable reference to a value of type `T`, and `&self` is a shorthand for `self: &Bar<T>`.

as a *proof search* for judgments in a certain logic (Manna and Waldinger 1980). Given a logical specification that is sufficiently precise, deductive synthesis produces programs that are *correct by construction*, and can be independently validated by third-party type checkers and verifiers.

Given the close connection between Rust types and Separation Logic (SL) developed in the context of program verification (Astrauskas et al. 2019), a promising direction for synthesizing Rust programs could be as simple as (1) translating Rust types and functional specifications into SL, and (2) using an SL-based deductive synthesizer such as SuSLik (Polikarpova and Sergey 2019) to synthesize programs from these SL specs. Unfortunately, this simple approach does not work: even though every program generated by SuSLik is provably memory-safe, not every such program satisfies the type rules of safe Rust. For example, SuSLik programs may use aliased mutable references (*e.g.*, for swapping pointer values), reinterpret a value of one data type as another (*e.g.*, take an integer as a pointer and vice versa), and pass around pointers to allocated but uninitialized memory—none of which is allowed in safe Rust. Moreover, since SuSLik is not aware of Rust's type rules, it cannot leverage them to restrict the search space. For instance, the postcondition in the example above is insufficient to ensure that the synthesized implementation returns a reference to the value of `self.data`; SuSLik requires a more comprehensive specification to synthesize the intended implementation, which increases the overhead for the user.

For deductive synthesis to generate well-typed and correct Rust code, its underlying logic must (1) capture the rules of safe Rust and (2) allow one to reason about the functional behavior of Rust programs. Unfortunately, none of the existing program logics and semantics for Rust satisfy both of these requirements. In particular, the logics used in the Rust verifiers Prusti (Astrauskas et al. 2019) and Creusot (Denis et al. 2021) can reason about functional correctness, but they assume that the program already type-checks, and hence do not formalize all the aspects of the type system in the logic. For example, Prusti's logic does not prevent two local variables from storing mutable references to the same heap location; hence, if used for synthesis, this logic might generate a program that is rejected by the type checker. On the other hand, Aeneas (Ho and Protzenko 2022) and Oxide (Weiss et al. 2019) capture the type rules of safe Rust, but do not allow one to reason about functional correctness within the logic. Finally, RustBelt (Jung et al. 2018) and RustHornBelt (Matsushita et al. 2022) formalize unsafe Rust, whereas we target safe Rust.

To overcome these limitations, we develop *Synthetic Ownership Logic* (SOL), the first program logic that faithfully reflects the type rules of safe Rust *and* can reason about the functional behavior of Rust programs. Like Prusti, SOL is based on Separation Logic, and it also incorporates ideas from Creusot and Aeneas. Unlike these existing logics, however, SOL is geared towards synthesis, purposely restricting the shape of programs it can derive, to make the search tractable.

Contributions. In summary, this paper makes the following contributions:

- Synthetic Ownership Logic (SOL), a variant of Separation Logic that is targeted to program synthesis of well-typed Rust programs from type signatures and functional specifications.
- RusSOL, the first synthesizer for Rust code from functional correctness specifications. We built RusSOL by integrating SOL into SuSLik's general-purpose proof search framework.
- An extensive evaluation of RusSOL with regard to utility and performance. We show that it is capable of synthesizing a large number of non-trivial heap-manipulating Rust programs, in a matter of seconds, and that required annotations are on average 27% shorter than the code.

2 OVERVIEW

In this section, we give an overview of our synthesizer, RusSOL, whose high-level workflow is outlined in Fig. 2. The synthesizer takes as input data type definitions (not shown in Fig. 2) and optional auxiliary functions, as well as the type signature of the function to be synthesized annotated

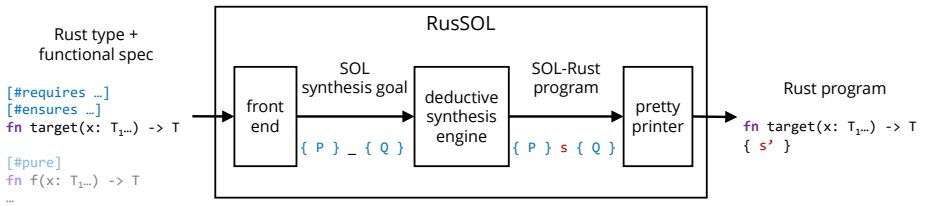


Fig. 2. RuSSOL workflow.

with pre- and postconditions, and produces a Rust implementation. We describe this workflow in two parts. Sec. 2.1 illustrates RuSSOL and its specification language on a series of examples. Sec. 2.2 peeks under the hood of the synthesizer to give a taste of its underlying program logic and deductive synthesis approach. The formal details are presented in Sec. 3.

Our running example throughout this section is a linked list data type:[3]

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

The definition above declares a list parameterized by the element type `T`; a list is a sum type (**`enum`**) with two variants: an empty list is denoted `Nil` and a non-empty list is denoted `Cons` and consists of a head element and a pointer to the tail (of type `Box<List<T>>`). Note that according to the Rust type system, a non-empty list *owns* both its head and its tail, which prevents uncontrolled aliasing among both list elements and list nodes. That is, any mutation of a list's head and tail will require exclusive ownership over the list itself.

2.1 RuSSOL by Example

In this section, we show how RuSSOL helps the user generate various operations over linked lists.

Programs from Types. Let us start with a very simple operation: creating a singleton list. Its type signature, shown below, states that `singleton` takes ownership of the element `elem` (which will be transferred to the new list), and returns an owned list. Instead of implementing the function by hand, we leave the body as a `todo!()` macro invocation, which is a placeholder for the code that the synthesizer will generate, shown on the right.

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

```
// Synthesis result:
let next = Box::new(List::Nil);
List::Cons { elem, next }
```

RuSSOL can synthesize the intended implementation just from the type signature, without any functional specification, for the following reasons. Since the `List` type is parameterized, there is no way of constructing a new value of type `T` "out of thin air". Furthermore, the value of `elem` cannot be duplicated (which would be possible only if `T` implemented, *e.g.*, the `Clone` trait). Therefore, there are only two possible well-typed solutions—returning an empty or a singleton list.[4] Out of the two, the empty list solution appears simpler, but requires dropping (de-allocating) the owned argument `elem`. The synthesizer tries to avoid dropping because it is unlikely that a function requires ownership of an argument without using it and, thus, generates the intended implementation. This example illustrates how Rust's type system—in this case, parametricity and linear types—vastly reduces the space of well-typed programs,[5] and with it the burden of specifications required for synthesis.

[3] We use the term *data type* to refer to both **`struct`**s and **`enum`**s in Rust. This particular definition of `List` is not the most efficient (Beingessner 2015), but we use it for simplicity. Our tool can also handle more efficient definitions.

[4] Technically, there are other well-typed programs, which perform unrelated computation before returning a list, but once again, thanks to the Rust type system, the effect of such unrelated computation would not be observable by clients. The synthesis tool usually picks the shortest program from those that satisfy the specification.

[5] Of course, in case of parametricity, this phenomenon has been studied already in the classic paper by Wadler (1989).

Preconditions and Pure Functions. The next operation we would like to implement is peek, which gives read-only access to the head element of the list. This intent is reflected in its type

```
fn peek(&self) -> &T
```

which takes an *immutable reference* to the list and returns an immutable reference to an element. The peek operation is meaningful only on non-empty lists, and, in fact, the Rust type system makes it impossible to produce a reference `&T` when the list is empty (so, synthesis would fail). To solve this problem, we annotate peek with a *precondition*, demanding that the list be non-empty.

There are multiple ways to express such a precondition in RusSOL, but the most idiomatic one is to define a *pure* (side-effect free) function that computes the length of the list, and then write the precondition abstractly in terms of this function, as shown in the `requires` annotation in Fig. 3. Pure functions are a standard approach in program verification for specifying functional behavior. They are reflected into the logic, enabling their use in specifications. RusSOL performs a syntactic check that pure functions do not perform any assignments or calls to non-pure functions.

```
#[pure]
fn len(&self) -> usize {
  match self {
    List::Nil => 0,
    List::Cons { next, .. } =>
      1 + next.len(),
  }}
```

```
#[requires(self.len() > 0)]
fn peek(&self) -> &T {
  todo!()
}
```

```
// Synthesis result:
match self {
  List::Nil => unreachable!(),
  List::Cons { elem, .. } =>
    elem,
}
```

Fig. 3. Peek at first element: pure function `len` (left), specification (middle), and synthesis result (right).

Pure functions need to be defined manually, which might be perceived as overhead. However, since these are regular, executable Rust functions, many of them occur in implementations anyway (such as a `len` operation on lists). Moreover, a small number of pure functions typically suffices to specify all operations of a data type. Therefore, the cost-benefit ratio of using the synthesizer is still favorable, especially since it guarantees that the synthesized code is provably correct (in this case, that the `unreachable!()` branch is truly unreachable).

Postconditions. Our next task is to implement push, which inserts a new element at the front of the list. Our first attempt at specifying push is depicted below. The type signature of push gives it write access to the list by *mutable reference*, which is a common pattern for mutator methods in Rust. In addition to the signature, we add a *postcondition*—the `ensures` annotation—which uses the pure function `len` we had previously defined to express that the length of the list has increased by one. Following Creusot (Denis et al. 2021), the notation `^r` in function specifications denotes the *final* value of the reference `r` at the time of its expiry, while the regular Rust dereferencing syntax `*r` denotes the value of `r` at the time of its creation. In this case, `(*self).len()` and `(^self).len()` denote the length of the list immediately before and after the call to push, respectively; as is customary in Rust, the former can be shortened to just `self.len()`.

```
#[ensures((^self).len() ==
          (*self).len() + 1)]
fn push(&mut self, elem T) {
  todo!()
}
```

```
// Synthesis result:
match self {
  List::Nil => {
    let next = Box::new(List::Nil);
    let self_new = List::Cons { elem, next };
    *self = self_new
  }
  List::Cons { next, .. } => next.push(elem),
}
```

```
#[ensures (match ^self {
  List::Nil => false,
  List::Cons { ref next, .. } =>
      **next === *self,
  })]
fn push(&mut self, elem T) { todo!() }
```

```
// Synthesis result:
let result = std::mem::replace(self, List::Nil);
let next = Box::new(result);
let self_new = List::Cons { elem, next };
*self = self_new
```

Fig. 4. Second attempt at specifying push (left); intended synthesis result (right).

While the synthesized implementation satisfies the specification, it does not have the intended behavior because it inserts the new element *at the end of the list*, as we can see from the recursive call to `push` on the tail. RuSSOL generates this program first because inserting the new element at the front is more complex than one might think. In particular, the following naive implementation is not type-correct in Rust and will, therefore, not be generated by RuSSOL:

```
1  fn push(&mut self, elem: T) {
2    let new_node = List::Cons(elem, Box::new(*self));
3    *self = new_node;
4  }
```

Line 2 is trying to transfer ownership of `self`'s value to `new_node`, which is not allowed because `self`'s value is passed via a mutable reference, which explicitly prevents such transfers.[6]

To obtain the intended behavior, we strengthen the postcondition of `push` as shown in Fig. 4. The new postcondition stipulates that at the end of the call, the list referenced by `self` must be non-empty, and its `next` field must point to a list that is *structurally equal* to the original value of `self` (we explain the notion of structural equality `===` in more detail in Sec. 2.2).

The generated code has the intended behavior. It also shows why this function is non-trivial to implement: it relies on the library function `std::mem::replace`, which uses unsafe Rust under the hood to take ownership of a value behind a mutable reference (here `self`) replacing it with another value (here `List::Nil`). Pointer manipulations in Rust frequently resort to library functions, such as `replace` and `swap`, to work around the limitations of the type system. RuSSOL contains specifications of these functions in its prelude and automatically invokes them when necessary, through the same mechanism it employs for user-defined helper functions. For example, `replace`'s specification indicates that the function returns the old value of `dest`, and sets `dest` to `src`'s value:

```
1  #[extern_spec] #[ensures (result === *dest && ^dest === src)]
2  fn std::mem::replace<T>(dest: &mut T, src: T) -> T;
```

Re-borrowing. In the last example of our tour, we show how our technique handles functions that create and return a reference into a data structure. Function `last_mut` below returns a mutable reference to the last node of a list (*i.e.*, the `Nil` node), such that callers can add nodes after it; for example, one can write `*(xs.last_mut()) = ys` to append a list `ys` to the end of the list `xs`.

```
fn last_mut(&mut self) -> &mut List<T> {
  todo!()
}
```

```
// Synthesis result:
self
```

The signature of `last_mut` forces the result value to be a reference to some node reachable from `self`, and cannot, for example, reference a freshly created list. This is because the input and output references implicitly have the same *lifetime*, which essentially means that the latter can only be created by *re-borrowing* from the former. Without a postcondition, however, there is no restriction on which node of the list to reference, and the simplest solution is just to return `self`.

[6] The thread at https://stackoverflow.com/q/28258548 documents a Rust programmer running into a similar problem.

Unfortunately, it is not sufficient to add a postcondition that the returned reference points to an empty list. Since `self` is a mutable reference, the synthesized implementation may modify the list. So instead of navigating to the end of the list, it may simply assign an empty list to the passed mutable reference and return that reference:

```
#[ensures((*result).len() == 0)]
fn last_mut(&mut self) -> &mut List<T> {
  todo!()
}
```

```
// Synthesis result:
*self = List::Nil;
self
```

A better way to specify functions that re-borrow from an argument reference is to relate the argument value in the initial state of the function to the value when the re-borrow expires:

```
#[ensures((^self).len() ==
          (*self).len() + (^result).len())]
fn last_mut(&mut self) -> &mut List<T> {
  todo!()
}
```

```
// Synthesis result:
match self {
  List::Nil => self,
  List::Cons { next, .. } => next.last_mut()
}
```

The postcondition in the specification above expresses that the list `self` should eventually grow by the same number of elements as are contained in the list `result`. A subtle point here is that, unlike in our previous example, `^self` *does not* refer to the program point immediately after the function call. Since `last_mut` returns a reference with the same lifetime as `self`, both `self` and `result` expire *later*, at the point when the caller is done using the returned reference, and `^self` refers to that point in the execution. This specification prevents the synthesizer from modifying the length of the list *inside* `last_mut`, which leads to the intended implementation. Interestingly, the postcondition also expresses a non-trivial property of the code *outside* this function: the caller can use `result` only to grow the original list; this property is guaranteed by RusSOL.

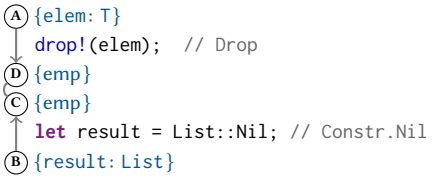
2.2 A Glimpse of Synthetic Ownership Logic

In this section, we illustrate how RusSOL solves the synthesis tasks described in the previous section, using deductive synthesis and a new program logic called Synthetic Ownership Logic (SOL). The general workflow of the synthesizer is shown in Fig. 2. Given a Rust type annotated with a functional specification—via `#[requires]` and `#[ensures]` clauses, possibly using `#[pure]` functions—the RusSOL front-end first translates this input into a *synthesis goal* expressed in SOL. The synthesizer also takes as input data type definitions, as well as specifications of helper functions that can be called from the target function; these are omitted from the figure to avoid clutter. The deductive synthesis engine then uses the synthesis goal to derive a program in SOL-Rust, which is a subset of Rust in a normal form akin to SSA, and with certain implicit operations made explicit (such as re-borrowing of references and dropping variables at the end of their scope). Finally, the pretty printer translates this program into more concise and human-readable Rust.

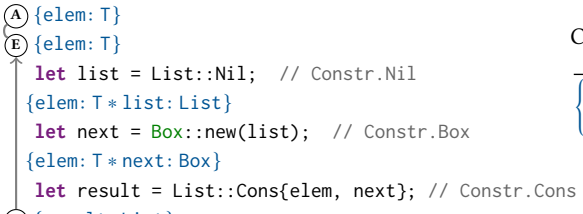
Deductive Synthesis as Symbolic Execution. Recall the `singleton` function from Sec. 2.1. Its type signature `fn singleton<T>(elem: T) -> List<T>` with no further annotations translates into the following synthesis goal in SOL (the type parameter `T` of `List` is omitted for brevity):

$$\{\texttt{elem}\colon \texttt{T}\}\ \_\ \{\texttt{result}\colon \texttt{List}\} \tag{1}$$

The synthesis goal is a pair of *symbolic heaps*, denoting the pre- and postcondition of the target program. For now, a symbolic heap is simply a set of type bindings $\{x_1\colon T_1 * x_2\colon T_2 * \cdots\}$; later in this section, we refine the bindings with information about the *values* of the variables. Our symbolic heaps correspond to symbolic heaps in Separation Logic, where $*$ is separating conjunction and $x\colon T$ is syntactic sugar for a predicate that describes the heap layout of an object of type T.



(a) The empty list program.



(b) The singleton list program.

Fig. 5. Two derivations for the synthesis goal (1).

Drop

$$\frac{}{\{x\colon \mathtt{T}\}\ \mathtt{drop!(x)}\ \{\mathtt{emp}\}}$$

Constr.Nil

$$\frac{}{\{\mathtt{emp}\}\ \mathbf{let}\ \mathsf{x} = \mathtt{List::Nil}\ \{\mathtt{x}\colon \mathtt{List}\}}$$

Constr.Cons

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

Constr.Box

$$\frac{}{\{\mathtt{l}\colon \mathtt{List}\}\ \mathbf{let}\ \mathsf{x} = \mathtt{Box::new(l)}\ \{\mathtt{x}\colon \mathtt{Box}\}}$$

Fig. 6. Rules of SOL used in Fig. 5.

Given the goal (1), the synthesizer attempts to generate a program that transforms the symbolic pre-heap {elem: T} into the post-heap {result: List}. To produce the owned list result, which is not present in the pre-heap, the synthesizer attempts to use one of the two list constructors, Nil or Cons. Assume that it chooses the former, emitting the statement **let** result = Nil, as shown in Fig. 5a. It will then *symbolically execute* this statement backwards starting from the current postcondition Ⓑ to obtain a new postcondition {emp} Ⓒ, *i.e.*, an empty heap. This backward step results in a new synthesis goal, namely to reach an empty heap from the pre-heap {elem: T} Ⓐ. The most immediate way to achieve this is to de-allocate elem. Forward symbolic execution of drop(elem) yields the heap {emp} Ⓓ, which concludes the program derivation.

Choosing the other list constructor, Cons, leads to the derivation shown in Fig. 5b. Constructing a non-empty owned List<T> requires an owned T and an owned Box<List<T>>; the latter, in turn, can be constructed from an owned List<T>, for which we pick the Nil constructor this time. Propagating the postcondition backwards through these three statements yields the heap {elem: T} Ⓔ, which matches our initial precondition Ⓐ, hence the derivation is complete.

This example illustrates how deductive synthesis derives a program one statement at a time, every time emitting a statement that can either *produce* some fragment of the current postcondition or *consume* some fragment of the current precondition, and then propagates the corresponding symbolic heap backwards or forwards, in a form of "bidirectional" symbolic execution. Of course, the real synthesizer does not always guess the right statement to emit on the first try, and instead implements cost-driven search in the space of program derivations, as we explain in Sec. 3.4.

The Rules of SOL. To choose candidate statements and to perform symbolic execution, the synthesizer needs to know the effect of each operation on the symbolic heap. Similarly to Separation Logic, SOL formalizes this knowledge as a set of inference rules about *Hoare triples* of the form $\{\mathcal{P}\}\ s\ \{\mathcal{Q}\}$, where $\mathcal{P}$ and $\mathcal{Q}$ are symbolic heaps and s is a statement. Fig. 6 shows the SOL axioms used in the derivations in Fig. 5: Drop and three instances of the axiom schema Constr that the synthesizer derives from the constructors of the given data types, here List and Box.[7]

[7] Box is actually a special case and is constructed via the new method, but we treat it as a constructor for simplicity.

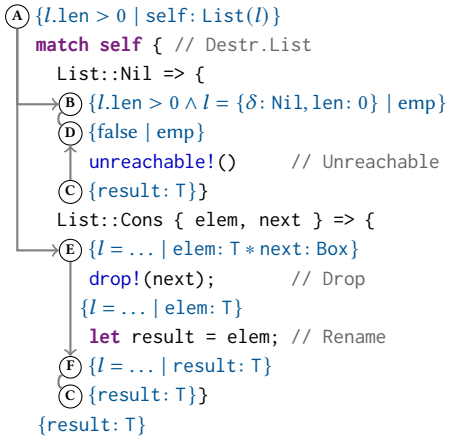


Fig. 7. Deriving function head.

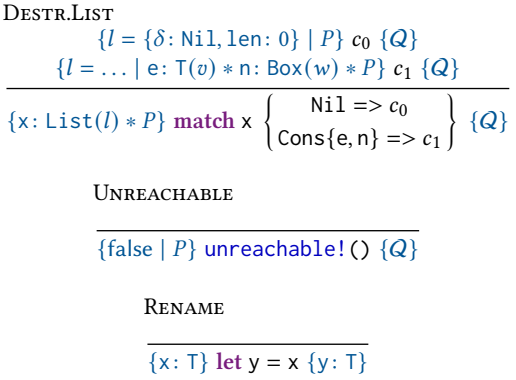


Fig. 8. SOL rules used in Fig. 7.

For example, in the *second* step of the derivation in Fig. 5b, the synthesizer detects a match between the postcondition of CONSTR.BOX and the fragment {next: Box} of the current postcondition. Hence, it applies the rule *backwards*, replacing the matched fragment with the precondition of the rule. On the other hand, in Fig. 5a, the rule DROP is applied *forwards*, by matching the rule's precondition with the precondition of the current synthesis goal. This bidirectional nature of the proof search gives the synthesizer the flexibility to proceed with whichever side of the goal can provide the most guidance on the next statement to be generated (for example, proceeding backwards from emp in our first derivation would be unwise).

Functional Specifications via Snapshots. So far, we have illustrated how SOL encodes Rust types; to handle functional specifications, we extend this encoding to include pre- and postconditions, as well as pure functions. Consider a function head that takes an owned non-empty list and returns its head element, while discarding the tail:

```
#[requires(self.len() > 0)]
fn head<T>(self) -> T
```

We now show how RUSSOL translates the precondition on the length of the list into SOL, and uses it to prove that matching `self` against `Nil` results in unreachable code.

To reason about functional properties in SOL, we refine each binding in the symbolic heap with a *snapshot* (Smans et al. 2010), written $x\colon T(v)$, which can be omitted if irrelevant. Intuitively, the snapshot contains all the information about an object of type T that can be mentioned in a functional specification, while abstracting away details such as the memory layout. More specifically, if T is a primitive type, *e.g.*, `i32`, then v is just a value of type T. If T is a data type, then v is a *record* that contains the value of the discriminant δ (*i.e.*, a value uniquely identifying the `enum` variant), the snapshots of all the fields, and the values of all pure functions that are defined on the type.

Using snapshots, we can translate the specification of head into the following goal in SOL:

$$\{l.\texttt{len} > 0 \mid \texttt{self}\colon \texttt{List}(l)\}\ \_\ \{\texttt{result}\colon \texttt{T}\} \tag{2}$$

Here l stands for the snapshot of the list in the pre-heap, and the `#[requires]` clause is translated into a *pure formula* $l.\texttt{len} > 0$ on the field `len` of the snapshot. Pure formulas in SOL are ordinary first-order formulas that can be added to any symbolic heap to constrain the values of snapshots from that heap (in practice we restrict them to SMT-decidable logics).

Fig. 7 shows a program derivation from this goal. We start by applying the DESTR.LIST rule in the forward direction to `self: List(`l`)` Ⓐ. This rule, shown in Fig. 8, is complementary to CONSTR and generates a **`match`** statement, which consumes **`self`**, leaving in its place its constituent fields. Importantly, in each branch we also learn something about l, the old snapshot of **`self`**; for example, in the `Nil` branch, we learn that $l = \{\delta\texttt{: Nil, len: 0}\}$ Ⓑ, where the value for `len` is extracted from the `Nil` case of the definition of the pure function `len`. Clearly, the pure formula in the new pre-heap is inconsistent, since l.`len` must be both positive and equal to zero.

At this point, RUSSOL fires the UNREACHABLE rule backwards from the post-heap Ⓒ, which allows replacing the symbolic heap with an arbitrary one, as long as the pure formula is replaced with false Ⓓ. Now the synthesizer can close the `Nil` branch: although the pre- and post-heaps have different pure formulas, the SMT solver can verify that one implies the other. More generally, in the presence of functional specifications, the two directions of symbolic execution are bridged by the rule of CONSEQUENCE (not shown in Fig. 8), familiar from Hoare logic, which requires that the pure precondition implies the pure postcondition (and that the impure conditions are the same); in our proof outlines, the applications of this rule are depicted as arcs connecting the two symbolic heaps.

The `Cons` branch Ⓔ does not require any pure reasoning: we fire the familiar DROP rule to dispose of the tail of the list, obtaining the symbolic heap `{elem: T}`, which is almost the same as the desired postcondition modulo variable renaming. This is a job for the RENAME rule, which emits a move assignment to rename variables appropriately Ⓕ. Finally, the `Cons` branch is closed via a trivial application of CONSEQUENCE, since the pure postcondition Ⓒ is simply true.

References and Prophecies. Having seen how SOL handles owned values in Rust, we now turn to references. We consider only *mutable* references in this section; the treatment of immutable references is similar, see Sec. 3. The main challenge for generating well-typed code with references is to satisfy Rust's *borrowing rules*, *e.g.*, that a variable that has been borrowed from cannot be used until the borrow expires.

Variables of a reference type `&'a mut T` are represented in our symbolic heap as a special kind of binding, $y \overset{'a}{\mapsto} T$, where $'a$ denotes the *lifetime* of the reference (which we omit if irrelevant or unchanged). To enforce borrowing rules, every variable in the symbolic heap is annotated with a *blocking set*, *i.e.*, the set of lifetimes of references that may have borrowed from it. For example, executing **`let`** `y = &`**`mut`** `x` from the heap `{x: List}` results in $\left\{\texttt{x}^{\{'a\}}\texttt{: List} * \texttt{y} \overset{'a}{\mapsto} \texttt{List}\right\}$. In the latter heap, we say that `x` is *blocked by* $'a$. Blocking sets are used *internally* by the synthesizer to ensure that a blocked binding is not used by any statement, but they do not complicate user-written specifications. We omit blocking sets when they are empty. Our blocked bindings are similar to loan values in AENEAS (Ho and Protzenko 2022), with lifetimes playing the role of their *loan identifiers*.

Recall from Sec. 2.1 that user specifications can refer to the *final* value of a mutable reference `r` via the syntax `^r`. To reason about final values in SOL, snapshots for reference bindings are pairs of the form $\cdot, \cdot$, *e.g.*, $x \mapsto T(c, \hat{x})$, where the left component is the *current* snapshot of the reference, and the right component is its *final* snapshot at the time of expiry. This treatment is inspired by CREUSOT (Denis et al. 2021), which internally models references as pairs of values. The final snapshot $\hat{x}$ is a so-called *prophecy variable*, whose value is initially unknown; in fact, only when the variable x is dropped do we learn what its final value is—it is x's current value just before it gets dropped. In SOL, this is captured by the rule DROPREF in Fig. 9; the generated `drop!` statement will later be erased by the pretty printer because references are not dropped explicitly in Rust. Fig. 9 also shows (simplified versions of) other rules for manipulating references; we will use these to derive our last two examples, `push` and `last_mut` (see Sec. 2.1). Note that because a single reference binding expires exactly once, the same prophecy variable is used for all occurrences of $\texttt{x} \mapsto \cdot$ in the same scope. We use the variable's name with an overhead wedge as our convention.

DropRef

$$\frac{}{\left\{\phi \mid \mathtt{x}^{\theta} \mapsto \mathtt{T}(c, \hat{x})\right\}\ \mathtt{drop!(x)}\ \{c = \hat{x} \land \phi \mid \mathtt{emp}\}}$$

Write

$$\frac{}{\left\{\mathtt{x} \mapsto \mathtt{T}(v, \hat{x}) * \mathtt{y}\colon \mathtt{T}(c)\right\}\ *\mathtt{x} = \mathtt{y}\ \left\{\mathtt{x} \mapsto \mathtt{T}(c, \hat{x})\right\}}$$

ReBorrow

$$\frac{}{\left\{\mathtt{x} \mapsto \mathtt{T}(c, \hat{x})\right\}\ \mathtt{let\ y = \&mut\ *x}\ \left\{\begin{array}{l}\mathtt{y} \overset{'a}{\longmapsto} \mathtt{T}(c, \hat{y}) * \\ \mathtt{x}^{\{'a\}} \mapsto \mathtt{T}(\hat{y}, \hat{x})\end{array}\right\}}$$

Fig. 9. SOL rules for reference manipulation. Note that, unlike an owned object, a reference may be dropped even while blocked since we can pick any blocking set for θ.

```
A {self ↦ List(l, ŝ) * elem: T}
  let tail = replace(&mut *self, Nil); // Call
B {self ↦ List({δ: Nil}, ŝ) * tail: List(l) * elem: T}
C {self ↦ List(_, ŝ) * tail: List(l) * elem: T}
  let next = Box::new(tail);          // Constr.Box
{self ↦ List(_, ŝ) * next: Box(l) * elem: T}
  let new = Cons { elem, next };      // Constr.Cons
{self ↦ List(_, ŝ) * new: List({δ: Cons, next: l, ...})}
  *self = new;                        // Write
{c.δ = Cons ∧ c.next = l | self ↦ List(c, ŝ)}
  drop!(self);                        // DropRef
D {ŝ.δ = Cons ∧ ŝ.next = l | emp}
```

Fig. 10. Deriving push from goal (3).

Synthesis with References. The specification of push from Fig. 4 (left) is translated as follows:[8]

$$\{\mathtt{self} \mapsto \mathtt{List}(l, \hat{s}) * \mathtt{elem}\colon \mathtt{T}\}\ \_\ \{\hat{s}.\delta = \mathtt{Cons} \land \hat{s}.\mathtt{next} = l \mid \mathtt{emp}\} \tag{3}$$

Since push has no return value, the symbolic heap in the postcondition is empty, but it has a non-trivial pure formula, generated from the `#[ensures]` clause. Note how `^self` from that clause gets translated into the prophecy variable $\hat{s}$ (which is introduced in the pre-heap as the final snapshot of `self`); note also that structural equality `===` is simply equality on snapshots.

The SOL derivation of push from (3) is shown in Fig. 10. We simplified some aspects of this derivation (*e.g.*, performing some pure reasoning implicitly) to focus on the key points. Starting from the post-heap, the synthesizer notices that the reference `self` is absent here but present in the pre-heap, and hence it has likely been dropped; executing DropRef backwards provides the information that the snapshot of `self`, c, before the drop must have been its final snapshot $\hat{s}$. Since the current snapshot of `self` does not match the one available in the pre-heap, the synthesizer decides to apply Write, to move an owned object—here called `new`—with the desired value into the reference `self`. To construct `new`, the synthesizer employs the familiar Constr.Cons rule, augmented with the information about snapshots of the `Box` and `List` values:

$$\{\mathtt{e}\colon \mathtt{T} * \mathtt{n}\colon \mathtt{Box}(n)\}\ \mathtt{let\ x = Cons\ \{\ e, n\ \}}\ \{\mathtt{x}\colon \mathtt{List}(\delta\colon \mathtt{Cons}, \mathtt{next}\colon n, \ldots)\}$$

In this rule, the type `Box` is treated in a special way: its snapshot is the snapshot of the object it points to. Using this information, the Constr.Cons step discharges the pure postcondition of push; it only remains to ensure that the snapshot of `next` is l.

Finally, our backwards symbolic execution arrives at a post-heap where the only mismatched binding is $\mathtt{tail}\colon \mathtt{List}(l)$. Recall from Sec. 2.1 that there is no way to obtain this list by dereferencing `self`, because one cannot turn a reference into an owned object. Neither can the synthesizer construct `tail` as a constant list, because it needs to have a specific snapshot. Consequently, RuSSOL invokes `std::mem::replace` from its standard library, whose specification we have shown in Sec. 2.1. It applies the (omitted) Call rule (in the forward direction), which essentially matches a fragment of the current pre-heap with the callee's precondition and then replaces that fragment into the callee's postcondition. Here, the call replaces the current snapshot of `self` with $\{\delta\colon \mathtt{Nil}\}$, and in return obtains the required owned list `tail`, completing the derivation.

[8] We are omitting the trivial `result: ()` from the postcondition for simplicity.

Program variable		x, y, z, ...		Logical variable			$x, y, z, \hat{x}, \hat{y}, \hat{z} \ldots$
Lifetime		$'a, 'b$		Snapshot	u, v, w	::=	$x \mid \{\delta\colon \mathtt{V}, \mathtt{0}\colon v, \ldots\} \mid (v, \hat{x})$
Mutability	μ	::=	mut | imm	*(primitive)*			| true | false | 0 | 1 | $\cdots$
Blocking set	θ	::=	$\{'a, 'b, \ldots\}$	Pure formula	ϕ, ψ	::=	$v \mid \phi = \psi \mid \cdots$
Type	T	::=	T | &$'a$ μ T | $(T \times \ldots)^{\mathtt{V}} + \ldots$	Symbolic heap	P, Q	::=	emp | $\mathtt{x}^{\theta}\colon T(v) \mid P * Q$
(primitive)			| bool | u8 | i8 | u16 | $\cdots$	Assertion	$\mathcal{P}, \mathcal{Q}$	::=	$\{\phi \mid P\}$

Fig. 12. SOL assertion syntax.

Re-borrowing. To handle re-borrowing, SOL needs to reason about the snapshots of parameters and result whose lifetimes extend *beyond* the end of the function. We illustrate this reasoning using last_mut from Sec. 2.1, which returns a mutable reference to the last node in the list. Its specification translates into the following synthesis goal:

$$\left\{\mathtt{self} \overset{'a}{\mapsto} \mathtt{List}(l, \hat{s})\right\} \_ \left\{\hat{s}.\mathtt{len} = l.\mathtt{len} + \hat{r}.\mathtt{len} \mid \mathtt{result} \overset{'a}{\mapsto} \mathtt{List}(c, \hat{r})\right\} \tag{4}$$

We omit the full derivation of last_mut for brevity, and consider only the base case (**self** = Nil), where the most interesting reasoning happens (Fig. 11). We explain the symbolic execution in the forward direction since it is easier to follow; the synthesizer performs this reasoning backwards.

In the Nil case, the symbolic pre-heap Ⓐ contains just **self**, whose length is known to be zero. The first statement *re-borrows* from **self** into result, using the rule ReBorrow from Fig. 9. The current snapshot of result becomes l, the original snapshot of **self**. More surprisingly, the current snapshot of **self** also changes to the *final* snapshot of result. This update reflects that the referent of **self** may be modified through result: the effect of such a modification is that *once* result *expires*, **self** will point to its final value (just before result expires); since **self** is blocked anyway, its *current* snapshot is irrelevant.

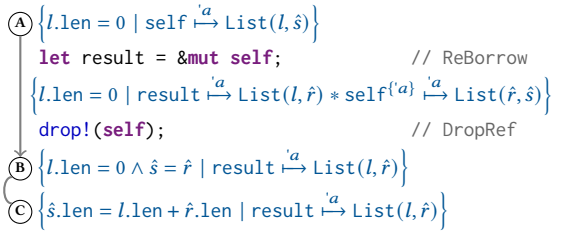


Fig. 11. Base case of last_mut derivation.

At the end of last_mut, the reference **self** is dropped. As per the rule DropRef, the effect of this statement, apart from removing **self** from the symbolic heap, is to learn that **self**'s final snapshot, $\hat{s}$, is its "current" snapshot $\hat{r}$ Ⓑ. Using this fact, we can automatically discharge the pure postcondition $\hat{s}.\mathtt{len} = l.\mathtt{len} + \hat{r}.\mathtt{len}$ Ⓒ. Since result re-borrows from **self**, the **self** reference is dropped at the end of the function, but *its lifetime does not end here* (it will only end when result's lifetime expires). DropRef reflects that, although the referent of **self** might very well be modified after **self** is dropped, it can be modified only via result (or its transitive re-borrows), and therefore the final snapshot of **self** must be the same as the final snapshot of result.

3 SYNTHETIC OWNERSHIP LOGIC, FORMALLY

This section formalizes the assertion language and inference rules of Synthetic Ownership Logic.

3.1 Assertion Language

SOL represents the state of a Rust program as an *assertion* of the form $\{\phi \mid P\}$, where P is a *symbolic heap* and ϕ is a *pure formula*. The syntax of the assertion language is given in Fig. 12.

Symbolic Heaps. A symbolic heap is a set of variable *bindings* connected by SL-style *separating conjunction* ($*$). Each binding $\mathtt{x}^{\theta}\colon T(v)$ associates a program variable x (potentially *blocked* by θ)

with a type T, refined by a *snapshot* v. Note that SOL differentiates between program variables, written in typewriter font, and logical variables, written in italic font. Inside an assertion, program variables appear only on the left of a binding, whereas logical variables can appear in snapshots and pure formulas. The *blocking set* θ of a binding for `x` is the set of lifetimes of all references that are currently borrowing from `x`; in most cases, θ contains at most one lifetime, but larger blocking sets can arise when multiple references are borrowing different fields of a data type.

The *types* of SOL include type variables, reference types, algebraic data types, and primitive Rust types. A data type is a sum of variants, where each variant is a product type tagged with its constructor `V`; for example, the type `Option<i32>` is represented as $()^{\texttt{None}} + (\texttt{i32})^{\texttt{Some}}$.[9] A reference type is annotated with a lifetime and mutability, which can be omitted if irrelevant. In the interest of readability, we introduce syntactic sugar for bindings with reference types, writing $\texttt{x}^\theta \overset{\texttt{'}a}{\mapsto}_\mu T(v)$ in place of $\texttt{x}^\theta\colon (\texttt{\&'}a\ \mu\ T)(v)$ (this syntactic sugar was used throughout the examples in Sec. 2).

Snapshots and Pure Formulas. A *snapshot* represents the value stored in a program variable, and can be a logical variable, a *record*, a *current-final pair*, or a primitive value. A record is a snapshot for a data type; it consists of the pre-defined discriminant field δ that stores the constructor, followed by a numbered field for each type inside the variant (in Sec. 2, we instead used named fields for readability). If a data type has associated pure functions, a snapshot is also extended with a *ghost field* for each such function. For example, the snapshot $\{\delta\colon \texttt{Some}, \texttt{0}\colon 42, \texttt{is\_some}\colon \texttt{true}\}$ represents the Rust value `Some(42)`, assuming that `is_some` is the only pure function defined for `Option`.

As we mentioned in Sec. 2.2, the snapshot of a reference `x` is a pair $(v, \hat{x})$, of its current value v and its final value $\hat{x}$. While the current value can be an arbitrary snapshot term, the final value is always a special kind of logical variable—a so-called a *prophecy variable* (Abadi and Lamport 1988; Denis et al. 2021)—which represents the value `x` will point to at the end of its lifetime. In SOL, prophecy variables are distinguished from other logical variables by the "hat" symbol ($\hat{x}, \hat{y}, \ldots$). Note that `x`'s current value can change during its lifetime (for example, if we write to `x`), but its final value never changes; when `x` is dropped, the prophecy is resolved, *i.e.*, we learn what $\hat{x}$ has been equal to all along. If a reference is immutable, its current value never changes and is always equal to the final value.

Pure formulas ϕ are quantifier-free logical formulas over snapshots. The exact theory used in ϕ is not important, as long as it is efficiently decidable by SMT solvers, and supports equality between snapshots, as well as the standard logical connectives.

Memory Model. A model of an SOL assertion $\{\phi \mid P\}$ is a *state* $\langle \sigma, \mathcal{H} \rangle$, where σ is a valuation of logical variables that satisfies ϕ, and $\mathcal{H}$ is a concrete heap that satisfies P. As is standard in SL with fractional permissions (Boyland 2013), a heap $\mathcal{H}\colon \mathrm{Loc} \rightharpoonup (0, 1] \times \mathrm{Val}$ is a partial map from locations to *permissions* and values.[10] A full permission $p = 1$ allows write access, whereas any $0 < p < 1$ allows only read access to the heap location.

A model of an *unblocked* binding $\texttt{x}\colon T$ where T is a non-reference or a mutable reference $\texttt{\&mut}\ T'$, includes full permissions to all heap locations occupied by the value of type T or T'. If T is an immutable reference $\texttt{\&imm}\ T'$, then the model includes partial permission to the footprint of T'.

On the other hand, a *blocked* binding $\texttt{x}^\theta\colon T$ with $\theta \neq \emptyset$ can be elaborated into a *separating implication* (aka "magic wand" (O'Hearn et al. 2001)) $P_\theta \mathrel{-\!\!*} \texttt{x}\colon T$, where P_θ includes all references with lifetimes in θ: $P_\theta = *_{\texttt{'}a \in \theta} \texttt{y}^{\theta'} \overset{\texttt{'}a}{\mapsto} T'$. A model of a separating implication $P_\theta \mathrel{-\!\!*} \texttt{x}\colon T$ is a model of $\texttt{x}\colon T$ *minus* a model of P_θ; this represents the fact that some of the permissions originally held by

[9] To avoid uninteresting notational overhead, we treat recursive types equi-recursively in the logic; our implementation instead unfolds recursive types whenever it applies the Destr and Constr inference rules.

[10] For our purposes, the distinction between stack and heap locations is not important: we can model stack variables as special heap locations addressed by a pair of scope and variable name.

$$\left[\!\!\left[\begin{array}{l} \texttt{\#[requires } r_1\texttt{]}\ldots \\ \texttt{\#[ensures } e_1\texttt{]}\ldots \\ \texttt{fn f(x: } T_1, \ldots\texttt{)} \to T \end{array} \right]\!\!\right] = \begin{array}{l} \{[\![r_1]\!]^e \wedge \ldots \mid [\![\texttt{x: } T_1]\!]^{\text{bnd}} * \ldots\} \\ \{[\![e_1]\!]^e \wedge \ldots \mid [\![\texttt{result: } T]\!]^{\text{bnd}}\} \end{array} \quad \begin{array}{l} [\![\texttt{x: \&'}a\ \mu\ T]\!]^{\text{bnd}} = \texttt{x} \overset{'a}{\mapsto}_{\mu} T(x, \hat{x}) \\ [\![\texttt{x: } T]\!]^{\text{bnd}} = \texttt{x: } T(x) \end{array}$$

$$[\![\texttt{x}]\!]^e = x \quad [\![\texttt{*x}]\!]^e = x \quad [\![\texttt{\^{}x}]\!]^e = \hat{x} \quad [\![e.\texttt{f()}]\!]^e = [\![e]\!]^e.\texttt{f} \quad [\![e_1 \texttt{ === } e_2]\!]^e = [\![e_1]\!]^e = [\![e_2]\!]^e \quad \ldots$$

Fig. 13. Translating surface specifications to SOL synthesis goals.

x are now held by its borrowers. When all lifetimes in θ expire, we can exchange the implication together with P_θ for an unblocked binding $\texttt{x}: T$: this is called *applying the magic wand* (in principle, a magic wand should be partially applied every time a lifetime in θ expires, but for the restricted set of programs that can be derived by SOL, we do not need such fine-grained accounting).

Finally, the semantics of emp and $*$ is entirely standard; in particular, a model of $P * Q$ is a *disjoint union* of the models of P and Q (more precisely, locations with partial permissions are allowed to overlap, as long as the total permission does not exceed 1). Giving a standard interpretation to $*$ is important because it gives us for free the soundness of the *frame rule* (discussed below).

Hoare Triples. The judgments of SOL are *Hoare triples* of the form $\{\mathcal{P}\}\ c\ \{\mathcal{Q}\}$, where $\mathcal{P}$ and $\mathcal{Q}$ are assertions and c is a *program statement*. We omit the syntax of statements since it is mostly a subset of Rust; importantly, our programs do not have loops, but do have (recursive) function calls. The interpretation of a Hoare triple is the familiar total correctness from SL—stating from a state that satisfies $\mathcal{P}$, c will execute safely and terminate in a state that satisfies $\mathcal{Q}$. Note however, that since SOL is a *synthesis* logic, we interpret c as unknown (to be discovered during proof search).

By common convention, free logical variables of $\mathcal{P}$ and $\mathcal{Q}$ are considered implicitly universally quantified across the triple if they occur in $\mathcal{P}$, and existentially quantified otherwise. Prophecy variables are an exception: they are *always* universally quantified. For example, recall the synthesis goal (4) for `last_mut` from Sec. 2.2: in this goal, the only existential is c, the "current" value of `result`, while all other logical variables ($'a, l, \hat{s}, \hat{r}$) are universal. This is because *the synthesizer* gets to pick the value of `result` at the end of the function, but the *caller* gets to pick everything else, including the value of `result` at the end of its lifetime (since `result`'s referent can be mutated after the function ends).

3.2 From Rust to Synthesis Goals

Deductive synthesis starts from a *goal* $\{\mathcal{P}\}\ \_\ \{\mathcal{Q}\}$, and attempts to derive a triple $\{\mathcal{P}\}\ c\ \{\mathcal{Q}\}$, for some program c, using the synthesis rules, described below. The goal is generated from the surface specification, *i.e.*, a Rust type signature annotated with `#[requires]` and `#[ensures]` clauses, as shown in Fig. 13. In the precondition, the symbolic heap has a binding for every argument; when translating a binding, the meta-function $[\![\cdot]\!]^{\text{bnd}}$ initializes its snapshot with a logical variable (or a pair of a regular and a prophecy variable, in the case of a reference). We use a convention that the names of prophecy variables correspond to those of their program variables; this simplifies the translation and also helps the proof search: because they are always universally quantified, the prophecies for the same reference coming from the forward and backward direction of the search need to match up, otherwise the search would fail.

In the postcondition, the return type is translated into a binding for the special variable `result`. In the presence of reference arguments, it might seem that objects passed by reference must be returned to the caller, and hence must also appear in the postcondition. But note that our translation agrees with the Rust semantics, where all arguments are dropped at the end of the function; dropping a reference, however, does not drop the object it points to.

DropRef

$$\left\{\phi \mid \mathsf{x}^{\theta} \mapsto T(v, \hat{x})\right\}\ \mathtt{drop!(x)}\ \{v = \hat{x} \land \phi \mid \mathsf{emp}\}$$

Write

$$\left\{\begin{array}{l}\mathsf{x} \mapsto_{\mathtt{mut}} T(v, \hat{x}) * \\ \mathsf{y} : T(w)\end{array}\right\}\ *\mathsf{x} = \mathsf{y}\ \{\mathsf{x} \mapsto_{\mathtt{mut}} T(w, \hat{x})\}$$

ReBorrow

$$\frac{'a : 'b \qquad \mu = \mathtt{mut}\ ?\ \theta = \{'b\} : \nu = \mathtt{imm}}{\left\{\mathsf{x} \overset{'a}{\mapsto}_{\mu} T(v, \hat{x})\right\}\ \mathbf{let}\ \mathsf{y} = \&\nu\ *\mathsf{x}\ \left\{\begin{array}{l}\mathsf{y} \overset{'b}{\mapsto}_{\nu} T(v, \hat{y}) * \\ \mathsf{x}^{\theta} \overset{'a}{\mapsto}_{\mu} T(\hat{y}, \hat{x})\end{array}\right\}}$$

CopyOut

$$\frac{T \text{ is primitive}}{\{\mathsf{x} \mapsto T(v, \hat{x})\}\ \mathbf{let}\ \mathsf{y} = *\mathsf{x}\ \left\{\begin{array}{l}\mathsf{x} \mapsto T(v, \hat{x}) * \\ \mathsf{y} : T(v)\end{array}\right\}}$$

DestrBorrow

$$\frac{\begin{array}{c}T = (T_y \times \ldots)^{\mathsf{V}_0} + (T_z \times \ldots)^{\mathsf{V}_1} + \ldots \qquad \{\mathsf{r}, \ldots\} = \mathrm{returns}(Q) \qquad 'b \text{ is fresh} \\ \left\{v = \{\delta : \mathsf{V}_0, 0 : u, \ldots\} \land \phi \mid \mathsf{x}^{\{'b\}} \overset{'a}{\mapsto} T(\{\delta : \mathsf{V}_0, 0 : \hat{y}, \ldots\}, \hat{x}) * \mathsf{y} \overset{'b}{\mapsto} T_y(u, \hat{y}) * \ldots * P\right\}\ c_0\ \{Q\} \\ \left\{v = \{\delta : \mathsf{V}_1, 0 : w, \ldots\} \land \phi \mid \mathsf{x}^{\{'b\}} \overset{'a}{\mapsto} T(\{\delta : \mathsf{V}_1, 0 : \hat{z}, \ldots\}, \hat{x}) * \mathsf{z} \overset{'b}{\mapsto} T_z(w, \hat{z}) * \ldots * P\right\}\ c_1\ \{Q\} \qquad \ldots\end{array}}{\left\{\phi \mid \mathsf{x} \overset{'a}{\mapsto} T(v, \hat{x}) * P\right\}\ \mathbf{let}\ (\mathsf{r}, \ldots) = \mathbf{match}\ \mathsf{x}\ \left\{\begin{array}{l}\mathsf{V}_0(\mathsf{y}, \ldots) \Rightarrow c_0, \\ \mathsf{V}_1(\mathsf{z}, \ldots) \Rightarrow c_1, \ \ldots\end{array}\right\}\ \{Q\}}$$

ConstrBorrow

$$\frac{T = (T_0 \times \ldots)^{\mathsf{V}_0} + (T_1 \times \ldots)^{\mathsf{V}_1} + \ldots \qquad \theta = \beta \cup \ldots}{\left\{\mathsf{x}^{\{'b\}} \overset{'a}{\mapsto} T(\{\delta : \mathsf{V}_n, 0 : \hat{y}, \ldots\}, \hat{x}) * \mathsf{y}^{\beta} \overset{'b}{\mapsto} \mathsf{T}_n(v, \hat{y}) * \ldots\right\}\ \mathtt{drop!(y)}; \ldots\ \left\{\mathsf{x}^{\theta} \mapsto T(\{\delta : \mathsf{V}_n, 0 : v, \ldots\}, \hat{x})\right\}}$$

Fig. 14. Core Borrowing Rules.

The `#[requires]` and `#[ensures]` annotations are translated into pure formulas using the meta-function $[\![\cdot]\!]^e$, which is mostly straightforward. The only cases worth mentioning are: current and final dereferencing syntax (`*x` and `^x`) are translated into the corresponding parts of the snapshot, a pure function invocation is translated into a ghost field access, and structural equality `===` is translated into logical equality on snapshots.

3.3 Synthesis Rules

Derivations of Hoare triples are built using *synthesis rules*. Many of them already appeared in Sec. 2.2, so in the interest of space we focus only on new and non-trivial details (the full version of all the rules is available in the supplementary material). The rules can be divided into three categories: *ownership rules* work with bindings of non-reference types, *borrowing rules* work with reference-type bindings, and *structural rules* are not specific to SOL or Rust.

Ownership Rules. Ownership rules include Drop, Rename, Unreachable, Constr, and Destr, all of which made an appearance in Sec. 2.2. Constr and Destr are rule schemas, which are instantiated for each data type. The only ownership rule we have not shown so far is Primitive, which constructs a value of a primitive type, but this rule is straightforward.

Borrowing Rules. Borrowing rules are the heart of SOL and its most complex part, since they encode the intricate borrowing semantics of Rust; the core borrowing rules are depicted in Fig. 14. The first three of them should be familiar from Sec. 2.2. DropRef is intended for dropping a reference-typed argument at the end of the function, thereby resolving its prophecy $\hat{x}$. As we mentioned above, this means deallocating the binding variable *but not its referent*, which is not an operation that can be explicitly invoked in Rust, hence this rule emits a custom macro `drop!`, which is later removed by the pretty-printer. Note that a reference can be dropped even if it is blocked; this prevents us from ever unblocking any reference it was re-borrowing from, which is not a problem when all references are dropped at the end of the function.

Write and CopyOut are the rules for writing and reading references. Write moves an owned value into a *mutable* reference. CopyOut copies a *primitive* value from a reference into a new owned value; the reason for this restriction is that *moving* from under a reference is not allowed in Rust, while *copying* is allowed only for primitive types.[11]

The ReBorrow rule is useful for re-borrowing (parts of) a reference-typed argument to either return it or pass it to another function (including a recursive call). If the original reference `x` is mutable ($\mu = \texttt{mut}$), the re-borrow `y` is allowed to be either mutable or immutable, but either way, `x` gets blocked; technically, with an immutable `y`, we could downgrade `x` to immutable instead of blocking it, but we did not find this useful for synthesis (we can always create more immutable references from `y`). If $\mu = \texttt{imm}$, however, then `y` must be immutable as well, but no blocking is required: both references co-exist in the heap holding partial permissions to the same referent. It is worth noting that a similar rule can be formulated for borrowing an owned value, but we did not find it necessary for our use cases.

The final two rules in Fig. 14, DestrBorrow and ConstrBorrow, are the counterparts of Constr and Destr for references. DestrBorrow pattern-matches on a reference `x` to a data type, which amounts to re-borrowing all of its fields; as a result, its effect is a combination of Destr and ReBorrow: like in the former, each branch of the match starts with all the fields of its respective variant and gets to assume the structure of `x`'s (current) snapshot; but in the spirit of the latter, `x` is still around inside the branches, albeit blocked. Since the code after the match might be using any unblocked bindings of Q, they must be returned from the match (denoted by returns(Q)). ConstrBorrow undoes the effect of DestrBorrow, by assembling the re-borrowed fields back together into a single data type reference (in the terminology of Sec. 3.1, this rule *applies a magic wand*, albeit in a restricted setting, where the left-hand side of the wand always consists of all fields of `x`). Two more borrowing rules are omitted from Fig. 14: they are similar to DestrBorrow and ConstrBorrow but work with nested references.

Structural Rules. Apart from rules that are specific to Rust, SOL inherits standard structural rules from Hoare logic and SL, such as the frame rule, the rule of consequence, and the rule for sequential composition. Additionally, from the latest version of Synthetic Separation Logic (Itzhaky et al. 2021a), it inherits the mechanism for recursive calls based on *cyclic proofs*; this mechanism handles proofs of termination and also supports discovering recursive auxiliary functions.

3.4 Proof Search

Gadgets. In the interest of clarity, we presented most SOL rules in the style of *small axioms* (O'Hearn et al. 2001), which only mention the minimal set of bindings they need. To build a full derivation, small axioms must be used in conjunction with structural rules, which are unwieldy in automated proof search because they apply to any goal in a number of ways. Instead, our proof search uses *gadgets*, *i.e.*, admissible rules that fuse small axioms and structural rules. For example, the small axiom DropRef cannot be directly applied to the goal:

$$\{v.\texttt{len} = 0 \mid \texttt{x} \mapsto \texttt{List}(v, \hat{x}) * \texttt{result: usize}(42)\}\ \_\ \{\hat{x}.\texttt{len} = 0 \mid \texttt{result: usize}(42)\}$$

because neither the pure postcondition nor the symbolic heaps are in the right shape. Fusing DropRef with the rules of *frame* and *consequence*, however, yields the gadget

$$\{\phi \mid \texttt{x} \mapsto T(v, \hat{x}) * R\}\ \texttt{drop!(x)}\ \{\psi \mid R\} \qquad \text{if } (v = \hat{x} \wedge \phi) \Rightarrow \psi$$

This gadget syntactically unifies with the goal above, yielding a proof obligation $\models (v = \hat{x} \wedge v.\texttt{len} = 0) \Rightarrow \hat{x}.\texttt{len} = 0$, which is easily discharged by the SMT solver.

[11] This is a simplification in our formalism: our implementation allows copying any type that implements the `Copy` trait.

Similarly, instead of using unrestricted *sequential composition* $c_1; c_2$ where c_1 and c_2 are arbitrary programs, we fuse the sequential composition rule with each of the small axioms, yielding the *forward* or *backward* version of the gadget by restricting either c_1 or c_2 to be a specific atomic statement. For example, the backward gadget for DropRef is shown above.

$$\textsc{DropRef-Backward}\quad \frac{\left\{\phi \mid \mathtt{x}^{\theta} \mapsto T(v', \hat{x}) * P\right\}\ c_1\ \left\{v = \hat{x} \Rightarrow \psi \mid \mathtt{x}^{\theta'} \mapsto T(v, \hat{x}) * R\right\}}{\left\{\phi \mid \mathtt{x}^{\theta} \mapsto T(v', \hat{x}) * P\right\}\ c_1; \mathtt{drop!(x)}\ \{\psi \mid R\}}$$

Recall that in Sec. 2 we frame synthesis as "bidirectional symbolic execution" and mention that some statements are more naturally generated in one direction than the other. For example, the Constr rule generates constructors backwards (at the end of the program), to allow the synthesizer to search for necessary constructor arguments in the following steps, like in Fig. 5b. We can now formalize this intuition: forward *vs.* backward symbolic execution simply corresponds to applying forward *vs.* backward gadgets during proof search. To speed up the search, we only create one direction of the gadget for each atomic statement—for example, there are no forward gadgets for Constr or DropRef and no backward gadgets for Destr or Call. Note that forward and backward gadgets can still be guided by both the pre- and the postcondition of the goal. For example, DropRef-Backward relies on the precondition to decide which reference x to drop—it does not conjecture x out of thin air, leaving c_1 to create it. This is an example of a synthesis rule being more restrictive than a verification rule would be, because it has no program to guide it.

Search Algorithm. To search the space of derivations composed of gadgets, RusSOL uses best-first backtracking search, similar to prior work on deductive synthesis (Kneuss et al. 2013; Polikarpova and Sergey 2019). At a high level, the algorithm maintains an and-or search tree, whose nodes are synthesis goals $\{\mathcal{P}\}\ \_\ \{\mathcal{Q}\}$. At each step, the lowest-cost leaf node is expanded according to all applicable SOL gadgets. The cost of a node is a heuristic function of the size of $\mathcal{P}$ and $\mathcal{Q}$.

To avoid exploring redundant orders of rule applications, the search is split into phases. First, we apply only forward gadgets to maximally decompose or leverage the precondition (*e.g.*, Destr, DestrBorrow, CopyOut). Then we apply only backward gadgets to decompose the postcondition (Constr and ReBorrow) or match its reference bindings and their values with the precondition (DropRef, ConstrBorrow, and Write). Lastly, when all the bindings in the pre- and post-heap are matched modulo variable names, primitive bindings, and extra owned bindings in the pre-state, we apply Drop, Rename, and Primitive to bridge the two heaps. A proof branch is closed once it reaches a goal where both symbolic heaps in the goal are empty and the pure precondition implies the pure postcondition (checked by an SMT solver). The search terminates when it finds a full derivation where all branches are closed.

4 EVALUATION

We have implemented our prototype synthesizer RusSOL in Scala, by integrating the gadget versions of SOL rules (Sec. 3.4) into the SuSLik deductive synthesis framework. To search the space of SOL derivations, we were able to reuse SuSLik's existing best-first search algorithm mostly unchanged.

We design our empirical evaluation to answer the following research questions:

(RQ1) Can RusSOL synthesize a variety of non-trivial Rust programs in a reasonable time?
(RQ2) Can RusSOL indeed leverage Rust types to reduce annotation overhead?
(RQ3) Are RusSOL's synthesized programs indeed well-typed and provably functionally correct?

4.1 Benchmarks and Setup

Since there is no established benchmark suite for Rust synthesis, we have assembled our own suite, using synthesis tasks from four different sources:

- Rust is a collection of 51 idiomatic Rust programs extracted from StackOverflow posts and the popular tutorial on linked lists (Beingessner 2015), and also written by us; these programs purposely exercise a variety of Rust features and include all examples form Sec. 1 and Sec. 2. For each of these programs, we retained its original type signature and added minimal functional annotations required to synthesize an implementation behaviorally equivalent to the original.
- SuSLik includes 19 synthesis tasks from the latest SuSLik benchmark suite (Itzhaky et al. 2021b). We excluded tasks that are not meaningful in safe Rust, either because they are trivial (de-allocating a data structure) or impossible due to aliasing (programs on doubly-linked lists). We also excluded tasks that require *branch abduction*, an advanced technique for synthesizing conditionals; this feature is orthogonal to the main focus of this paper, and yet requires non-trivial implementation effort, so we leave it for future work. For the remaining tasks, we manually translated them into Rust data types and signatures, and added minimal annotations required to synthesize the intended implementation.
- Verifier contains all 47 functions from the Prusti and Creusot test suites that came with sufficiently complete functional specifications (the remaining test cases are code snippets with assertions but no meaningful end-to-end behavior); we left these specifications unchanged.
- 100-Crates consists of 2671 Rust function signatures automatically extracted from the top 100 crates on crates.io. We excluded signatures with unsupported types, such as slices, raw pointers, characters, and floats. Because these signatures have no functional annotations, we can rarely expect the synthesizer to guess the intended program, but returning a value of the intended type can still be non-trivial, especially when dealing with data types, generic types, and references. The purpose of this suite is to stress-test the synthesizer's ability to generate well-typed Rust.

All experiments were conducted on a consumer-grade laptop with an Intel i7 CPU and 16GB RAM.

4.2 Results

RQ1: Generality and Efficiency. To answer RQ1, we analyze RuSSOL's performance on annotated benchmarks (*i.e.*, all except 100-Crates). The results are summarized in Tab. 1. These synthesis tasks include functions that manipulate a variety of data types, including linked lists, binary and n-ary (rose) trees, and binary search trees. Many of these functions are recursive, and some—such as *non-destructive append* and *rose tree copy*, discussed below—require the synthesizer to discover (mutually-)recursive auxiliary functions; RuSSOL is able to handle such functions thanks to the *cyclic synthesis* approach by Itzhaky et al. (2021a). The annotated benchmarks also cover a range of challenging Rust features, such as generics, mutable and shared references, re-borrowing parts of a data structure, implementing iterators, and data types that store and nest references.

As reported in Tab. 1, RuSSOL is *able to solve 115 out of 117 synthesis tasks*. Of the two failed tasks, one is from StackOverflow[12] and is unsolvable without unsafe code (not even with internally-unsafe standard library functions). The other failure—calculating the length of a list of lists—is unrelated to Rust ownership rules and is caused by RuSSOL's inability to synthesize a pure arithmetic expression. The 115 solved tasks have non-trivial complexity: derivations typically contain tens of rule applications, up to a maximum of 103 for the *rose tree copy* benchmark. The final pretty-printed Rust programs are 5.4 lines long on average, and up to a maximum of 30 lines. For all of these tasks, we manually checked that the synthesized programs indeed match the user intent or the original implementation. This analysis was done as a sanity check and to confirm that the partial specifications were sufficiently strong.

Most programs were synthesized within only a few seconds (1.8s on average), with *rose tree copy* as the only long-running outlier at 28s. We also compared RuSSOL's results on SuSLik benchmarks

[12] https://stackoverflow.com/q/29570781

Table 1. Synthesis results in 4 categories. For each category, we give the total number of attempted functions along with how many were successfully synthesized, the mean time it took to synthesize each function, the mean number of rules required to find the solution, the mean LOC of the synthesized body, and the mean ratio between synthesis result and spec size (where applicable, this ratio is compared to that of SuSLik).

Source	Group	# Tasks	Synthesis result				Annotations	
			# Solved	Time (s)	SOL rules	Rust LOC	Code/Spec	
Rust	SLL Tutorial	7	7	1.6	36.7	6.3	1.1	
	StackOverflow	5	4	0.9	28.5	4.8	2.0	
	Custom	39	39	2.7	32.8	6.3	1.3	
SuSLik	Integers	1	1	0.4	19	4	2.3	(+125%)
	Singly linked list	7	7	3.5	41.4	10.1	2.9	(+170%)
	Sorted list	2	2	0.7	26.5	5.0	0.8	(+293%)
	List of lists	2	1	2.7	50	15	1.0	(+117%)
	Binary tree	5	5	2.9	58.6	16.0	3.5	(+196%)
	Rose tree	2	2	18.3	91.0	25.0	2.3	(+70%)
Verifier	Prusti	41	41	0.2	9.4	1.4	0.6	
	Creusot	6	6	0.4	20.8	3.2	0.8	
Total annotated		117	115	1.8	26.5	5.4	1.4	
100-crates	Primitive	1036	1036	0.3	18.5	1.0	∞	
	Non-Primitive	1635	1328	0.5	20.8	1.4	∞	

with those reported for SuSLik (Itzhaky et al. 2021b) (they were run on different but comparable hardware). Like RusSOL, SuSLik also fails on one of the 19 benchmarks, albeit a different one: it times out on *rose tree copy* after 30 minutes. Among the commonly solved benchmarks, the synthesis times are similar: 3.1s for RusSOL vs. 2.2s for SuSLik on average. Note, however, that SuSLik has an unfair advantage: it requires the user to provide bounds on the unfolding depth of inductive predicates (its equivalent of Destr and Constr rules), and without these bounds, it is *only able to solve 14 out of 19 benchmarks* before timing out. RusSOL does not impose any such bounds, and moreover, typically requires more Destr/Constr applications than SuSLik, because Rust types have more levels of nesting (*e.g.*, a `List`'s tail is inside a `Box`). We attribute RusSOL's strong performance despite the absence of bounds to the extra guidance provided by the Rust types.

RQ2: Annotation Overhead. To address RQ2, we compute the size ratio of the synthesized Rust code to user-provided annotations, measured in AST nodes (that is, larger is better). Annotations include everything that the user has to write only if they are using the synthesizer: the `#[requires]` and `#[ensures]` clauses, as well as those pure functions that we judged to not be independently useful outside of specifications (for example, the `len` function on lists is excluded from annotations, while the `elems` function, which returns the mathematical set of lists' elements, is included). Moreover, if multiple synthesis tasks in a group used the same pure function, we count it only once.

Last column of Tab. 1 reports the results. On average annotations are *27% more concise* than the synthesized code. This is an encouraging result, especially since, in our experience, annotations are often easier to write because the user has to worry less about ownership. The Verifier category records a particularly low ratio as these benchmarks are designed to test verifiers, generally with small functions and many annotations. On the other hand, in the SuSLik category, the code is on average 2.8 times larger than the specification—an unsurprising result as these tests were designed to showcase the benefits of synthesis. Finally, the Rust suite is the most representative of the tasks we envision RusSOL to be used for, with its annotation ratio in between the two extremes.

For the SuSLik benchmarks, we also compared our Code/Spec ratio to that of SuSLik and found ours to be *172% higher* on average. This confirms our intuition that a lot of information can be extracted from the Rust data type definitions and function signatures. Decoupling pure functions from data type definitions and making use of Rust's generics to reuse data types between benchmarks further reduces the amount of input a user has to provide: for example, for all 19 SuSLik benchmarks we need to define only *three* data types (a linked list, a binary tree, and a rose tree), whereas SuSLik requires *eight* distinct inductive predicates.

RQ3: Well-Typing and Functional Correctness. The programs synthesized by RuSSOL are guaranteed to be correct against the SOL rules by construction, because RuSSOL extracts programs from SOL derivations. Proving soundness of the SOL rules themselves would require an established formal semantics for safe Rust that captures both the borrowing rules and functional properties. Unfortunately, as of now, no such semantics exists. Therefore, to increase our confidence in the soundness of SOL, we seek to confirm empirically that the synthesized programs are indeed well-typed and can be certified correct by an independent verifier. To this end, we ran all programs synthesized for the annotated benchmarks through the Rust compiler and a supported subset thereof through the Creusot verifier; *neither tool uncovered any errors.* Many of the Verifier benchmarks as well as a few from the Custom group exhibit intricate behavior, at the cutting edge of existing Rust verification tools. For example, RuSSOL can reason about arbitrarily-nested borrows, and restrict what clients can do with reference-typed results (much like the `assert_on_expiry` pledge in Prusti, something which Creusot does not yet support). The latter feature enables RuSSOL to synthesize a `get_root` function for a binary search tree (BST), which guarantees that the BST invariant will be preserved even if the returned reference to the root is mutated.

Finally, in order to test RuSSOL's ability to handle real-world Rust types, we turn to the 100-Crates benchmark suite. We ran the synthesizer on all the 2671 supported type signatures, and, whenever synthesis succeeded, we type-checked the solution using the Rust compiler. The results are reported at the bottom of Tab. 1. We separate the tasks into two categories, depending on whether the return type is primitive. Tasks with primitive return types admit trivial solutions (immediately return a constant value); hence in the following, we focus on the "non-primitive" category where many synthesized programs are non-trivial and some are very close or identical to the original code. One example, the `factor_first` function from `Either`, is shown above, and more can be found in the accompanying artifact (Fiala et al. 2023). The Rust compiler did not uncover any errors in the solved benchmarks in either category. As one can see from the table, RuSSOL is able to solve 1328 out of 1635 non-primitive tasks. There are three main reasons why the remaining 307 tasks fail: (1) they require Unreachable in some branches (*e.g.*, `unwrap`) but with no precondition RuSSOL cannot use this rule, (2) they need unsafe code, or (3) the return type is not constructible (*e.g.*, because it has private fields) and we did not give RuSSOL the required functions for creating it.

```
fn factor_first(x: Either<(T,L),(T,R)>)
    -> (T, Either<L,R>) {
  match x {
    Either::Left(tpl) => {
      let res = Either::Left(tpl.1);
      (tpl.0, res) }
    Either::Right(tpl) => {
      let res = Either::Right(tpl.1);
      (tpl.0, res)
    }}}
```

4.3 Discussion

Our experiments show that RuSSOL can synthesize a broad range of Rust implementations with a modest annotation overhead. We identify three categories of results that are especially noteworthy:

(1) *Synthesis from types alone.* Apart from `singleton`, discussed in Sec. 2, we found several non-trivial real-world examples from the 100-Crates benchmark suite, whose behavior could be reproduced by RuSSOL using only their type signatures.

(2) *Complex reasoning about references.* In addition to `last_mut` from Sec. 2 and `get_root` mentioned above, both of which require reasoning about re-borrowing and what the client can do with the returned reference, another notable example is the `next` function for a linked list iterator. This function operates on a datatype (`Iter`) whose field has a reference type, and as such requires subtle reasoning known to be hard both for human programmers and for verifiers (for example, Prusti does not support such datatypes). Indeed, the tutorial we take this from highlights four plausible ill-typed implementations before showing the correct result.
(3) *Complex recursion.* As we mentioned above, RusSOL can synthesize not only self-recursive functions, but also functions that call recursive auxiliaries. One example is *non-destructive append*, which takes as input immutable references to two linked lists and creates a new list containing elements of both. The main function recursively traverses one of the lists, and upon reaching the end, it calls an auxiliary, whose task is to recursively copy the other list. This auxiliary is not provided by the user, but is automatically discovered by RusSOL. Another example is *rose tree copy*, whose implementation consists of two mutually recursive functions: the main function that copies a tree and an auxiliary that copies its list of children.

At the same time, our experiments highlight several *limitations* of the current implementation:

(1) *Traits.* RusSOL does not support traits and associated functions, which makes it incomplete when working with generics or datatypes with private fields. This limitation is the main reason for failures on the 100-Crates benchmark suite.
(2) *Conditionals.* RusSOL cannot generate arbitrary conditionals, as it does not implement branch abduction. As a result, it cannot handle SuSLik benchmarks such as sorting or BST insertion.
(3) *Arithmetic.* RusSOL cannot synthesize non-trivial arithmetic expressions, because it focuses on heap manipulation rather than synthesizing pure expressions.
(4) *Unsafe code.* RusSOL cannot generate unsafe code; any unsafe code required to solve the task needs to be wrapped inside a safe wrapper provided to RusSOL as a library function.

The former three limitations are not fundamental: these features can be implemented with more engineering effort, albeit at the cost of extending the search space. The last restriction is more fundamental, since RusSOL uses the safe Rust type system to guide the search.

5 RELATED WORK

Synthesis for Rust. To our knowledge, RusSOL is the first tool to synthesize Rust programs that satisfy user-provided specifications. The only other Rust synthesizer we are aware of—SyRust (Takashima et al. 2021)—generates code snippets to test unsafe libraries. SyRust targets straight-line sequences of API calls and does not support conditionals, loops, or recursion. It also does not support any kind of functional specifications and is only concerned with well-typing. Moreover, because SyRust generates hundreds of thousands of snippets per API under test, it does not need to always get the typing perfectly right, as long as the rejection rate by the Rust compiler is low enough. For these reasons, SyRust can get away with a lightweight encoding of Rust's typing constraints into SAT, whereas RusSOL requires a full-fledged program logic, in order to reason about functional properties and guarantee well-typing by construction.

Synthesis of Heap-Manipulating Programs. One line of work we directly build upon is *Synthetic Separation Logic* (SSL) (Itzhaky et al. 2021a; Polikarpova and Sergey 2019), which is a logic for deriving heap-manipulating C-like programs from Separation Logic specifications, implemented in the SuSLik tool. One aspect of Rust that SSL captures natively is the treatment of *the heap* as a linear resource. It had also been previously extended with immutable pointers (Costea et al. 2020), which can be used to encode (some aspects of) Rust's shared references.

The main impedance mismatch between SSL and Rust is that Rust also treats *stack variables* as linear resources, which own parts of the heap and cannot alias (whereas in SSL, stack variables are just a way to name heap locations). For that reason, SOL models stack variables very differently from SSL: each stack variable has its unique binding in the symbolic heap and cannot appear in pure formulas. In addition, to model borrowing and shared references, SOL extends SSL with advanced heap assertions akin to *magic wands* (O'Hearn et al. 2001) and *fractional permissions* (Boyland 2013); the former model variables that have been borrowed from, the latter model shared references.

Program Logics for Rust. Recent years have seen a proliferation of verification tools for Rust (Astrauskas et al. 2019; Denis et al. 2021; Ho and Protzenko 2022; Jung et al. 2018; Lattuada et al. 2023; Lehmann et al. 2023; Matsushita et al. 2022, 2021; Wolff et al. 2021) and formalizations of its ownership and borrowing discipline (Jung et al. 2019; Weiss et al. 2019). Unfortunately, none of these formalisms could be directly used for synthesis, because a synthesizer needs to reason about functional properties, *and* enforce the borrowing rules of safe Rust. No prior work supports both.

In particular, Oxide (Weiss et al. 2019) and StackedBorrows (Jung et al. 2019) formalize Rust's borrowing rules as a static and dynamic semantics, respectively, but are not meant for reasoning about functional properties. Aeneas (Ho and Protzenko 2022) positions itself as a verification tool, but targets *extrinsic* verification: it translates a Rust program into a functional language, to enable proving its properties within a proof assistant.

On the other hand, Prusti (Astrauskas et al. 2019; Wolff et al. 2021) and Creusot (Denis et al. 2021) specifically target *intrinsic* reasoning about functional correctness, but their logics do not capture all aspects of Rust's type system. More specifically, Creusot encodes Rust into a pure language, and the soundness of this encoding relies on the program already being well-typed in safe Rust. Prusti translates Rust into the Viper intermediate language (Müller et al. 2016) and reasons about it using a variant of Separation Logic, which has the same limitations as SSL: Prusti's logic can verify a program that, although memory-safe, does not actually type-check in Rust.

Our new logic SOL was inspired by several of these prior formalisms. Our translation from Rust types to Separation Logic builds upon ideas from Prusti, but we use Creusot's prophetic encoding of references, (which in turn evolved from RustHorn (Matsushita et al. 2021)), instead of Prusti's pledges. Our encoding of the borrowing rules using *blocking sets* is similar to Aeneas (and, to a lesser extent, to StackedBorrows). Specifically, a SOL state like $\left\{ \mathsf{x}^{\{'a\}} : \mathsf{T} * \mathsf{y} \overset{'a}{\mapsto} \mathsf{T}(\sigma) \right\}$ would be written in Aeneas as an environment $\mathsf{x} \mapsto \mathsf{loan}\ 'a, \mathsf{y} \mapsto \mathsf{borrow}\ 'a\ (\sigma : \mathsf{T})$ (where σ is a symbolic value). Both assertions indicate that the variable x is currently on loan and therefore inaccessible; the main difference is that our notation (in combination with the prophetic encoding of snapshots) makes it easier to establish what the snapshot of x will be once the loan expires.

6 CONCLUSION

We presented the first automatic synthesis of safe Rust programs from type signatures and formal specifications. Its core is Synthetic Ownership Logic, a new program logic for deriving programs that are guaranteed to satisfy Rust's type and borrowing rules, as well as user-provided functional specifications. This logic unlocks new ways to prune the search by using Rust types, which often severely limit the range of operations applicable to a variable. Compared to synthesis techniques for other languages, this pruning reduces the annotation overhead and speeds up the search.

Possible directions for future work include an extension of the supported Rust subset, optimizations of the search algorithm, as well as exploring various applications of our synthesis approach.

Acknowledgments. We thank the anonymous PLDI'23 reviewers for their insightful comments. This work has been supported by the National Science Foundation under Grant No. 1911149, and by a Singapore MoE Tier 3 grant "Automated Program Repair", MOE-MOET32021-0001.

DATA AVAILABILITY

The software artifact accompanying this paper is available online (Fiala et al. 2023). The artifact contains the source code and build scripts for RusSOL, a corpus of case studies that can be used to reproduce the experimental results described in Sec. 4, and a `README` file in markdown that provides detailed step-by-step instructions for running RusSOL and the experiments. The artifact also contains an appendix for this article (in PDF) that describes additional synthesis rules and discusses notable synthesis results from our case studies.

REFERENCES

Martín Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *LICS*. IEEE Computer Society, 165–175. https://doi.org/10.1109/LICS.1988.5115

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 147:1–147:30. https://doi.org/10.1145/3360573

Aria Beingessner. 2015. Learning Rust With Entirely Too Many Linked Lists. https://rust-unofficial.github.io/too-many-lists/

John Boyland. 2013. Fractional Permissions. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. LNCS, Vol. 7850. Springer, 270–288. https://doi.org/10.1007/978-3-642-36946-9_10

Andreea Costea, Amy Zhu, Nadia Polikarpova, and Ilya Sergey. 2020. Concise Read-Only Specifications for Better Synthesis of Programs with Pointers. In *ESOP (LNCS, Vol. 12075)*. Springer, 141–168. https://doi.org/10.1007/978-3-030-44914-8_6

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2021. *The CREUSOT Environment for the Deductive Verification of Rust Programs*. Research Report RR-9448. Inria Saclay - Île de France. https://hal.inria.fr/hal-03526634

Jonáš Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. 2023. *Reproduction Package for Article "Leveraging Rust Types for Program Synthesis"*. https://doi.org/10.5281/zenodo.7811786

Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP (2022), 711–741. https://doi.org/10.1145/3547647

Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021a. Cyclic Program Synthesis. In *PLDI*. ACM, 944–959. https://doi.org/10.1145/3453483.3454087

Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021b. Deductive Synthesis of Programs with Pointers: Techniques, Challenges, Opportunities - (Invited Paper). In *CAV (LNCS, Vol. 12759)*. Springer, 110–134. https://doi.org/10.1007/978-3-030-81685-8_5

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (dec 2019), 32 pages. https://doi.org/10.1145/3371109

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo recursive functions. In *OOPSLA*. ACM, 407–426. https://doi.org/10.1145/2509136.2509555

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* OOPSLA (2023). To appear.

Nico Lehmann, Adam Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.* 7, PLDI (2023). To appear.

Zohar Manna and Richard J. Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (1980), 90–121. https://doi.org/10.1145/357084.357090

Nicholas D. Matsakis and Felix S. Klock II. 2014. The Rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014*. ACM, 103–104. https://doi.org/10.1145/2663171.2663188

Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI*. ACM, 841–856. https://doi.org/10.1145/3519939.3523704

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* 43, 4 (2021), 15:1–15:54. https://doi.org/10.1145/3462205

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS, Vol. 9583)*. Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS, Vol. 2142)*. Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 72:1–72:30. https://doi.org/10.1145/3290385

The Rust Team. 2017. The Rust programming language. http://rust-lang.org.

Jan Smans, Bart Jacobs, and Frank Piessens. 2010. Heap-Dependent Expressions in Separation Logic. In *FMOODS/FORTE (LNCS, Vol. 6117)*. Springer, 170–185. https://doi.org/10.1007/978-3-642-13464-7_14

Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. 2021. SyRust: Automatic Testing of Rust Libraries with Semantic-Aware Program Synthesis. In *PLDI*. ACM, 899–913. https://doi.org/10.1145/3453483.3454084

Philip Wadler. 1989. Theorems for Free!. In *FPCA*. ACM, 347–359. https://doi.org/10.1145/99370.99404

Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *CoRR* abs/1903.00982 (2019). arXiv:1903.00982 http://arxiv.org/abs/1903.00982

Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021. Modular specification and verification of closures in Rust. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485522