Using Verification Techniques to Synthesise Rust Programs

Master's Thesis Project Description

Fabian Bösiger
Supervised by Jonás Fiala and Prof. Dr. Peter Müller

Programming Methodology Group
ETH Zürich

January 26, 2023

Introduction

Rust is a modern programming language with a strong type system and ownership model that guarantees memory-safety and thread-safety at compile time. However, this type system has the downside of making Rust code hard to write, especially for new developers. To aid new developers, a code synthesiser can be used to automatically generate function based based on the functions signature and specification. The strict type system not only simplifies writing specifications, but also reduces the search space for program synthesis. This makes the Rust language an ideal candidate for program synthesis.

One way to build a program synthesiser is to follow the deductive approach in which a proof search based on formal logical specifications emits a corresponding witness program such that the program satisfies the formal specification. The resulting witness program is correct by construction. Synthetic Ownership Logic (SOL) is a variant of Separation Logic (O'Hearn et al., 2001) that is targeted to program synthesis of well typed Rust programs. SOL is used in RUSSOL, the first synthesiser for Rust code. RUSSOL was built by integrating SOL into the SUSLIK (Polikarpova and Sergey, 2019) general-purpose proof search framework.

The SUSLIK proof search algorithm explores the space of all valid proof derivations by applying derivation rules based on some heuristic. Every leaf of the search tree corresponds to a synthesis goal that includes a precondition and a postcondition. The search starts from the root, i.e. the initial synthesis goal, and always applies a rule to a leaf that isn't closed by a terminal rule. Each leaf closed by a terminal rule denotes either a correctly synthesised program or an inconsistency, in which case the search simply continues. After finding a valid solution, the search can be continued to find other possible solutions. Although SUSLIK is optimised to find proofs in Synthetic Separation Logic (SSL), this method of proof searching is not efficient in searching for SOL proofs.

Approach

To overcome the weaknesses in using SUSLIK as a proof searcher for RUSSOL, the aim of this project is to build a new version of the RUSSOL synthesis tool from scratch written Rust based on the synthesis rules from the existing prototype. The new tool should make use of an improved proof search algorithm optimised for SOL.

SOL derivation rules can be expressed in such a way that they either modify the precondition or the postcondition, but not both. That way, we can split up the search space into two separate search trees, the first one using forward rules exclusively exploring derivations starting from the original precondition, and the second one using backward rules exclusively exploring derivations starting from the original postcondition. In the final step, we can match the derived preconditions with the derived postconditions to obtain the final derivation and the corresponding witness program.

Code Snippet 1: An example function signature to synthesise.

```
enum Either<L, R> {
    Left(L),
    Right(R),
}
```

```
fn select<U, V>(fst: Box<U>, snd: Box<V>) -> Either<(U, V), (V, U)>;
```

As an example, consider the function signature in code snippet 1. To synthesise an appropriate implementation, the original algorithm starts by applying forward rules such as DESTR for the input parameters `fst` and `snd`. Here it is possible to first dereference `fst` and then `snd`, or the other way around. This results in two separate branches in the search tree, one for each possible order. After this first phase is done, the search algorithm starts applying backward rules, in this case CONSTR to construct the result. The function signature expects the result type `Either<(U, V), (V, U)>`, so we can construct either the `Either::Left((fst, snd))` or the `Either::Right((snd, fst))` variant. Note that the backward rules are applied for all open leaves. As a consequence, the search tree may repeat some work in some subtrees as we can see in figure 1. After the search is done, we can construct all the possible programs that satisfy the function signature by following all the possible paths to the terminal leaves. For example, by following the leftmost path we obtain the program shown in code snippet 2.

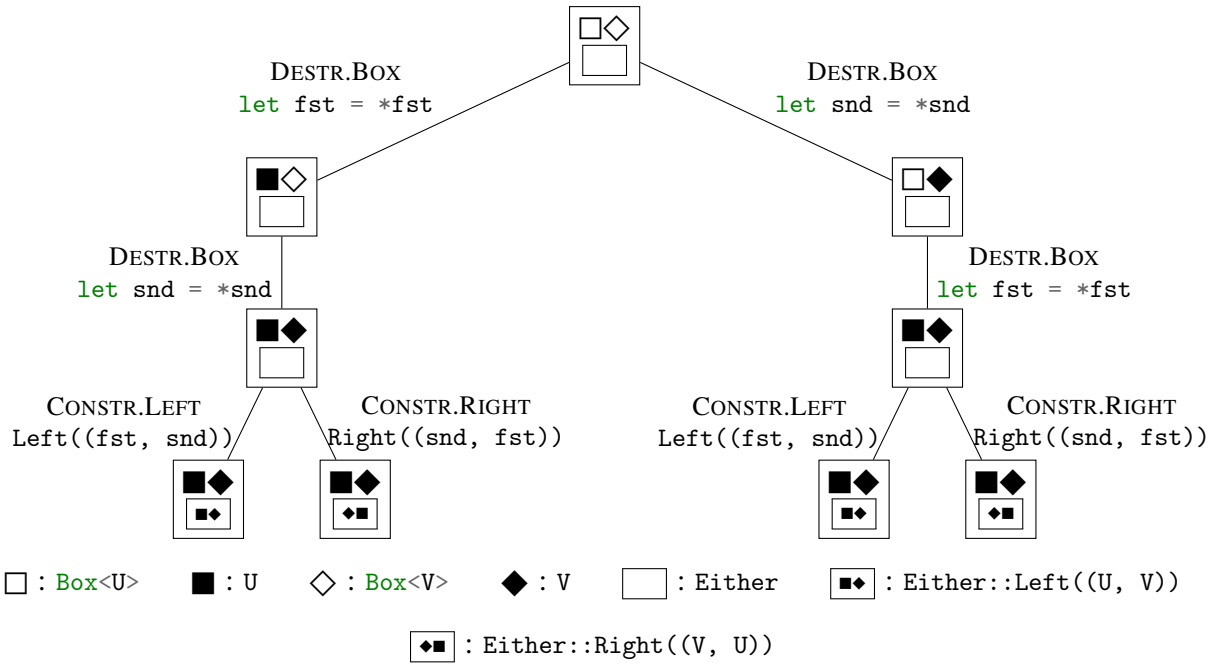


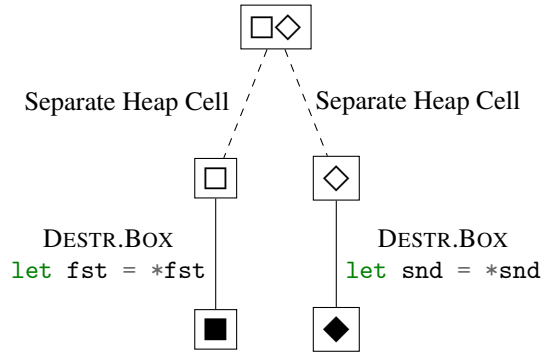
Figure 1: Search tree using the original approach. The nodes represent the program state at each program point. The edges are annotated with the applied rules and the corresponding witness program.

Code Snippet 2: One of the possible solutions that satisfy the constraints of the function signature.

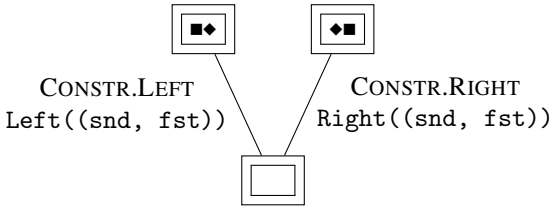
```
fn select<U, V>(fst: Box<U>, snd: Box<V>) -> Either<(U, V), (V, U)> {
    let fst = *fst;
    let snd = *snd;
    Either::Left((fst, snd))
}
```

In contrast, the new search algorithm implements some changes that aim to accelerate its execution time. First, the forward and backward rules are applied on separate search trees as depicted in figure 2. Second, the search algorithm separates preconditions and postconditions into independent heap cells such that we avoid the application of rules in different orders, for example when dereferencing `fst` and `snd`. In a final step, we can combine the leaves from the forward search tree and the backward search tree to obtain our final solution to the search and a corresponding witness program. Concretely, we select both leaves from the forward search tree

in an arbitrary order, and combine it with either one of the leaves from the backward search tree to obtain all possible programs that satisfy the function signature. This again allows us to construct the program shown in code snippet 2. Applying these changes to our new search algorithm allows us to minimise the amount of work that is repeated.



(a) Forward search tree



(b) Backward search tree

Figure 2: Search trees using our new approach. Note that the number of rule applications is reduced significantly compared to figure 1.

The number of search steps saved with the changes implemented as above highly depends on the function signature and specification that constrains the search. However, we can make a rough estimate on the number of search steps saved by changing from a single search tree to separated forward and backward search trees. Assuming the original approach using a single search tree uses n nodes, the height of the tree can be estimated by $\log(n)$. Further assuming that the number of applicable forward and backward rules are roughly the same for any proof search, we can estimate the height of our forward and backward search trees in the new approach by $\log(n)/2$ and a total number of nodes of $2\sqrt{n}$ in both search trees, considerably lowering the number of nodes in the new approach.

Challenges

The new approach of our envisioned search algorithm naturally comes with some challenges. It is unclear how to split up compute time between applying forward rules and backward rules in an efficient way. Furthermore, it is unclear when to stop applying forward and backward rules and start the final phase of connecting the open leaves of forward and backward search trees together to obtain the final solution. Finally, the overhead introduced by this final phase is hard to estimate and it is crucial to find an efficient implementation to reduce this overhead.

An additional challenge will be the handling and implementation of more complicated rules that cannot be easily classified as either a forward or backward rule. Some borrowing rules or rules to call external functions can have multiple program state inputs and outputs, effectively turning the search tree into a directed acyclic graph.

Goals

To summarise, the goal of this project is to create a synthesis tool for the Rust language by designing and implementing an efficient search algorithm whilst adapting logic rules to complement this.

Core Goals

C1 **Find a new proof search algorithm.**
Find a fitting representation for the program state at each program point, i.e. node of the search tree, and implement a search with an efficient is-valid-solution check.
Estimated Time: 4 Weeks

C2 **Implement the chosen approach for ownership and structural rules.**
Implement the rules for updating the program state, optimising them for the chosen approach above. For the implementation, the Rust programming language is used. At this point, synthesising simple programs that work exclusively with owned data should be possible.
Estimated Time: 4 Weeks

C3 **Add support for borrowing rules.**
Add rules that work on borrowed data such that synthesising programs that work with owned and shared data is possible.
Estimated Time: 4 Weeks

C4 **Add support for external functions.**
To enable the synthesis of more complex programs, we add support for calling pre-specified external functions.
Estimated Time: 2 Weeks

C5 **Evaluation and benchmarking.**
Evaluate the new approach on a set of real-world benchmarks. Compare the benchmark results to the results from the original implementation that uses SuSLik as a proof searcher.
Estimated Time: 2 Weeks

Extension Goals

E1 **Add support for recursive function calls.**
The use of recursive function calls enables additional program implementations. To support the synthesis of such programs, the synthesiser can look back in the search tree to check whether a problem is solvable by using recursion.
Estimated Time: 2 Weeks

E2 **Runtime-cost heuristic.**
Usually, there are multiple valid synthesised programs that can be synthesised. To help chose between these programs, we extend the search with a heuristic based on runtime-cost analysis.
Estimated Time: 2 Weeks

E3 **Add support for if-branching.**
Use a technique called "condition abduction" to synthesise code with if-branching, enabling support for programs such as "find the maximum value of the list".
Estimated Time: 4 Weeks

References

O'Hearn, P., Reynolds, J., and Yang, H. (2001). Local reasoning about programs that alter data structures. In Fribourg, L., editor, *Computer Science Logic*, pages 1–19, Berlin, Heidelberg. Springer Berlin Heidelberg.

Polikarpova, N. and Sergey, I. (2019). Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 3(POPL).