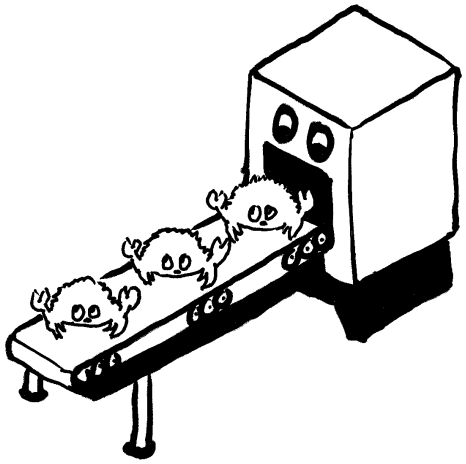
Using Verification Techniques to Synthesise Rust Programs

Master's Thesis

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

Programming Methodology Group
ETH Zürich



Abstract

Rust's strong type system is advantageous for automatic synthesis techniques. There exists a prototype tool for the synthesis of Rust programs based on the SuSLik solver that uses a search tree to find valid programs. We build a new tool that employs a novel directed acyclic graph-based search algorithm with the goal of improving the performance of the search of valid programs while retaining most of the features from the previous approach. In the evaluation, we show that this new search algorithm can significantly improve the performance over the traditional tree-like search algorithm for the subset of programs that are supported by both search algorithms.

Acknowledgements

Major thanks to my supervisor Jonáš Fiala for the interesting and helpful discussions about the topic and the general support during the writing of this thesis. The time and effort he expended to help and advise me was highly appreciated.

A big thanks to Prof. Dr. Peter Müller and the Programming Methodology Group for sparking my interest in the field of program verification and synthesis and providing with me the opportunity to work on such topics.

Contents

1 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 synthesizer can be used to automatically generate the function body based on the signature and specification of the function. 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.

The set of valid solutions for a given function signature can be further constrained by introducing functional specification annotations. During the synthesis of a type-correct solution program to a given function signature, we can also collect facts about the functional behaviour of that program. Then it is a matter of passing these facts together with the functional specification to an SMT solver to verify whether the solution program satisfies the functional constraints.

There exists a prototype tool called RUSSOL (Fiala et al., 2023) for the synthesis of Rust programs based on the SUSLIK program synthesis tool (Polikarpova and Sergey, 2019) that uses a search tree to find valid programs. The search tree is formed by applying synthesis rules to synthesis goals, expanding the search tree until a solution is found. In this thesis, our main contribution is to implement a new search algorithm for RUSSOL that replaces the SUSLIK solver with the goal of improving the synthesis speed. The new algorithm employs a novel directed acyclic graph-based search algorithm, which has the advantage of reducing the number of applied synthesis rules, thus decreasing the number of search steps required to reach a valid solution program. Furthermore, we implement our search algorithm in a way that makes it possible to prioritise certain solutions based on a heuristic, which can help find the most important solution first.

2 Background

2.1 Prusti

Prusti (Astrauskas et al., 2022) is a formal verification tool to verify Rust programs. It introduces functional specification annotations to specify the functional behaviour of programs that resemble the syntax of Rust program expressions.

Prusti is specifically designed for the Rust programming language. By capitalising on Rust's robust type guarantees, Prusti simplifies the process of specifying and verifying Rust programs. Formal verification of system software, which involves reasoning about pointers, aliasing, and mutable state, is known to be challenging and typically requires expertise in complex logics such as separation logic. However, Prusti leverages Rust's ownership type system to enhance the specification and verification process. For example, ownership and framing-related information crucial for separation logic proofs are extracted directly from the program's type-checking process performed by the Rust compiler.

Although we do not use Prusti directly in this work, we use its testing and benchmarking examples that are annotated with functional specifications in our evaluation.

2.2 Creusot

Creusot (Denis et al., 2022) is, similar to Prusti, another verification tool for Rust programs. It introduces its own flavour of functional specification annotations that we use in our work to annotate function signatures with pre- and postconditions.

Creusot's functional specification annotations also resemble the syntax of Rust program expressions, but also introduce some further notation. The expression `^x` is used to denote the final value of some reference `x` at the time of its expiry, while the regular Rust dereferencing syntax `*x` denotes the value at the time of its creation. It is also possible to use pure functions in the functional specifications, allowing the user to write specifications such as `#[ensures((^list).len() == (*list).len() + 1)]`. This postcondition expresses that the annotated function adds one element to `list`, increasing its length by one. Here, `(*list).len()` expresses the length of the list when the function was called, while `(^list).len()` expresses the length of the list at the end of the function call where the reference to `list` expires.

Besides using the same notation as Creusot for functional specifications, we also Creusot's testing and benchmarking examples in our evaluation.

2.3 SuSLik

SuSLik (Polikarpova and Sergey, 2019) is a program synthesis tool for generating heap-manipulating C-like programs from separation logic specifications. Synthetic Separation Logic (SSL) is a variant of Separation Logic (O'Hearn et al., 2001; Reynolds, 2002) that is targeted to program synthesis of well-typed Rust programs and is used by SuSLik to derive solution programs.

The synthesis algorithm operates by taking a pair of assertions, representing pre- and postconditions that describe different states of the symbolic heap and attempts to transform one state into the other

by repeatedly applying SSL synthesis rules. These synthesis rules are used to generate program statements, that in the end make up a valid solution program for the given synthesis problem. The derived programs are inherently correct as they satisfy the assigned pre- and postconditions and are accompanied by complete proof derivations that can be independently verified.

2.4 RUSSOL

RUSSOL (Fiala et al., 2023) is the first synthesizer for Rust code that can satisfy user-provided specifications.

The logics used in both of the previously introduced program verifiers can reason about functional correctness. However, they assume that the program already type-checks and thus do not formalise 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. If this logic was used for program synthesis, it might generate programs with mutable references that point to the same location. Thus, RUSSOL introduces program logic called Synthetic Ownership Logic (SOL) to solve synthesis tasks and generate solution programs that are correct by construction.

The type- and functional specification of a target function make up pre- and postconditions for the synthesis process. They are translated into a synthesis goal that is expressed in SOL. RUSSOL solves this synthesis goal by integrating SUSLIK's proof search framework.

3 Approach

The SuSLik proof search algorithm explores the space of all valid proof derivations by repeatedly applying derivation rules to synthesis goals, thus forming a search tree. Every node 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 node that isn't closed by a terminal rule. The search continues until there are no open nodes left to explore.

Although SuSLik is good at finding solutions in Synthetic Separation Logic (SSL), this method of proof searching is not efficient in searching for SOL proofs. To overcome the weaknesses in using SuSLik as a proof searcher for RuSSOL, this project builds a new version of the RuSSOL synthesis tool based on the synthesis rules from the existing prototype. The new tool makes use of an improved proof search algorithm optimised for SOL.

SOL derivation rules are generally designed 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 last step, we can match the derived preconditions with the derived postconditions to obtain the final derivation and the corresponding solution program.

As an example, consider the function signature in figure 1. To synthesise an appropriate implementation, the original algorithm that uses the SuSLik solver starts by applying forward rules such as DestrStruct for the input parameters `u` and `v` to unwrap the wrappers. Here it is possible to first apply DestrStruct to `u` and then `v`, or the other way around. This results in two separate branches in the search tree, one for each possible order. After this first phase of working on the preconditions is done, the search algorithm starts applying backward rules to work on the postconditions, in this case, ConstrEnum and ConstrStruct to the `result` until the pre- and postconditions match. Note here that the SuSLik solver applies backward rules to all open leaves. As a consequence, the search tree may repeat some work in some subtrees as we can see in figure 3. After the search is done, we can construct all the possible programs that satisfy the function signature by following all the possible paths from the root to the leaves. For example, by following the leftmost path we obtain the solution program shown in figure 2.

In contrast, the new search algorithm implements multiple changes that aim to accelerate its execution time. First, the forward and backward rules are applied on separate search trees. 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 destructing the inputs `u` and `v`. In the 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 solution program. Applying these changes to our new search algorithm allows us to minimise the amount of work that is repeated. Instead of a search tree, this results in a search graph as depicted in figure 6. From this search graph, we can again construct the same solution program shown in figure 2.

```
struct Wrapper<T>(T);

struct Both<A, B>(A, B);

enum Either<A, B> {
    A(A),
    B(B),
}


fn select<U, V>(u: Wrapper<U>, v: Wrapper<V>) -> Either<Both<U, V>, Both<V, U>> {
    todo!();
    result
}
```

Figure 1: A first example, the goal is to replace the `todo!()` with a synthesised function body.

```
fn select<U, V>(u: Wrapper<U>, v: Wrapper<V>) -> Either<Both<U, V>, Both<V, U>> {
    let Wrapper { content: u } = u;
    let Wrapper { content: v } = v;
    let a = Both(u, v);
    let result = Either::A(a);
    result
}
```

Figure 2: One of the possible solutions that satisfy the constraints of the function signature of figure 1.

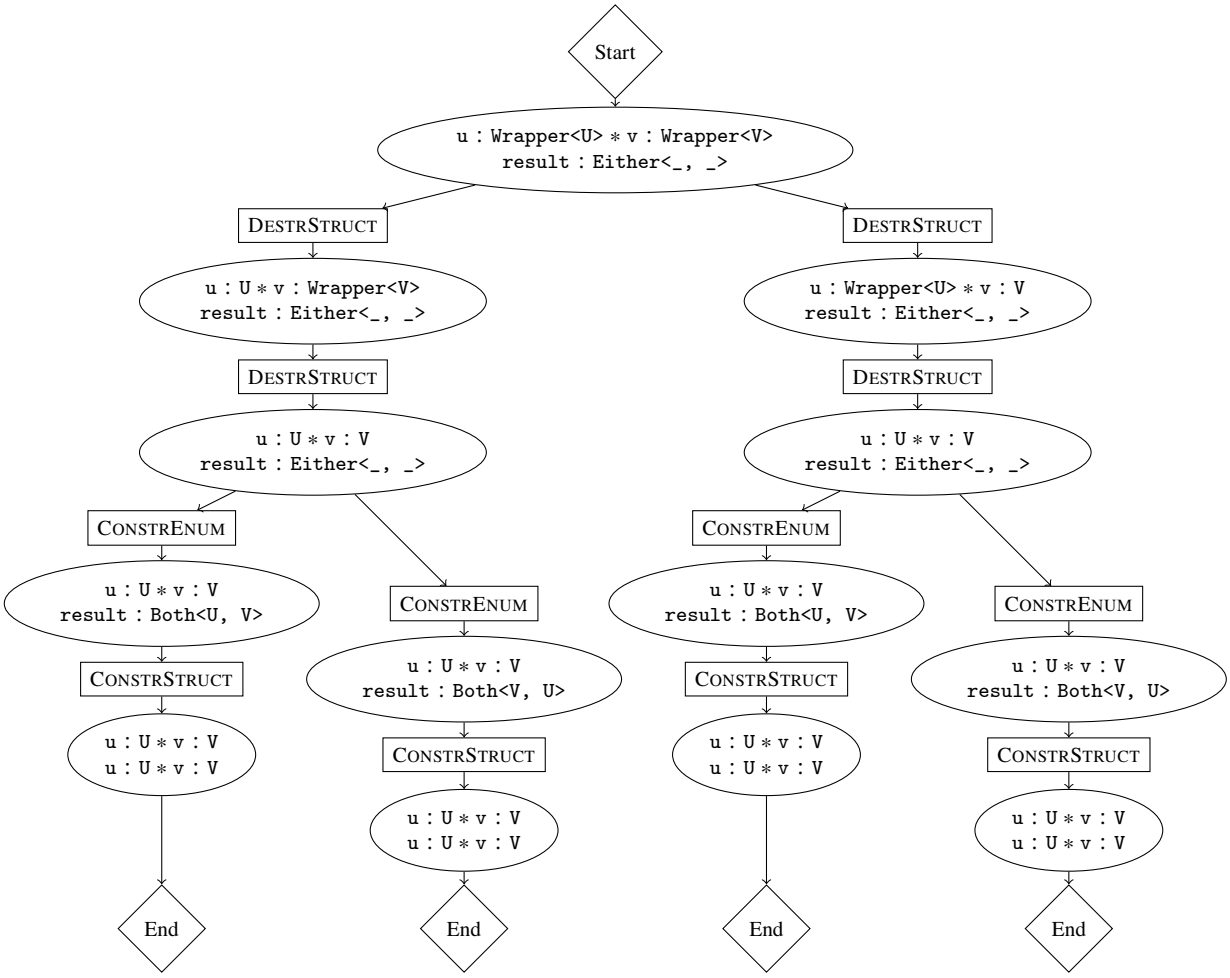


Figure 3: A graphic illustration of SUSLIK's approach for figure 1. Ellipses represent synthesis goals, rectangles represent rule applications. Synthesis goals contain a precondition (first line) and a post-condition (second line).

4 Implementation

We start by introducing the program state representation and how the synthesis rules are used to find possible solution programs by building a synthesis graph. We then discuss how this approach can be extended to support functional specifications, as well as external and recursive function calls. Finally, to show how solution prioritisation can be used, we introduce a runtime-cost based heuristic.

4.1 Program State Representation

To represent the effects of a program on the program state, SOL formalises these changes using inference rules that operate on a *symbolic heap*. A symbolic heap $\{\phi \mid \mathtt{x}_1 : T_1 * \mathtt{x}_2 : T_2 * \cdots\}$ is a set of *heap cells* that represent a binding of the form $\mathtt{x} : T$, where $\mathtt{x}$ is a program variable of type T. A binding $\mathtt{x} : T(v)$ can be refined by a snapshot v that contains all the information about an object of type T. As an example, if T is a primitive type, then the snapshot v contains the concrete value of that primitive type. If T is an enum type, then v contains the value of the discriminant δ that uniquely identifies the enum variant and the snapshots of all its fields. Heap cells can additionally contain a *pure formula* ϕ that constrains the values of snapshots from that heap.

A *synthesis goal* is a type of node in our graph. It is defined as a structure holding a single heap cell, plus some additional data. Goals can be divided into *forward goals* and *backward goals*, where forward goals are used to build the forward search tree, and backward rules to build the backward search tree.

The *synthesis rules* take as an input a number of synthesis goals and output again a number of new synthesis goals. In general, we can classify synthesis rules as *forward rules* that take one forward goal as an input and produce a number of new forward goals as outputs, *backward rules* that analogously take one backward goal as an input and again produce a number of new backward goals. Having only one goal that is taken as an input simplifies the rule application process significantly as the algorithm can just select a single synthesis goal and attempt to apply the rule there. There also exist *matching rules* that match forward goals with backward goals, thus combining the forward and backward tree and forming a directed acyclic graph (DAG).

4.2 Working with Owned Data

The program synthesizer starts by building a *synthesis graph* by adding the input and output arguments of the target function as synthesis goals and then attempting to apply synthesis rules to further expand the graph. The expanded synthesis graph represents all possible program states of a program. The nodes in the synthesis graph represent either *goal nodes* that represent synthesis goals or *rule nodes* that connect goal nodes with each other. The first step in building the synthesis graph is to add a node representing a forward goal for each of the target function's input arguments and a node representing a backward goal for the return argument. The added forward goals representing the input arguments build the root of the potentially multiple forward search trees, while the backward goal represents the root of the backward search tree. Next, the synthesizer attempts to apply synthesis rules to extend the synthesis graph.

For this, any new goal node that is added to the synthesis graph is checked for applicable rules by checking if the concrete types and variables can be substituted into a rule definition. If an applicable rule is found, the rule is added to the graph by adding a corresponding rule node and further goal nodes

$$\textsc{ConstrStruct}\ \frac{T = T_0 \times \cdots \times T_n}{\{\mathtt{f}_0 : T_0(v_0) * \cdots * \mathtt{f}_n : T_n(v_n)\}\ \mathtt{let\ x\ =\ (f_0, \ldots, f_n)}\ \{\mathtt{x} : T(\{\mathtt{f}_0 : v_0, \ldots, \mathtt{f}_n : v_n\})\}}$$

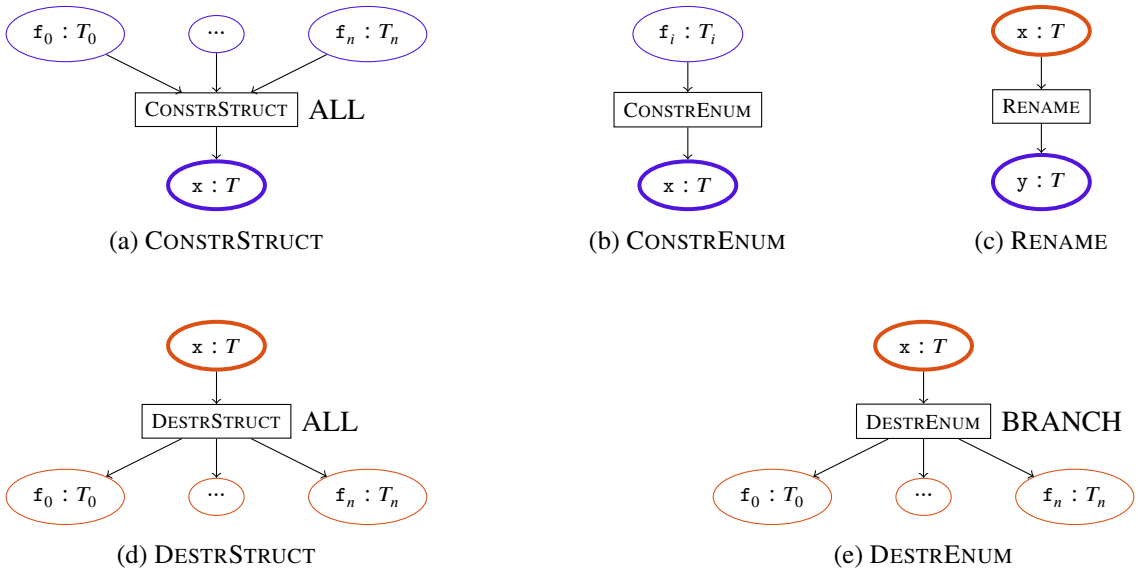
$$\textsc{ConstrEnum}\ \frac{T = (T_0)^{V_0} + \cdots + (T_n)^{V_n}}{\{\mathtt{f}_i : T_i(v_i)\}\ \mathtt{let\ x\ =}\ V_i(\mathtt{f}_i)\ \{\mathtt{x} : T(\{\delta : V_i, \mathtt{f}_i : v_i\})\}}$$

$$\textsc{DestrStruct}\ \frac{T = T_0 \times \cdots \times T_n}{\{\mathtt{x} : T(\{\mathtt{f}_0 : v_0, \ldots, \mathtt{f}_n : v_n\})\}\ \mathtt{let}\ (\mathtt{f}_0, \ldots, \mathtt{f}_n)\ \mathtt{=\ x}\ \{\mathtt{f}_0 : T_0(v_0) * \cdots * \mathtt{f}_n : T_n(v_n)\}}$$

$$\textsc{DestrEnum}\ \frac{T = (T_0)^{V_0} + \cdots + (T_n)^{V_n} \qquad g_i := \{\mathtt{f}_i : T_i(v_i)\}}{\{\mathtt{x} : T(\{\delta : V_i, \mathtt{f}_i : v_i\})\}\ \mathtt{match\ x\ \{}\ V_0(\mathtt{f}_0)\ \mathtt{=>}\ g_0, \ldots V_n(\mathtt{f}_n)\ \mathtt{=>}\ g_n,\ \mathtt{\}}}$$

$$\textsc{Drop}\ \frac{T \text{ not ref}}{\{\mathtt{x} : T(v)\}\ \mathtt{drop!(x)}\ \{\mathrm{emp}\}} \qquad \textsc{Rename}\ \frac{T \text{ not ref}}{\{\mathtt{x} : T(v)\}\ \mathtt{let\ y\ =\ x}\ \{\mathtt{y} : T(v)\}}$$

Figure 4: Synthesis rules for owned types.



(a) ConstrStruct (b) ConstrEnum (c) Rename

(d) DestrStruct (e) DestrEnum

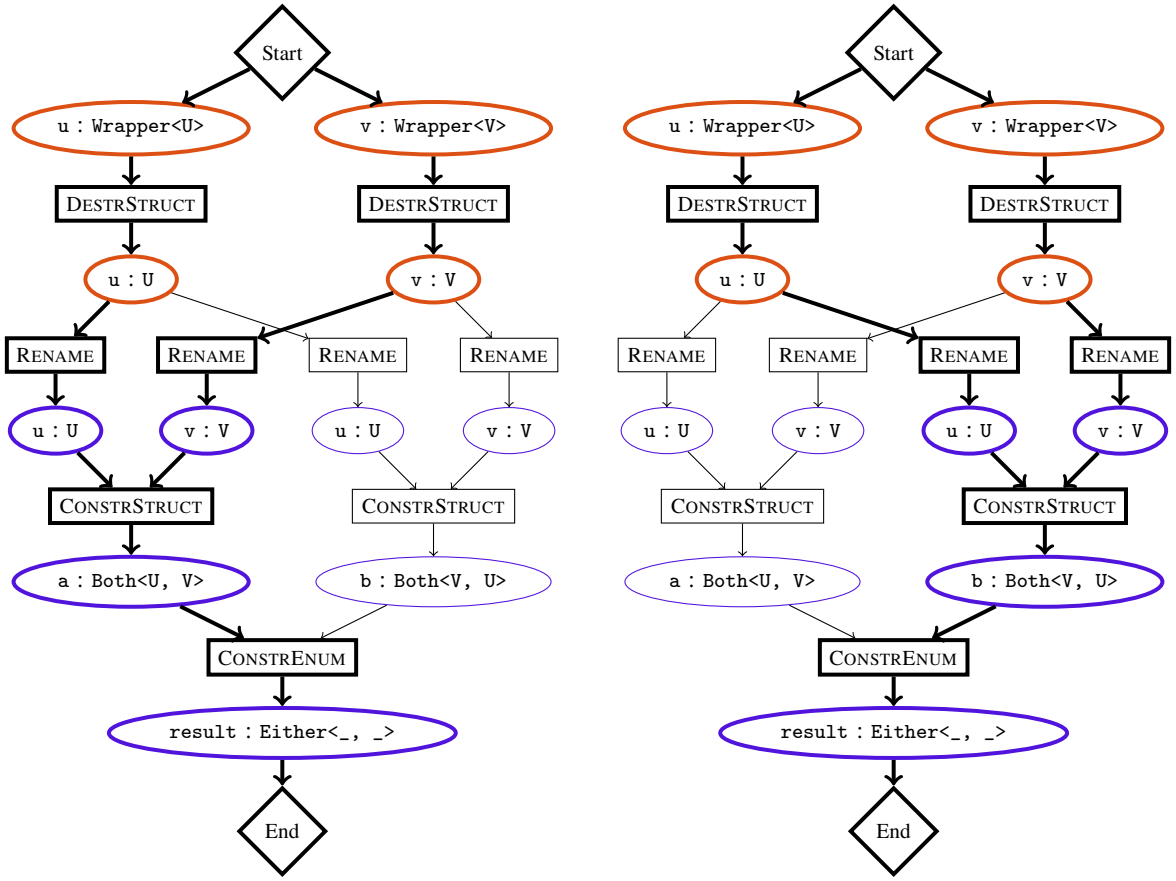
Figure 5: Synthesis rules for owned types as graph components.

for the outputs of that rule. How these nodes are added and connected with each other can be seen in the graph representation of the rules in figure 5. The goal node that triggers a rule application is called the *trigger node* for a rule, highlighted with a thick border. As an example, consider the DestrStruct rule shown in figure 4. This rule can be applied whenever a goal node is of struct type, thus becoming a trigger node for the DestrStruct rule. In figure 5d, we can see how the rule is then added to the synthesis graph. The DestrStruct rule node adds n new goals, one for each field of the struct.

Whenever a new goal is added, it is also checked for matches in the opposite tree by applying match rules such as Rename. In figure 5c, we can see how the Rename rule is added to the synthesis graph. Note that for match rules, we have both a forward and a backward node that can act as trigger nodes.

That way, goal nodes are matched immediately to all possible goal nodes of the opposite direction. This combines the forward and backward search trees, thus effectively forming a directed acyclic graph (DAG) that we call the synthesis graph.

The DROP rule is not an explicit rule that is applied to the synthesis graph. Instead, goal nodes that are dropped simply are not connected to any further rules. This significantly reduces the number of nodes in the synthesis graph and thus increases the performance of the synthesis algorithm compared to when DROP would be implemented as an explicit rule.



(a) First Solution

(b) Second Solution

Figure 6: Synthesis graph for figure 1 with both of the possible flows highlighted.

As an example, consider again figure 1 and the corresponding synthesis graph in figure 6. In the synthesis graph, we draw forward nodes an orange border, backward nodes with a purple border, goal nodes as ellipses and rule nodes as rectangles. The algorithm starts by adding nodes for the input and return arguments, the forward goal nodes `u` and `v` are added to the start node and the backward goal node `result` to the end node. Next, the process of applying rules starts. Here we only consider the rules for owned data as shown in figures 4 and 5 respectively. `u` and `v` are forward goals and of struct type, so the forward rule DESTRSTRUCT can be applied to both of them. This results in two new forward goals `u` and `v` that correspond to their respective inner fields. Further, `result` is a backward goal of enum type, so the backward rule CONSTRENUM can be applied. Similarly to the previous rule application, this creates two new backward goals `a` and `b` that correspond to the field of each variant in `result`. Finally, the goal nodes that have the same type are matched using the RENAME matching rule. There are no other rules that can be applied in a valid manner, thus the algorithm has completed the construction of the synthesis graph.

4.3 Working with Borrowed Data

$$\textsc{DestrBorrowStruct}\ \frac{T = T_0 \times \cdots \times T_n \qquad {}'a_0, \ldots, {}'a_n \text{ fresh}}{\{\mathtt{x} \mapsto T(\{\mathtt{f}_0 : v_0, \ldots, \mathtt{f}_n : v_n\})\}\ \ \mathtt{let}\ (\mathtt{f}_0, \ldots, \mathtt{f}_n) = \mathtt{x}\ \ \{\mathtt{f}_0 \overset{{}'a_0}{\longmapsto} T_0(v_0) * \cdots * \mathtt{f}_n \overset{{}'a_n}{\longmapsto} T_n(v_n) * \mathtt{x}^{\{{}'a_0,\ldots,{}'a_n\}} \mapsto T\}}$$

$$\textsc{DestrBorrowEnum}\ \frac{T = (T_0)^{V_0} + \cdots + (T_n)^{V_n} \qquad {}'a_0, \ldots, {}'a_n \text{ fresh} \qquad g_i := \{\mathtt{f}_i \overset{{}'a_i}{\longmapsto} T_i(v_i)\}}{\{\mathtt{x} \mapsto T(\{\delta : V_i, \mathtt{f}_i : v_i\})\}\ \ \mathtt{match}\ \mathtt{x}\ \{\ V_0(\mathtt{f}_0) \Rightarrow g_0, \ldots, V_n(\mathtt{f}_n) \Rightarrow g_n\ \}\ \ \{\mathtt{x}^{\{{}'a_0,\ldots,{}'a_n\}} \mapsto T\}}$$

$$\textsc{Expire}\ \frac{}{\{\mathtt{x}^{\{{}'a_0,\ldots,{}'a_n\}} \mapsto T * \mathtt{f}_0 \overset{{}'a_0}{\longmapsto} T_0 * \cdots * \mathtt{f}_n \overset{{}'a_n}{\longmapsto} T_n)\}\ \mathtt{skip!()}\ \{\mathtt{x} \mapsto T\}}$$

$$\textsc{DropRef}\ \frac{}{\{\phi \mid \mathtt{x} \mapsto T(v, \hat{x})\}\ \mathtt{drop!(x)}\ \{\hat{x} = v \wedge \phi \mid \mathrm{emp}\}}$$

$$\textsc{Write}\ \frac{}{\{\mathtt{x} \mapsto_{\mathtt{mut}} T(v, \hat{x}) * \mathtt{y} : T(w)\}\ *\mathtt{x} = \mathtt{y}\ \{\mathtt{x} \mapsto_{\mathtt{mut}} T(w, \hat{x})\}}$$

$$\textsc{CopyOut}\ \frac{}{\{\mathtt{x} \mapsto_{\mathtt{mut}} T(v, \hat{x})\}\ \mathtt{let}\ \mathtt{y} = *\mathtt{x}\ \{\mathtt{x} \mapsto_{\mathtt{mut}} T(v, \hat{x}) * \mathtt{y} : T(v)\}}$$

Figure 7: Synthesis rules for borrowed types.

Analogously to the rules that work with owned data, there are rules that work with borrowed data that are shown in figure 7 with their corresponding graph representations in figure 8. However, a few new concepts need to be introduced to deal with borrowed data. While a binding of the form $\mathtt{x} : T$ denotes that the program variable x has type T, we write $\mathtt{x} \mapsto_{\mathtt{imm}} T$ when x is an immutable reference to T and $\mathtt{x} \mapsto_{\mathtt{mut}} T$ for mutable references respectively. Furthermore if a reference has lifetime ${}'a$, we write $\mathtt{x} \overset{{}'a}{\longmapsto} T$. A reference can also be blocked by a set of lifetimes ${}'a_0, \ldots, {}'a_n$, denoted as $\mathtt{x}^{\{{}'a_0,\ldots,{}'a_n\}} \mapsto T$.

There are some rules that require special consideration when applying them to the synthesis graph. For a binding $\mathtt{x} \mapsto T$, the rules DestrBorrowStruct and DestrBorrowEnum introduce fresh lifetimes ${}'a_0, \ldots, {}'a_n$ for their fields and emit a blocked binding $\mathtt{x}^{\{{}'a_0,\ldots,{}'a_n\}} \mapsto T$ such that x can be unblocked by using the Expire rule once its fields expire. In figure 8, blocked nodes are highlighted using dashed borders.

The Expire rule takes a blocked reference and the synthesis goals that contain the blocked set's lifetimes. The references contained in the set of blocking goals are expired which means that the blocked reference can be safely unblocked and reused later.

Contrary to the Drop rule, the DropRef rule is not implemented implicitly and has a graph component representation as shown in figure 8d. This is because there needs to be an explicit rule that connects mutable references to the end node. Otherwise, a mutable reference would always be dropped implicitly due to the inner working of the program search algorithm that is explained later in section 4.4. The DropRef rule connects either a mutable reference or the begin node directly to the end node. This

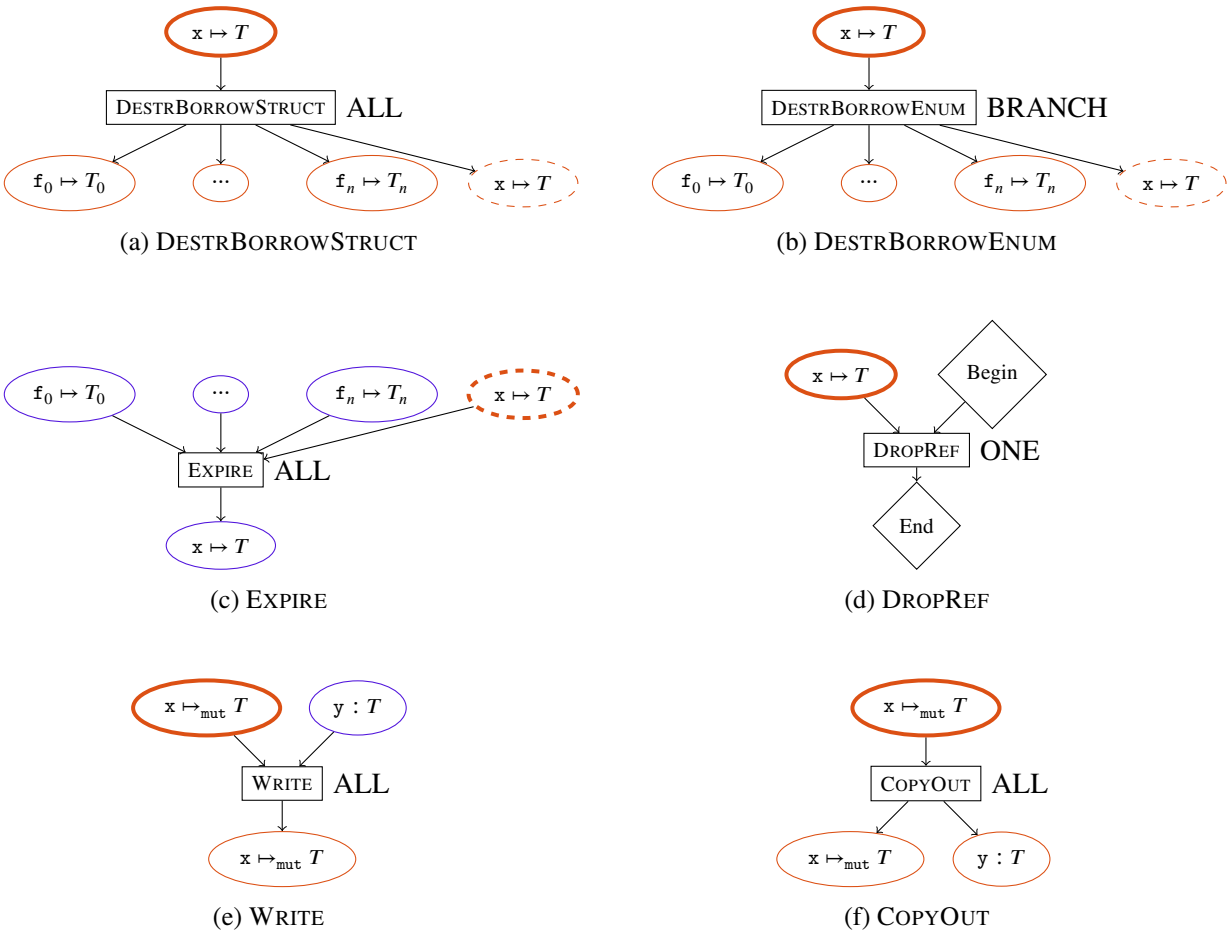


Figure 8: Synthesis rules for borrowed n types as graph components.

indicates that a mutable reference can be dropped either right at the start or after it was used otherwise within the solution program. As an optimisation, the DropRef rule is only applied to goals where the mutable reference was modified. If a mutable reference is never modified, the DropRef rule is never added to connect it to the end node, which is the same as dropping it at the start of the program.

The Write rule in figure 8e takes two goal nodes as an input instead of only one as is usual with forward rules. The first input node corresponds to the mutable reference to write to. The second input goal node corresponds to the value that is written into the mutable reference and is encoded as a backward goal node. This enables our algorithm to apply further backward rules to it or match it with other forward goal nodes, for example by using the Rename rule as seen in figure 11.

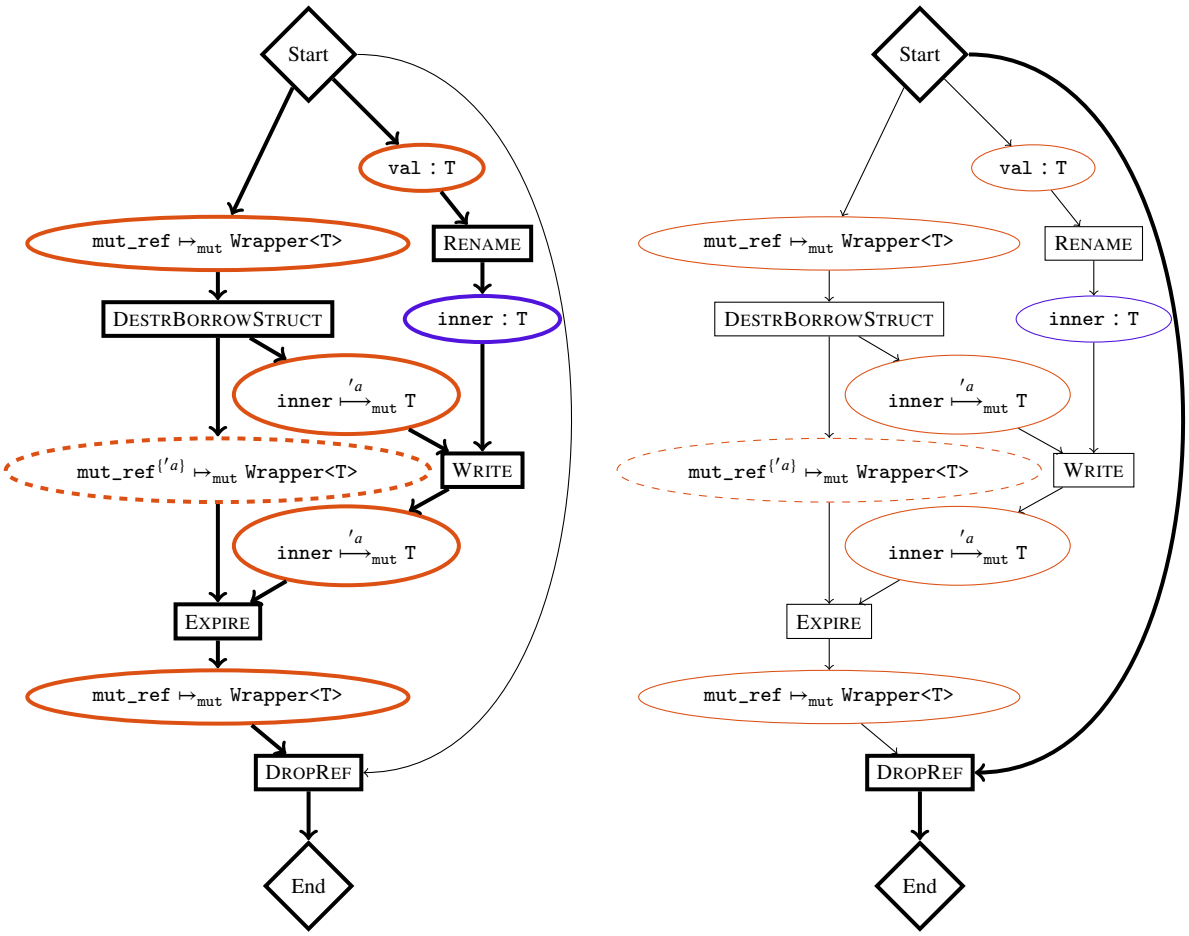
To better understand this mechanism and the borrowing rules in general, let's introduce a new example in figure 9 that uses the newly introduced borrowing rules. In this example, the rules are applied analogously to the first example to build the synthesis graph in figure 11, but also considering the newly introduced borrowing rules. The function `unwrap_write` takes a mutable reference to a wrapped value `mut_ref: &mut Wrapper<T>` and an owned value `val: T`. The first rule that can be applied is the DestrBorrowStruct rule that gives a mutable reference to the inner value `inner: &mut T` of the wrapper and blocks the mutable reference to the wrapper itself, denoted as a node with dashed borders. Next, the Write rule can be applied to the `inner` and is matched using the Rename rule `val: T` that can be written into `inner`. Next, `inner` can be used by the Expire rule to unblock the blocked value `mut_ref`. Finally, the DropRef rule is used to connect the mutable reference to the end node.

```rust
struct Wrapper<T>(T);

fn unwrap_write<T>(mut_ref: &mut Wrapper<T>, val: T) {
    todo!();
}
```

Figure 9: A second example to demonstrate the borrowing rules.

```rust
fn unwrap_write<T>(mut_ref: &mut Wrapper<T>, val: T) {
    let Wrapper(inner) = mut_ref;
    *inner = val;
}
```

Figure 10: One of the solutions that satisfy the type constraints of figure 9.



(a) First Solution where the owned value `val` is written into the mutable reference `mut_ref`.

(b) Second solution where the mutable reference `mut_ref` is not modified.

Figure 11: Synthesis graph for figure 9 with two of the multiple possible flows highlighted.

4.4 Program Search Algorithm

In section 4.1 we have seen how the program state is represented as a synthesis graph. The next step is to find all possible valid flows through this synthesis graph. A *valid flow* through the synthesis graph denotes a way in which we can build the type of the return argument of a target function signature from its input arguments by applying synthesis rules in the order that they were visited by the flow. From this valid flow, a valid solution program can be built from the order of synthesis rule applications.

In order to build a valid flow, the traversal through every node in the synthesis graph has certain restrictions that need to be satisfied. To formalise these restrictions more precisely, we introduce a set of *traversal rules* that specify the conditions that need to hold when traversing a node. Note that traversal rules are a different concept than synthesis rules: While synthesis rules dictate how to build the synthesis graph, traversal rules dictate how to traverse the synthesis graph to build a valid flow.

There are two kinds of traversal rules. The *input traversal rules* specify how multiple input flows are combined at a node. There are three input traversal rules, the ONE rule, the ALL rule and the JOIN rule. The *output traversal rules* specify how an output flow continues from a node. Again, there are three output traversal rules, the ONE rule, the ALL rule and the BRANCH rule. The graph components in figure 5 and 8 are annotated with their respective traversal rules.

To better understand how the traversal rules work, let's look at a few examples. The ONE traversal rules dictate that every incoming flow is kept separate and creates one outgoing flow. In general, this rule is used for goal nodes to enforce Rust's ownership system as we don't want a valid flow to pass the same resource to multiple consumers. An exception are goal nodes whose type implements the `Copy` marker. In that case, the goal node uses the ALL rule such that the object can trivially be copied and passed to multiple consumers. The ONE traversal rule is also used for the CONSTRENUM rule to indicate that an enum can be constructed if only one of its variants can be constructed.

The ALL input traversal rule is used whenever a node requires valid flows for all input nodes and combines them into a valid output flow. Rule nodes such as CONSTRSTRUCT employ this rule to indicate that a valid flow is needed for all goal nodes corresponding to struct fields in order to create a struct. Analogously, the ALL output traversal rule is used for nodes that create multiple valid output flows for one valid input flow. Rule nodes such as DESTRSTRUCT employ this rule to indicate that all the fields of the deconstructed struct can be used at the same time, thus resulting in multiple valid flows, one for each field of the struct.

The BRANCH traversal rule is used to model the behaviour of conditional statements and expressions such as match expression introduced by the DESTRENUM rule. It indicates that the resulting output flows need to be combined at some point further down the flow at a goal node by using the JOIN rule. Each output flow corresponds to a match arm, and the goal node where the corresponding JOIN rule is used corresponds to the output type of the match expression.

Note that all the input traversal rules have the same effect if there is only one incoming edge for some node. This is due to the fact that combining all incoming flows is the same as keeping every incoming flow separate if there is only ever one incoming flow. Analogously, the output traversal rules all have the same effect if there is only one outgoing flow. Thus, we annotate forward rules that have one incoming edge and potentially multiple outgoing edges only with their respective output traversal rule, and backward rules that have multiple incoming edges and one outgoing edge only with their respective input traversal rule.

In order to define the construction of a valid flow more formally, we first define a *scope* as a stack of

branch points $s := [b_1, \ldots, b_m]$. Each branch point b_i has a corresponding set of identifiers $\text{arms}(b_i) := \{a_{i,0}, \ldots, a_{i,n}\}$ that denote the set of n arms originating from branch point b_i. The function $\text{arm}(b_i) := a_{i,j}$ denotes the arm the flow has taken at that branch point. We define $\text{last}(s) := b_m$ to be the last branch point within the scope s.

Next, we introduce operators that act on a scope. Whenever a new scope is entered, the function $\text{push}(s, b_i, a_{i,j})$ adds the branch point b_i to the end of the scope and sets $\text{arm}(b_i) = a_{i,j}$, indicating that arm $a_{i,j}$ was taken at branch point b_i. When leaving a scope, the function $\text{pop}(s)$ removes the last tuple at the end of the scope. $\text{cond}(a_{i,j})$ is an expression representing the branch condition on which arm $a_{i,j}$ at branch point b_i is taken. This is needed for expressing facts about the functional behaviour of a program within a scope.

Using the notion of scopes, we can define a flow $f_i := (\text{scope}_i, E_i, U_i)$ where $\text{scope}(n_i) := s_i$ is a mapping from nodes to scopes and denotes the scope s_i of some node n_i. The scope is used to make sure that the resulting solution programs respect the program scopes of variables. $E_i := \{(n_0^{\text{src}}, n_0^{\text{dst}}), \ldots\}$ is a set of edges that were already traversed in this flow. $U_i := \{(n_0^{\text{src}}, n_0^{\text{dst}}), \ldots\} \subseteq E_i$ is a subset of edges that can only occur once per flow. This mechanism can be used to make sure that the same goal node corresponding to an owned resource is not used multiple times, thus conflicting with Rust's ownership rules. We define the union of two flows to be the union of their set of mappings of nodes to scopes, edges and unique edges $f_i \cup f_j = (\text{scope}_i \cup \text{scope}_j, E_i \cup E_j, U_i \cup U_j)$. When computing the union of scopes, we can assume by the construction of the algorithm that the same keys (in this case nodes of the synthesis graph) always contain the same value (scopes of the respective nodes).

Before joining two flows together, we need to make sure that the uniqueness requirement is met. For this, we introduce the $\text{unique}(f_0, \ldots, f_m)$ function that checks whether the union of the flows $f_0, \ldots, f_m$ does not pass through an edge that can be only used once.

$$\text{unique}(f_1, \ldots, f_m) := \quad \forall U_i, U_j \in \{U_1, \ldots, U_m\}. U_i \cap U_j = \emptyset$$

Figure 12 shows an example where the unique requirement fails. $\text{output}_{\text{ONE}}$ adds the edge e into the unique set. When attempting to apply the $\text{input}_{\text{ALL}}$ traversal rule, this fails as both U_1 and U_2 contain the edge e.

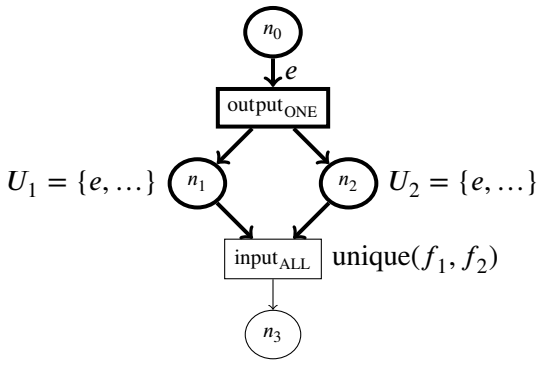


Figure 12: Example where the unique requirement fails.

Before joining flows back together that come from different scopes, the function $\text{joinable}(n_1^{\text{src}}, f_0, \ldots, n_m^{\text{src}}, f_m)$ checks whether the flows $f_1, \ldots, f_m$ coming from the nodes $n_1^{\text{src}}, \ldots n_m^{\text{src}}$ can be joined back into a single flow. For this to be the case, the respective scopes $s_1, \ldots, s_m$ need to originate from the same branch point b and all match arms $\text{arms}(b)$ of this branch point need to be covered by the flows.

$$\text{joinable}(n_1^{\text{src}}, f_1, \ldots, n_m^{\text{src}}, f_m) := \exists b : \left(\forall s_i \in \{\text{scope}(n_1^{\text{src}}), \ldots, \text{scope}(n_m^{\text{src}})\} : \text{last}(s_i) = b\right) \wedge \left(\bigcup_{i=1}^{m} \text{arm}(\text{last}(\text{scope}(n_i^{\text{src}}))) = \text{arms}(b)\right)$$

Figure 13 shows an example where the joinable requirement fails. $\text{output}_{\text{BRANCH}}$ creates a fresh branch point b and two scopes, one for each match arm $\text{arms}(b) = \{n_2, n_3\}$. However, the flow doesn't traverse through the second match arm for the $\text{input}_{\text{JOIN}}$ rule. Both input edges have different scopes which invalidates the joinable requirement.

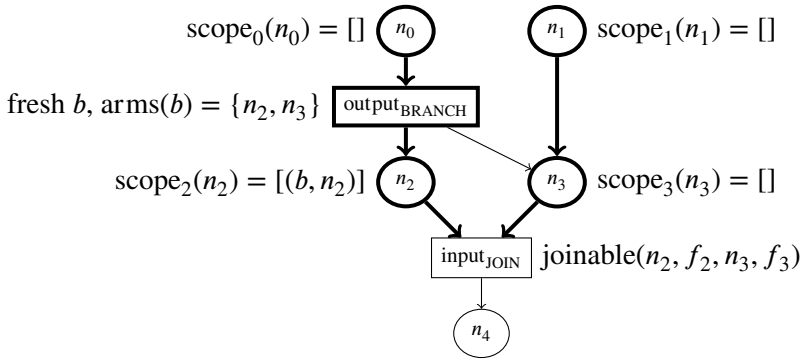


Figure 13: Example where the joinable requirement fails.

Using the newly introduced definition of a flow, we can define the ONE, ALL and BRANCH output traversal rules, and the ONE, ALL and JOIN input traversal rules more formally. All the output traversal rules take a flow f coming from node n^{src} and pass it to the next node n^{dst}. $\text{output}_{\text{ONE}}$ adds an edge from n^{src} to n^{dst}. These edges are also added to the unique set, meaning they can only be used once. $\text{output}_{\text{ALL}}$ also adds an edge from n^{src} to n^{dst} but doesn't add this edge to the unique set. $\text{output}_{\text{BRANCH}}$ works just like $\text{output}_{\text{ONE}}$, but also creates a new scope for the node n^{dst}.

$$\text{output}_{\text{ONE}}(n^{\text{src}}, f, n^{\text{dst}}) := f \cup \left(\emptyset, \{(n^{\text{src}}, n^{\text{dst}})\}, \{(n^{\text{src}}, n^{\text{dst}})\}\right)$$

$$\text{output}_{\text{ALL}}(n^{\text{src}}, f, n^{\text{dst}}) := f \cup \left(\emptyset, \{(n^{\text{src}}, n^{\text{dst}})\}, \emptyset\right)$$

$$\text{output}_{\text{BRANCH}}(n^{\text{src}}, f, n^{\text{dst}}) := f \cup \left(\{n^{\text{dst}} \mapsto \text{push}(s, n^{\text{src}}, n^{\text{dst}})\}, \{(n^{\text{src}}, n^{\text{dst}})\}, \{(n^{\text{src}}, n^{\text{dst}})\}\right)$$

The input traversal rules take multiple flows $f_1, \ldots, f_m$ coming from the respective nodes $n_0^{\text{src}}, \ldots, n_m^{\text{src}}$ and join them into one or potentially multiple flows at node n_j^{dst}. $\text{input}_{\text{ONE}}$ does not combine the incoming flows, but just adds the node n_j^{dst} to each flow, keeping it in the respective scope of that flow. $\text{input}_{\text{ALL}}$ combines the incoming flows into a single flow. This is only allowed if the uniqueness requirement for the edges is met and the scope is the same for all flows at the node n_j^{dst}. It also adds the node n_j^{dst} to each flow within the same scope. Finally, $\text{input}_{\text{JOIN}}$ joins multiple flows coming from the same branch point back together into a single flow. This is only allowed if the uniqueness requirement is met and the flows are joinable.

$$\text{input}_{\text{ONE}}(n_1^{\text{src}}, f_1, \ldots, n_m^{\text{src}}, f_m, n_j^{\text{dst}}) \quad := f_1 \cup \left(\{n_j^{\text{dst}} \mapsto s_1\}, \emptyset, \emptyset\right), \ldots, f_m \cup \left(\{n_j^{\text{dst}} \mapsto s_m\}, \emptyset, \emptyset\right)$$

$$\text{input}_{\text{ALL}}(n_1^{\text{src}}, f_1, \ldots, n_m^{\text{src}}, f_m, n_j^{\text{dst}}) \quad := \begin{cases} f_1 \cup \cdots \cup f_m \cup \left(\{n_j^{\text{dst}} \mapsto s_j\}, \emptyset, \emptyset\right) & \text{unique}(f_1, \ldots, f_m), \\ & s_j = s_1 = \cdots = s_m \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{input}_{\text{JOIN}}(n_1^{\text{src}}, f_1, \ldots, n_m^{\text{src}}, f_m, n_j^{\text{dst}}) \quad := \begin{cases} f_1 \cup \cdots \cup f_m \cup \left(\{n_j^{\text{dst}} \mapsto s_j\}, \emptyset, \emptyset\right) & \text{unique}(f_1, \ldots, f_m), \\ & \text{joinable}(f_1, \ldots, f_n), \\ & s_j = \text{pop}(s_1) = \cdots = \text{pop}(s_m), \\ \emptyset & \text{otherwise} \end{cases}$$

To see how the traversal rules work in a more concrete example, let's examine again the synthesis graph from figure 6 and construct the valid flows. The algorithm starts with a single valid flow from the begin node. The begin node uses the ALL traversal rule as all input arguments can be used at the same time. The next goal nodes `u` and `v` do not implement `Copy` and thus use the ONE rule. However, as the nodes only have a single input flow and a single output flow, they do not create any new flows. The same goes for both of the next DESTRSTRUCT rules. At this point, there is still a single flow that contains the uppermost five nodes in the synthesis graph. The following two goal nodes both have a single input flow but have two outputs. As both of the goal nodes use the ONE rule. This results in a total of four flows, the first flow with the first and second RENAME, the second with the first and fourth RENAME, the third with the second and third RENAME and the fourth with the third and fourth RENAME. The next four goal nodes are again trivial as they have a single input and a single output. The algorithm now arrives at the CONSTRSTRUCT rule nodes that use ALL traversal rule as they require flows for all inputs. This eliminates the second and the third flow as they do not provide resources for all fields of the struct, leaving only two remaining flows, one that traverses through the first CONSTRSTRUCT and another that traverses through the second CONSTRSTRUCT rule. These two remaining valid flows are shown in figure 6a and 6b.

In the second synthesis graph from figure 11, the algorithm starts again with a single valid flow from the begin node. As the begin node uses the ALL traversal rule, both `mut_ref` and `val` can be used in the same flow. The mutable reference `mut_ref` can be destructed using the DESTRBORROWSTRUCT rule, yielding a mutable reference to its inner value `inner`, and a blocked version of `mut_ref`. Next, the `Write` rule can be used to write the owned value `val` into the mutable reference `inner`. Then, the blocked reference `mut_ref` can be unblocked by using the EXPIRE rule. Finally, the DROPREF rule uses the ONE traversal rule and thus creates two flows: Either the flow will be connected through the mutable reference `mut_ref` to the end node, resulting in the flow highlighted in figure 11a where the mutable reference `inner` is written to. Or the flow is connected directly from the begin node to the end node, resulting in the flow highlighted in figure 11b where the mutable reference is not written to.

4.5 Functional Specifications

Building a solution program from the synthesis graph by following the traversal rules ensures that the type specification given by the signature of the target function is satisfied. However, the algorithm also supports synthesising functions that are annotated with functional specifications. In the next section, we take a look at how the algorithm ensures that functional specifications hold for the solution programs.

Rust type signatures can be annotated with `#[requires]` and `#[ensures]` clauses that represent the pre- and postconditions of the annotated function where the precondition can be assumed as a fact and the postcondition needs to be proven to hold after the execution of the synthesised program. All rules specify further facts. For example, the RENAME rule specifies on how program variables are related to each other. If we rename a program variable `let y = x`, we know that the value of `y` is that of `x`. Furthermore, some rules such as FNCALL (discussed later in section 4.6) impose a precondition that needs to be proven in order for it to be used in a valid way. Finally, rules such as DESTRENUM may also add branch conditions. Throughout the traversal of the synthesis graph, these facts and statements to prove are accumulated under their respective branch conditions and transformed into a single expression that we call the *flow condition*. If a valid flow is found that satisfies the type specification of a function signature, the next step is to prove whether the found solution program satisfies the functional specifications. For this, the algorithm checks whether the flow condition is satisfiable by invoking an SMT solver. Note that in this approach it is only necessary to check the flow condition for solutions that already satisfy the type specification. If no functional specification is given, the flow condition does not need to be checked as the program search algorithm already constructs only type-valid solution programs by design.

```
#[ensures(matches!(input, Ok(_)) == matches!(result, Ok(_)))]
fn map_err<T, E1, E2>(input: Result<T, E1>) -> Result<T, E2> {
    todo!();
    result
}

#[extern_spec]
#[requires(matches!(input, Err(_)))]
#[ensures(matches!(result, Err(_)))]
fn convert_err<T, E1, E2>(input: Result<T, E1>) -> Result<T, E2>
```

Figure 14: An example to demonstrate functional specifications and external function calls.

To demonstrate how the construction of the flow condition works in more detail, consider a new example in figure 14. The `map_err` function takes as an input a `Result<T, E1>` enum and output the `Result<T, E2>` enum. To satisfy the specifications, the synthesiser can utilise the external function `convert_err` that expects an `Result<T, E1>` error variant as its input and outputs the corresponding `Result<T, E2>` error variant. To see how facts, statements to prove and branch conditions work in that case, consider the annotated solution program in figure 15. From the start, we know as a fact that the discriminant of the input enum $\delta_{\texttt{input}}$ and the output enum $\delta_{\texttt{result}}$ are within a valid range $\delta_{\texttt{input}} \in \{0,1\} \land \delta_{\texttt{result}} \in \{0,1\}$, where a value of 0 indicates the first enum variant `Ok` and a value of 1 the second variant `Err`. The solution program matches on the input enum which results in two match arms with respective branch conditions. In the first match arm, the program returns the `Ok` variant, yielding the fact $\delta_{\texttt{input}} = 0 \implies \delta_{\texttt{result}} = 0$. In the second match arm, the input for the `convert_err` function call is constructed. Calling `convert_err` yields a trivial

statement to prove that verifies its precondition $\delta_{\texttt{input}} = 1 \implies \delta_{\texttt{input}} = 1$, plus a fact from its postcondition $\delta_{\texttt{input}} = 1 \implies \delta_{\texttt{result}} = 1$, both under the respective branch condition from the match arm. Finally, to verify the postcondition of the `map_err` function, it needs to be proven that $\delta_{\texttt{input}} = 0 \iff \delta_{\texttt{result}} = 0$. This holds because we know that $\delta_{\texttt{input}} \in \{0,1\} \wedge \delta_{\texttt{result}} \in \{0,1\}$, as well as $\delta_{\texttt{input}} = 0 \implies \delta_{\texttt{result}} = 0$ and $\delta_{\texttt{input}} = 1 \implies \delta_{\texttt{result}} = 1$.

```
fn map_err<T, E1, E2>(input: Result<T, E1>) -> Result<T, E2> {
    // fact: δ_input ∈ {0,1} ∧ δ_result ∈ {0,1}
    let result = match input {
        Ok(t) => {
            // branch condition: δ_input = 0
            Ok(t)
            // fact: δ_input = 0 ⟹ δ_result = 0
        }
        Err(e) => {
            // branch condition: δ_input = 1
            let input = Err(e);
            convert_err(input)
            // to prove: δ_input = 1 ⟹ δ_input = 1
            // fact: δ_input = 1 ⟹ δ_result = 1
        }
    };
    // to prove: δ_input = 0 ⟺ δ_result = 0
    result
}
```

Figure 15: The solution that satisfies the type and functional constraints of figure 14.

The use of pure functions is also supported within functional specification declarations. A pure function's body can be expressed as a pure formula. Whenever a fresh program variable is introduced by a DESTR or CONSTR rule, it is also checked whether a pure function is defined that uses this program variable. If so, the pure function's body that is translated as a pure formula is substituted with the respective caller's input and output arguments that were used for that function. The substituted pure formula's body is then added as a fact to the already existing facts ϕ in the following synthesis goals.

4.6 External Function Calls

External functions are annotated with the `#[extern_spec]` clause. We implement external function calls by translating an external function's input arguments and return argument to goal nodes, similar to the arguments of the function that is to be synthesised. However, instead of translating input arguments as forward nodes and the return argument as a backward node, we invert the direction such that input arguments are translated as backward nodes and the return argument as a forward node. The goal nodes that correspond to the input arguments are connected by the FNCALL rule to the goal node representing the output argument. Whenever there exists a solution for all input arguments, the output argument of the function can be computed by invoking the external function call.

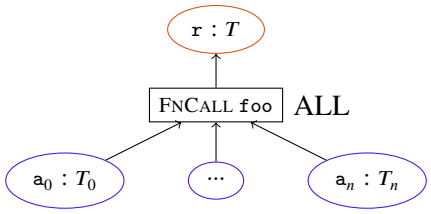
Figure 16 and 17a show the FNCALL rule and its corresponding synthesis graph component. It utilises the ALL traversal rule to express that all input arguments are needed in order to compute the output argument. There is no trigger node as the FNCALL rule is added to the synthesis graph only when

$$\textsc{FnCall}\ \frac{\phi \implies \phi' \qquad \{\phi' \mid \mathtt{a}_0 : T_0(v_0) * \cdots * \mathtt{a}_n : T_n(v_n)\}\ \mathtt{foo}\ \{\psi \mid \mathtt{r} : T(w)\}}{\{\phi \mid \mathtt{a}_0 : T_0(v_0) * \cdots * \mathtt{a}_n : T_n(v_n)\}\ \mathtt{let\ r = foo(a_0, \ldots, a_n)}\ \{\phi \wedge \psi \mid \mathtt{r} : T(w)\}}$$

Figure 16: Synthesis rule for a function call.

translating an external function's arguments and is not dynamically added as the synthesis graph is extended with new synthesis goals, contrary to all the other rules.

Furthermore, if the external function call is annotated with functional specifications, the FnCall rule adds the function's precondition as a statement that needs to be proven. This ensures that the program search algorithm returns only solutions where the external function's precondition is satisfied. The function's postcondition is added as a given fact.



(a) FnCall

Figure 17: Synthesis rule for function calls as a graph component.

Let's take a look at how the synthesis graph of figure 18 is created from the example in figure 14. First, the input and return arguments of the function `map_err` are translated into their respective goal nodes that are connected to the begin and end nodes. Then, the input and return arguments of the function `convert_err` are translated, but in an inverted manner. After all the function arguments are translated into goal nodes, the synthesis rules are applied as usual, eventually forming the complete synthesis graph.

4.7 Recursive Function Calls

To allow recursive function calls, we can reuse the same mechanism that allows the call of external functions described in the previous section 4.6. First, the target function's input arguments are added to the synthesis graph as forward goal nodes and the return argument as a backward goal node as usual. Additionally, the target function is also added as a function call just as external functions, adding the return argument as a forward goal that is connected via a FnCall rule to its input arguments as backward goals.

To avoid recursive function calls that don't make any progress, the number of Destr and Constr rules that are applied before the invocation of the recursive function call are counted. The recursive function call is only allowed if the number of Destr rules is greater than the number of Constr rules. This uses the size of the object as a termination measure, guaranteeing that progress is made. We now have all prerequisites to synthesise recursive functions, for example the one shown in figure 19. Before calling itself recursively, the function applies the DestrEnum rule by matching on the list, thus making progress and eventually reaching the end of the list.

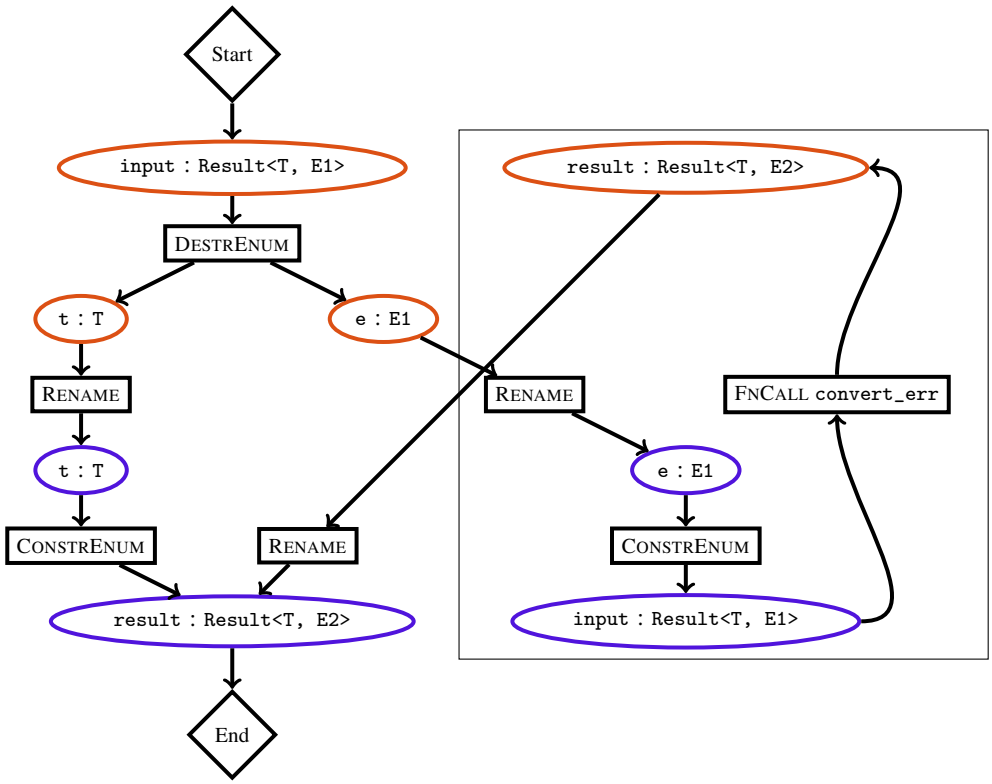


Figure 18: Synthesis graph for figure 14. The nodes that are generated from the external function call are enclosed in the rectangle.

```
enum List<T> {
    Next(Box<List<T>>),
    End(T),
}

fn get_end<T>(list: List<T>) -> T {
    let result = match list {
        List::Next(list) => {
            let list = *list;
            get_end(list)
        }
        End(t) => {
            t
        }
    };
    result
}
```

Figure 19: An example to illustrate recursive function calls.

4.8 Runtime Cost Heuristic

At any given time during the execution of the program search algorithm described in section 4.4, there may exist multiple incomplete flows that did not yet reach the end node. Each incomplete flow contains

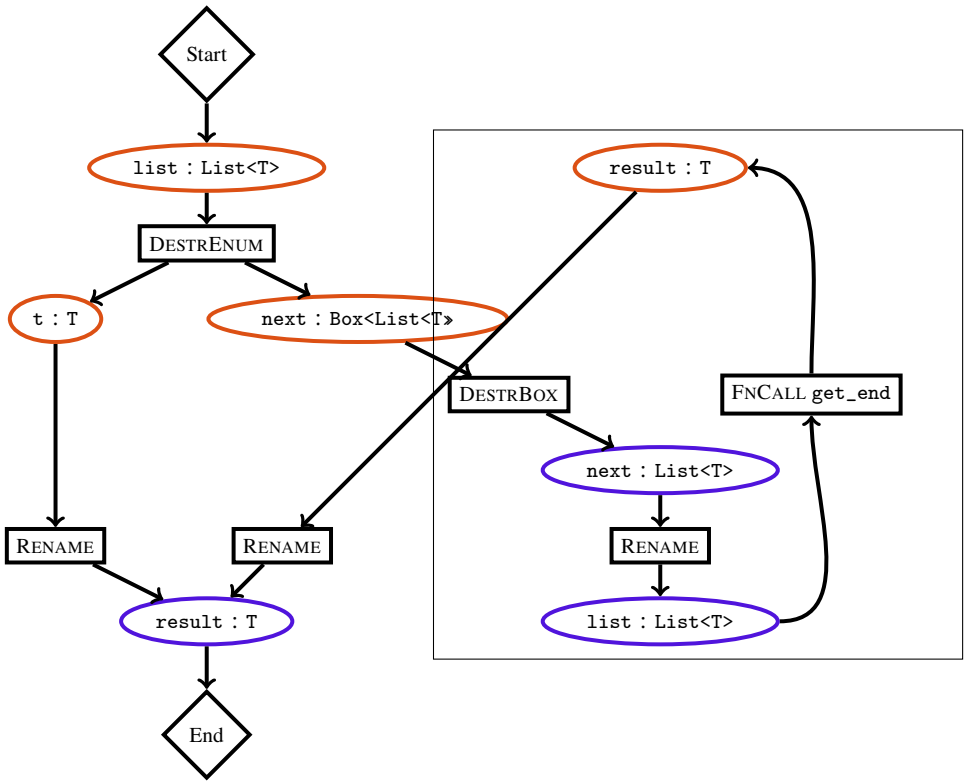


Figure 20: Synthesis graph for figure 27. The nodes that are generated from the recursive function call are enclosed in the rectangle.

a set of nodes and edges that it traverses. Based on the rule nodes that the flow traverses and the goal nodes that are connected to that rule node, it is possible to define various heuristics that assign a priority to each incomplete flow. The algorithm then picks the incomplete flow with the highest priority which is then extended by further traversing the synthesis graph. In general, the incomplete flow with the highest priority is the first that reaches the end node and gets completed. However, the priority of an incomplete flow is recomputed whenever the flow is extended with additional rule nodes and may thus change its priority during its construction.

Using this method of changing priorities for incomplete flows, various heuristics can be implemented. Concretely, we implement a *runtime cost heuristic* that prioritises synthesis solutions with the smallest runtime cost when run. Each rule generates a program statement that defines an estimate of its runtime cost, and the total runtime cost of a flow is the sum of the runtime cost of each rule.

$$\text{mov}(T) := \begin{cases} \text{sizeof}(T) & \text{if the size of } T \text{ is known,} \\ h_{\text{mov}} & \text{otherwise} \end{cases}$$

$$\text{alloc}(T) := h_{\text{alloc}}$$

$$\text{call}(f) := \begin{cases} h_{\text{call}} & \text{if } f \text{ is not a recursive function call,} \\ h_{\text{rec}} & \text{otherwise} \end{cases}$$

$$\text{drop}(T) := \begin{cases} h_{\text{call}} & \text{if } T \text{ is not a primitive,} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{where } h_{\text{mov}} < h_{\text{alloc}} < h_{\text{call}} < h_{\text{rec}}$$

The estimation of the runtime cost of a rule works by estimating the number of trivial operations emitted for the respective program statement that is generated from that rule. mov(T) estimates the runtime cost as the size of the type T in bytes if it is known and some value h_{mov} otherwise. alloc(T) is set to some high value h_{alloc} such that allocations are avoided if the size of T is smaller than h_{alloc}. call(f) calls the function f. As the function body of f is unknown to the algorithm, we estimate it to be some high value h_{call} similar to the allocation, and if f is a recursive function call we estimate it to be an even higher value h_{rec} as recursive function calls are generally very expensive. Finally, the cost drop(T) of dropping a non-primitive value of type T is estimated to be as expensive as an external function call. If the type T is primitive, we know that it has no drop implementation and the runtime cost is estimated as 0.

As a second example that contains allocations, consider the `singleton` function in figure 21 that creates a list from a single element of type `T`. There exist two solutions, the first solution in figure 22 just creates an empty list. It contains no allocations and just constructs an element `List::Nil` for which we estimate the runtime cost as mov(`List<T>`). Furthermore, `elem` is not used and thus dropped, which has the runtime cost drop(`T`).

The second solution in figure 23 constructs a list with a single element of type `T`. To achieve this, the first step is again to construct `List::Nil` for which we estimate the runtime cost as mov(`List<T>`), which is then boxed, where constructing a box has runtime cost alloc(`List<T>`). Constructing `List::Cons` is again estimated to have a runtime cost of mov(`List<T>`).

We can see that the runtime cost of the first solution is dominated by the call to the drop function, while the runtime cost of the second solution is dominated by the allocation. In general, we assume that calling the drop function is just as expensive as calling any unknown function, and thus more expensive than an allocation as the unknown function itself may allocate some data. This means that in this case, our runtime cost heuristic prefers the second solution where `elem` is not dropped. However, if we assume that `T` is a primitive that has no drop implementation, we can assume that dropping the primitive is cheaper than an allocation. In that case, the first solution is preferred.

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

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

Figure 21: A function that creates a list from a single element.

```
fn singleton(elem: T) -> List<T> { // drop(T)
    let result = List::Nil;        // mov(List<T>)
    result
}
```

Figure 22: First solution for figure 21.

```
fn singleton(elem: T) -> List<T> {
    let next = List::Nil;      // mov(List<T>)
    let next = Box::new(next); // alloc(List<T>)
    let result = List::Cons {  // mov(List<T>)
        elem,
        next,
    };
    result
}
```

Figure 23: Second solution for figure 21.

5 Evaluation

The new synthesizer that we call DAG solver was implemented in the Rust programming language itself. It uses Creusot's (Denis et al., 2022) annotations for functional specifications. All the benchmarks were done on a consumer-grade PC with an AMD Ryzen 7 5800 8-Core CPU and 16 GB of RAM running Fedora Linux. Our evaluation aims to answer the following three main questions:

(Q1) Does the new search algorithm support the same features as the SuSLik implementation?

(Q2) Can the new search algorithm compute the same number of solutions as the SuSLik implementation on the same subset of supported examples?

(Q3) Can the new search algorithm outperform the SuSLik implementation on the same subset of supported examples?

For the benchmarks, we have set a time limit of ten seconds per synthesis problem. As the DAG solver supports only a subset of the functionality that the SuSLik solver supports, we only include problems where both of the solvers find at least one solution. We list the total number of solutions found, the added time per solution and the time to the first solution. The time to the first solution measures the time difference from the invocation of the algorithm to the time when the first valid solution is found. This measure is especially useful in combination with a program search heuristic or with functional specifications where the first solution is usually the one that is desired. The added time per solution is the average of the added time it takes from one solution to compute the next solution, or the time from starting the algorithm until computing the first solution respectively. Intuitively, this is the average time a user needs to wait until they receive the next solution.

5.1 Top 100 Crates

This benchmark contains Rust function signatures extracted from the top 100 crates from the Rust community's crate registry. Signatures with unsupported types are excluded. Note that the function signatures extracted from the top 100 crates have no functional specifications. The goal of this benchmark is to evaluate the performance of our new program search algorithm on a wide range of real-world examples compared to the previous approach that uses the SuSLik solver. The benchmark results are shown in table 1.

Note that although we were able to extract 14'138 function signatures from the top 100 crates, we only consider the ones where both the SuSLik Solver and the DAG Solver find at least one solution. As our implementation at the moment only supports a small subset of rules (see section 5.4 for all of the limitations), the set of problems is reduced significantly to 224 function signatures. This explains the lower number of solutions found compared to the earlier work of Fiala et al. (2023).

5.2 Functional Specifications

As our program search algorithm supports functional specifications and the previous benchmark does not contain any, we also include examples with functional specifications extracted from different sources. It includes a collection of idiomatic rust programs that were taken from the previous work of Fiala et al.

Benchmark	Data Point Description	SuSLik Solver	DAG Solver No Heuristic	DAG Solver Runtime Cost
Top 100 Crates	Number of Solutions Found	853	313 (-63%)	313 (-63%)
	Time to First Solution	142 ms	3 ms (-98%)	4 ms (-97%)
	Added Time per Solution	58 ms	4 ms (-94%)	5 ms (-92%)

Table 1: Benchmark results of the functions from the top 100 crates.

(2023), a collection of synthesis tasks from the SuSLik benchmark suite (Itzhaky et al., 2021) and finally a collection of programs from the Prusti (Astrauskas et al., 2022) and Creusot (Denis et al., 2022) test suites. The benchmark results are shown in table 2.

Benchmark	Data Point Description	SuSLik Solver	DAG Solver No Heuristic	DAG Solver Runtime Cost
Functional Specifications	Number of Solutions Found	77	33 (-57%)	30 (-61%)
	Time to First Solution	111 ms	2 ms (-98%)	7 ms (-93%)
	Added Time per Solution	50 ms	8 ms (-82%)	4 ms (-91%)

Table 2: Benchmark results of the examples with functional specifications.

5.3 Custom Test Cases

Together with the top 100 crates and the examples with functional specifications, we also include a benchmark on the newly created test cases for the DAG Solver. These test cases contain a mix of diverse function signatures, including mutable references, functional specifications and recursive function calls to thoroughly test the new solver's implementation. The benchmark results are shown in table 3.

Benchmark	Data Point Description	SuSLik Solver	DAG Solver No Heuristic	DAG Solver Runtime Cost
Test Cases	Number of Solutions Found	321	170 (-47%)	170 (-47%)
	Time to First Solution	151 ms	4 ms (-97%)	3 ms (-98%)
	Added Time per Solution	77 ms	4 ms (-95%)	3 ms (-96%)

Table 3: Benchmark results of the custom test cases.

5.4 Limitations

At the moment, our synthesis rules allow only the synthesis of programs where any enum's variants contain at most one field. This is due to the fact that synthesis graph nodes can only utilise either the BRANCH traversal rule or the ALL traversal rule. An enum with multiple fields would need to utilise both of these rules simultaneously. The introduction of scopes as proposed in the following section 6.1 would eliminate this limitation entirely as the need for BRANCH and JOIN traversal rules would be eliminated.

Our implementation of external function calls supports at most one function call per synthesis result. Recall from section 4.6 that each external function is added once into the synthesis graph. At the mo-

Group	Feature	SuSLik Solver	DAG Solver
Data Structures	Structs	Yes	Yes
	Tuples	Yes	Yes
	Enums	Yes	Yes[1]
Primitives	Finding Concrete Values	Yes[2]	No
Mutable References	Writing	Yes	Yes
	Expiring	Yes	Yes
	Passing to External Functions	Yes	Yes
Function Calls	External Function Calls	Yes	Yes[3]
	Pure Functions in Specification	Yes	Yes[4]
Recursion	Recursion on the Target Function	Yes	Yes
	Synthesising Helper Functions	Yes	No
Solution Prioritisation	Based on Runtime Cost	No	Yes

Table 4: Overview over all supported features of both solvers.

ment, there exists no way to dynamically add more function calls during the expansion of the synthesis graph. Furthermore, the FnCall rule is unique in the sense that its edges are directed from the backward goals that represent its inputs to the forward goals that represent its output. The reversed direction may introduce cycles to the synthesis graph if multiple functions are used.

Contrary to the SuSLik search algorithm, our new algorithm does not yet support the synthesis of helper functions. However, recursive calls to the target function itself are supported as described in section 4.7.

Although there is support for annotating target functions with functional specifications, the synthesis using functional specifications is limited. Currently, concrete values for primitive types can not be found from the functional specifications. Although it can be proven that there exists some primitive value that satisfies the functional specification, the concrete value itself is not found and in the synthesised program code, a default value is inserted instead that can be later substituted by the user.

5.5 Summary

Table 5 summarises the three benchmarks from sections 5.1, 5.2 and 5.3. The new solver demonstrates significantly faster performance compared to the SuSLik solver, as indicated by the time taken to find the initial solution and the additional time required for each subsequent solution. On average, the time required to find the first solution is reduced by 97%, while the additional time per solution is reduced by 93% when employing the runtime cost heuristic.

We can also observe that depending on the heuristic used for the DAG solver, a different number of solutions are found. Internally, the DAG solver stores incomplete solution flows in a priority queue. During the execution of the algorithm, this priority queue may grow relatively large. To avoid filling the queue with too many incomplete flows, the incomplete flows with the least priority may be dropped after the queue reaches a certain size. As the priority changes with the heuristic that is used, this may

[1] Enum variants with one field.
[2] Only some specific values are tested.
[3] Only one external function call per solution is supported.
[4] Generics are not supported.

Benchmark	Data Point Description	SuSLik Solver	DAG Solver No Heuristic	DAG Solver Runtime Cost
Total	Number of Solutions Found	1251	516 (-59%)	513 (-59%)
	Time to First Solution	142 ms	3 ms (-98%)	4 ms (-97%)
	Added Time per Solution	62 ms	4 ms (-93%)	4 ms (-93%)

Table 5: Combined results of all the benchmarks.

lead to different incomplete flows being dropped and ultimately to a different number of total solutions that are found.

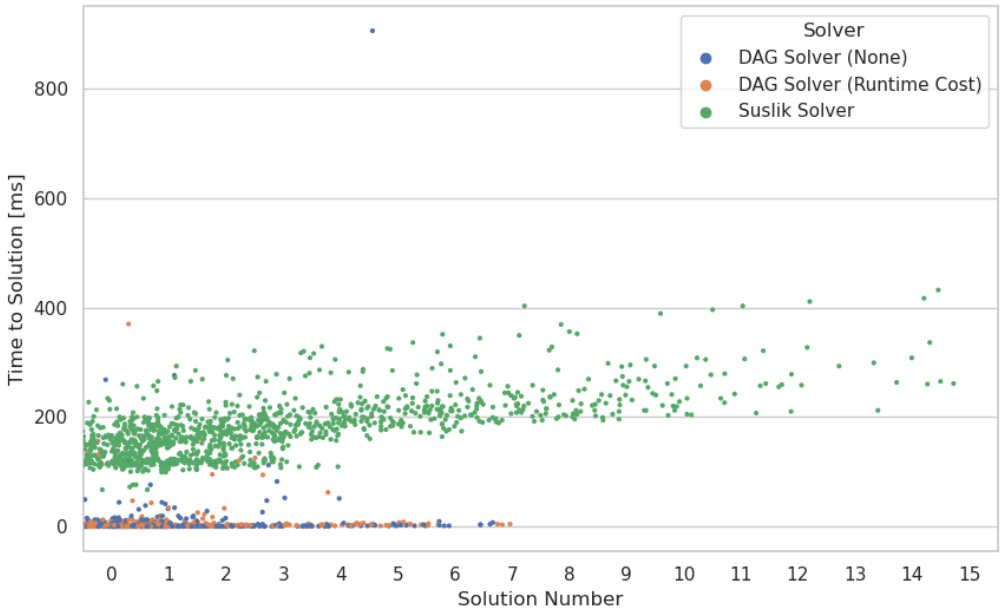


Figure 24: Combined results of all the benchmarks.

In figure 24 a dot is drawn for each solution that was found, and different solvers are highlighted with different colours. The x-axis tells us the solution number that indicates how many solutions were found for the same function signature beforehand. The y-axis shows the time that was needed to find the solution. In general, the new approach (DAG Solver) finds solutions much faster than the old approach (SuSLik Solver). However, there are a few outliers where the DAG solver takes considerably longer to find a solution. We have found that most of these outliers deal with mutable references. This is most likely due to the fact that the DropRef rule gives the program search algorithm the choice of either dropping the mutable reference from the start or only after applying a set of rules to it, for example writing to it. This causes the runtime of the algorithm to increase exponentially when dealing with multiple mutable references.

A further observation that can be made is that the longest time to solve a problem was 906 ms, despite setting the timeout to ten seconds. We ascribe this observation to the fact that the program synthesis algorithm has an exponential time complexity. However, we believe that it would be possible to find solutions to larger problems more efficiently by introducing a suitable solution prioritisation heuristic,

In figure 25, we divide the benchmark programs into three categories. *Functional Specs* contains all

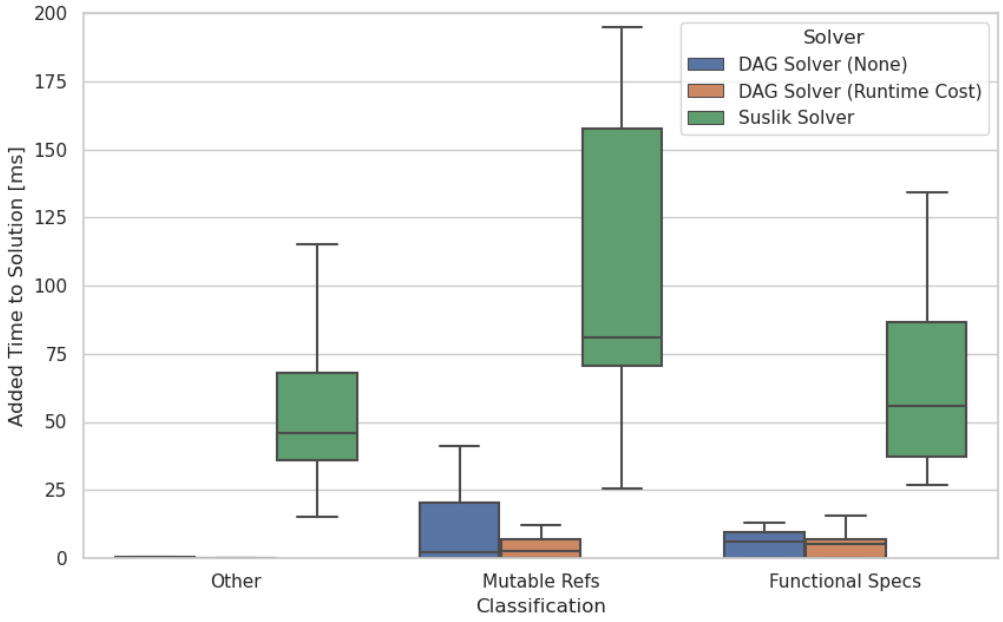


Figure 25: Combined results of all the benchmarks divided into categories.

the solutions where the target function has non-trivial functional specifications. *Mutable Refs* contains all the solutions where the target function deals with mutable references. Finally, *Other* contains all the remaining solutions. As expected, we can clearly see that the DAG solver is the fastest when used on target functions that contain neither mutable references nor functional specifications. When dealing with mutable references and functional specifications, the time to find solutions increases a bit, although still being faster than the SuSLik solver. When solving functional specifications, the advantage of the DAG solver compared to the SuSLik solver decreases the most as both solvers have to invoke the SAT solver to prove whether the functional specifications are satisfied. We can also observe that using the runtime heuristic for the DAG solver seems especially advantageous when dealing with mutable references compared to using no heuristic.

To answer the first question of our evaluation **(Q1)**, we take a look at table 4 that lists all supported features of both solvers. Most of the features supported by the SuSLik solver are also supported by the DAG solver, except for the synthesis of recursive helper functions and the insertion of concrete primitive values that satisfy the functional specifications. At the moment, our solver can decide whether there exists a primitive value that satisfies the functional specification, but it can not yet find the concrete value and insert it. This is due to a limitation of the interaction with the SMT solver. Enum variants can contain exactly one field, however, by using a tuple within that field one can easily emulate the behaviour of enum variants with multiple fields. Additionally, our solver introduces the runtime cost heuristic feature that can prioritise certain solutions over others. This is not possible with the SuSLik solver.

To answer the second question **(Q2)**, the number of solutions found suggests that the SuSLik solver finds a lot more solutions than the DAG solver for the top 100 crates benchmark. However, we note that the SuSLik solver also generates programs that include statements without any effect, for example constructing objects that are dropped immediately or calling functions whose return value is never used. The DAG solver on the other hand discards programs that contain effect-free statements. Furthermore,

the SuSLik solver may return the structurally equivalent solution multiple times, but with different primitive values inserted. In contrast, the DAG solver just uses the one default value defined on a primitive type.

To answer the final question **(Q3)**, the time to the first solution and the added time per solution show that the new solver is much faster compared to the SuSLik solver. The time to find the first solution is lowered by 97% on average, while the added time per solution is also lowered by 93% when using the runtime cost heuristic. Some small part of this significant speedup can be explained from a more efficient implementation and the difference in the programming languages used. Our solver was implemented in Rust, while the SuSLik solver is implemented in Scala, which adds the overhead of more heap allocations, garbage collection and others.

As we have discussed earlier in section 3, our new synthesis algorithm had the goal of significantly reducing the number of rules that are applied in the synthesis process. If we try to estimate the number of search steps saved with the changes implemented, we can make a rough estimate of 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. As this approach could be implemented as discussed for the new DAG solver, we can be certain that this algorithmic difference is the main explanation for the observed speedup.

6 Future Work

6.1 Introducing Scoped Synthetic Ownership Logic

With the previous tree-like search, keeping track of the scope in which a variable is defined was not needed. With our novel approach of having a forward and a backward search tree that are combined into a search graph, keeping track of scopes becomes a necessity to avoid synthesising malformed programs that use program variables in invalid scopes. With our attempt, scopes were kept track of during the execution of the program search algorithm (see section 4.4) with the use of BRANCH and JOIN traversal rules. However, it should also be possible to create a scope-aware program state representation such that variable scopes are respected even during the creation of the synthesis graph. For this end, we propose the introduction of *scopes* for the SOL inference rules, creating an extension of SOL that we call Scoped Synthetic Ownership Logic (SSOL).

Recall the definition of scopes from section 4.4. Synthesis goals are slightly modified to also contain a scope. We annotate program variables with their respective scopes s with the subscript notation $\mathtt{x}_s : T$. Consequently, some of the synthesis rules also need to be adapted to take into account the new logic.

In particular, rules that previously used the BRANCH traversal rule such as DESTRENUM need to be modified to keep track of scopes. The necessary modifications are shown in figure 26. We further introduce two new rules, MOVE and JOIN. The JOIN rule joins together branches that were created by rules such as DESTRENUM and thus replaces the JOIN traversal rule that was introduced with the program search algorithm in section 4.4. The MOVE rule is used to move a program variable from an outer scope into an inner scope. Note that both of these rules are now enforced directly during the construction of the synthesis graph and not only during the program search. This should not only enable much easier implementation of the program search algorithm but will most likely further increase the performance of the program search significantly.

Note that the DESTRENUM and DESTRBORROWENUM rules now not only change the contents of the heap cell but also change the scope by adding a branch point. The RENAME match rule now only matches variables in the same scope. The DROP rule can drop variables from any scope. Furthermore, two new rules are introduced that act on the scope. The first rule JOIN acts as a join point for rules that introduce branch points. The second rule MOVE takes a value and moves it into the arms of a branch point, but unlike DESTRENUM or DESTRBORROWENUM, it does not modify the value itself. The remaining rules that we do not discuss here are assumed to only act within the same scope.

To demonstrate how the SSOL rule change the synthesis graph compared to the usual SOL rules, we take a look at a new example in figure 27. The synthesis graph in figure 28a is constructed with the usual SOL rules. On the other hand, the second synthesis graph in figure 28b is constructed using the newly introduced SSOL rules. Note that when using the SSOL rules, the goals originating from the DESTRENUM rule have distinct scopes u and v. Thus, they can't be matched with the goals from the CONSTRSTRUCT rule that both expect the same scope. With the first approach, it is only clear that no valid solution exists after the program search algorithm is executed. With the SSOL rules, it is clear from the synthesis graph alone that there exists no valid program as the forward tree is not connected to the backward tree at all.

$$\textsc{DestrEnum}\ \frac{T = (T_0)^{V_0} + \cdots + (T_n)^{V_n} \qquad b \text{ fresh} \qquad \text{arms}(b) = \{a_0, \ldots, a_n\} \qquad s_i := \text{push}(s, b, a_i) \qquad g_i := \{(\mathtt{f}_i)_{s_i} : T_i(v_i)\}}{\{\mathtt{x}_s : T(\{\delta : V_i, \mathtt{f}_i : v_i\})\}\ \mathtt{match\ x\ \{}\ V_0(\mathtt{f}_0)\ \mathtt{=>}\ g_0, \ldots V_n(\mathtt{f}_n)\ \mathtt{=>}\ g_n,\ \mathtt{\}}}$$

$$\textsc{DestrBorrowEnum}\ \frac{T = (T_0)^{V_0} + \cdots + (T_n)^{V_n} \qquad {}'a_0, \ldots, {}'a_n \text{ fresh} \qquad b \text{ fresh} \qquad \text{arms}(b) = \{a_0, \ldots, a_n\} \qquad s_i := \text{push}(s, b, a_i) \qquad g_i := \{(\mathtt{f}_i)_{s_i} \overset{{}'a_i}{\longmapsto} T_i(v_i)\}}{\{\mathtt{x}_s \mapsto T(\{\delta : V_i, \mathtt{f}_i : v_i\})\}\ \begin{array}{l}\mathtt{match\ x\ \{}\ V_0(\mathtt{f}_0)\ \mathtt{=>}\ g_0, \ldots, \\ \qquad V_n(\mathtt{f}_n)\ \mathtt{=>}\ g_n\ \mathtt{\}}\end{array}\ \{\mathtt{x}^{\{{}'a_0, \ldots, {}'a_n\}} \mapsto T\}}$$

$$\textsc{Move}\ \frac{\text{arms}(b) = \{a_0, \ldots, a_n\} \qquad s_i := \text{push}(s, b, a_i) \qquad g_i := \{(\mathtt{f}_i)_{s_i} : T_i(v_i)\}}{\{\mathtt{x}_s : T(\{\delta : V_i, \mathtt{f}_i : v_i\})\}\ \mathtt{skip!()}\ g_0, \ldots, g_n}$$

$$\textsc{Join}\ \frac{\text{arms}(b) = \{a_0, \ldots, a_n\} \qquad s := \text{pop}(s_0) = \cdots = \text{pop}(s_n) \qquad g_i := \{\phi_i \mid \mathtt{x}_{s_i} : T(v_i)\}}{g_0, \ldots, g_n\ \mathtt{skip!()}\ \begin{array}{l}\{(\text{cond}(a_0) \implies (\phi_0 \wedge v = v_0)) \wedge \cdots \\ \qquad \wedge (\text{cond}(a_n) \implies (\phi_n \wedge v = v_n)) \mid \mathtt{x}_s : T(v)\}\end{array}}$$

$$\textsc{Rename}\ \frac{T \text{ not ref}}{\{\mathtt{x}_s : T(v)\}\ \mathtt{let\ y\ =\ x}\ \{\mathtt{y}_s : T(v)\}}$$

Figure 26: Newly introduced or modified rules in SSOL.

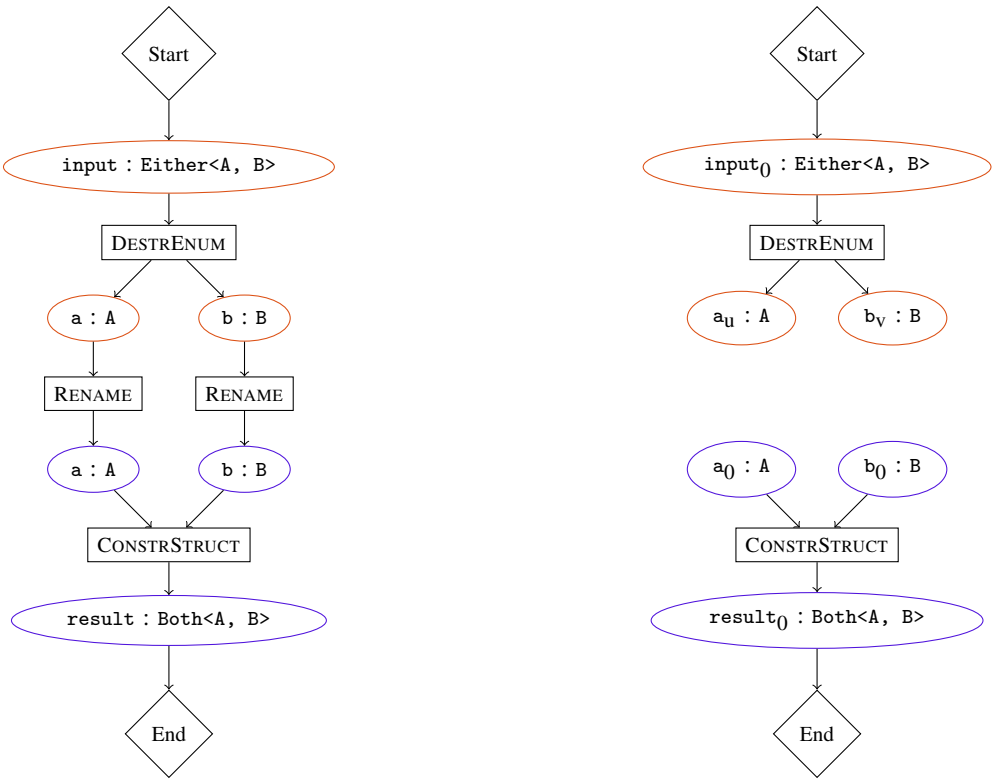
```
struct Both<A, B>(A, B);

enum Either<A, B> {
    A(A),
    B(B),
}

fn either_to_both<A, B>(input: Either<A, B>) -> Both<A, B> {
    todo!();
    result
}
```

Figure 27: A partially complete function body.

6.2 Partially Complete Programs for Better Synthesis Guesses

We use functional specifications to reduce the set of solution programs. However, the vast majority of programmers in practice do not annotate functions with functional specifications. We propose an alternative way to restrict the set of solution programs to better capture the intentions of the programmer.

Recall that the synthesizer builds a synthesis graph with rule nodes and goal nodes. A valid program for the type specification corresponds to a flow through this synthesis graph. Assume that instead of



(a) Using SOL rules

(b) Using SSOL rules

Figure 28: Synthesis graphs for figure 27.

an empty function body, we have a partially complete function body as seen in figure 29. The `todo!()` denotes the remaining function body that needs to be synthesised. The first statement of the function body constrains the set of valid flows to those that first travel through a DestrBorrowStruct rule. The remaining flows that do not travel through a DestrBorrowStruct rule can be discarded as they do not satisfy the constraints given by the partial function body. If further disallow statements that are effect-free, for example destructing rules where none of the fields are used, this leaves only one possible solution program that writes to the content of the wrapper by replacing the `todo!()` with `*inner = val`.

```
fn unwrap_write<T>(mut_ref: &mut Wrapper<T>, val: T) {
    let Wrapper(inner) = mut_ref;
    todo!();
}
```

Figure 29: A partially complete function.

6.3 Generative AI Guided Search

Large language models (LLM) have proven themselves useful to quickly generate whole programs or auto-complete computer programs. However, this method of generating programs has two major disadvantages. First, these models are commonly not aware of the underlying type system rules and occasionally suggest programs that do not compile. Second, they do not consider functional specifications and thus may generate programs that do not satisfy the requirements.

On the other hand, our approach has the advantage that it respects the target language's type system and annotated functional specifications. We have also seen in section 4.8 that it is possible to guide the program search algorithm by introducing some heuristics. We propose combining both of the advantages of these different approaches for program generation by adding a heuristic based on an LLM. Based on previously seen data, the LLM would be able to efficiently prioritise the synthesis of the most "useful" program, while the fact that the synthesis still uses the underlying synthesis graph traversal guarantees that the solution program satisfies type and functional specifications.

6.4 Parallelising the Program Search Algorithm

The current implementation of the program search algorithm described in section 4.4 can in principle be easily parallelised. However, some of the data types that are used by the Rust compiler are not usable in a multi-threaded environment. There is an effort to make the Rust compiler parallel that naturally uses thread-safe data structures, but is currently unstable and did not work with our approach of multi-threading. The idea of parallelising the search algorithm can either be revisited at a later stage when the Rust compiler is multi-threaded, or the search algorithm itself can be rewritten in a way that it does not rely on the Rust compiler's data structures in the first place.

6.5 Code Editor Support

To further allow a better adaptation of the code synthesizer, a plugin for code editors can be made available. The plugin either scans rust code for `todo!()`'s and provides appropriate solutions, or provides solutions on the fly while typing. Together with the proposal for synthesising partially complete programs from section 6.2, the code synthesizer becomes a feasible tool to speed up development times in practice.

7 Conclusion

The goal of this work was to introduce a new proof search framework optimised for SOL to increase the performance of the synthesis process for Rust programs by replacing the SuSLik's proof search algorithm in the RuSSOL implementation.

The new search algorithm takes advantage of the design of SOL derivation rules. By splitting the search space into two separate search trees, one for forward rules and one for backward rules, we can explore derivations starting from the original precondition and postcondition separately. The search algorithm also separates preconditions and postconditions into independent heap cells such that we avoid the application of rules in different orders. In the final step, the forward and backward trees are matched to obtain the synthesis graph of a problem that can then be traversed to obtain valid solution programs for the given functional and type specification.

By implementing these changes, our new search algorithm significantly improves execution time by minimising the number of synthesis rules that are applied. We found that the new DAG solver outperformed the SuSLik solver significantly. The time taken to find the first solution was reduced by an average of 97%, and the additional time per solution was decreased by 93%. We attribute this considerable speedup mainly to the algorithmic difference achieved by transitioning from a single search tree to separated forward and backward search trees.

Our evaluation also showed that the new solver supports most of the features provided by the SuSLik solver, with the main exceptions of recursive helper functions and the insertion of concrete primitive values satisfying functional specifications.

Overall, the combination of Rust's strong type system and the novel program search introduced by our new search algorithm, as well as the ability to introduce heuristics for solution prioritisation and the proposed changes for a future version are very promising for making the synthesis of Rust programs more efficient.

References

Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., and Summers, A. J. (2022). The prusti project: Formal verification for rust. In Deshmukh, J. V., Havelund, K., and Perez, I., editors, *NASA Formal Methods*, pages 88–108, Cham. Springer International Publishing.

Denis, X., Jourdan, J.-H., and Marché, C. (2022). Creusot: A foundry for the deductive verification of rust programs. In Riesco, A. and Zhang, M., editors, *Formal Methods and Software Engineering*, pages 90–105, Cham. Springer International Publishing.

Fiala, J., Itzhaky, S., Müller, P., Polikarpova, N., and Sergey, I. (2023). Leveraging rust types for program synthesis. *Proc. ACM Program. Lang.*, 7(PLDI).

Itzhaky, S., Peleg, H., Polikarpova, N., Rowe, R. N. S., and Sergey, I. (2021). Deductive synthesis of programs with pointers: Techniques, challenges, opportunities. In Silva, A. and Leino, K. R. M., editors, *Computer Aided Verification*, pages 110–134, Cham. Springer International Publishing.

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).

Reynolds, J. (2002). Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74.