Realizability in Semantics-Guided Synthesis Done Eagerly

ROLAND MEYER, TU Braunschweig, Germany
JAKOB TEPE, TU Braunschweig, Germany
SEBASTIAN WOLFF, New York University, USA

We present realizability and realization logic, two program logics that jointly address the problem of finding solutions in semantics-guided synthesis. What is new is that we proceed eagerly and not only analyze a single candidate program but a whole set. Realizability logic computes information about the set of candidate programs in a forward fashion. Realization logic uses this information as guidance to identify a suitable candidate in a backward fashion. Realizability logic is able to analyze a set of programs due to a new form of assertions that tracks synthesis alternatives. Realizability logic then picks alternatives to arrive at a program, and we give the guarantee that this process will not need backtracking. We show how to implement the program logics using verification conditions, and report on experiments with a prototype in the context of safe memory reclamation for lock-free data structures.

1 INTRODUCTION

The syntax-guided synthesis (SyGuS) initiative [2, 3] has been instrumental in pushing the development of program synthesis technology. Key to this success has been the definition of a standardized input format that is *solver independent*: the format only refers to the synthesis task but does not constrain the synthesis technology. This means every SyGuS solver can, in principle, be applied to the entire SyGuS benchmark set, and the community can focus on comparing and improving the synthesis technology. A SyGuS task consists of a specification of the desired program behavior and a grammar for the programs that may be chosen. The inherent limitation of SyGuS is that the grammar should refer to SMT expressions, expressions from logical theories that are supported by SMT solvers. SyGuS cannot handle expressions that fall outside the known logical theories, and for the expressions it can handle it assumes the standard semantics.

Semantics-guided synthesis (SemGuS) [23] has recently been proposed as a successor of SyGuS that is meant to overcome the dependence on SMT expressions and describe synthesis problems in a *domain-independent* way. The key to domain independence is to add a third parameter to the definition of the synthesis problem, namely a definition of the program semantics. Giving the user the ability to define the program semantics dramatically increases the reach of SemGuS over SyGuS. The user can now define loop constructs and synthesize programs with complex control flow. In fact, synthesizing full programs rather than logical expressions has been one of the goals behind SemGuS. It is a goal that SemGuS shares with solver-aided languages like Rosette [42] or Sketch [39]. Note, however, that Rosette and Sketch are neither solver nor domain independent. Rosette will not be able to read Sketch input and vice versa, and both can only draw conclusions about the standard semantics of the language. SemGuS allows the user to provide approximate semantics, and applications abound.

Giving the user the ability to define the program semantics also dramatically increases the computational effort of solving SemGuS tasks over SyGuS tasks. To come up with efficient synthesis algorithms, it has turned out helpful to decompose the problem and develop algorithms dedicated to proving synthesis tasks unrealizable and algorithms for synthesizing solutions. For unrealizability, the state-of-the-art approach, implemented in the tool Messy [23] and its SyGuS precursors nay [19] and nope [18], is *unrealizability logic* [22]. Unrealizability logic tries to prove Hoare triples of the form $\{r\}\mathsf{N}\{s\}$. The novelty is that N denotes a set of programs, and by proving the triple one shows that all programs in this set satisfy the pre-post specification. The pre-post specification is set-up in a way that validity of the triple means no program in the set can solve the SemGuS task at hand, hence the name unrealizability.

Our contribution is a new algorithm for realizability, for synthesizing solutions to SemGuS tasks. The state-of-the-art tool for realizability, called Messy-Enum [23], implements the CEGIS algorithm [40] algorithm for search-based synthesis [3]. Guided by knowledge about failed synthesis attempts, Messy-Enum iteratively constructs candidate programs and checks every single one of them for validity, i.e., for whether it solves the synthesis task. There is, however, an important difference between SyGuS and SemGuS. In SyGuS, checking the validity of a candidate program is cheap, it is an SMT query. In SemGuS, checking the validity of a candidate program is expensive: Messy-Enum reduces it to a constraint Horn clause (CHC) query, and solving them may turn out as hard as conducting a full verification run. This is not unexpected, after all SemGuS aims to synthesize full programs. But it means that search-based solvers have no chance to scale without new strategies to reduce (i) the number of iterations and (ii) the cost of each iteration.

Our algorithm reduces the number of iterations by analyzing sets of candidate programs for validity, in an *eager* fashion, rather than a single one. What makes this scale is compositionality. We construct the sets of programs bottom-up, starting from sets of commands to sets of more and more complex programs. The key idea is to abstract the sets of programs to their input-output behavior. Abstracting single programs to their input-output behavior to compute over equivalence classes is an idea that has been widely used in synthesis [1, 15, 36, 44, 46]. The novelty in our work is eagerness: we abstract sets of programs whose input-output behavior does not match. Compositionality requires that the input-output is rich enough (i) to only reason with this abstraction, without having to resort to the underlying programs, and (ii) to answer the realizability problem.

We develop this idea in a new program logic [17]. Our *realizability logic* reasons over triples of the form $\langle R\rangle P\langle S\rangle$, where P is a set of programs whose input-output behavior is given in the form of a precondition R and a postcondition S. What is new is that the pre- and postcondition are not single predicates, but sets of predicates. This reflects the fact that the programs in P are synthesis alternatives. To achieve compositionality, it is important to get the notion of validity right. Given $\langle R\rangle P\langle S\rangle$ and $\langle S\rangle Q\langle T\rangle$, we want to be able to conclude $\langle R\rangle P; Q\langle T\rangle$, where $P; Q$ contains all programs $\texttt{prog}; \texttt{prog}'$ with $\texttt{prog} \in P$ and $\texttt{prog}' \in Q$. The right choice is to reason backwards and define validity

$$\models_a \langle R\rangle P\langle S\rangle \quad \text{by} \quad \forall s \in S.\, \exists r \in R.\, \exists \texttt{prog} \in P.\ \models_d \{r\}\texttt{prog}\{s\}\ .$$

It is worth contrasting this definition with the notion of validity in the recent unrealizability logic [22]. Their triples $\{r\}P\{s\}$ are valid, if $\models_d \{r\}\texttt{prog}\{s\}$ holds for all programs $\texttt{prog} \in P$. Moreover, note that r and s are single predicates, and not sets of predicates as in our case. In fact, the paper explicitly asks for a realizability analogue of their unrealizability logic, and we named our program logic after that proposal.

The relation $\models_d \{r\}\texttt{prog}\{s\}$ is validity in classical Hoare logic. The index d stands for *demonic*, and indicates that the choice of the initial state in r as well as the choice of the execution in program prog are made demonically, and the postcondition s has to over-approximate all the resulting states. The index a for validity $\models_a \langle R\rangle P\langle S\rangle$ in our synthesis logic stands for *angelic*, and indicates that the choice of the predicate r in the precondition R as well as the choice of the program $\texttt{prog} \in P$ are made *angelically*, and the postcondition S is an under-approximation of all predicates s that can be guaranteed. Assertions in our program logic are thus angelic choices over predicates, and these predicates are demonic choices over states. To the best of our knowledge, there is no program logic that would reason over such alternations in related work.

Consider the program proof given in Figure 1. The program has two Boolean variables x and y and two non-terminals, M that may be rewritten to x=0 or x=1 and N that may be rewritten to y=0 or y=1. Starting from the set of all states, represented by the singleton predicate *true*, the two programs represented by M give us two synthesis alternatives: we can guarantee $\texttt{x} = 0$ or we can guarantee $\texttt{x} = 1$. In both cases, we still do not have any knowledge about the value of y. We do

```
1 ⟨true⟩
2 M(⟨true⟩x = 0⟨x = 0⟩ |
3   ⟨true⟩x = 1⟨x = 1⟩);
4 ⟨x = 0, x = 1⟩
5 N(⟨x = 0, x = 1⟩y = 0⟨x = 0 ∧ y = 0, x = 1 ∧ y = 0⟩ |
6   ⟨x = 0, x = 1⟩y = 1⟨x = 0 ∧ y = 1, x = 1 ∧ y = 1⟩);
7 ⟨x = 0 ∧ y = 0, . . . , x = 1 ∧ y = 1⟩
8 assert(x = 1 ∧ y = 1)
9 ⟨x = 1 ∧ y = 1, fail⟩
```

Fig. 1. Proof outline in realizability logic.

```
1 ⟨true⟩
2 M(⟨true⟩x = 0⟨x = 0⟩ |
3   ⟨true⟩x = 1⟨x = 1⟩);
4 ⟨x = 1⟩
5 N(⟨x = 0, x = 1⟩y = 0⟨x = 1 ∧ y = 1⟩ |
6   ⟨x = 1⟩y = 1⟨x = 1 ∧ y = 1⟩);
7 ⟨x = 1 ∧ y = 1⟩
8 assert(x = 1 ∧ y = 1)
9 ⟨x = 1 ∧ y = 1⟩
```

Fig. 2. Proof outline derived from Figure 1.

not yet make a decision, but use the two alternatives to consider the programs that can be derived from N. This yields four synthesis alternatives, namely x=0 ∧ y=0 to x=1 ∧ y=1. Only the last of these alternatives passes the assertion, the others lead to a failure of the execution.

A first point of criticism one may have about realizability logic is that the alternation between angelic and demonic choice will explode too quickly. In the end, all we have done is to replace the choice of a program by a powerset construction over predicates, a technique pioneered in automata theory [37]. This merely translates a complex search into complex assertions. We argue that, for SemGuS where validity checks are expensive, the assertions may be the right place to keep the complexity. Formal methods has developed expressive logical languages and abstract domains that can denote complex sets of states with very concise assertions. In our experiments, we have worked with Cartesian abstraction [8, Chapter 9].

A more severe point of criticism is that our realizability triples $\langle R\rangle P\langle S\rangle$ drop the relationship between the single programs $\texttt{prog} \in P$ and the pre- and postconditions $r \in R$ and $s \in S$ that this program can achieve. Hence, when we have given a proof in realizability logic, like the one in Figure 1, all we know is that the program sketch M; N; assert(x = 1 ∧ y = 1) can be completed to a program that passes the assertion, but *we do not know the program*. What we have, however, is a proof outline in realizability logic that annotates the program with intermediary assertions. The idea is to use this proof outline as guidance of how to instantiate the non-terminals.

Our second contribution is *realization logic*, a program logic to derive rewriting steps $\texttt{po} \vdash \texttt{po}'$ between proof outlines in realizability logic. The guarantee given by realization logic is that valid proof outlines are rewritten to valid proof outlines. The rewriting process will not only eliminate alternatives from the definition of non-terminals until a program that satisfies the specification has been found. An equally important step is to eliminate predicates that will not be helpful to satisfy the desired postcondition, or may even fail in the future. The elimination of predicates is done backwards, starting from the postcondition, and we illustrate it on our example:

$$\langle \texttt{x} = 0 \wedge \texttt{y} = 0, \ldots, \texttt{x} = 1 \wedge \texttt{y} = 1\rangle \texttt{assert(x} = 1 \wedge \texttt{y} = 1\texttt{)} \langle \texttt{x} = 1 \wedge \texttt{y} = 1, \textit{fail}\rangle \tag{1}$$

$$\vdash \langle \texttt{x} = 0 \wedge \texttt{y} = 0, \ldots, \texttt{x} = 1 \wedge \texttt{y} = 1\rangle \texttt{assert(x} = 1 \wedge \texttt{y} = 1\texttt{)} \langle \texttt{x} = 1 \wedge \texttt{y} = 1\rangle \tag{2}$$

$$\vdash \langle \texttt{x} = 1 \wedge \texttt{y} = 1\rangle \texttt{assert(x} = 1 \wedge \texttt{y} = 1\texttt{)} \langle \texttt{x} = 1 \wedge \texttt{y} = 1\rangle\ .$$

The first step drops the alternative *fail*. As there are less synthesis options to choose from, this weakens the postcondition and the step is thus sound by the standard rule of consequence in Hoare logic. The second step propagates this elimination backwards by *weakening* the precondition, which is rather uncommon in program logics. We can weaken the precondition, because x = 1 ∧ y = 1 is sufficient to obtain the postcondition. For the following rewriting steps, it will be helpful to consider Figure 2. We propagate the precondition x = 1 ∧ y = 1 to the postconditions of the alternatives for non-terminal N. Note that the proof outline remains valid although the alternative

$\{x = 0, x = 1\}$`y=0`$\{x = 1 \wedge y = 1\}$ is invalid. The point is that there is still the alternative `y=1`. We eliminate the incorrect alternative in the next rewriting step. The backwards reasoning thus revealed that the alternative `y=0` will not help to satisfy the specification. We then weaken the precondition of the alternative `y=1` and obtain the proof outline shown in the figure. The logic will proceed along the same lines and identify the program `x=1; y=1; assert(x = 1 ∧ y = 1)` as a solution to the synthesis task.

Our third contribution is an algorithm to construct valid proof outlines in realizability logic. The ambition is to avoid expensive CHC queries and get away with standard SMT queries, very much like SyGuS. To achieve this, we are willing to rely on user guidance, namely the loop invariants that are known to be crucial for verification. Our algorithm is then a generalization of deductive verification to realizability logic. We start from a realizability triple whose program sketch comes with invariant annotations. We show how to compute the missing intermediary assertions with the help of strongest postconditions. From the resulting proof outline, we compute verification conditions: a set of constraints whose validity entails the validity of the proof outline (and the annotations). Given that our assertions are new, we had to adapt the strongest postconditions and the verification conditions. The new definitions are made such that they can be handled by standard technology: the postconditions can be computed in a symbolic way and the verification conditions can be discharged by an SMT solver.

Our next contribution is to automate realization logic: we give an algorithm that rewrites proof outlines until a solution to the synthesis problem has been found. Our algorithm is again based on deductive verification and solver support. The idea is to derive from the given proof outline (ordinary) verification conditions (over predicates rather than sets of predicates) whose validity proves that a certain program is a solution to the synthesis problem. What is new is that the verification condition checks have to be interleaved with their construction, because they control the choice of the program and thus the future verification conditions. The choice of verification conditions is not unique but depends on the program as much as the argument why this program solves the synthesis problem. Interestingly, we can show that there is a *backtracking freedom* guarantee: no matter the choice, the synthesis is guaranteed to succeed. At a high level, this is a consequence of the notion of validity in realzability logic.

We implemented our algorithms for realizability and realization logic in a new tool. Given that we have not yet developed generic assertion languages to deal with sets of sets of predicates, our implementation is tied to one application domain: memory management in lock-free data structures with the help of a safe memory reclamation algorithm. Due to the lock-free processing, protecting a memory cell is not a mere call to the safe memory reclamation algorithm, but a complicated sequence of calls followed by checks of global invariants that indicate whether a call has been successful. The data structures include tricky ones like the ORVYY set and the DGLM queue, the challenging protection is with ordered hazard pointers (the first has priority over the second), but we can also handle epochs. This set of case studies is interesting in several respects. It works over an abstract data domain, namely the SMR types introduced in [27], and therefore cannot be handled by solver-aided languages out-of-the-box (an encoding would be possible). The programs contain unbounded loops, and therefore cannot be handled by SyGuS solvers. The places in which to synthesize information are far apart, which means we have to capture the influence of complex code on the to-be-synthesized information. Our approach handles all instances in a matter of seconds. Moreover, we did not need any user annotations but were able to infer the required loop invariants automatically.

It may have become clear that we develop our algorithms in the context of an imperative programming language that is parameterized in the data domain, the set of commands, and the semantics of commands. We fix the semantics of choice, sequential composition, and Kleene star.

In its original formulation, SemGuS is more liberal and would also take the operators for building programs as a parameter. To generalize our work to that setting, the user would have to come up with appropriate proof rules for the new operators. This is also the approach taken by unrealizability logic [22]. We believe, however, that this should only be a second step. The parameterization we work with is the standard assumption in program logics [5], and it has proven rich enough to handle a large variety of benchmarks. In fact, it is rich enough to handle all examples that are typically given to motivate SemGuS (C, Python, regular expressions, and bounded loops).

To sum up, our contributions are the following.

- Realizability logic (Section 3), the first program logic to reason about realizability in SemGuS. The new idea is to have sets of predicates as assertions to represent synthesis alternatives.
- Realization logic (Section 4), the first program logic to compute a solution to a SemGuS problem from a proof outline in realizability logic. The key idea of realization logic is to propagate failing synthesis attempts backwards, and eliminate unsuitable alternatives.
- An algorithm to find proofs in realizability logic (Section 5). We use deductive verification and develop appropriate strongest postconditions and verification conditions.
- An algorithm to find program derivations in realization logic (Section 6). Also here we rely on deductive verification, and we can give a backtracking freedom guarantee.
- We implemented our algorithms and applied them to synthesize the protection of memory accesses in a lock-free data structure by a safe memory reclamation algorithm (Section 8). The synthesis works over an abstract semantics of protection types, and the problem cannot be handled by state-of-the-art synthesizers.

We start with a recapitulation of program logic, which forms the foundation of our work.

2 PRELIMINARIES

We introduce an imperative programming language that is parameterized in the data domain and in the set of commands, and give a Hoare logic for it that will form the basis of our development.

2.1 Parameters

The development in this paper is parameterized in a triple $(\mathit{States}, \mathtt{COM}, [\![-]\!])$ which is given as part of the synthesis task. The set *States* is the data domain, the set of states programs operate over. There are no requirements, the set may be finite or infinite. The set COM contains the commands com that can be used in programs. The function $[\![-]\!] : \mathtt{COM} \to \mathit{States} \to \mathit{Predicates}$ assigns to each command a function from states to the set of predicates we define in a moment.

2.2 Programs

We use a classical while-language whose commands stem from the given set COM. Apart from that, we have sequential composition, choice, and Kleene star:

$$\mathtt{prog} \quad ::= \quad \mathtt{com} \mid \mathtt{prog}; \mathtt{prog} \mid \mathtt{prog} + \mathtt{prog} \mid \mathtt{prog}^* \,.$$

We use Programs for the set of all programs. We use Execs for the set of all executions ex, programs that neither contain choices nor loops. The function *exec* yields all executions that can originate from a given program. For example, $\mathit{exec}((\mathtt{com}_1 + \mathtt{com}_2); \mathtt{com})$ is the set $\{\mathtt{com}_1; \mathtt{com}, \mathtt{com}_2; \mathtt{com}\}$. Loops are unrolled. Formally, we see the program as a regular expression and take the language.

To give the program semantics, we first define $\mathit{Predicates} = \mathcal{P}(\mathit{States}) \cup \{\mathit{fail}\}$. Predicates are sets of states or a distinguished element *fail*, and we typically use r, s, t for predicates. We have sets of states to support non-determinism, and failure to track program crashes, due to problems like segfaults or failing assertions. The predicates form a complete lattice with inclusion as the

ordering and *fail* as the top element: $r \preceq_d s$ is defined by $s = \mathit{fail} \vee (r \neq \mathit{fail} \wedge r \subseteq s)$, and we say that r is *more precise* than s. We also write $\bigsqcup_d$ for the join. The index d reminds us that the non-determinism in programs is resolved demonically. Note that we formulated predicates in a semantic way, there is no assertion language to denote sets of states. Nevertheless, we emphasize that the inclusion in the formulation of $r \preceq_d s$ corresponds to implication in logic, meaning the definition can be implemented with solver technology right away.

To give a semantics to executions, we first lift the semantics of commands from states to predicates. We define $[\![\mathtt{com}]\!](\mathit{fail}) = \mathit{fail}$ and $[\![\mathtt{com}]\!](r) = \bigsqcup_d \{[\![\mathtt{com}]\!](s) \mid s \in r\}$, where $r \neq \mathit{fail}$. We then set $[\![\mathtt{ex}_1; \mathtt{ex}_2]\!](r) = [\![\mathtt{ex}_2]\!]([\![\mathtt{ex}_1]\!](r))$. A consequence is that programs cannot recover from failure.

2.3 Hoare Logic

We specify the correctness of programs by Hoare triples of the form $\{r\}\mathtt{prog}\{s\}$. The triple is valid, if every possible execution of prog that starts in a state from r ends in a state from s. We define

$$\models_d \{r\}\mathtt{prog}\{s\} \quad \text{by} \quad \forall \mathtt{ex} \in \mathit{exec}(\mathtt{prog}).\ [\![\mathtt{ex}]\!](r) \preceq_d s\ .$$

To derive valid triples in a compositional way, Hoare logic offers proof rules similar to the black ones in Figure 3, we just have to replace all angle brackets by set brackets, the indices a by d, and assume R, S, T are predicates as we have defined them above. Rule (**COM**) requires that the postcondition over-approximates the effect of the command on the precondition. Rule (**SEQ**) composes programs sequentially if the intermediary assertion matches. Rule (**LOOP**) checks that the predicate is invariant under the loop body. Rule (**DEM**) is commonly called choice and lets the demon choose the alternative with which to continue the execution. The rule of consequence (**CSQ**) lets us strengthen the precondition and weaken the postcondition. We use $\vdash_d \{r\}\mathtt{prog}\{s\}$ to indicate that a Hoare triple can be derived with these proof rules. It is readily checked that these rules are sound in that $\vdash_d \{r\}\mathtt{prog}\{s\}$ implies $\models_d \{r\}\mathtt{prog}\{s\}$.

One can also show that the rules are complete, $\models_d \{r\}\mathtt{prog}\{s\}$ implies $\vdash_d \{r\}\mathtt{prog}\{s\}$. As we will need it later, we explain the proof. The idea is to consider the full state space of program prog from states in r. Now for each line of code, we collect all states from the state space that decorate this line, and let them form a predicate. We construct a proof outline in which every line of code carries the predicate we have just constructed as a precondition. One can now check that this proof outline can be derived with the above rules.

3 REALIZABILITY LOGIC

We extend programs to program sketches that contain yet to be resolved non-terminals, and present our program logic to reason about realizability.

3.1 Sketches

We extend the grammar of Programs by non-terminals N from a finite set $\mathcal{N}$. The resulting *sketches* are given by the grammar:

$$\mathtt{sketch} \quad ::= \quad \mathtt{com} \mid \mathtt{sketch}; \mathtt{sketch} \mid \mathtt{sketch} + \mathtt{sketch} \mid \mathtt{sketch}^* \mid \mathtt{N}\ .$$

We use Sketches for the set of all sketches. We assume each non-terminal $\mathtt{N} \in \mathcal{N}$ comes with a set of productions $\emptyset \neq \mathit{prod}(\mathtt{N}) \subseteq \mathtt{Sketches}$ and usually write $\mathtt{N} ::= \mathtt{sketch}_1 \mid \mathtt{sketch}_2$ rather than $\mathit{prod}(\mathtt{N}) = \{\mathtt{sketch}_1, \mathtt{sketch}_2\}$. There may be more than two alternatives, and the sketches may themselves contain non-terminals so that we have proper recursion. The pair $(\mathcal{N}, \mathit{prod})$ of non-terminals and their productions belong to the synthesis task and are given by the user.

To lift the semantics from programs to sketches, we let the function *drv* determine all programs (without non-terminals) that can be derived from a sketch by rewriting the non-terminals. For

$$\textbf{(COM)}\ \frac{[\![\mathtt{com}]\!](r) \preceq_d s}{\vdash_a \langle\{r\}\rangle \mathtt{com} \langle\{s\}\rangle} \qquad \textbf{(SEQ)}\ \frac{\vdash_a \langle R\rangle \mathtt{sketch}_1 \langle S\rangle \quad \vdash_a \langle S\rangle \mathtt{sketch}_2 \langle T\rangle}{\vdash_a \langle R\rangle \mathtt{sketch}_1; \mathtt{sketch}_2 \langle T\rangle}$$

$$\textbf{(LOOP)}\ \frac{R = \{r\} \quad \vdash_a \langle R\rangle \mathtt{sketch} \langle R\rangle}{\vdash_a \langle R\rangle \mathtt{sketch}^* \langle R\rangle} \qquad \textbf{(CSQ)}\ \frac{R \preceq_a R' \qquad S' \preceq_a S \quad \vdash_a \langle R'\rangle \mathtt{sketch} \langle S'\rangle}{\vdash_a \langle R\rangle \mathtt{sketch} \langle S\rangle}$$

$$\textbf{(DEM)}\ \frac{R = \{r\} \quad \vdash_a \langle R\rangle \mathtt{sketch}_1 \langle S\rangle \quad \vdash_a \langle R\rangle \mathtt{sketch}_2 \langle S\rangle}{\vdash_a \langle R\rangle \mathtt{sketch}_1 + \mathtt{sketch}_2 \langle S\rangle} \qquad \textbf{(ANG)}\ \frac{\mathtt{N} ::= \mathtt{sketch} \mid \ldots \quad \vdash_a \langle R\rangle \mathtt{sketch} \langle S\rangle}{\vdash_a \langle R\rangle \mathtt{N} \langle S\rangle}$$

$$\textbf{(GATHER)}\ \frac{\vdash_a \langle R_1\rangle \mathtt{sketch} \langle S_1\rangle \quad \vdash_a \langle R_2\rangle \mathtt{sketch} \langle S_2\rangle}{\vdash_a \langle R_1 \cup R_2\rangle \mathtt{sketch} \langle S_1 \cup S_2\rangle}$$

Fig. 3. Rules of realizability logic. Extensions of Hoare logic are highlighted in blue.

```
1 ⟨true⟩   N(x = 0 | x = 1);   ⟨x = 0, x = 1⟩
2 (assert(x==0) + assert(x==1))
3 ⟨fail⟩
```

Fig. 4. Demonic choice.

$$\textbf{(SHARE)}\ \frac{\vdash_a \langle T\rangle \mathtt{N} \langle U\rangle \qquad \vdash_a \langle R\rangle \mathtt{sketch}[\mathtt{N}/\langle T\rangle\langle U\rangle] \langle S\rangle}{\vdash_a \langle R\rangle \mathtt{sketch} \langle S\rangle}$$

Fig. 5. Subproof sharing.

example, with $\mathtt{N} ::= \mathtt{com}_1 \mid \mathtt{com}_2$ we have $drv(\mathtt{N}; \mathtt{com}) = \{\mathtt{com}_1; \mathtt{com}, \mathtt{com}_2; \mathtt{com}\}$. Formally, we see the sketch as a sentential form in a context-free grammar and take the language.

3.2 Realizability Logic

Realizability logic reasons over correctness specification of the form $\langle R\rangle \mathtt{sketch} \langle S\rangle$. Here, R, S are so-called *selections* from the set $Selections = \mathcal{P}_{fin}(Predicates)$. A selection is a finite set of predicates as we have defined them above. The idea is this. When we encounter a non-terminal in a sketch, we do not yet determine how to resolve it to a program with the function drv. Instead, we collect in a selection the predicates that these programs may justify as a postcondition. Since also the precondition is a selection, we can not only choose the program but also the precondition to justify the postcondition. We thus define validity of realizability triples via validity in Hoare logic:

$$\models_a \langle R\rangle \mathtt{sketch} \langle S\rangle \qquad \text{by} \qquad \forall s \in S.\ \exists r \in R.\ \exists \mathtt{prog} \in drv(\mathtt{sketch}).\ \models_d \{r\}\mathtt{prog}\{s\}\ .$$

To derive valid realizability triples, we use the proof rules in Figure 3. We write $\vdash_a \langle R\rangle \mathtt{sketch} \langle S\rangle$ if the realizability triple can be derived with the help of these rules. Rule (**COM**) checks whether the effect of a command com on a predicate r is captured by s. The rule allows us to derive the realizability triple $\langle\{r\}\rangle \mathtt{com} \langle\{s\}\rangle$ in which the selections are singleton sets. The rule plays together with (**GATHER**): if we can also derive $\langle\{r'\}\rangle \mathtt{com} \langle\{s'\}\rangle$ for another pre- and postcondition, then we can join the selections and obtain $\langle\{r, r'\}\rangle \mathtt{com} \langle\{s, s'\}\rangle$. Rule (**GATHER**) is also important to join the pre- and postconditions for the various programs that can be derived from a non-terminal. To obtain a realizability triple $\langle R\rangle \mathtt{N} \langle S\rangle$, Rule (**ANG**) unwinds the non-terminal to a sketch sketch that is given by a production and then proves $\langle R\rangle \mathtt{sketch} \langle S\rangle$. Note that the sketch may again contain N, and may contain more than one occurrence. We explain at the end of the section how to avoid the repetition of proof trees and handle non-terminals in an efficient way.

Rule (**DEM**) looks like the standard rule for choice operators in Hoare logic. In realizability logic, the precondition has to be a singleton selection, and the rule is unsound without this side condition. If there were multiple predicates in R, then the synthesis from $\mathtt{sketch}_1$ and the synthesis from $\mathtt{sketch}_2$ could use different predicates. This would be incorrect, because the synthesis is

done at compile-time while the branch is chosen at runtime, and therefore the synthesis is not allowed to react on the branch. To see this, consider the sketch in Figure 4 and let `choice` stand for `assert(x==0) + assert(x==1)`. Without the side condition, we could derive the realizability triple $\vdash_a \langle x = 0, x = 1\rangle \texttt{choice} \langle \texttt{x} = 0, \texttt{x} = 1\rangle$, because we could enter the left branch with the predicate $x = 0$ and the right branch with the predicate $x = 1$. However, the programs we can synthesize, namely `x=0; choice` and `x=1; choice`, have to commit to one of the assignments, and therefore `choice` is doomed to fail. A similar reasoning applies for Rule (**LOOP**). The way to handle multiple predicates in a precondition is to consider them one by one and join the results, again using (**GATHER**).

Rule (**SEQ**) reinforces the definition of validity for realizability triples. Consider $\models_a \langle R\rangle \texttt{sketch}_1 \langle S\rangle$ and $\models_a \langle S\rangle \texttt{sketch}_2 \langle T\rangle$. If validity was defined by an existential instead of a universal quantifier over the postcondition, then it would not be sound to derive $\models_a \langle R\rangle \texttt{sketch}_1\texttt{; sketch}_2 \langle T\rangle$. The point is that the existentially quantified predicate $s_1 \in S$ that is chosen to justify the validity of the first triple could differ from the predicate $s_2 \in S$ needed for the second. There is an alternative definition of validity that admits compositionality. We discuss it at the end of the following section when we have developed the overall synthesis technique.

It remains to discuss (**CSQ**). The rule of consequence needs an ordering on selections. We say that selection R is *more versatile* than selection S, if for every predicate in S there is a more precise predicate in R, so we define

$$R \preceq_a S \qquad \text{by} \qquad \forall s \in S.\, \exists r \in R.\, r \preceq_d s\,.$$

Having more predicates available makes a selection more versatile and having less predicates makes a selection less versatile. It is also possible to have a more versatile selection with less predicates if they are more precise. With this, $\emptyset$ is the least versatile and $\{\emptyset\}$ is the most versatile selection.

The rules are sound and, together with a rule for the edge case of an empty precondition that we give in the appendix, also complete.

Theorem 3.1 (Sound-And-Complete). $\vdash_a \langle R\rangle \texttt{sketch} \langle S\rangle$ *if and only if* $\models_a \langle R\rangle \texttt{sketch} \langle S\rangle$.

Proof. Soundness holds by an induction on the height of the proof tree. We argue for completeness and consider $\models_a \langle R\rangle \texttt{sketch} \langle S\rangle$. By the definition of validity, this means for every predicate $s \in S$ there is a predicate $r \in R$ and a program $\texttt{prog} \in \mathit{drv}(\texttt{sketch})$ so that $\models_d \{r\}\texttt{prog}\{s\}$ holds. We invoke the completeness of Hoare logic and obtain $\vdash_d \{r\}\texttt{prog}\{s\}$. The derivation can be mimicked in realizability logic and yields $\vdash_a \langle r\rangle \texttt{prog} \langle s\rangle$. Since the program has been derived from `sketch`, we can also obtain $\vdash_a \langle r\rangle \texttt{sketch} \langle s\rangle$ with finitely many applications of (**ANG**). Rule (**GATHER**) allows us to join the triples $\langle r\rangle \texttt{sketch} \langle s\rangle$ and obtain the desired $\vdash_a \langle R\rangle \texttt{sketch} \langle S\rangle$. □

3.3 Discussion

On SemGuS. The synthesis tasks we consider have the following input: the states, the commands, and the semantics $(\mathit{States}, \texttt{COM}, [\![-]\!])$ of the programming language, the non-terminals with their production rules $(\mathcal{N}, \mathit{prod})$, and the realizability triple $\langle R\rangle \texttt{sketch} \langle S\rangle$ of interest. In its original formulation [23], SemGuS would be more liberal and allow the user to also define the operators (their syntax and their semantics) from which programs can be built. We fix those operators to concatenation, choice, and Kleene star, instead. This allows us to work with an extension of Hoare logic. To lift our approach to the more general setting, the user would have to specify the proof rules that are sound for the new operators, which would be in-line with the approach in unrealizability logic [22].

The reader may also note that we have not made assumptions about the correctness specification. In synthesis, it is common to work with sets of examples. The work on unrealizability logic has

shown that such sets can be captured by so-called vector assertions [22]. Vector assertions are predicates of a particular form, and our construction of selections readily applies to them.

On search-based synthesis. Realizability logic is formulated such that it can be readily combined with search-based synthesis. The point is that Rule (**ANG**) does not force us to consider all programs a non-terminal can be rewritten to. Instead, we can consider a set of programs that appear most promising, as can be judged from a probabilistic grammar [20].

On the implementation of non-terminals. A non-terminal may occur multiple times in a proof in realizability logic, it may even occur recursively. Rather than duplicating the subproofs for this non-terminal, we would like to share them in the various places the non-terminal is used. This is made possible by Rule (**SHARE**) given in Figure 5. If we have already derived the triple $\langle T\rangle \mathsf{N} \langle U\rangle$, then we can eliminate the non-terminal from a program sketch and replace it by the pair $\langle T\rangle\langle U\rangle$. The pair stands for the fact that we can synthesize a program that transforms T to U. Rule (**SHARE**) is readily checked to be sound. However, it is a derived rule whose application can easily be mimicked by Rule (**ANG**), at the cost of blowing-up the proof. An implementation would refine the rule so that different occurrences of the non-terminal can be replaced in different ways. Moreover, one would maintain a pointer to the subproof that should be inserted, to be able to derive the program with the technique given next.

4 REALIZATION LOGIC

We define realization logic, a program logic to derive programs from sketches. The key insight is that a proof outline in realizability logic for the sketch of interest is helpful guidance to find a program that solves the synthesis task. To make use of this guidance, realization logic rewrites the given proof outline until a suitable program is found. Realization logic thus reasons over rewriting steps of the form $\mathsf{po} \vdash \mathsf{po}'$ between proof outlines. The strategy behind the rewriting is to propagate information about failed synthesis attempts backwards, and thereby iteratively eliminate predicates from selections and productions from the definition of non-terminals.

As we need a precise understanding of proof outlines, we give the definition:

$$\mathsf{po} \quad ::= \quad \langle R\rangle \mathsf{com} \langle S\rangle \mid \mathsf{po}; \mathsf{po} \mid \mathsf{po} + \mathsf{po} \mid \mathsf{po}^* \mid \mathsf{N}(\mathsf{po}) \mid (\mathsf{po} \mid \mathsf{po}) \ .$$

We write $\vdash_a \mathsf{po}$ to indicate that the proof outline has been derived with realizability logic. The assertions we track are the ones that have been used in the application of (**SEQ**), (**LOOP**), and (**ANG**). The formal definition is in the appendix. We abuse the notation a bit and write $\langle R\rangle \mathsf{po} \langle S\rangle$ to indicate that R is the precondition and S is the postcondition of the proof outline po. This is well-defined, because $\vdash_a \mathsf{po}_1 + \mathsf{po}_2$ implies that po_1 and po_2 have the same pre- and postcondition.

The proof rules of realization logic are listed in Figure 6, we discuss them below. The rules give the following soundness guarantee: if we start from a proof outline in realizability logic and rewrite it to another proof outline, then also this proof outline can be derived with realizability logic.

Theorem 4.1 (Soundness). $\vdash_a \mathsf{po}$ *and* $\mathsf{po} \vdash \mathsf{po}'$ *together imply* $\vdash_a \mathsf{po}'$.

This allows us to invoke the soundness of realizability logic from Theorem 3.1 and obtain the desired semantic guarantee for the program we have derived.

A feature all rules share is that they are guided by the original proof outline: the selections in the proof outline that results from rewriting are limited to the selections in the original proof outline. To ease the notation, we use R^{w} to denote a selection that is known to be less versatile than R, meaning $R \preceq_a R^{\mathsf{w}}$ holds. This may be used repeatedly, so $R \preceq_a R^{\mathsf{w}} \preceq_a R^{\mathsf{ww}}$.

The rules in Figure 6 rewrite the given proof outline in a compositional fashion. The base case (**RCOM**) allows us to pick a pre- and postcondition from a selection. This is the moment we realize

$$\textbf{(RCOM)}\ \frac{[\![\mathtt{com}]\!](r) \preceq_d s \qquad R \preceq_a \{r\} \qquad S \preceq_a \{s\}}{\langle R\rangle \mathtt{com} \langle S\rangle \vdash \langle \{r\}\rangle \mathtt{com} \langle \{s\}\rangle}$$

$$\textbf{(RANG)}\ \frac{\mathtt{po} \vdash \mathtt{po}'}{\mathtt{N}(\mathtt{po}_1 \mid \mathtt{po} \mid \mathtt{po}_2) \vdash \mathtt{N}(\mathtt{po}_1 \mid \mathtt{po}' \mid \mathtt{po}_2)}$$

$$\textbf{(RSELECT)}\ \frac{}{\mathtt{N}(\mathtt{po}_1 \mid \mathtt{po} \mid \mathtt{po}_2) \vdash \mathtt{po}}$$

$$\textbf{(RSEQL)}\ \frac{\langle R\rangle \mathtt{po}_1 \langle S\rangle \vdash \langle R^{\mathrm{w}}\rangle \mathtt{po}_1' \langle S\rangle}{\langle R\rangle \mathtt{po}_1 \langle S\rangle ; \mathtt{po}_2 \vdash \langle R^{\mathrm{w}}\rangle \mathtt{po}_1' \langle S\rangle ; \mathtt{po}_2}$$

$$\textbf{(RSEQR)}\ \frac{\langle S\rangle \mathtt{po}_2 \langle T\rangle \vdash \langle S^{\mathrm{w}}\rangle \mathtt{po}_2' \langle T^{\mathrm{w}}\rangle}{\langle R\rangle \mathtt{po}_1 \langle S\rangle ; \langle S\rangle \mathtt{po}_2 \langle T\rangle \vdash \langle R\rangle \mathtt{po}_1 \langle S^{\mathrm{w}}\rangle ; \langle S^{\mathrm{w}}\rangle \mathtt{po}_2' \langle T^{\mathrm{w}}\rangle}$$

$$\textbf{(RDEM)}\ \frac{R^{\mathrm{w}} = \{r\} \qquad \langle R\rangle \mathtt{po}_i \langle S\rangle \vdash \langle R^{\mathrm{w}}\rangle \mathtt{po}_i' \langle S^{\mathrm{w}}\rangle \quad \text{for } i = 1, 2}{\langle R\rangle \mathtt{po}_1 \langle S\rangle + \langle R\rangle \mathtt{po}_2 \langle S\rangle \vdash \langle R^{\mathrm{w}}\rangle \mathtt{po}_1' \langle S^{\mathrm{w}}\rangle + \langle R^{\mathrm{w}}\rangle \mathtt{po}_2' \langle S^{\mathrm{w}}\rangle}$$

$$\textbf{(RLOOP)}\ \frac{\langle I\rangle \mathtt{po} \langle I\rangle \vdash \langle I^{\mathrm{w}}\rangle \mathtt{po}' \langle I^{\mathrm{w}}\rangle \qquad I^{\mathrm{w}} = \{i\}}{\langle I\rangle \mathtt{po}^* \langle I\rangle \vdash \langle I^{\mathrm{w}}\rangle \mathtt{po}^* \langle I^{\mathrm{w}}\rangle}$$

$$\textbf{(RTRANS)}\ \frac{\mathtt{po}_1 \vdash \mathtt{po}_2 \qquad \mathtt{po}_2 \vdash \mathtt{po}_3}{\mathtt{po}_1 \vdash \mathtt{po}_3}$$

$$\textbf{(RCSQ)}\ \frac{\langle R\rangle \mathtt{po} \langle S\rangle \vdash \langle R^{\mathrm{ww}}\rangle \mathtt{po}' \langle S^{\mathrm{w}}\rangle}{\langle R\rangle \mathtt{po} \langle S\rangle \vdash \langle R^{\mathrm{w}}\rangle \mathtt{po}' \langle S^{\mathrm{ww}}\rangle}$$

$$\textbf{(RGATHER)}\ \frac{R \preceq_a (R_1 \cup R_2) \qquad S \preceq_a (S_1 \cup S_2) \qquad \mathit{sketch}(\mathtt{po}_1') = \mathit{sketch}(\mathtt{po}_2') \qquad \langle R\rangle \mathtt{po} \langle S\rangle \vdash \langle R_1\rangle \mathtt{po}_1' \langle S_1\rangle \qquad \langle R\rangle \mathtt{po} \langle S\rangle \vdash \langle R_2\rangle \mathtt{po}_2' \langle S_2\rangle}{\langle R\rangle \mathtt{po} \langle S\rangle \vdash \langle R_1 \cup R_2\rangle \mathit{gather}(\mathtt{po}_1', \mathtt{po}_2') \langle S_1 \cup S_2\rangle}$$

Fig. 6. Rules of realization logic.

that predicates from the precondition can be dropped because they are not needed to obtain the postcondition, as shown in the introduction. We have to explicitly check the over-approximation because realizability triples do not maintain the interplay between the predicates in the pre- and in the postcondition. Using (**RANG**), we can rewrite proofs for the right-hand sides of non-terminals. The heart of the calculus is (**RSELECT**), which allows us to replace non-terminals by their right-hand sides. The Rules (**RSEQL**) and (**RSEQR**) rewrite the left resp. the right part of a sequence. In (**RSEQL**), it is important that the new intermediary assertion matches the original intermediary assertion. This does not have to be the case in (**RSEQR**), which uses the rule of consequence to automatically weaken the postcondition on the left. As in realizability logic, Rule (**RDEM**) forces us to commit to a predicate when reasoning about demonic choices. It also produces the same postcondition S^{w} in both branches. Rule (**RLOOP**) for rewriting loop bodies has a similar behavior. The rule of consequence (**RCSQ**) is used like in realizability logic. The same is true for (**RGATHER**), which uses function *gather* to recursively join the intermediary assertions in the proof outlines. The definition is in the appendix. The need for this function stems from the fact that (**RDEM**) can only work with single predicates. Rule (**TRANS**) is the transitivity of rewriting. This rule is needed to combine rewrites, for example when rewriting the left and the right part of a sequence.

Realization logic always rewrites a proof outline into one that is weaker in the sense that it has less synthesis options or, phrased differently, is closer to a program.

Lemma 4.2. $\mathtt{po} \vdash \mathtt{po}'$ *implies* $\mathtt{po} \preceq_p \mathtt{po}'$.

The ordering $\mathtt{po} \preceq_p \mathtt{po}'$ states that $\mathtt{po}'$ has less versatile assertions than $\mathtt{po}$ and a sketch or even program that can be derived from the sketch in $\mathtt{po}$. The definition is by induction on the structure of proof outlines. The base case is $\langle R\rangle \mathtt{com} \langle S\rangle \preceq_p \langle R^{\mathrm{w}}\rangle \mathtt{com} \langle S^{\mathrm{w}}\rangle$. The step cases are similar to the one for choice: $\mathtt{po}_1 + \mathtt{po}_2 \preceq_p \mathtt{po}_1' + \mathtt{po}_2'$ if $\mathtt{po}_1 \preceq_p \mathtt{po}_1'$ and $\mathtt{po}_2 \preceq_p \mathtt{po}_2'$. Non-terminals are an exception, where we use $\mathtt{N}(\mathtt{po}_1 \mid \mathtt{po} \mid \mathtt{po}_2) \preceq_p \mathtt{N}(\mathtt{po}_1 \mid \mathtt{po}' \mid \mathtt{po}_2)$ if $\mathtt{po} \preceq_p \mathtt{po}'$. A proof outline over a non-terminal is also stronger than a proof outline over a derivative, $\mathtt{N}(\mathtt{po}_1 \mid \mathtt{po} \mid \mathtt{po}_2) \preceq_p \mathtt{po}$.

Realization logic also has a completeness property: proof outlines can be rewritten into all weaker proof outlines that are derivable in realizability logic.

Theorem 4.3 (Completeness). $\vdash_a \mathtt{po}$ *and* $\vdash_a \mathtt{po}'$ *and* $\mathtt{po} \preceq_p \mathtt{po}'$ *together imply* $\mathtt{po} \vdash \mathtt{po}'$.

A final guarantee that is interesting from an algorithmic point of view is that realization logic provably does not need backtracking: from every proof outline and for every predicate in the postcondition, one can derive a program that satisfies this postcondition. By Theorem 4.1, the guarantee continues to hold after a rewriting step (that should not remove the predicate of interest), hence the name.

Theorem 4.4 (Backtracking Freedom). *Let* $\vdash_a \langle R\rangle \mathsf{po} \langle S\rangle$ *and* $s \in S$. *Then there are* r *and* po' *so that* $\langle R\rangle \mathsf{po} \langle S\rangle \vdash \langle\{r\}\rangle \mathsf{po}' \langle\{s\}\rangle$, $r \in R$, *and* $\mathit{sketch}(\mathsf{po}') \in \mathit{drv}(\mathit{sketch}(\mathsf{po}))$.

It is tempting to try to prove backtracking freedom with the help of completeness in Theorem 3.1. To understand why this does not work, consider the valid proof outline given in Figure 7 and suppose both commands have the same semantics, namely the identity function. When we want to derive a program for the postcondition $\{\mathtt{x} = 0\}$, realization logic can only propose $\mathtt{skip}_1$. The notion of validity in realizability logic, however, may justify the postcondition by $\models_d \{\mathtt{x} = 0\}\mathtt{skip}_2\{\mathtt{x} = 0\}$. Completeness in Theorem 3.1 now shows that there is a proof outline for $\vdash_a \langle \mathtt{x} = 0\rangle \mathtt{skip}_2 \langle \mathtt{x} = 0\rangle$. The point is that this proof outline is not weaker than the one in Figure 7. This also means that there is no contradiction to Theorem 4.3. In short, Theorem 3.1 reasons about sketches and Theorem 4.4 reasons about proof outlines, and the two are incomparable.

For an example application of realization logic, we refer to Section 1. Eliminating the predicate *fail* in Equation (1) is an application of Rule (**RCSQ**). Weakening the precondition in Equation (2) is an application of Rule (**RCOM**). Resolving the non-terminal to the left production is an application of rule (**RSELECT**). The resulting proof outline fragments are put together using Rules (**RSEQL**), (**RSEQR**), and (**RTRANS**).

```
1 ⟨x = 0⟩
2 N((⟨x = 0⟩ skip₁ ⟨x = 0⟩) |
3   (⟨x = 0⟩ skip₂ ⟨∅⟩))
4 ⟨x = 0⟩
```

Fig. 7. Guidance.

Overall approach and the notion of validity. The overall approach to synthesizing a program from a sketch `sketch`, a precondition R, and postcondition S is this. We prove the triple $\langle R\rangle \mathtt{sketch} \langle S\rangle$ in realizability logic. One may see this proof as a symbolic execution that annotates the code by assertions in a *forward* fashion, starting from the precondition and ending at the postcondition. When we have proven the triple, we in particular have a proof outline at hand. We rewrite this proof outline to a program using the realization logic we have just developed. The rewriting proceeds *backwards*: starting from failures and predicates that do not guarantee the postcondition, we eliminate predicates from selections and productions from the definition of non-terminals.

We need the rewriting phase because realizability logic abstracts away the link between the programs that can be synthesized from the sketch and the pre- and postconditions that these programs can guarantee. This abstraction is precisely what makes realizability logic scale, and the link is precisely what realization logic recovers. What makes realization logic efficient is that it is guided by the proof outline in realizability logic. Moreover, realizability logic provably does not need backtracking (the rewriting cannot go wrong).

It would be possible to define the notion of validity in realizability logic differently, namely by universally quantifying over the precondition and existentially quantifying over the postcondition. Our synthesis approach can be adapted to this definition as follows. We imagine the proof in realizability logic as being constructed *backwards*, the symbolic execution starts from the postcondition and proceeds to the precondition. Realization logic would then start from unsuitable preconditions and proceed *forwards*. Our approach has the advantage that forward reasoning tends to be more deterministic than backwards reasoning, which leads to smaller assertions in realizability logic.

5 IMPLEMENTING REALIZABILITY LOGIC

We give an algorithm that checks the validity of triples in realizability logic. The algorithm also yields a proof outline that will be handed over to the implementation of realization logic developed in the next section. The algorithm is a deductive verification: we compute a set of verification conditions whose validity entails the validity of the given realizability triple. Importantly, although the assertions in realizability logic are selections rather than mere predicates, the validity check can be discharged with an off-the-shelf SMT solver. At the technical level, our contribution lies in the definition of suitable verification conditions. We proceed forwards, using a tailor-made strongest post. Deductive verification needs loop invariants, and this is no different in our case. We assume these invariants are given by the user. Moreover, we assume the user gives us information about the non-terminals, either in the form of an annotated assertion transformer, or in the form of a recursion depth to which the non-terminal should be unwound in the search for a program. Taking these user annotations as an input is a compromise: it is a burden on the user, but it improves the applicability of the method. Having said that, we remark that in our experiments we were able to automatically determine the loop invariants and handle the non-terminals.

5.1 Verification Conditions and Strongest Post

In Figure 8, we define the function $vc : \mathit{annRealizabilityTriples} \to \mathcal{P}(\mathit{VerifConds})$ which takes an annotated realizability triple and produces a set of verification conditions. The annotation we assume is an invariant I in the case of loops $\texttt{sketch}^*$, denoted by $\texttt{sketch}^*[I]$, and a selection transformer $\Gamma : \mathit{Selections} \to \mathit{Selections}$ in the case of non-terminals N, written as $\mathsf{N}[\Gamma]$. A verification condition is an inequality between selections from the set $\mathit{VerifConds} = \mathit{Selections} \times \mathit{Selections}$.

We rely on the correctness of the annotations. A loop invariant is sound, if $\models_a \langle\{i\}\rangle\texttt{sketch}\langle\{i\}\rangle$ holds for all $i \in I$. It is complete if $\models_a \langle\{j\}\rangle\texttt{sketch}\langle\{j\}\rangle$ implies $j \in I$. A selection transformer is sound, if $\Gamma(R) = S$ implies $\models_a \langle R\rangle\mathsf{N}\langle S\rangle$. It is complete, if $\models_a \langle R\rangle\mathsf{N}\langle S\rangle$ implies $\Gamma(R) \preceq_a S$.

Function vc is sound: when all verification conditions hold, then the realizability triple is valid. Note that we do not need to assume the soundness of the annotations, but will check this as part of the verification conditions. For completeness, however, we need to make this assumption.

Theorem 5.1 (VC-Sound-And-Complete). *$\models vc(\langle R\rangle\texttt{sketch}\langle S\rangle)$ implies $\models_a \langle R\rangle\texttt{sketch}\langle S\rangle$. If all annotations are complete, $\models_a \langle R\rangle\texttt{sketch}\langle S\rangle$ implies $\models vc(\langle R\rangle\texttt{sketch}\langle S\rangle)$.*

The benefit of using verification conditions $R \preceq_a S$ as the proof obligation is that they can be discharged with off-the-shelf SMT technology. For every predicate $s \in S$, we have to go through the predicates $r \in R$ until we find $r \subseteq s$. This means at most quadratically many SMT queries. Future assertion languages for selections may suggest a different strategy.

Function vc makes use of the strongest post function sp which is also defined in Figure 8. It takes as input a selection and a sketch and outputs a selection, $sp : (\mathit{Selections} \times \texttt{Sketches}) \to \mathit{Selections}$. The output is a selection that can be guaranteed when running the sketch on the input. Moreover, it is the most versatile selection, if the annotations are complete. Since the result does not refer to the verification conditions, we have to require soundness.

Theorem 5.2 (SP-Sound-And-Complete). *If the annotations are sound, $sp(R, \texttt{sketch}) \preceq_a S$ implies $\models_a \langle R\rangle\texttt{sketch}\langle S\rangle$. If the annotations are sound and complete, $\models_a \langle R\rangle\texttt{sketch}\langle S\rangle$ implies $sp(R, \texttt{sketch}) \preceq_a S$.*

The functions vc and sp are defined inductively over the structure of sketches. The novelty is in the cases for demonic choices, non-terminals, and loops. For demonic choices, recall that Rule (**DEM**) reasons over single predicates. The strongest post thus iterates through the predicates in the given precondition. For each $r \in R$, it computes the strongest post for both branches. If we have $s_1 \in$

$$sp(R, \texttt{com}) := \{[\![\texttt{com}]\!](r) \mid r \in R\} \qquad sp(R, \texttt{N}[\Gamma]) := \Gamma(R)$$
$$sp(R, \texttt{sketch}_1; \texttt{sketch}_2) := sp(sp(R, \texttt{sketch}_1), \texttt{sketch}_2) \qquad sp(R, \texttt{sketch}^*[I]) := \{i \in I \mid R \preceq_a \{i\}\}$$
$$sp(R, \texttt{sketch}_1 + \texttt{sketch}_2) := \{s_1 \sqcup_d s_2 \mid s_k \in sp(\{r\}, \texttt{sketch}_k) \wedge r \in R \wedge k \in \{1, 2\}\}$$

$$vc(\langle R\rangle \texttt{com} \langle S\rangle) := \{sp(R, \texttt{com}) \preceq_a S\} \qquad vc(\langle R\rangle \texttt{N}[\Gamma] \langle S\rangle) := \{sp(R, \texttt{N}[\Gamma]) \preceq_a S\} \cup ca(R, \texttt{N}, \Gamma)$$
$$vc(\langle R\rangle \texttt{sketch}_1; \texttt{sketch}_2 \langle S\rangle) := vc(\langle R\rangle \texttt{sketch}_1 \langle sp(R, \texttt{sketch}_1)\rangle) \cup vc(\langle sp(R, \texttt{sketch}_1)\rangle \texttt{sketch}_2 \langle S\rangle)$$
$$vc(\langle R\rangle \texttt{sketch}^*[I] \langle S\rangle) := \{sp(R, \texttt{sketch}^*[I]) \preceq_a S\} \cup ca(I, \texttt{sketch}^*)$$
$$vc(\langle R\rangle \texttt{sketch}_1 + \texttt{sketch}_2 \langle S\rangle) := (\cup\{vc(\langle\{r\}\rangle \texttt{sketch}_k \langle sp(\{r\}, \texttt{sketch}_k)\rangle) \mid r \in R \wedge k \in \{1, 2\}\}) \cup$$
$$\{sp(R, \texttt{sketch}_1 + \texttt{sketch}_2) \preceq_a S\}$$

$$ca(R, \texttt{N}, \Gamma) := \cup\{vc(\langle R\rangle \texttt{prog} \langle sp(R, \texttt{prog})\rangle) \mid \min j : (\cup\{sp(R, \texttt{prog}') \mid \texttt{prog}' \in \mathbb{O}(\texttt{N}, j)\}) \preceq_a \Gamma(R) \wedge \texttt{prog} \in \mathbb{O}(\texttt{N}, j)\}$$
$$ca(I, \texttt{sketch}^*) := \{vc(\langle\{i\}\rangle \texttt{sketch} \langle\{i\}\rangle) \mid i \in I\}$$

Fig. 8. Definitions of strongest post, verification conditions, and check annotation functions.

$sp(\{r\}, \texttt{sketch}_1)$, then we can guarantee the predicates s_1 when running $\texttt{sketch}_1$ on r. The same holds for $s_2 \in sp(\{r\}, \texttt{sketch}_2)$. But as the branch will be chosen demonically, we can only enforce the least upper bound of s_1 and s_2. However, we are free to combine all s_1 with all s_2. This explains the definition. The verification conditions check each branch individually, again with a singleton precondition. Moreover, they check whether the strongest post is sufficient to prove the postcondition.

For non-terminals, the strongest post is given by the annotated transformer. The verification conditions check whether the strongest post entails the postcondition. Moreover, they check the soundness of the annotation with the function $ca(R, \texttt{N}, \Gamma)$. We elaborate on it in a moment. For loops, we rely on an annotated invariant. The strongest post keeps only those predicates from the invariant that follow from the precondition. The verification conditions again check whether the strongest post entails the postcondition. Moreover, they check whether the invariant annotation is sound via $ca(I, \texttt{sketch}^*)$.

Soundness of a loop invariant is easy to check. Since Rule (**LOOP**) can only enter a loop with a singleton, we check each predicate in I for being an invariant. The soundness of transformers is more difficult to check. Function $ca(R, \texttt{N}, \Gamma)$ tries to derive a set of programs from $\texttt{N}$ which justifies the postcondition $\Gamma(R)$. To this end, it repeatedly invokes an oracle $\mathbb{O}$ which returns programs from $drv(N)$. The function $\mathbb{O}(\texttt{N}, j)$ returns the first j programs the oracle produces for the non-terminal $\texttt{N}$. If the set of programs that have been returned so far is able to produce the postcondition, $ca(R, \texttt{N}, \Gamma)$ collects the corresponding verification conditions and terminates. The only guarantee we need about the oracle is that every program will eventually be returned. If the transformer is sound, we will eventually obtain a large enough set of programs. Otherwise, the verification condition generation will not terminate. A simple implementation for an oracle is to return the programs that can be obtained by unrolling each non-terminal at most k times, for larger and larger k.

5.2 Example: Checking Realizability of the Factorial Function

We illustrate the verification condition computation on an example. Consider the sketch depicted in Figure 9. We check whether we can derive from it a program that computes the factorial of 42: the precondition of interest is $R = \{\texttt{x} = 42 \wedge \texttt{y} = 1\}$ and the postcondition is $S = \{\texttt{y} = 42!\}$. The sketch is already annotated by the results of the strongest post in grey, the loop is annotated by the invariant $I = \{\texttt{y} * \texttt{x}! = 42! \wedge \texttt{x} \geq 0\}$, and the non-terminal $\texttt{N}$ `::= y=y+x | y=y*x` is annotated by the transformer Γ with $\Gamma(R) = sp(R, \texttt{y=y+x}) \cup sp(R, \texttt{y=y*x})$. We refer to the sketch as `sketch`,

the loop body as body, and the commands by their line number, so x-- is 17. We abbreviate the non-terminal including Lines 6 to 14 by N.

We explain the computation of the verification conditions, which returns

$$vc(\langle R\rangle \texttt{sketch}\langle S\rangle) = \{sp(I, 22) \preceq_a S, \qquad I \preceq_a I \;\; (I = sp(R, \texttt{body}^*[I])), \tag{3}$$
$$sp(\{i\}, 4\,;\texttt{N};17) \preceq_a \{i\}, \qquad sp(\{i\}, 4) \preceq_a sp(\{i\}, 4), \tag{4}$$
$$sp(\{i\}, 4\,;\texttt{N}) \preceq_a sp(\{i\}, 4\,;\texttt{N}), \qquad sp(\{i\}, 4\,;8) \preceq_a sp(\{i\}, 4\,;8), \tag{5}$$
$$sp(\{i\}, 4\,;12) \preceq_a sp(\{i\}, 4\,;12)\} \,. \tag{6}$$

Executing $vc(\langle R\rangle \texttt{sketch}\langle S\rangle)$ yields two recursive calls:

$$vc(\langle R\rangle \texttt{sketch}\langle S\rangle) = vc(\langle R\rangle \texttt{body}^*[I]\langle sp(R, \texttt{body}^*[I])\rangle) \cup vc(\langle sp(R, \texttt{body}^*[I])\rangle 22\langle S\rangle) \,.$$

To calculate the strongest post of the loop, R and each individual element of the loop invariant have to be compared. Since $I = \{i\}$ is a singleton and $R \preceq_a \{i\}$, we have that I itself is the strongest post of the loop. Therefore, the second recursive call boils down to checking $sp(I, \texttt{assume x = 0}) \preceq_a S$, Inequality (3)(left). Next, we invoke $vc(\langle R\rangle \texttt{body}^*[I]\langle sp(R, \texttt{body}^*[I])\rangle)$. We already argued that the strongest post of the loop is the loop invariant. Thus, $I \preceq_a I$ is added as Inequality (3)(right). Checking the loop invariant annotation with $ca(I, \texttt{body}^*)$ returns $vc(\langle\{i\}\rangle \texttt{body}\langle\{i\}\rangle)$. This, in turn, yields two calls to the verification conditions function:

$$vc(\langle\{i\}\rangle \texttt{body}\langle\{i\}\rangle) = vc(\langle i\rangle 4\,;\texttt{N}\langle sp(\{i\}, 4\,;\texttt{N})\rangle) \cup vc(\langle sp(\{i\}, 4\,;\texttt{N})\rangle 17\langle i\rangle) \,.$$

For the second recursive call, the base case of function $vc(-)$ applies and adds $sp(\{i\}, 4\,;\texttt{N};17) \preceq_a \{i\}$ to the output, Inequality (4)(left). The first recursive call yields

$$vc(\langle i\rangle 4\,;\texttt{N}\langle sp(\{i\}, 4\,;\texttt{N})\rangle) = vc(\langle i\rangle 4\langle sp(\{i\}, 4)\rangle) \cup vc(\langle sp(\{i\}, 4)\rangle \texttt{N}\langle sp(\{i\}, 4\,;\texttt{N})\rangle) \,.$$

The first of these calls adds $sp(\{i\}, 4) \preceq_a sp(\{i\}, 4)$, Inequality (4)(right). The second recursive call is for the non-terminal. Inequality $sp(\{i\}, 4\,;\texttt{N}) \preceq_a sp(\{i\}, 4\,;\texttt{N})$ checks entailment of the postcondition, Inequality (5)(left). For soundness of the annotation, $ca(sp(\{i\}, 4), \texttt{N}, \Gamma)$ is called. For $j = 2$, the oracle proposes the programs y=y+x and y=y*x. See that by our definition of the selection transformer Γ, the inequality $sp(R, \texttt{y=y+x}) \cup sp(R, \texttt{y=y*x}) \preceq_a \Gamma(R)$ holds. Thus, the function ca outputs two more inequalities (after resolving the recursive calls to the base case of vc): the inequality $sp(\{i\}, 4; 8) \preceq_a sp(\{i\}, 4; 8)$ considers the first production resolving the nonterminal and the inequality $sp(\{i\}, 4; 12) \preceq_a sp(\{i\}, 4; 12)$ considers the second production. Both inequalities are added to the output, Inequality (5)(right) and Inequality (6).

All inequalities of $vc(\langle R\rangle \texttt{sketch}\langle S\rangle)$ hold and so $\models_a \langle R\rangle \texttt{sketch}\langle S\rangle$ by Theorem 5.1. This means, the non-terminal can be resolved such that the resulting program will compute the factorial of 42. We present next an algorithm to automatically choose the correct branch.

6 IMPLEMENTING REALIZATION LOGIC

We give an algorithm to compute a program from a proof outline in realizability logic. It is a deductive verification that collects a set of verification conditions whose validity shows that the program is a solution to the synthesis task. What is unconventional is that the validity has to be checked in the course of the verification condition computation. The reason is that the validity checks steer the program construction, and also the verification conditions that will be collected in the future. Despite these dynamics, we can show that a small number of verification condition checks will be sufficient to derive the program. Moreover, the verification conditions compare ordinary predicates (rather than selections in the previous section), and therefore can be implemented as single SMT solver queries. All this makes our algorithm efficient in practice.

```
1  ⟨x = 42 ∧ y = 1⟩
2  (
3    ⟨y ∗ x! = 42! ∧ x ≥ 0⟩
4    assume(x > 0);
5    ⟨y ∗ x! = 42! ∧ x > 0⟩
6    N[Γ](
7      ⟨y ∗ x! = 42! ∧ x > 0⟩
8      y = y + x;
9      ⟨(y − x) ∗ x! = 42! ∧ x > 0⟩
10     |
11     ⟨y ∗ x! = 42! ∧ x > 0⟩
12     y = y * x;
13     ⟨(y/x) ∗ x! = 42! ∧ x > 0⟩
14   )
15   ⟨(y − x) ∗ x! = 42! ∧ x > 0,
16     (y/x) ∗ x! = 42! ∧ x > 0⟩
17   x--;
18   ⟨(y − (x + 1)) ∗ (x + 1)! = 42! ∧ x + 1 > 0,
19     (y/(x + 1)) ∗ (x + 1)! = 42! ∧ x + 1 > 0⟩
20 )*[{y ∗ x! = 42! ∧ x ≥ 0}]
21 ⟨y ∗ x! = 42! ∧ x ≥ 0⟩
22 assume(x = 0);
23 ⟨y ∗ x! = 42! ∧ x = 0⟩
```

Fig. 9. Proof outline of a sketch for computing the factorial function.

Our algorithm implements the proof rules in realization logic, more precisely the non-backtracking strategy stated in Theorem 4.4. Given a proof outline and a predicate in the postcondition, it traverses the proof backwards to determine suitable preconditions and programs that justify the postcondition. Our algorithm is thus a function of type $syn : prfOutls \times Predicates \rightarrow Predicates \times \texttt{Programs}$. It is defined in Figure 10 and we discuss it in a moment. An important feature of the function is to discard predicates from selections, in which case it may return $(fail, -)$ and an arbitrary program.

The function gives strong guarantees: when we start from a sketch that has been derived in realizability logic and a predicate in the postcondition, then the function is guaranteed not to fail but return a precondition and a program that, together with the given postcondition, will form a valid Hoare triple. It is also very efficient: the number of verification conditions that have to be checked is linear, actually bounded by the size of the proof outline.

Theorem 6.1 (*syn*-Sound-And-Complete). *Consider* $\vdash_a \langle R\rangle \texttt{po} \langle S\rangle$ *and* $s \in S$ *with* $s \neq fail$. *Then* $syn(\langle R\rangle \texttt{po} \langle S\rangle, s) = (r, \texttt{prog})$ *with* $r \in R$, $r \neq fail$, $\texttt{prog} \in drv(sketch(\texttt{po}))$, *and* $\models_d \{r\}\texttt{prog}\{s\}$. *The number of SMT solver calls is at most* $|\texttt{po}|$.

Function *syn* is given in Figure 10. If several cases apply, the topmost one will be taken. This means the invocation will always return $(fail, -)$ on an empty precondition, and the *fail* predicate in a precondition will always be skipped. When we have a command, we go through the predicates r in the precondition until we find one that justifies the given postcondition s. Since $s \in S$ and we started from a proof outline in realizability logic, the search for r is guaranteed to be successful. Moreover, for the guarantees given by Theorem 6.1 it does not matter which predicate r we take. The notion of soundness in realizability logic gives the guarantee that the earlier (in the program text) assertions can handle every r. But of course the choice has an influence on the shape of the program.

For a sequential composition $\texttt{po}_1;\texttt{po}_2$, we take the given predicates s, propagate it through $\texttt{po}_2$ to obtain $(t, \texttt{prog}_2)$, propagate t through $\texttt{po}_1$ to get $(r, \texttt{prog}_1)$, and return $(r, \texttt{prog}_1;\texttt{prog}_2)$. Importantly, also here we have the guarantee that both calls will be successful. When we have a non-terminal, we consult the proof outlines for the right-hand sides. For an angelic choice among two right-hand sides, we first try to synthesize a program from the left proof outline, and if we fail we know that we will be successful on the right. Whether the synthesis with the left proof outline will succeed can be foreseen by checking if the target post condition s is in the post condition of the left proof outline.

$$
\begin{aligned}
syn(\langle\emptyset\rangle \mathsf{po} \langle S\rangle, s) &:= (fail, -)\\
syn(\langle\{fail\} \cup R\rangle \mathsf{po} \langle S\rangle, s) &:= syn(\langle R\rangle \mathsf{po} \langle S\rangle, s)\\
syn(\langle\{r\} \cup R\rangle \mathsf{com} \langle S\rangle, s) &:= [\![\mathsf{com}]\!](r) \preceq_d s \ \texttt{?}\ (r, \mathsf{com}) \ \texttt{:}\ syn(\langle R\rangle \mathsf{com} \langle S\rangle, s)\\
syn(\mathsf{po}_1; \mathsf{po}_2, s) &:= \ \texttt{let}\ (t, \mathsf{prog}_2) = syn(\mathsf{po}_2, s)\ \texttt{and}\ (r, \mathsf{prog}_1) = syn(\mathsf{po}_1, t)\ \texttt{in}\ (r, \mathsf{prog}_1; \mathsf{prog}_2)\\
syn(\mathsf{N}(\mathsf{po}), s) &:= syn(\mathsf{po}, s)\\
syn(\mathsf{po}_1 \mid \mathsf{po}_2, s) &:= \ \texttt{let}\ (r, \mathsf{prog}) \in syn(\mathsf{po}_1, s)\ \texttt{in}\ r \neq fail\ \texttt{?}\ (r, \mathsf{prog})\ \texttt{:}\ syn(\mathsf{po}_2, s)\\
syn(\langle\{r\} \cup R\rangle \mathsf{po}_1 \langle S\rangle + \langle\{r\} \cup R\rangle \mathsf{po}_2 \langle S\rangle, s) &:= outl(\langle\{r\}\rangle sketch(\mathsf{po}_i) \langle\{s\}\rangle) = \mathsf{abort}\ \texttt{?}\ syn(\langle R\rangle \mathsf{po}_1 \langle S\rangle + \langle R\rangle \mathsf{po}_2 \langle S\rangle, s)\ \texttt{:}\\
&\quad \texttt{let}\ \ \mathsf{po}_i' = outl(\langle\{r\}\rangle sketch(\mathsf{po}_i) \langle\{s\}\rangle)\ \texttt{and}\ (r, \mathsf{prog}_i) = syn(\mathsf{po}_i', s)\ \texttt{in}\ \ (r, \mathsf{prog}_1 + \mathsf{prog}_2)\\
syn(\langle\{i\} \cup I\rangle \mathsf{po}^* \langle\{i\} \cup I\rangle, s) &:= i \not\preceq_d s\ \texttt{?}\ syn(\langle I\rangle \mathsf{po}^* \langle I\rangle, s)\ \texttt{:}\\
&\quad \texttt{let}\ \ \mathsf{po}' = outl(\langle\{i\}\rangle sketch(\mathsf{po}) \langle\{i\}\rangle)\ \texttt{and}\ (i, \mathsf{prog}) = syn(\mathsf{po}', i)\ \texttt{in}\ (i, \mathsf{prog}^*)
\end{aligned}
$$

Fig. 10. Definition of the synthesis function.

The involved cases are for demonic choices and loops. These are the cases in which the program logics had to consider single predicates. This is mimicked here, and we first discuss the demonic choice. We go through all predicates r in the given precondition. We try to construct proof outlines for both branches in which r is the precondition and s is the postcondition. This is the task of the calls $outl(\langle\{r\}\rangle sketch(\mathsf{po}_1) \langle\{s\}\rangle)$ and $outl(\langle\{r\}\rangle sketch(\mathsf{po}_2) \langle\{s\}\rangle)$. We argue in a moment why these calls can actually be looked up in the given proof outline, and therefore mean no effort. If one of these calls aborts, we continue with the next r. Otherwise, we obtain the proof outlines po_1' and po_2'. We invoke $syn(\mathsf{po}_1', s)$ and $syn(\mathsf{po}_2', s)$. They will for sure return r as it is the only precondition, but we need the programs to be able to return $(r, \mathsf{prog}_1 + \mathsf{prog}_2)$.

For loops, we have to make an assumption on the form of the proof outline: the loop invariant should not be lost through weakening. If assertions have to be weakened, we assume the proof outline takes the form $\langle R\rangle \langle I\rangle \mathsf{po}^* \langle I\rangle \langle S\rangle$. Like in the previous case, we go through the predicates $i \in I$ to find one that is stronger than the given postcondition s. If it has been found, it remains to synthesize the program for the loop body. We reconstruct the proof outline with i as the pre and postcondition, and rely on syn to determine the program. If the proof outline takes the form $\langle R\rangle \langle I\rangle \mathsf{po}^* \langle I\rangle \langle S\rangle$, instead of returning an i of I, we return an r of R with $r \preceq_d i$.

We elaborate on why the function calls $outl(\langle\{r\}\rangle sketch(\mathsf{po}) \langle\{s\}\rangle)$ mean no overhead. The point is that the Rules (**DEM**) and (**LOOP**) can only deal with single predicates in the precondition. This means when we constructed the proof outline of interest, we have constructed the proof outlines that we now need as a byproduct. We just have to store them explicitly to be able to look them up now. This explains the case of loops.

In the case of demonic choices, the proof outline construction may abort, and we explain how to handle this with hashing. Since Rule (**DEM**) requires a singleton as the precondition, we must have done proofs of the form $\langle\{r_i\}\rangle sketch(\mathsf{po}_1') + sketch(\mathsf{po}_2') \langle S_i\rangle$. From the requirements of Rule (**DEM**), we also have the proof outlines $\langle\{r_i\}\rangle \mathsf{po}_j' \langle S_i\rangle$ for $j = 1, 2$. The proofs might have been gathered to form the proof outline $\langle R\rangle \mathsf{po}_1 + \mathsf{po}_2 \langle S\rangle$. Since the predicate s is in S, there must be a selection S_i with $s \in S_i$. We can reuse the corresponding proof outlines $\langle\{r_i\}\rangle \mathsf{po}_j' \langle S_i\rangle$ by weakening the postcondition S_i to the singleton $\{s\}$.

6.1 Example: Synthesizing Factorial Function

Recall the sketch `sketch` from last section's example, Figure 9. In the last section, we have proven $\models_a \langle R\rangle \mathtt{sketch} \langle S\rangle$ with $R = \{\mathtt{x} = 42 \wedge \mathtt{y} = 1\}$ and $S = \{\mathtt{y} = 42!\}$. In this example, we will concretize the sketch to a program `prog` such that $\models_d \{r\}\mathtt{prog}\{s\}$ holds. To do so, we use the

function *syn*. The whole proof outline is called po. We use the same abbreviations as in the last example. Additionally, with T8 we abbreviate the full realizability triple of Line 8. That means T8 stands for Lines 7 through 9. Other realizability triples are abbreviated similarly. The realizability triple around the nonterminal is abbreviated by TN, i.e. TN stands for Lines 5 to 16. We abbreviate the selection of Lines 15 and 16 with A_{15}. Other selections are abbreviated similarly. Let $\{r\} = R$ and $\{s\} = S$. We start by calling $syn(\langle R\rangle\texttt{sketch}\langle S\rangle, s)$. More specifically, the call is $syn(\langle R\rangle\texttt{body}^*\langle I\rangle\,;\texttt{T22}, s)$. The selection I is the singleton $\{i\}$. The function invokes the call $syn(\texttt{T22}, s)$ which returns $(i, \texttt{assume(x = 0)})$. Next, it invokes $syn(\langle R\rangle\texttt{body}^*\langle I\rangle, i)$ which, in turn, invokes $syn(\langle\{i\}\rangle\texttt{body}\langle\{i\}\rangle, i)$ because $i \preceq_d i$ holds trivially. We can reuse the already done proof $\langle\{i\}\rangle\texttt{body}\langle\{i\}\rangle$ as po′. The call $syn(\langle\{i\}\rangle\texttt{body}\langle\{i\}\rangle, i)$ invokes $syn(\langle A_{15}\rangle\texttt{x--}\langle\{i\}\rangle, i)$. The call $syn(\langle\{A_{15}[0]\} \cup \{A_{15}[1]\}\rangle\texttt{x--}\langle\{i\}\rangle, i)$ will invoke $syn(\langle\{A_{15}[1]\}\rangle\texttt{x--}\langle\{i\}\rangle, i)$ because the result of $[\![\texttt{x--}]\!](A_{15}[0])$ is not more precise than i. However, the call $syn(\langle\{A_{15}[1]\}\rangle\texttt{x--}\langle\{i\}\rangle, i)$ returns $(A_{15}[1], \texttt{x--})$. Here, we witness a foreshadowing on how the wrong angelic choice will be eliminated. The eliminated predicate stems from the left production of the nonterminal which will not be able to produce the now selected predicate $A_{15}[1]$. We continue the next recursive call of $syn(\langle\{i\}\rangle\texttt{body}\langle\{i\}\rangle, i)$, i.e. $syn(\texttt{T4}\,;\texttt{TN}, A_{15}[1])$. It invokes $syn(\texttt{TN}, A_{15}[1])$ This call first checks whether the left branch can produce the target predicate. This is not the case, so eventually $(\mathit{fail}, -)$ is returned, eliminating the left branch. The right branch is now considered, where $syn(\texttt{T12}, A_{15}[1])$ resolves to $(A_5[0], \texttt{y = y*x})$. Therefore, the next call is $syn(\texttt{T4}, A_5[0])$ which resolves to the pair $(i, \texttt{assume (x>0)})$. Getting back to the recursive call of the loop, $syn(\langle R\rangle\texttt{body}^*\langle I\rangle, i)$ yields the pair $(r, (4\,;12\,;17)^*)$ because $r \preceq_d i$ with $\{r\} = R$. In total, we get $(r, \texttt{prog}) = (r, (4; 12; 17)^*; 22)$ as output. We know that $\models_d \{r\}\texttt{prog}\{s\}$ holds. This means, the program indeed calculates 42 factorial.

7 APPLICATION: MEMORY MANAGEMENT IN LOCK-FREE DATA STRUCTURES

We show how to employ realizability and realization logic to automatically generate code for the memory management in lock-free data structures. This will also be the setting for our experiments. There are several aspects that make this setting interesting for benchmarks: the states are type assignments and the semantics of commands is an abstract one. So SemGuS [23] is an ideal choice, while SyGuS [2] and solver-aided languages [39, 42] need an encoding to capture the synthesis tasks. The programs contain loops, which is difficult for SyGuS solvers. Finally, there are many synthesis options, and the right ones may be far apart. This creates a large space of potential solutions [3] that our compositional method manages to explore in a matter of seconds.

Safe Memory Reclamation. The asynchronous nature of lock-free data structures makes manual memory management difficult: if the threads do not synchronize, how does a thread know that it is safe to free a memory cell, or safe to access it? The solution is to add to the data structure a safe memory reclamation algorithm (SMR) which acts as a central instance that is informed about the intentions of all threads. If a thread wants to access a memory cell, it asks the reclamation algorithm to protect the cell. If it wants to free a memory cell, it asks the SMR to do so. As the reclamation algorithm manages the protections, it can defer the free until it is safe. Garbage collection is an SMR, but there are more efficient solution. Epochs are a simple model [13, 16], we consider here the hazard pointers that have been proposed for addition to the next C++ standard [31]. To be precise, we support the elaborate version in which the first hazard pointer is stronger than the second.

Reclamation algorithms tend to implement lock-free data structures. This means protection calls are non-atomic, and may be successful or fail due to interference. The reclamation algorithm will not be able to determine success. Instead, the thread that issued the protection call will have to find out by checking global invariants: if, for example, a sentinal node has not changed when

returning from the call, then the call was successful. Unfortunately, finding the right combination of protection calls and checks has turned out error-prone.

Meyer and Wolff [26, 27] gave a type system that checks memory safety. It can be instantiated to different reclamation algorithms and applies to a variety of data structures. We report on experiments with the queues MS and DGLM, and the sets ORVYY, Michael, VechevCAS, and VechevDCAS. The type system is control-flow sensitive, which means the typing can be written as a proof outline in Hoare logic. A type check on a program prog is successful, if the triple $\vdash_t \langle \neg fail \rangle \texttt{prog} \langle \neg fail \rangle$ can be derived: starting from no assumptions about the protection of pointers, the program will not fail. This means all pointers were protected before they were dereferenced. Appendix B.1 provides more background on [26, 27]. An important detail is that they use code annotations to inform the type system about protections that can otherwise otherwise only be inferred with a shape analysis. These annotations have to be discharged separately. We omit these checks in our experiments, and therefore our synthesis is only valid relative to the validity of the annotations.

Instantiation. We instantiate the parameters of our development. Meyer and Wolff [26, 27] track the protection of memory cells by assigning types from a set $\mathbb{T}$ to the pointers in the program *Vars*. Our states, predicates, and selections are thus $States = Vars \to \mathbb{T}$, $Predicates = \mathcal{P}(Vars \to \mathbb{T}) \cup \{fail\}$, and $Selections = \mathcal{P}(\mathcal{P}(Vars \to \mathbb{T}) \cup \{fail\})$.

Our implementation needs an assertion language to represent predicates and selections. We use $Predicates^{\#} = (Vars \to \mathbb{T}) \cup \{fail\}$ and $Selections^{\#} = \mathcal{P}(Predicates^{\#})$. The definition of the abstract predicates is due to [26, 27]. The first step is to apply a Cartesian abstraction on the predicates, which yields $Vars \to \mathcal{P}(\mathbb{T})$. A Cartesian predicate $\{\mathsf{x} : \{t_1, t_2\}\} \times \{\mathsf{y} : \{t\}\}$ represents the concrete predicate $\{\mathsf{x} : t_1 \wedge \mathsf{y} : t, \mathsf{x} : t_2 \wedge \mathsf{y} : t\}$. The second step is to exploit the fact that the types form a lattice, and represent $\mathsf{x} : \{t_1, t_2\}$ by $\mathsf{x} : t_1 \sqcup t_2$. The details are in Appendix B.5.

The commands are the usual ones in C. The semantics is from [27] and defined in a way that works with the above abstraction.

Synthesis. Given a lock-free data structure and a reclamation algorithm, our goal is to synthesize in the data structure calls to the reclamation algorithm and code annotations that, together, guarantee memory safety, more precisely, make the above type check go through.

First, we enrich every line of code with a non-terminal from which calls to the SMR algorithm and invariant annotations may be generated. In the case of a single hazard pointer, for example, the non-terminal has the definition to the left. For each variable v, we may issue a protection, add an annotation saying that the variable has not been freed, or simply skip the annotation. For global invariants, we use heuristics of where to put them, which improves the scalability. The result of this phase is a sketch sketch. Next, we try to prove $\vdash_a \langle \neg fail \rangle \texttt{sketch} \langle \neg fail \rangle$ in realizability logic. We use the algorithm from Section 5 and check the validity of the corresponding verification conditions. If successful, we also get a proof outline that we feed to realization logic, implemented in the synthesis algorithm from Section 6. The algorithm has the guarantee to be successful, and returns a program that corresponds to the original code with suitable annotations added. What is interesting is that we can control the synthesis function to avoid reclamation calls and code annotations whenever possible. We give details on the implementation and its behavior on experiments.

```
1 AC ::= skip; |
2 atomic {@inv active(v);} |
3 (in:protect(v);re:protect(v);)
```

8 EVALUATION

We implemented our synthesis algorithm in a C++ program. As we do not yet have an assertion language for selections, our implementation targets the memory reclamation problem discussed

Table 1. Results of experiments of the Pessimistic and Optimistic Approach conducted on an Apple M2.

			Max/Avg/Med $\lvert R\rvert$	Max/Avg/Med $\preceq_d$ per $\preceq_a$	Pessimistic		Optimistic	
					⏱ *vc*	⏱ *syn*	⏱ *syn*	⏱ *vc*
HP1	Treiber's [43]	Push	6 / 1.57 / 1	12 / 1.85 / 1	< 0.1*s*	< 0.1*s*	< 0.1*s*	< 0.1*s*
		Pop	6 / 1.43 / 1	18 / 1.71 / 1	< 0.1*s*	< 0.1*s*	< 0.1*s*	< 0.1*s*
	MS [30]	EnQ	5 / 1.27 / 1	12 / 1.40 / 1	< 0.1*s*	< 0.1*s*	< 0.1*s*	< 0.1*s*
	DGLM [9]	EnQ	17 / 1.3 / 1	64 / 1.9 / 1	0.7*s*	< 0.1*s*	< 0.1*s*	< 0.1*s*
HP2	MS [30]	DeQ	90 / 1.6 / 1	1K / 2.92 / 1	17*s*	0.6*s*	0.6*s*	< 0.1*s*
	DGLM [9]	DeQ	29 / 1.74 / 1	5K / 18.9 / 1	249*s*	6.3*s*	5.9*s*	< 0.1*s*
	ORVYY [33]	Add	158 / 3.85 / 1	276 / 3.6 / 1	1.4*s*	< 0.1*s*	< 0.1*s*	< 0.1*s*
		Rm	32 / 1.29 / 1	276 / 1.96 / 1	0.6*s*	0.1*s*	0.1*s*	< 0.1*s*
	Michael [28]	Add	127 / 3.06 / 1	5K / 13.4 / 1	3.1*s*	2.1*s*	2.2*s*	< 0.1*s*
		Rm	127 / 3.15 / 1	5K / 13.7 / 1	1.5*s*	2.1*s*	2.1*s*	< 0.1*s*
	VechevCAS [45]	Add	25 / 1.7 / 1	148 / 3.1 / 1	0.6*s*	< 0.1*s*	< 0.1*s*	< 0.1*s*
		Rm	21 / 1.7 / 1	178 / 3.1 / 1	0.1*s*	< 0.1*s*	< 0.1*s*	< 0.1*s*
	VechevDCAS [45]	Add	45 / 2.64 / 1	341 / 6.68 / 1	0.4*s*	< 0.1*s*	< 0.1*s*	< 0.1*s*
		Rm	21 / 1.7 / 1	145 / 2.9 / 1	0.1*s*	< 0.1*s*	< 0.1*s*	< 0.1*s*

above. It can handle different reclamation algorithms and challenging data structures, though. We give details on the implementation and elaborate on the performance, which was surprisingly good.

Implementation. The most important technique we implemented to improve the scalability is an optimistic variant of our approach. The pessimistic variant is the one from Section 7 that first validates the proof outline and then derives a program. The optimistic variants derives the program without having validated the proof outline. The point is that function *syn* checks the verification conditions it needs to construct the program anyhow, and therefore the output will be sound. The verification conditions for proof outlines have to compare selections, and each such comparison may create quadratically many solver calls.

We moreover implemented the following application domain specific heuristics. We assume local pointers do not need to be protected. We only insert annotations `@inv active(v)` for sentinel nodes. The point is that sentinel nodes will never be freed. As discussed above, this would have to be confirmed by a check that we omit. We automatically compute loop invariants from the predicates that are available when entering a loop. There is no saturation nor widening. Still, the approach worked for all data structures we examined. For resolving non-terminals, we prefer `skip` over SMR calls and SMR calls over annotations.

Results. We experimented with the seven lock-free data structures and two hazard pointer variants discussed in Section 7. The environment is an Apple M2 with 8GB RAM. We do not have a solver backend but hand the bitvectors we need within our implementation. The results are given in Table 1. In some methods we used hints that can be deduced by a lightweight shape analysis.

The first column shows the maximum, average, and median size of a selection. The next shows the number of invocations of the comparison function on predicates per comparison of selection. For example, $\{a\} \preceq_a \{b_1, b_2, b_3\}$ may yield three comparisons $a \preceq_d b_i$. This only applies to the pessimistic approach. The next two columns display the time for the functions *vc* and *syn* in the pessimistic approach. The last two columns are for the optimistic approach. It clearly outperforms the pessimistic approach, but both are quite fast.

The key insight of our experiments is that there are two classes of synthesis options. The options that are handled well by our approach lead to similar predicates and thus small selections, or they fail early. The options that are difficult for our approach lead to new predicates that fail late. This can be seen in the benchmarks with two hazard pointers. In the set implementations, the order

among the hazard pointers matters, and if we try to synthesize a protection with the wrong order we fail early. For DGLM, the order does not matter and the synthesis time increases dramatically.

Taking a step back, the experiments indicate that our realizability and realization logics should be combined with an outer search [3]. The search would concentrate on the synthesis options that are difficult for our approach and only fix them. Our approach would handle the remaining, easier options. We see this as a promising direction for future work.

9 RELATED WORK

We have already discussed the related work on SemGuS, SyGuS, and solver-aided programming in the introduction. There is a body of work on programming models with angelic non-determinism, dating back to Floyd [12]. The common case, however, is that the angelic choices are made at runtime, and therefore can react to the demonic choices. These models are related to two-player zero-sum graph games [14] typically used in reactive synthesis, as originally proposed by Church [7] and later developed by Pnueli and Rosner [35]. In SemGuS, the angelic choices have to be made up-front, which adds a form of imperfect information [38] that the common case does not have to deal with. This imperfect information shines through in Rules (**DEM**) and (**LOOP**), where the angelic player cannot react. Two works, however, are closer to our development.

Mamouras [24] presents a Hoare logic to reason about the correctness of programs with (runtime) angelic non-determinism. His assertions are classical predicates, which means the proof construction is forced to decide on a synthesis option early on, may later fail due to incompatible decisions, and in this case has to backtrack. To avoid precisely this backtracking, we proposed selections as assertions in realizability logic. As a consequence, the resulting theories are largely different.

Celiku and von Wright [6] consider the same class of programs as Mamouras, but try to refine angelic choices to conditionals and loops. The refinement is guided by a proof outline, and may be compared to our realization logic. An important difference is that they work with ordinary predicates, and try to synthesize appropriate ones. We work with the new selections, instead, in which we have gathered the relevant predicates.

Also related is the recent [32] that uses angelic non-determinism to synthesize recursive functions. During recursive calls, they select an output value for the function and later check whether the choice can be justified. The work [21, 41] infers a program from verification conditions that contain unknown program parts which, in the case of [21], should be filled by library components. The crucial difference to realization logic is that we have eagerly analyzed the program space, and the task of *syn* amounts to a concretization rather than a search. We have not seen a backtracking freedom guarantee in related work.

The deductive approach decomposes the task of synthesizing a program into synthesis tasks for subprograms [25]. A popular idea is to factorize the subprograms along input-output equivalence so that the computation can proceed with equivalence classes [1, 11, 15, 34, 36, 44]. What is different in our work is that we try to be eager in the bottom-up construction, and consider a set of synthesis options whose behavior does not have to match. Related to our approach are [10, 46] that weaken the input-output equivalence by considering an abstract domain. A solution for the abstract domain is then a restricted class of programs that can be searched more efficiently. Realizability and realization logic may prove useful for this task.

We introduced our eager approach to reduce the number of costly verification queries, a prominent endeavor in search-based synthesis [3]. An important development is to inform the solver about semantic properties of the program sought [4]. In the context of example-based specifications, a semantic property is the invariance to perturbations of the example set, which can be incorporated by enriching the example set.

REFERENCES

[1] A. Albarghouthi, S. Gulwani, and Z. Kincaid. 2013. Recursive Program Synthesis. In *CAV (LNCS, Vol. 8044)*. Springer, 934–950.

[2] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. 2013. Syntax-guided synthesis. In *FMCAD*. IEEE, 1–8.

[3] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama. 2018. Search-based program synthesis. *CACM* 61, 12 (2018), 84–93.

[4] S. An, R. Singh, S. Misailovic, and R. Samanta. 2020. Augmented example-based synthesis using relational perturbation properties. *PACMPL* 4, POPL (2020), 56:1–56:24.

[5] C. Calcagno, P. W. O'Hearn, and H. Yang. 2007. Local Action and Abstract Separation Logic. In *LICS*. IEEE, 366–378.

[6] O. Celiku and J. von Wright. 2003. Implementing angelic nondeterminism. In *Asia-Pacific Software Engineering Conference*. IEEE, 176–185.

[7] A. Church. 1957. Applications of recursive arithmetic to the problem of circuit synthesis. *Summaries of the Summer Institute of Symbolic Logic* 1 (1957), 3–50.

[8] P. Cousot. 2021. *Principles of Abstract Interpretation.* The MIT Press.

[9] S. Doherty, L. Groves, V. Luchangco, and M. Moir. 2004. Formal Verification of a Practical Lock-Free Queue Algorithm. In *FORTE (LNCS, Vol. 3235)*. Springer, 97–114.

[10] D. Drachsler-Cohen, S. Shoham, and E. Yahav. 2017. Synthesis with Abstract Examples. In *CAV (LNCS, Vol. 10426)*. Springer, 254–278.

[11] J. K. Feser, S. Chaudhuri, and I. Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *PLDI*. ACM, 229–239.

[12] R. W. Floyd. 1967. Nondeterministic Algorithms. *JACM* (1967).

[13] K. Fraser. 2004. *Practical lock-freedom.* Ph. D. Dissertation. University of Cambridge.

[14] E. Grädel, W. Thomas, and T. Wilke (Eds.). 2002. *Automata, Logics, and Infinite Games: A Guide to Current Research.* LNCS, Vol. 2500. Springer.

[15] S. Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *POPL*. ACM, 317–330.

[16] T. L. Harris. 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *DISC (LNCS, Vol. 2180)*. Springer, 300–314.

[17] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *CACM* 12, 10 (1969), 576–580.

[18] Q. Hu, J. Breck, J. Cyphert, L. D'Antoni, and T. W. Reps. 2019. Proving Unrealizability for Syntax-Guided Synthesis. In *CAV (LNCS, Vol. 11561)*. Springer, 335–352.

[19] Q. Hu, J. Cyphert, L. D'Antoni, and T. W. Reps. 2020. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In *PLDI*. ACM, 1128–1142.

[20] Q. Hu and L. D'Antoni. 2018. Syntax-Guided Synthesis with Quantitative Syntactic Objectives. In *CAV (LNCS, Vol. 10981)*. Springer, 386–403.

[21] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. 2010. Oracle-guided component-based program synthesis. In *ICSE*. ACM, 215–224.

[22] J. Kim, L. D'Antoni, and T. Reps. 2023. Unrealizability Logic. *PACMPL* 7, POPL, 659–688.

[23] J. Kim, Q. Hu, L. D'Antoni, and T. Reps. 2021. Semantics-Guided Synthesis. *PACMPL* 5, POPL, 1–32.

[24] K. Mamouras. 2016. Synthesis of Strategies Using the Hoare Logic of Angelic and Demonic Nondeterminism. *LMCS* 12, 3.

[25] Z. Manna and R. J. Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM TOPLAS* 2, 1 (1980), 90–121.

[26] R. Meyer and S. Wolff. 2019. Decoupling Lock-Free Data Structures from Memory Reclamation for Static Analysis, In POPL. *PACMPL*, Article 58, 31 pages.

[27] R. Meyer and S. Wolff. 2020. Pointer Life Cycle Types for Lock-Free Data Structures with Memory Reclamation, In POPL. *PACMPL*, Article 68, 36 pages.

[28] M. M. Michael. 2002. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *SPAA*. ACM, 73—-82.

[29] M. M. Michael. 2002. Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes. In *PODC*. ACM, 10 pages.

[30] M. M. Michael and M. L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*. ACM, 267–275.

[31] M. M. Michael, M. Wong, P. McKenney, A. Hunter, D. S. Hollman, JF Bastien, H. Boehma, D. Goldblatt, F. Birbacher, and M. Stearn. 2023. Hazard Pointers for C++26.

[32] A. Miltner, A. T. Nuñez, A. Brendel, S. Chaudhuri, and I. Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. *PACMPL* 6, Article 21 (2022), 29 pages.

[33] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. 2010. Verifying Linearizability with Hindsight. In *PODC*. ACM, 85—-94.

[34] P. M. Osera and S. Zdancewic. 2015. Type-and-example-directed program synthesis. In *PLDI*. ACM, 619–630.

[35] A. Pnueli and R. Rosner. 1989. On the Synthesis of a Reactive Module. In *POPL* (Austin, Texas, USA). ACM, 179—-190.
[36] O. Polozov and S. Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *OOPSLA*. ACM, 107–126.
[37] M. O. Rabin and D. S. Scott. 1959. Finite Automata and Their Decision Problems. *IBM J. Res. Dev.* 3, 2 (1959), 114–125.
[38] J. H. Reif. 1984. The complexity of two-player games of incomplete information. *J. Comput. System Sci.* 29, 2 (1984), 274–301.
[39] A. Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. U.C. Berkeley.
[40] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. 2006. Combinatorial Sketching for Finite Programs. *ASPLOS* 41, 11 (2006), 404—-415.
[41] S. Srivastava, S. Gulwani, and J. S. Foster. 2010. From program verification to program synthesis. In *POPL*. ACM, 313–326.
[42] E. Torlak and R. Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Onward!* ACM, 135–152.
[43] R. K. Treiber. 1986. Systems programming: Coping with parallelism. *Technical Report RJ 5118* (1986). https://dominoweb.draco.res.ibm.com/reports/rj5118.pdf
[44] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur. 2013. TRANSIT: specifying protocols with concolic snippets. In *PLDI*. ACM, 287–296.
[45] M. Vechev and E. Yahav. 2008. Deriving Linearizable Fine-Grained Concurrent Objects. In *PLDI*. ACM, 125—-135.
[46] X. Wang, I. Dillig, and R. Singh. 2018. Program Synthesis Using Abstraction Refinement. *PACMPL* 2, POPL (2018), 63:1–63:30.
[47] S. Wolff. 2021. *Verifying Non-blocking Data Structures with Manual Memory Management*. Ph. D. Dissertation. TU Braunschweig.

A PROOFS

We present omitted proofs and details of this paper.

A.1 Proofs for Section 3

Lemma A.1. *The function* $[\![\mathsf{com}]\!]$ *is monotonic for all commands* com*, i.e. the following equation holds:*

$$r \preceq_d s \implies [\![\mathsf{com}]\!](r) \preceq_d [\![\mathsf{com}]\!](s) \ .$$

Proof. Let r and s be elements of *Predicates* with $r \preceq_d s$.
Case 1: $[\![\mathsf{com}]\!](s) = \mathit{fail}$. Then $[\![\mathsf{com}]\!](r) \preceq_d [\![\mathsf{com}]\!](s)$.
Case 2: $[\![\mathsf{com}]\!](s) \neq \mathit{fail}$. Then, $s \neq \mathit{fail}$ and there is no p in s with $[\![\mathsf{com}]\!](p) = \mathit{fail}$. Because $s \neq \mathit{fail}$, we have $r \neq \mathit{fail}$ and $r \subseteq s$. Therefore, there also is no p in r with $[\![\mathsf{com}]\!](p) = \mathit{fail}$. Now, let p' be an element of $[\![\mathsf{com}]\!](r)$. Thus, there exists a p in r with $p' \in [\![\mathsf{com}]\!](p)$. Since $r \subseteq s$, this p also is in s and therefore, p' also is in $[\![\mathsf{com}]\!](s)$, making $[\![\mathsf{com}]\!](r)$ a subset of $[\![\mathsf{com}]\!](s)$. In total, $[\![\mathsf{com}]\!](r) \preceq_d [\![\mathsf{com}]\!](s)$ holds. □

Lemma A.2. *The relation* $\preceq_d$ *is a partial order relation on Predicates. This means, the order is reflexive, transitive and antisymmetric, i.e. the following equations hold:*

$$\forall\, r \in \mathit{Predicates} : r \preceq_d r$$
$$\forall\, r, s, t \in \mathit{Predicates} : r \preceq_d s \wedge s \preceq_d t \implies r \preceq_d t$$
$$\forall\, r, s \in \mathit{Predicates} : r \preceq_d s \wedge s \preceq_d r \implies r = s$$

Proof. Let r, s, t be elements of *Predicates*.
Reflexivity ($r \preceq_d r$):
Case 1: $r = \mathit{fail}$. We get $r \preceq_d r$ immediately. Case 2: $r \neq \mathit{fail}$. So r is a set of states. Then, $r \subseteq r$ holds.
Transitivity ($r \preceq_d s \wedge s \preceq_d t \implies r \preceq_d t$):
Case 1: $t = \mathit{fail}$. Then $r \preceq_d t$ holds. Case 2: $t \neq \mathit{fail}$. Then $s \neq \mathit{fail}$ and $r \neq \mathit{fail}$. By transitivity of the relation$\subseteq$, we get $r \preceq_d t$.
Anti-symmetry ($r \preceq_d s \wedge s \preceq_d r \implies r = s$):
If either one of r and s is *fail* the other one also has to be fail for both inequalities to hold. If neither one is *fail*, anti-symmetry follows from the anti-symmetry of $\subseteq$. □

Lemma A.3. *The partial order* $(\mathit{Predicates}, \preceq_d)$ *is a complete lattice. That means for any subset R of Predicates there exists a join and a meet. In fact, the join and meet can be computed by the following equations: Let R be a subset of Predicates.*

$$\sqcup(R) = \begin{cases} \mathit{fail} & , \mathit{fail} \in R \\ \bigcup_{r \in R} r & , \mathit{else} \end{cases}$$

$$\sqcap(R) = \begin{cases} \mathit{fail} & , R = \{\mathit{fail}\} \\ \bigcap_{r \in (R \setminus \{\mathit{fail}\})} r & , \mathit{else} \end{cases}$$

Proof. Let $R \subseteq \mathit{Predicates}$
We show that $\sqcup(R)$ is an upper bound of R: Let r be in R. Case 1: Assume $\mathit{fail} \in R$. Then $\sqcup(R) = \mathit{fail}$. Thus, $r \preceq_d \sqcup(R)$. Case 2: Assume *fail* is not in R. Then $\sqcup(R) = \bigcup_{r \in R} r$. Thus, $r \subseteq \sqcup(R)$ and $r \preceq_d \sqcup(R)$.
We show that $\sqcup(R)$ is the least upper bound of R: Let u be an upper bound of R. Case 1: Assume $\mathit{fail} \in R$. Then $\mathit{fail} \preceq_d u$. Thus, $\sqcup(R) \preceq_d u$. Case 2: Assume *fail* is not in R. Then $\sqcup(R) = \bigcup_{r \in R} r$. Since u is an upper bound, $\forall r \in R : r \subseteq u$. Thus, $(\bigcup_{r \in R} r) \subseteq u$. So, $\sqcup(R) \preceq_d u$.

We show that $\sqcap(R)$ is a lower bound of R: Let r be in R. Case 1: Assume $R = \{fail\}$. Then $r = fail = \sqcap(R)$. Thus $\sqcap(R) \preceq_d r$. Case 2: Assume $R \neq \{fail\}$. If $r = fail$, we get $\sqcap(R) \preceq_d r$. Otherwise, $\sqcap(R) = \cap_{r \in R \setminus \{fail\}} r$. Therefore, $\sqcap(R) \subseteq r$ and thus $\sqcap(R) \preceq_d r$.

We show that $\sqcap(R)$ is the greatest lower bound of R: Let l be a lower bound of R. Case 1: Assume $R = \{fail\}$. Then $\sqcap(R) = fail$. Thus $l \preceq_d \sqcap(R)$. Case 2: Assume $R \neq \{fail\}$. Let r be in $R \setminus \{fail\}$. Then $l \subseteq r$. This holds for every r. Thus, $l \subseteq \sqcap(R)$. Therefore, $l \preceq_d \sqcap(R)$. □

The Missing EMPTY Rule:

For completeness, we need to add the following rule to the calculus:

$$\frac{}{\vdash_a \langle R \rangle \mathtt{sketch} \langle \emptyset \rangle} \textbf{(EMPTY)}$$

Proof of Theorem 3.1.

We prove soundness by structural induction over the proof tree.

Base Case: We have $\vdash_a \langle \{r\} \rangle \mathtt{com} \langle \{s\} \rangle$ through the rule (**COM**). Thus, we have $[\![\mathtt{com}]\!](r) \preceq_d s$. Therefore, the realizability triple $\models_a \langle \{r\} \rangle \mathtt{com} \langle \{s\} \rangle$ holds.

Induction Step: In the first case, we have $\vdash_a \langle R \rangle \mathtt{sketch}_1; \mathtt{sketch}_2 \langle T \rangle$ through the rule (**SEQ**). Thus, we have the realizability triples $\vdash_a \langle R \rangle \mathtt{sketch}_1 \langle S \rangle$ and $\vdash_a \langle S \rangle \mathtt{sketch}_2 \langle T \rangle$. By applying the induction hypothesis, we get $\models_a \langle R \rangle \mathtt{sketch}_1 \langle S \rangle$ and $\models_a \langle S \rangle \mathtt{sketch}_2 \langle T \rangle$. Let t be an element of T. Then, there exists a predicate $s \in S$ and a program $\mathtt{prog}_2 \in drv(\mathtt{sketch}_2)$ with $\models_d \{s\} \mathtt{prog}_2 \{t\}$. And, there exists $r \in R$ and $\mathtt{prog}_1 \in drv(\mathtt{sketch}_1)$ with $\models_d \{r\} \mathtt{prog}_1 \{s\}$. By completeness of Hoare logic, we get $\vdash_d \{r\} \mathtt{prog}_1 \{s\}$ and $\vdash_d \{s\} \mathtt{prog}_1 \{t\}$. Using rule (**SEQ**), we get $\vdash_d \{r\} \mathtt{prog}_1; \mathtt{prog}_2 \{t\}$.

In the next case we have the triple $\vdash_a \langle R \rangle \mathtt{sketch} \langle S \rangle$ through the rule (**CSQ**). Thus we have $\vdash_a \langle R' \rangle \mathtt{sketch} \langle S' \rangle$ with $R \preceq_a R'$ and $S' \preceq_a S$. Through the induction hypothesis, we get $\models_a \langle R' \rangle \mathtt{sketch} \langle S' \rangle$. Now, let s be an element of S. Since $S' \preceq_a S$, there is an s' in S' with $s' \preceq_d s$. From $\models_a \langle R' \rangle \mathtt{sketch} \langle S' \rangle$ we know, that there is an r' in R' and a program $\mathtt{prog} \in drv(\mathtt{sketch})$ such that $\models_d \{r'\} \mathtt{prog} \{s'\}$ holds. Through transitivity of the relation $\preceq_d$, we also get that $\models_d \{r'\} \mathtt{prog} \{s\}$ holds. Due to monotonicity of executions, and since $R \preceq_a R'$ holds, there is an r in R for which $\models_d \{r\} \mathtt{prog} \{s\}$ is true. That means, $\models_a \langle R \rangle \mathtt{sketch} \langle S \rangle$ holds.

In the next case, we have $\vdash_a \langle R \rangle \mathtt{sketch}_1 + \mathtt{sketch}_2 \langle S \rangle$ through the rule (**DEM**). Therefore, we know $R = \{r\}$ and the realizability triples $\vdash_a \langle R \rangle \mathtt{sketch}_1 \langle S \rangle$ and $\vdash_a \langle R \rangle \mathtt{sketch}_2 \langle S \rangle$ hold. From the induction hypothesis we know that $\models_a \langle R \rangle \mathtt{sketch}_1 \langle S \rangle$ and $\models_a \langle R \rangle \mathtt{sketch}_2 \langle S \rangle$ hold. Now, let s be an element of S. Then, there exists $\mathtt{prog}_1 \in drv(\mathtt{sketch}_1)$ for which $\models_d \{r\} \mathtt{prog}_1 \{s\}$ is true. Analogously, there exists $\mathtt{prog}_2 \in drv(\mathtt{sketch}_2)$ for which $\models_d \{r\} \mathtt{prog}_2 \{s\}$ is true. We now show that $\models_d \{r\} \mathtt{prog}_1 + \mathtt{prog}_2 \{s\}$ is true. This directly implies that $\models_a \langle R \rangle \mathtt{sketch}_1 + \mathtt{sketch}_2 \langle S \rangle$ holds. Let $\mathtt{ex}$ be an execution of $\mathtt{prog}_1 + \mathtt{prog}_2$. Thus, it is either an execution of $\mathtt{prog}_1$ or $\mathtt{prog}_2$. W.l.o.g. let $\mathtt{ex}$ be in $exec(\mathtt{prog}_1)$. Since $\models_d \{r\} \mathtt{prog}_1 \{s\}$ is true, we also get that $[\![\mathtt{ex}]\!](r)$ is more precise than s. Therefore, the Hoare triple $\models_d \{r\} \mathtt{prog}_1 + \mathtt{prog}_2 \{s\}$ is true and thus $\models_a \langle R \rangle \mathtt{sketch}_1 + \mathtt{sketch}_2 \langle S \rangle$ holds.

In the next case, we have $\vdash_a \langle R \rangle \mathtt{sketch}^* \langle R \rangle$ through the rule (**LOOP**). Thus we know the realizability triple $\vdash_a \langle R \rangle \mathtt{sketch} \langle R \rangle$ holds and $R = \{r\}$. The induction hypothesis yields $\models_a \langle R \rangle \mathtt{sketch} \langle R \rangle$. From this we know that there is a $\mathtt{prog} \in drv(\mathtt{sketch})$ with $\models_d \{r\} \mathtt{prog} \{r\}$. We now show that $\models_d \{r\} \mathtt{prog}^* \{r\}$ is true. For this, we inductively show $\models_d \{r\} \mathtt{prog}^i \{r\}$ for every $i \in \mathbb{N}$. Since $\mathtt{prog}^0$ is $\mathtt{skip}$, $\models_d \{r\} \mathtt{prog}^0 \{r\}$ immediately follows. In the induction step, we show $\models_d \{r\} \mathtt{prog}^i; \mathtt{prog} \{r\}$. From the hypothesis, we know that $\models_d \{r\} \mathtt{prog}^i \{r\}$ and $\models_d \{r\} \mathtt{prog} \{r\}$ hold. Referring to the case where we have proven sequences in this proof, this implies that $\models_d \{r\} \mathtt{prog}^{i+1} \{r\}$ is true. Thus, we know that $\models_d \{r\} \mathtt{prog}^* \{r\}$ holds. Therefore, $\models_a \langle R \rangle \mathtt{sketch}^* \langle R \rangle$ is true.

Fig. 11. Definition of the Program Logic for Synthesis over proof outlines. Extensions of traditional Hoare logic are blue.

$$\textbf{(PCOM)}\ \frac{[\![\mathtt{com}]\!](r) \preceq_d s}{\vdash_p \langle\{r\}\rangle \mathtt{com} \langle\{s\}\rangle} \qquad \textbf{(PSEQ)}\ \frac{\vdash_p \langle R\rangle \mathtt{po}_1 \langle S\rangle \qquad \vdash_p \langle S\rangle \mathtt{po}_2 \langle T\rangle}{\vdash_p \langle R\rangle \mathtt{po}_1 \langle S\rangle ; \langle S\rangle \mathtt{po}_2 \langle T\rangle}$$

$$\textbf{(PLOOP)}\ \frac{\vdash_p \langle R\rangle \mathtt{po} \langle R\rangle \qquad R = \{r\}}{\vdash_p \langle R\rangle (\langle R\rangle \mathtt{po} \langle R\rangle)^* \langle R\rangle}$$

$$\textbf{(PDEM)}\ \frac{\vdash_p \langle R\rangle \mathtt{po}_1 \langle S\rangle \qquad \vdash_p \langle R\rangle \mathtt{po}_2 \langle S\rangle \qquad R = \{r\}}{\vdash_p \langle R\rangle \mathtt{po}_1 \langle S\rangle + \langle R\rangle \mathtt{po}_2 \langle S\rangle} \qquad \textbf{(PCSQ)}\ \frac{\vdash_p \langle R'\rangle \mathtt{po} \langle S'\rangle \qquad R \preceq_a R' \qquad S' \preceq_a S}{\vdash_p \langle R\rangle \mathtt{po} \langle S\rangle}$$

$$\textbf{(PANG)}\ \frac{\vdash_p \langle R\rangle \mathtt{rhs} \langle S\rangle \qquad \mathtt{N} ::= \mathtt{rhs} \mid \ldots}{\vdash_p \mathtt{N}(\langle R\rangle \mathtt{rhs} \langle S\rangle)} \qquad \textbf{(PEMPTY)}\ \frac{}{\vdash_p \langle R\rangle \mathtt{po} \langle \emptyset\rangle}$$

$$\textbf{(PGATHER)}\ \frac{\vdash_p \langle R_1\rangle \mathtt{po}_1 \langle S_1\rangle \qquad \vdash_p \langle R_2\rangle \mathtt{po}_2 \langle S_2\rangle \qquad \mathit{sketch}(\mathtt{po}_1) = \mathit{sketch}(\mathtt{po}_2)}{\vdash_p \langle R_1 \cup R_2\rangle \mathit{gather}(\mathtt{po}_1, \mathtt{po}_2) \langle S_1 \cup S_2\rangle}$$

In the next case, we have $\vdash_a \langle R\rangle \mathtt{N} \langle S\rangle$ through the rule (**ANG**). From this, we know $\vdash_a \langle R\rangle \mathtt{rhs} \langle S\rangle$ holds with $\mathtt{N} ::= \mathtt{rhs} \mid \ldots$. From the induction hypothesis we get $\models_a \langle R\rangle \mathtt{rhs} \langle S\rangle$. Since $\mathit{drv}(\mathtt{rhs})$ is a subset of $\mathit{drv}(\mathtt{N})$, we get $\models_a \langle R\rangle \mathtt{N} \langle S\rangle$ immediately.

In the last case, we have $\vdash_a \langle R_1 \cup R_2\rangle \mathtt{sketch} \langle S_1 \cup S_2\rangle$ through the rule (**GATHER**). From this, we know $\vdash_a \langle R_1\rangle \mathtt{sketch} \langle S_1\rangle$ and $\vdash_a \langle R_2\rangle \mathtt{sketch} \langle S_2\rangle$ hold. Let s be an element of $S_1 \cup S_2$. W.l.o.g. let s be an element of S_1. From the induction hypothesis we know that there is an r in $R_1 \subseteq R_1 \cup R_2$ and a program $\mathtt{prog} \in \mathit{drv}(\mathtt{sketch})$ with $\models_d \{r\}\mathtt{prog}\{s\}$. Thus, we get $\models_a \langle R_1 \cup R_2\rangle \mathtt{sketch} \langle S_1 \cup S_2\rangle$. This concludes the soundness proof.

We proceed with the proof for completeness. In the first case, the postcondition is empty. We have $\models_a \langle R\rangle \mathtt{sketch} \langle\emptyset\rangle$. Using the rule (**EMPTY**), we get $\vdash_a \langle R\rangle \mathtt{sketch} \langle\emptyset\rangle$.

In the second case, the postcondition is not empty. We have $\models_a \langle R\rangle \mathtt{sketch} \langle S\rangle$. Thus, for every $s \in S$ there is a program $\mathtt{prog} \in \mathit{drv}(\mathtt{sketch})$ and a predicate $r \in R$ for which $\models_d \{r\}\mathtt{prog}\{s\}$. Since Hoare logic is complete, we have $\vdash_d \{r\}\mathtt{prog}\{s\}$. We can mimic this proof in realizability logic for sketch by using rule (**ANG**) mimicking the derivation of prog from sketch. Thus, we have $\vdash_a \langle r\rangle \mathtt{sketch} \langle s\rangle$. Since S is finite, we only need to do the proof for finitely many programs. These proofs are then combined using the rule (**GATHER**) multiple times to derive $\vdash_a \langle R'\rangle \mathtt{sketch} \langle S\rangle$ for the subset R' of R containing the used predicates r of R. Then, using rule (**CSQ**), we get $\vdash_a \langle R\rangle \mathtt{sketch} \langle S\rangle$. □

A.2 Proofs for Section 4

To properly define the notation of a valid proof outline, we state a a proof system for that ranges over proof outlines in Figure 11. When we write $\vdash_p \mathtt{N}(\mathtt{po})$ we mean $\vdash_p \langle R\rangle \mathtt{N}(\mathtt{po}) \langle S\rangle$ where R resp. S are the union of all pre- resp. postconditions of the subproofs in po. The function *sketch* extracts the program sketch out of the proof outline:

Definition 1. *The function gather combines two proofs over the same sketch.*

$$\mathit{gather} : (\mathit{prfOutls} \times \mathit{prfOutls}) \rightharpoonup \mathit{prfOutls}$$

$$
\begin{aligned}
\mathit{gather}(\langle R_1\rangle \mathsf{com}\langle S_1\rangle, \langle R_2\rangle \mathsf{com}\langle S_2\rangle) &= \langle R_1 \cup R_2\rangle \mathsf{com}\langle S_1 \cup S_2\rangle \\
\mathit{gather}(\langle R_1\rangle \mathsf{p1}^*\langle S_1\rangle, \langle R_2\rangle \mathsf{p2}^*\langle S_2\rangle) &= \langle R_1 \cup R_2\rangle \mathit{gather}(\mathsf{p1}, \mathsf{p2})^*\langle S_1 \cup S_2\rangle \\
\mathit{gather}(\mathsf{po}_1 ; \mathsf{po}_2, \mathsf{po}_1' ; \mathsf{po}_2') &= \mathit{gather}(\mathsf{po}_1, \mathsf{po}_1') ; \mathit{gather}(\mathsf{po}_2, \mathsf{po}_2') \\
\mathit{gather}(\mathsf{po}_1 + \mathsf{po}_2, \mathsf{po}_1' + \mathsf{po}_2') &= \mathit{gather}(\mathsf{po}_1, \mathsf{po}_1') + \mathit{gather}(\mathsf{po}_2, \mathsf{po}_2') \\
\mathit{gather}(\mathsf{po}_1 \mid \mathsf{po}_2, \mathsf{po}_1' \mid \mathsf{po}_2') &= \mathit{gather}(\mathsf{po}_1, \mathsf{po}_1') \mid \mathit{gather}(\mathsf{po}_2, \mathsf{po}_2') \\
\mathit{gather}(\mathsf{N}(\mathsf{po}_1 \mid \mathsf{po}_2), \mathsf{N}(\mathsf{po}_1' \mid \mathsf{po}_2')) &= \mathsf{N}(\mathit{gather}(\mathsf{po}_1, \mathsf{po}_1') \mid \mathsf{po}_2 \mid \mathsf{po}_2')
\end{aligned}
$$

In the last case, po_1 *and* po_1' *are the productions where the right hand side in both proof outlines are the same. The proof outlines* po_2 *and* po_2' *reason over different sketches and their proof outlines therefore cannot be combined.*

In order for the function $\mathit{gather}(\mathsf{po}_1, \mathsf{po}_2)$ *to be defined, it is required that* po_1 *and* po_2 *are proofs over the same sketch, i.e.* $\mathit{sketch}(\mathsf{po}_1) = \mathit{sketch}(\mathsf{po}_2)$.

We need the following theorem:

Theorem A.4 (Equivalence). *The following implication holds for any proof* po *of prfOutls and any angels R and S:*

$$\vdash_p \langle R\rangle \mathsf{po} \langle S\rangle \implies \vdash_a \langle R\rangle \mathit{sketch}(\mathsf{po}) \langle S\rangle \ .$$

The following implication holds for any sketch sketch *of* Sketches *and any angels R and S:*

$$\vdash_a \langle R\rangle \mathsf{sketch} \langle S\rangle \implies \exists\, \mathsf{po} \in \mathit{prfOutls} : \mathit{sketch}(\mathsf{po}) = \mathsf{sketch} \wedge \vdash_p \langle R\rangle \mathsf{po} \langle S\rangle \ .$$

Proof of Theorem A.4.

We start with the first implication. We show the implication by structural induction over the proof tree.

Base Case:: We have $\vdash_p \langle\{r\}\rangle \mathsf{com} \langle\{s\}\rangle$ through (**PCOM**). Thus, we know $[\![\mathsf{com}]\!](r) \preceq_d s$ is true. With rule (**COM**) we directly get $\vdash_a \langle\{r\}\rangle \mathsf{com} \langle\{s\}\rangle$.

Induction Step: Since all induction steps are only applying the induction hypothesis and rebuilding the proof in the other proof system, we demonstrate it on one case only. We have the proof outline $\vdash_p \langle R\rangle \mathsf{po}_1 ; \mathsf{po}_2 \langle T\rangle$ through (**PSEQ**). With the preconditions and the induction hypothesis we get $\vdash_a \langle R\rangle \mathsf{sketch}_1 \langle S\rangle$ and $\vdash_a \langle S\rangle \mathsf{sketch}_2 \langle T\rangle$ where the sketches are of the corresponding proof. Using the rule (**SEQ**), we get $\vdash_a \langle R\rangle \mathsf{sketch}_1 ; \mathsf{sketch}_2 \langle T\rangle$

We continue with the second implication. Again, we show the implication by structural induction over the proof tree.

Base Case:: We have $\vdash_a \langle\{r\}\rangle \mathsf{com} \langle\{s\}\rangle$ through (**COM**). Thus, we know $[\![\mathsf{com}]\!](r) \preceq_d s$ is true. With rule (**PCOM**) we directly get $\vdash_p \langle\{r\}\rangle \mathsf{com} \langle\{s\}\rangle$.

Induction Step: Since all induction steps are only applying the induction hypothesis and rebuilding the proof in the other proof system, we demonstrate it on one case only. We have the realizability triple $\vdash_a \langle R\rangle \mathsf{sketch}_1 ; \mathsf{sketch}_2 \langle T\rangle$ through (**SEQ**). With the preconditions and the induction hypothesis we get two proofs po_1 and po_2 for which the following holds: $\vdash_p \langle R\rangle \mathsf{po}_1 \langle S\rangle$ and $\vdash_p \langle S\rangle \mathsf{po}_2 \langle T\rangle$. Furthermore, $\mathit{sketch}(\mathsf{po}_i) = \mathsf{sketch}_i$. With rule (**PSEQ**) we get $\vdash_p \langle R\rangle \mathsf{po}_1 ; \mathsf{po}_2 \langle T\rangle$. □

We show several properties of a valid proof outline: When we write $\vdash_p \mathsf{po}$ with $\mathsf{po} = \mathsf{po}_1 \mid \ldots \mid \mathsf{po}_n$ we mean $\vdash_p \mathsf{po}_1 \wedge \ldots \wedge \vdash_p \mathsf{po}_n$.

Lemma A.5. *In a sketch proof outline, all realizability triples hold, i.e. the following implications are true:*

$$
\begin{aligned}
\vdash_p \mathsf{po}_1 ; \mathsf{po}_2 &\implies \vdash_p \langle R\rangle \mathsf{po}_1 \langle S\rangle \wedge \vdash_p \langle S'\rangle \mathsf{po}_2 \langle T\rangle \wedge S = S' \\
\vdash_p \mathsf{po}_1 + \mathsf{po}_2 &\implies \vdash_p \langle R\rangle \mathsf{po}_1 \langle S\rangle \wedge \vdash_p \langle R'\rangle \mathsf{po}_2 \langle S'\rangle \wedge R = R' \wedge S = S'
\end{aligned}
$$

$$\vdash_p \mathsf{N}(\mathsf{po}) \implies \vdash_p \mathsf{po}$$
$$\vdash_p \langle R\rangle(\langle I\rangle \mathsf{po} \langle I'\rangle)^* \langle S\rangle \implies \vdash_p \langle I\rangle \mathsf{po} \langle I'\rangle \wedge I = I' \wedge R \preceq_a I \wedge I \preceq_a S\,.$$

Proof. We use structural induction to prove the implications. We proof one implication at a time.

For the first one, the only rules that need to be considered are (**PSEQ**), (**PCSQ**) and (**PGATHER**). The base case holds trivially. Rule (**PSEQ**) immediately yields the sought after result. Through transitivity of $\preceq_a$, (**PCSQ**) also yields the implication after applying the induction hypothesis. When rule (**PGATHER**) is applied, the proof has the following shape:

$$\vdash_p \langle R \cup R'\rangle \mathit{gather}(\mathsf{po}_1, \mathsf{po}_1')\langle S \cup S'\rangle ; \langle S \cup S'\rangle \mathit{gather}(\mathsf{po}_2, \mathsf{po}_2')\langle T \cup T'\rangle$$

We can use the gather rule to show $\vdash_p \mathit{gather}(\mathsf{po}_i, \mathsf{po}_i')$ because the individual proofs hold according to the induction hypothesis. That fact that the intermediary assertions match is a result of the induction hypothesis.

The other cases all follow the same pattern: The proofs for the rule corresponding to the program (here (**PSEQ**) for a program sequence) and rule (**PCSQ**) hold trivially. For rule (**PGATHER**), the rule itself has to be applied to the result of the induction hypothesis, as seen here. □

Proof of Theorem 4.1.

With the newly introduced notation, the theorem is the following:

Theorem A.6 (Soundness). $\vdash_p \mathsf{po}$ *and* $\mathsf{po} \vdash \mathsf{po}'$ *together imply* $\vdash_p \mathsf{po}'$.

We conduct the proof using structural induction over the proof tree in realization logic.

Base Case: We know that $\vdash_p \langle R\rangle \mathsf{com} \langle S\rangle$ and $\langle R\rangle \mathsf{com} \langle S\rangle \vdash \langle \{r\}\rangle \mathsf{com} \langle \{s\}\rangle$ hold. From the rule (**RCOM**) of the realization logic we know that $[\![\mathsf{com}]\!](r) \preceq_d s$ is true. This directly implies that $\vdash_p \langle \{r\}\rangle \mathsf{com} \langle \{s\}\rangle$ holds.

Induction Step: Consider the rule (**RSELECT**). The implication immediately holds because the nested proofs are sound per Lemma A.5.

Consider the rule (**RAC**). Again, since the nested proofs are sound, we get $\vdash_p \mathsf{po}_1$ and $\vdash_p \mathsf{po}$ and $\vdash_p \mathsf{po}_2$. Applying the induction hypothesis on the precondition of the rule yields the triple $\vdash_p \mathsf{po}'$ Applying rule (**PANG**) on each subproof, yields $\vdash_p N(\mathsf{po}_1)$ and $\vdash_p N(\mathsf{po}')$ and $\vdash_p N(\mathsf{po}_2)$. Applying rule (**PGATHER**) multiple times yields the sought after result: $\vdash_p N(\mathsf{po}_1 \mid \mathsf{po}' \mid \mathsf{po}_2)$.

Consider rule (**RSEQL**). Since the nested proofs are sound, we get $\vdash_p \langle R\rangle \mathsf{po}_1 \langle S\rangle$ and also $\vdash_p \mathsf{po}_2$. The induction hypothesis yields the realizability triple $\vdash_p \langle R^{\mathsf{w}}\rangle \mathsf{po}_1' \langle S\rangle$. Using rule (**PSEQ**), we get $\vdash_p \langle R^{\mathsf{w}}\rangle \mathsf{po}_1' \langle S\rangle ; \mathsf{po}_2$.

Consider rule (**RSEQR**). Again, by soundness of the subproofs we get $\vdash_p \langle R\rangle \mathsf{po}_1 \langle S\rangle$ and also $\vdash_p \langle S\rangle \mathsf{po}_2 \langle T\rangle$. Applying the induction hypothesis we get $\vdash_p \langle S^{\mathsf{w}}\rangle \mathsf{po}_2' \langle T^{\mathsf{w}}\rangle$. Since $S \preceq_a S^{\mathsf{w}}$, the infer rule can be applied to the first realizability triple resulting in $\vdash_p \langle R\rangle \mathsf{po}_1 \langle S^{\mathsf{w}}\rangle$ Then applying rule (**PSEQ**) yields: $\vdash_p \langle R\rangle \mathsf{po}_1 ; \mathsf{po}_2 \langle T^{\mathsf{w}}\rangle$.

Consider rule (**RDEM**). Again, through the soundness of the sub proofs (Lemma A.5), we get $\vdash_p \langle R\rangle \mathsf{po}_1 \langle S\rangle$ and $\vdash_p \langle R\rangle \mathsf{po}_2 \langle R_p\rangle$. The induction hypothesis yields $\vdash_p \langle R^{\mathsf{w}}\rangle \mathsf{po}_i' \langle S^{\mathsf{w}}\rangle$ with $R^{\mathsf{w}} = \{r\}$. Applying rule (**PDEM**) results in

$$\vdash_p \langle R^{\mathsf{w}}\rangle \mathsf{po}_1 + \mathsf{po}_2 \langle S^{\mathsf{w}}\rangle\,.$$

Consider rule (**RLOOP**). Through the soundness of the sub proofs (Lemma A.5), we get the realizability triple $\vdash_p \langle I\rangle \mathsf{po} \langle I\rangle$. Applying the induction hypothesis, we get the realizability triple $\vdash_p \langle I^{\mathsf{w}}\rangle \mathsf{po}' \langle I^{\mathsf{w}}\rangle$ with $I^{\mathsf{w}} = \{i\}$. Applying rule (**PLOOP**) yields the realizability triple

$$\vdash_p \langle I^{\mathsf{w}}\rangle \mathsf{po}'^* \langle I^{\mathsf{w}}\rangle\,.$$

Consider rule (**RTRANS**). The induction hypothesis yields soundness of $\vdash_p \mathsf{po}_2$. Applying the induction hypothesis again yields the sought after result: $\vdash_p \mathsf{po}_3$.

Consider rule (**RCSQ**). The induction hypothesis yields $\vdash_p \langle R^{ww}\rangle \mathsf{po}'\langle S^{w}\rangle$. Using rule (**PCSQ**), we get $\vdash_p \langle R^{w}\rangle \mathsf{po}'\langle S^{ww}\rangle$.

Lastly, consider rule (**RGATHER**). Applying the induction hypothesis yields $\vdash_p \langle R_1\rangle \mathsf{po}'_1\langle S_1\rangle$ and $\vdash_p \langle R_2\rangle \mathsf{po}'_2\langle S_2\rangle$ with $\mathit{sketch}(\mathsf{po}'_1) = \mathit{sketch}(\mathsf{po}'_2)$. Applying the rule (**PGATHER**), we get

$$\vdash_p \langle R_1 \cup R_2\rangle \mathit{gather}(\mathsf{po}'_1, \mathsf{po}'_2)\langle S_1 \cup S_2\rangle$$

□

Proof of Theorem 4.3. Again, we state theorem with the new notation:

Theorem A.7. $\vdash_p \mathsf{po}$ *and* $\vdash_p \mathsf{po}'$ *and* $\mathsf{po} \preceq_p \mathsf{po}'$ *together imply* $\mathsf{po} \vdash \mathsf{po}'$.

Proof. For this proof, we conduct a structural induction over the syntactic structure of the proof po. Then, some cases require their own additional structural induction over the proof tree of po' in the proof outline calculus. In the following, we also have $X \preceq_a X^{w}$ for any selections X and X^{w}.

Base Case: $\mathsf{po} = \mathsf{com}$. Then, $\mathsf{po}' = \mathsf{com}$. We show $\langle R\rangle \mathsf{po}\langle S\rangle \vdash \langle R^{w}\rangle \mathsf{po}'\langle S^{w}\rangle$ by induction over the proof tree of $\vdash_p \langle R^{w}\rangle \mathsf{po}'\langle S^{w}\rangle$. In the base case, we have the realizability triple $\vdash_p \langle r\rangle \mathsf{com}\langle s\rangle$. Thus, we know that $[\![\mathsf{com}]\!](r) \preceq_d s$ holds. Therefore, $\langle R\rangle \mathsf{com}\langle S\rangle \vdash \langle \{r\}\rangle \mathsf{com}\langle \{s\}\rangle$ holds. In the induction step, there are two cases. Either the rule (**PGATHER**) was used, or the rule (**PCSQ**) was used. If (**PGATHER**) was used, we have $\vdash_p \langle R_1 \cup R_2\rangle \mathsf{com}\langle S_1 \cup S_2\rangle$. From this, we know that $\vdash_p \langle R_i\rangle \mathsf{com}\langle S_i\rangle$ hold. Applying the induction hypothesis yields $\langle R\rangle \mathsf{com}\langle S\rangle \vdash \langle R_i\rangle \mathsf{com}\langle S_i\rangle$. Using the rule (**RGA**), we get $\langle R\rangle \mathsf{com}\langle S\rangle \vdash \langle \cup_i R_i\rangle \mathsf{com}\langle \cup_i R_i\rangle$. If (**PCSQ**) was used, we have $\vdash_p \langle R^{w}\rangle \mathsf{com}\langle S^{ww}\rangle$. From the precondition, we know $\vdash_p \langle R^{ww}\rangle \mathsf{com}\langle S^{w}\rangle$ holds. Using the induction hypothesis, we get $\langle R\rangle \mathsf{com}\langle S\rangle \vdash \langle R^{ww}\rangle \mathsf{com}\langle S^{w}\rangle$. Applying rule (**RCSQ**) yields $\langle R\rangle \mathsf{com}\langle S\rangle \vdash \langle R^{w}\rangle \mathsf{com}\langle S^{ww}\rangle$.

Induction Step:

If the proof is a sequence, i.e. $\langle R\rangle \mathsf{po}\langle T\rangle = \langle R\rangle \mathsf{po}_1\langle S\rangle ; \langle S\rangle \mathsf{po}_2\langle T\rangle$, then the other proof is $\langle R^{w}\rangle \mathsf{po}'\langle T^{w}\rangle = \langle R^{w}\rangle \mathsf{po}'_1\langle S^{w}\rangle ; \langle S^{w}\rangle \mathsf{po}'_2\langle T^{w}\rangle$.

Thus we know that, the realizability triples $\vdash_p \langle S\rangle \mathsf{po}_2\langle T\rangle$ and $\vdash_p \langle S^{w}\rangle \mathsf{po}'_2\langle T^{w}\rangle$ hold. Applying the induction hypothesis, we get $\mathsf{po}_2 \vdash \mathsf{po}'_2$. Applying the rule (**RSEQR**) yields

$$\langle R\rangle \mathsf{po}_1\langle S\rangle ; \langle S\rangle \mathsf{po}_2\langle T\rangle \vdash \langle R\rangle \mathsf{po}_1\langle S^{w}\rangle ; \langle S^{w}\rangle \mathsf{po}'_2\langle T^{w}\rangle .$$

Because $\vdash_p \langle R\rangle \mathsf{po}_1\langle S\rangle$ holds, the realizability triple $\vdash_p \langle R\rangle \mathsf{po}_1\langle S^{w}\rangle$ also holds due to the consequence rule. We also have that $\vdash_p \langle R^{w}\rangle \mathsf{po}'_1\langle S^{w}\rangle$ is true. Applying the induction hypothesis, we get $\langle R\rangle \mathsf{po}_1\langle S^{w}\rangle \vdash \langle R^{w}\rangle \mathsf{po}'_1\langle S^{w}\rangle$. Using rule (**RSEQL**) yields

$$\langle R\rangle \mathsf{po}'_1\langle S^{w}\rangle ; \langle S^{w}\rangle \mathsf{po}'_2\langle T^{w}\rangle \vdash \langle R^{w}\rangle \mathsf{po}'_1\langle S^{w}\rangle ; \langle S^{w}\rangle \mathsf{po}'_2\langle T^{w}\rangle .$$

Combined with the rewrite from before and rule (**RTRANS**), we have

$$\langle R\rangle \mathsf{po}_1\langle S\rangle ; \langle S\rangle \mathsf{po}_2\langle T\rangle \vdash \langle R^{w}\rangle \mathsf{po}'_1\langle S^{w}\rangle ; \langle S^{w}\rangle \mathsf{po}'_2\langle T^{w}\rangle .$$

In the next case, the proof is a demonic choice, i.e.

$$\langle R\rangle \mathsf{po}\langle S\rangle = \langle R\rangle \mathsf{po}_1\langle S\rangle + \langle R\rangle \mathsf{po}_2\langle S\rangle .$$

Then, the other proof is

$$\langle R^{w}\rangle \mathsf{po}'\langle S^{w}\rangle = \langle R^{w}\rangle \mathsf{po}'_1\langle S^{w}\rangle + \langle R^{w}\rangle \mathsf{po}'_2\langle S^{w}\rangle .$$

We proceed by induction over the proof tree of

$$\langle R^{w}\rangle \mathsf{po}'_1\langle S^{w}\rangle + \langle R^{w}\rangle \mathsf{po}'_2\langle S^{w}\rangle .$$

If the proof was done with the rule (**PDEM**), it has the following shape:

$$\langle R^{\mathrm{w}}\rangle \mathsf{po}_1'\langle S^{\mathrm{w}}\rangle + \langle R^{\mathrm{w}}\rangle \mathsf{po}_2'\langle S^{\mathrm{w}}\rangle$$

with $R^{\mathrm{w}} = \{r\}$. With the overall induction hypothesis and because the subproofs are sound, we know that

$$\langle R\rangle \mathsf{po}_1\langle S\rangle \vdash \langle R^{\mathrm{w}}\rangle \mathsf{po}_1'\langle S^{\mathrm{w}}\rangle \text{ and } \langle R\rangle \mathsf{po}_2\langle S\rangle \vdash \langle R^{\mathrm{w}}\rangle \mathsf{po}_2'\langle S^{\mathrm{w}}\rangle$$

hold. Using the rule (**RDEM**), we get

$$\langle R\rangle \mathsf{po}_1\langle S\rangle + \langle R\rangle \mathsf{po}_2\langle S\rangle \vdash \langle R^{\mathrm{w}}\rangle \mathsf{po}_1'\langle S^{\mathrm{w}}\rangle + \langle R^{\mathrm{w}}\rangle \mathsf{po}_2'\langle S^{\mathrm{w}}\rangle \ .$$

Next, we, again, have two possibilities. The proof was conducted using (**PGATHER**) or (**PCSQ**). Starting with rule (**PGATHER**), the proof has the following shape:

$$\vdash_p \langle R_1^{\mathrm{w}} \cup R_2^{\mathrm{w}}\rangle \mathsf{po}_1'\langle S_1^{\mathrm{w}} \cup S_2^{\mathrm{w}}\rangle + \langle R_1^{\mathrm{w}} \cup R_2^{\mathrm{w}}\rangle \mathsf{po}_2'\langle S_1^{\mathrm{w}} \cup S_2^{\mathrm{w}}\rangle$$

From preconditions of the gather rule we get

$$\vdash_p \langle R_1^{\mathrm{w}}\rangle \mathsf{po}_1'\langle S_1^{\mathrm{w}}\rangle + \langle R_1^{\mathrm{w}}\rangle \mathsf{po}_2'\langle S_1^{\mathrm{w}}\rangle$$

$$\vdash_p \langle R_2^{\mathrm{w}}\rangle \mathsf{po}_1''\langle S_2^{\mathrm{w}}\rangle + \langle R_2^{\mathrm{w}}\rangle \mathsf{po}_2''\langle S_2^{\mathrm{w}}\rangle$$

and Applying the induction hypothesis then yields

$$\langle R\rangle \mathsf{po}_1\langle S\rangle + \langle R\rangle \mathsf{po}_2\langle S\rangle \vdash \langle R_i^{\mathrm{w}}\rangle \mathsf{po}_1'\langle S_i^{\mathrm{w}}\rangle + \langle R_i^{\mathrm{w}}\rangle \mathsf{po}_2'\langle S_i^{\mathrm{w}}\rangle \ .$$

Similarly for po_1'' and po_2''. Using the rule (**RGATHER**) we get

$$\langle R\rangle \mathsf{po}_1\langle S\rangle + \langle R\rangle \mathsf{po}_2\langle S\rangle \vdash \langle \cup_i R_i^{\mathrm{w}}\rangle gather(\mathsf{po}_1', \mathsf{po}_1'')\langle \cup_i S_i^{\mathrm{w}}\rangle + \langle \cup_i R_i^{\mathrm{w}}\rangle gather(\mathsf{po}_1', \mathsf{po}_1'')\langle \cup_i S_i^{\mathrm{w}}\rangle \ .$$

Closing with the rule (**PCSQ**), the proof has the following shape:

$$\langle R^{\mathrm{w}}\rangle \mathsf{po}_1'\langle S^{\mathrm{ww}}\rangle + \langle R^{\mathrm{w}}\rangle \mathsf{po}_2'\langle S^{\mathrm{ww}}\rangle \ .$$

From the precondition of the infer rule, we have

$$\langle R^{\mathrm{ww}}\rangle \mathsf{po}_1'\langle S^{\mathrm{w}}\rangle + \langle R^{\mathrm{ww}}\rangle \mathsf{po}_2'\langle S^{\mathrm{w}}\rangle \ .$$

Using the induction hypothesis, we get the following rewrite:

$$\langle R\rangle \mathsf{po}_1\langle S\rangle + \langle R\rangle \mathsf{po}_2\langle S\rangle \vdash \langle R^{\mathrm{ww}}\rangle \mathsf{po}_1'\langle S^{\mathrm{w}}\rangle + \langle R^{\mathrm{ww}}\rangle \mathsf{po}_2'\langle S^{\mathrm{w}}\rangle \ .$$

Now, the rule (**RCSQ**) can be applied to get the sought after result:

$$\langle R\rangle \mathsf{po}_1\langle S\rangle + \langle R\rangle \mathsf{po}_2\langle S\rangle \vdash \langle R^{\mathrm{w}}\rangle \mathsf{po}_1'\langle S^{\mathrm{ww}}\rangle + \langle R^{\mathrm{w}}\rangle \mathsf{po}_2'\langle S^{\mathrm{ww}}\rangle \ .$$

This concludes the proof for demonic choices.

In the next case, the proof is a loop. This case is analogous to demonic choices.

Finally, consider the proof is over a nonterminal. So, the first proof is

$$\vdash_p \langle R\rangle \mathsf{N}(\langle R_1\rangle \mathsf{po}_1\langle S_1\rangle \mid \ldots \mid \langle R_n\rangle \mathsf{po}_n\langle S_n\rangle \mid \mathsf{po}_r)\langle S\rangle \ .$$

Then, the other proof can either also be a weaker proof over a nonterminal or it is only one production of the nonterminal. We start with the first case. The second proof is then

$$\vdash_p \langle R^{\mathrm{w}}\rangle \mathsf{N}(\langle R_1^{\mathrm{w}}\rangle \mathsf{po}_1'\langle S_1^{\mathrm{w}}\rangle \mid \ldots \mid \langle R_n^{\mathrm{w}}\rangle \mathsf{po}_n'\langle S_n^{\mathrm{w}}\rangle)\langle S^{\mathrm{w}}\rangle \ .$$

We know that all the subproofs hold. Applying the induction hypothesis then yields the following rewrites for all i:

$$\langle R_i\rangle \mathsf{po}_i\langle S_i\rangle \vdash \langle R_i^{\mathrm{w}}\rangle \mathsf{po}_i'\langle S_i^{\mathrm{w}}\rangle \ .$$

We apply the rule (**RANG**) on the rewritten part several times. Together with rule (**TRANS**), we combine the rewrites and get:

$$\langle R\rangle \mathsf{N}(\langle R_1\rangle \mathsf{po}_1\langle S_1\rangle \mid \ldots \mid \langle R_n\rangle \mathsf{po}_n\langle S_n\rangle \mid \mathsf{po}_r)\langle S\rangle \vdash \langle \cup_i R_i^{\mathsf{w}}\rangle \mathsf{N}(\langle R_1^{\mathsf{w}}\rangle \mathsf{po}_1'\langle S_1^{\mathsf{w}}\rangle \mid \ldots \mid \langle R_n^{\mathsf{w}}\rangle \mathsf{po}_n'\langle S_n^{\mathsf{w}}\rangle)\langle \cup_i S_i^{\mathsf{w}}\rangle \,.$$

Moving on to the case where one production is chosen. Without loss of generality, let this be the first production. The proof therefore is $\vdash_p \langle R_1^{\mathsf{w}}\rangle \mathsf{po}_1'\langle S_1^{\mathsf{w}}\rangle$. Since we know that $\vdash_p \langle R_1\rangle \mathsf{po}_1\langle S_1\rangle$ holds, we can apply the induction hypothesis and get the rewrite $\langle R_1\rangle \mathsf{po}_1\langle S_1\rangle \vdash \langle R^{\mathsf{w}}\rangle \mathsf{po}_1'\langle S_1^{\mathsf{w}}\rangle$. Then, we apply rule (**RAC**) and get the following:

$$\langle R\rangle \mathsf{N}(\langle R_1\rangle \mathsf{po}_1\langle S_1\rangle \mid \ldots \mid \langle R_n\rangle \mathsf{po}_n\langle S_n\rangle)\langle S\rangle \vdash \langle R_1^{\mathsf{w}} \cup (\cup_i R_i)\rangle \mathsf{N}(\langle R_1^{\mathsf{w}}\rangle \mathsf{po}_1'\langle S_1^{\mathsf{w}}\rangle \mid \ldots \mid \langle R_n\rangle \mathsf{po}_n\langle S_n\rangle)\langle S_1^{\mathsf{w}} \cup (\cup_i S_i)\rangle \,.$$

Furthermore, the rule (**RSELECT**) can be applied to get

$$\langle R_1^{\mathsf{w}} \cup (\cup_i R_i)\rangle \mathsf{N}(\langle R_1^{\mathsf{w}}\rangle \mathsf{po}_1'\langle S_1^{\mathsf{w}}\rangle \mid \ldots \mid \langle R_n\rangle \mathsf{po}_n\langle S_n\rangle)\langle S_1^{\mathsf{w}} \cup (\cup_i S_i)\rangle \vdash \langle R_1^{\mathsf{w}}\rangle \mathsf{po}_1'\langle S_1^{\mathsf{w}}\rangle \,.$$

Lastly, applying the rule (**RTRANS**) yields the sought after result:

$$\langle R\rangle \mathsf{N}(\langle R_1\rangle \mathsf{po}_1\langle S_1\rangle \mid \ldots \mid \langle R_n\rangle \mathsf{po}_n\langle S_n\rangle)\langle S\rangle \vdash \langle R_1^{\mathsf{w}}\rangle \mathsf{po}_1'\langle S_1^{\mathsf{w}}\rangle \,.$$

This concludes the proof. □

We proof the following lemma:

Lemma A.8. *The following implication holds:*

$$\vdash_p \langle R\rangle \mathsf{po}_1\langle S\rangle + \langle R\rangle \mathsf{po}_2\langle S\rangle \wedge S \preceq_a \{s\} \implies \exists\, r, \mathsf{po}_1', \mathsf{po}_2' : r \in R \wedge \vdash_p \langle\{r\}\rangle \mathsf{po}_1'\langle\{s\}\rangle \wedge \vdash_p \langle\{r\}\rangle \mathsf{po}_2'\langle\{s\}\rangle \wedge \mathsf{po}_1 \preceq_p \mathsf{po}_1' \wedge \mathsf{po}_2 \preceq_p \mathsf{po}_2' \wedge sketch(\mathsf{po}_1) = sketch(\mathsf{po}_1') \wedge sketch(\mathsf{po}_2) = sketch(\mathsf{po}_2') \,.$$

Proof. We conduct the proof by induction over the proof tree of

$$\vdash_p \langle R\rangle \mathsf{po}_1\langle S\rangle + \langle R\rangle \mathsf{po}_2\langle S\rangle$$

The base case holds trivially.

In the first case of the induction step, the proof was done using the rule (**PDEM**). Then, we have $\vdash_p \langle R\rangle \mathsf{po}_1\langle S\rangle$ and also $\vdash_p \langle R\rangle \mathsf{po}_2\langle S\rangle$ with $R = \{r\}$. Thus, we know $\vdash_p \langle\{r\}\rangle \mathsf{po}_i\langle S\rangle$. Using the rule (**PCSQ**) on both proofs, we get the realizability triples $\vdash_p \langle\{r\}\rangle \mathsf{po}_i\langle\{s\}\rangle$.

We have two remaining cases, either the proof was finished using the rule (**PGATHER**) or rule (**PCSQ**). In the first case, we have

$$\vdash_p \langle \cup_i R_i\rangle \mathsf{po}_1\langle \cup_i S_i\rangle + \langle \cup_i R_i\rangle \mathsf{po}_2\langle \cup_i S_i\rangle$$

Since $\cup_i S_i \preceq_a \{s\}$, one assertion S_i is stronger than $\{s\}$. Without loss of generality, let this be S_1. From the above realizability triple, we know that the realizability triple

$$\vdash_p \langle R_1\rangle \mathsf{po}_1'\langle S_1\rangle + \langle R_1\rangle \mathsf{po}_2'\langle S_1\rangle$$

holds. Applying the induction hypothesis, we get an r and the proofs po_1'' and po_2'' for which the realizability triples $\vdash_p \langle\{r\}\rangle \mathsf{po}_1''\langle\{s\}\rangle$ and $\vdash_p \langle\{r\}\rangle \mathsf{po}_2''\langle\{s\}\rangle$ hold.

In the second case, we have

$$\vdash_p \langle R\rangle \mathsf{po}_1\langle S^{\mathsf{w}}\rangle + \langle R\rangle \mathsf{po}_2\langle S^{\mathsf{w}}\rangle$$

The precondition of the rule (**PCSQ**) requires

$$\vdash_p \langle R^{\mathsf{w}}\rangle \mathsf{po}_1\langle S\rangle + \langle R^{\mathsf{w}}\rangle \mathsf{po}_2\langle S\rangle$$

Applying the induction hypothesis yields an $r \in R^{\mathrm{w}}$ and $\vdash_p \langle\{r\}\rangle \mathsf{po}_i' \langle\{s\}\rangle$. Since $R \preceq_a R^{\mathrm{w}}$ there is an r' of R with $r' \preceq_d r$. Thus we can strengthen the precondition of the realizability triples to get $\vdash_p \langle\{r'\}\rangle \mathsf{po}_i' \langle\{s\}\rangle$. □

We proof the following lemma:

Lemma A.9. *The following implication holds:*

$$\vdash_p \langle R\rangle(\langle I\rangle \mathsf{po} \langle I\rangle)^* \langle S\rangle \wedge i \in I \implies \exists\, \mathsf{po}' \colon \vdash_p \langle\{i\}\rangle \mathsf{po}' \langle\{i\}\rangle \wedge \mathsf{po} \preceq_p \mathsf{po}' \wedge \mathit{sketch}(\mathsf{po}) = \mathit{sketch}(\mathsf{po}') \,.$$

Proof. The proof is analogous to one of the previous lemma. □

Proof of Theorem 4.4. The proof is done by an induction over the structure of the `prf`. Again, we first state the theorem using the new notation:

Theorem A.10 (Backtracking Freedom). *Let* $\vdash_p \langle R\rangle \mathsf{po} \langle S\rangle$ *and* $s \in S$. *Then there are* r *and* po' *so that* $\langle R\rangle \mathsf{po} \langle S\rangle \vdash \langle\{r\}\rangle \mathsf{po}' \langle\{s\}\rangle$, $r \in R$, *and* $\mathit{sketch}(\mathsf{po}') \in \mathit{drv}(\mathit{sketch}(\mathsf{po}))$.

Proof. **Base Case:** The proof is $\vdash_p \langle R\rangle \mathtt{com} \langle S\rangle$. Applying soundness yields a predicate r with $[\![\mathtt{com}]\!](r) \preceq_d s$ and $r \in R$. Therefore, we can apply rule (**RCOM**) and get $\langle R\rangle \mathtt{com} \langle S\rangle \vdash \langle\{r\}\rangle \mathtt{com} \langle\{s\}\rangle$.

Induction Step: If the proof is a sequence, we have

$$\langle R\rangle \mathsf{po}_1 \langle S\rangle \,;\, \langle S\rangle \mathsf{po}_2 \langle T\rangle \,.$$

Then, we know that the subproofs are true so po_1 and po_2 hold. Applying the induction hypothesis on the second realizability triple from above, we get an s with $s \in S$ and a proof po_2' for which the following holds:

$$\langle S\rangle \mathsf{po}_2 \langle T\rangle \vdash \langle\{s\}\rangle \mathsf{po}_2' \langle\{t\}\rangle \,.$$

Next, we apply the induction hypothesis on the other sequence and get a predicate r with $r \in R$ and a proof po_1' for which the following holds:

$$\langle R\rangle \mathsf{po}_1 \langle S\rangle \vdash \langle\{r\}\rangle \mathsf{po}_1' \langle\{s\}\rangle \,.$$

Applying rule (**RSEQR**), we get the following rewrite:

$$\langle R\rangle \mathsf{po}_1 \langle S\rangle \,;\, \langle S\rangle \mathsf{po}_2 \langle T\rangle \vdash \langle R\rangle \mathsf{po}_1 \langle\{s\}\rangle \,;\, \langle\{s\}\rangle \mathsf{po}_2' \langle\{t\}\rangle \,.$$

Then, applying rule (**RSEQL**), we get the following rewrite:

$$\langle R\rangle \mathsf{po}_1 \langle\{s\}\rangle \,;\, \langle\{s\}\rangle \mathsf{po}_2' \langle\{t\}\rangle \vdash \langle\{r\}\rangle \mathsf{po}_1' \langle\{s\}\rangle \,;\, \langle\{s\}\rangle \mathsf{po}_2' \langle\{t\}\rangle \,.$$

Finally, with rule (**RTRANS**), we get the sought after result:

$$\langle R\rangle \mathsf{po}_1 \langle S\rangle \,;\, \langle S\rangle \mathsf{po}_2 \langle T\rangle \vdash \langle\{r\}\rangle \mathsf{po}_1' \langle\{s\}\rangle \,;\, \langle\{s\}\rangle \mathsf{po}_2' \langle\{t\}\rangle \,.$$

If the proof is a demonic choice, so we have

$$\langle R\rangle \mathsf{po}_1 \langle S\rangle + \langle R\rangle \mathsf{po}_2 \langle S\rangle \,,$$

and a predicate s with $s \in S$. Then we can apply Lemma A.8 to get a predicate r and the proofs $\vdash_p \langle\{r\}\rangle \mathsf{po}_1' \langle\{s\}\rangle$ and $\vdash_p \langle\{r\}\rangle \mathsf{po}_2' \langle\{s\}\rangle$ with $\mathsf{po}_i \preceq_p \mathsf{po}_i'$. . Using Theorem 4.3, the following two rewrites are sound:

$$\langle R\rangle \mathsf{po}_1 \langle S\rangle \vdash \langle\{r\}\rangle \mathsf{po}_1' \langle\{s\}\rangle \text{ and } \langle R\rangle \mathsf{po}_2 \langle S\rangle \vdash \langle\{r\}\rangle \mathsf{po}_2' \langle\{s\}\rangle \,.$$

An application of the rule (**RDEM**) then yields the sought after result:

$$\langle R\rangle \mathsf{po}_1 \langle S\rangle + \langle R\rangle \mathsf{po}_2 \langle S\rangle \vdash \langle\{r\}\rangle \mathsf{po}_1' \langle\{s\}\rangle + \langle\{r\}\rangle \mathsf{po}_2' \langle\{s\}\rangle \,.$$

Moving on with loops. We have the proof $\vdash_p \langle R\rangle(\langle I\rangle \mathsf{po} \langle I\rangle)^* \langle S\rangle$ and a predicate s with $s \in S$. Since $I \preceq_a S$, there also is an i of I with $i \preceq_d s$. Applying Lemma A.9, we get a proof po' for which $\vdash_p \langle\{i\}\rangle \mathsf{po}' \langle\{i\}\rangle$ holds and $\mathsf{po} \preceq_p \mathsf{po}'$. Theorem 4.3 then yields

$$\langle I\rangle \mathsf{po} \langle I\rangle \vdash \langle\{i\}\rangle \mathsf{po}' \langle\{i\}\rangle \,.$$

Applying rule (**RLOOP**) yields

$$\langle R\rangle(\langle I\rangle \mathsf{po} \langle I\rangle)^* \langle S\rangle \vdash \langle\{i\}\rangle(\langle\{i\}\rangle \mathsf{po}' \langle\{i\}\rangle)^* \langle\{i\}\rangle \,.$$

Since $R \preceq_a I$, we know there is an r in R for which $r \preceq_d^{\#} i$ holds. Applying rule (**RCSQ**), we get the sought after result:

$$\langle R\rangle(\langle I\rangle \mathsf{po} \langle I\rangle)^* \langle S\rangle \vdash \langle\{r\}\rangle(\langle\{i\}\rangle \mathsf{po}' \langle\{i\}\rangle)^* \langle\{s\}\rangle \,.$$

In the last case, we consider proofs over nonterminals. We have a predicate s and the proof

$$\vdash_p \langle R\rangle \mathsf{N}(\langle R_1\rangle \mathsf{po}_1 \langle S_1\rangle \mid \ldots \mid \langle R_n\rangle \mathsf{po}_n \langle S_n\rangle) \langle S\rangle$$

with $s \in S$. Since $(S_1 \cup \ldots \cup S_n) = S$, there is a selection S_i with $s \in S_i$. Without loss of generality, let this i be 1. We know the realizability triple $\vdash_p \langle R_1\rangle \mathsf{po}_1 \langle S_1\rangle$ holds. Using the induction hypothesis, we get the following for an r_1 of R_1:

$$\langle R_1\rangle \mathsf{po}_1 \langle S_1\rangle \vdash \langle\{r_1\}\rangle \mathsf{po}_1' \langle\{s\}\rangle \,.$$

Using the rule (**RANG**), we get the following rewrite:

$$\mathsf{N}(\langle R_1\rangle \mathsf{po}_1 \langle S_1\rangle \mid \ldots \mid \langle R_n\rangle \mathsf{po}_n \langle S_n\rangle) \vdash \mathsf{N}(\langle\{r_1\}\rangle \mathsf{po}_1' \langle\{s\}\rangle \mid \ldots \mid \langle R_n\rangle \mathsf{po}_n \langle S_n\rangle) \,.$$

Then, with rule (**RSELECT**), we get

$$\mathsf{N}(\langle\{r_1\}\rangle \mathsf{po}_1' \langle\{s\}\rangle \mid \ldots \mid \langle R_n\rangle \mathsf{po}_n \langle S_n\rangle) \vdash \langle\{r_1\}\rangle \mathsf{po}_1' \langle\{s\}\rangle \,.$$

Finally, applying rule (**RTRANS**) twice, we get

$$\langle R\rangle \mathsf{N}(\langle R_1\rangle \mathsf{po}_1 \langle S_1\rangle \mid \ldots \mid \langle R_n\rangle \mathsf{po}_n \langle S_n\rangle) \langle S\rangle \vdash \langle\{r_1\}\rangle \mathsf{po}_1' \langle\{s\}\rangle \,.$$

Because $R_1 \subseteq R$, we have $r_1 \in R$. This concludes the proof. □

A.3 Proofs for Section 5

We show the following lemma:

Lemma A.11. *The function sp is monotonic (if the specifications of the nonterminals are monotonic):*

$$R \preceq_a R' \implies sp(R, \mathtt{sketch}) \preceq_a sp(R', \mathtt{sketch}) \,.$$

Proof. We proof this lemma by induction over the structure of sketch.

Base Case: We show $R \preceq_a R'$ implies $sp(R, \mathtt{com}) \preceq_a sp(R', \mathtt{com})$. Let the predicate s be of $sp(R', \mathtt{com})$. Then, $s = [\![\mathtt{com}]\!](r')$ for some predicate r' of R'. Since we know $R \preceq_a R'$, there is a predicate r in R with $r \preceq_d r'$. Due to the interpretation of commands being monotonic, we get that $[\![\mathtt{com}]\!](r) \preceq_d [\![\mathtt{com}]\!](r') = s$. Because $[\![\mathtt{com}]\!](r)$ is in $sp(R, \mathtt{com})$, $sp(R, \mathtt{com}) \preceq_a sp(R', \mathtt{com})$ follows.

Induction Step: First, we consider the case where the sketch is a sequence. We show $R \preceq_a R'$ implies $sp(R, \mathtt{sketch}_1; \mathtt{sketch}_2) \preceq_a sp(R', \mathtt{sketch}_1; \mathtt{sketch}_2)$. From the induction hypothesis, we know that $sp(R, \mathtt{sketch}_1) \preceq_a sp(R', \mathtt{sketch}_1)$ holds. Applying the induction hypothesis again yields

$$sp(sp(R, \mathtt{sketch}_1), \mathtt{sketch}_2) \preceq_a sp(sp(R', \mathtt{sketch}_1), \mathtt{sketch}_2) \,.$$

Inserting the definition of the strongest post function, we get

$$sp(R, \mathtt{sketch}_1; \mathtt{sketch}_2) \preceq_a sp(R', \mathtt{sketch}_1; \mathtt{sketch}_2) \,.$$

Next, we consider the case where the sketch is a choice. We show

$$R \preceq_a R' \implies sp(R, \texttt{sketch}_1 + \texttt{sketch}_2) \preceq_a sp(R', \texttt{sketch}_1 + \texttt{sketch}_2) \ .$$

Let the predicate s_j be of $sp(R', \texttt{sketch}_1 + \texttt{sketch}_2)$. Thus, there is an r' of R' with $s'_j = s'_1 \sqcup s'_2$ and s'_1 is of $sp(\{r'\}, \texttt{sketch}_1)$ and s'_2 is of $sp(\{r'\}, \texttt{sketch}_2)$. Since $R \preceq_a R'$, there is an r of R with $\{r\} \preceq_a \{r'\}$. Applying the induction hypothesis yields

$$sp(\{r\}, \texttt{sketch}_1) \preceq_a sp(\{r'\}, \texttt{sketch}_1) \text{ and } sp(\{r\}, \texttt{sketch}_2) \preceq_a sp(\{r'\}, \texttt{sketch}_2)$$

Therefore, there is an $s_1 \in sp(\{r\}, \texttt{sketch}_1)$ and an $s_2 \in sp(\{r\}, \texttt{sketch}_2)$ with $\{s_1\} \preceq_a \{s'_1\}$ and $\{s_2\} \preceq_a \{s'_2\}$. Therefore, $s_j = s_1 \sqcup s_2 \preceq_d s'_1 \sqcup s'_2 = s'_j$. Since s_j is of $sp(R, \texttt{sketch}_1 + \texttt{sketch}_2)$, we have

$$sp(R, \texttt{sketch}_1 + \texttt{sketch}_2) \preceq_a sp(R', \texttt{sketch}_1 + \texttt{sketch}_2) \ .$$

Next, consider the sketch is a loop. We show

$$R \preceq_a R' \implies sp(R, \texttt{sketch}^*[I]) \preceq_a sp(R', \texttt{sketch}^*[I]) \ .$$

Let i be of $sp(R', \texttt{sketch}^*[I])$. Thus, $R' \preceq_a \{i\}$. By transitivity we have $R \preceq_a \{i\}$ and thus i is also in $sp(R, \texttt{sketch}^*[I])$. The inequality $i \preceq_d i$ holds trivially.

Lastly, consider the sketch is a nonterminal. Monotonicity follows directly from the required monotonicity of the specification.

This concludes the proof. □

We show the following lemma:

Lemma A.12 (Soundness). *For any R of Selections and any sketch* sketch *of* Sketches*, the following realizability triple holds (if the specifications of the nonterminals and loops are sound):*

$$\vdash_a \langle R \rangle \texttt{sketch} \langle sp(R, \texttt{sketch}) \rangle \ .$$

Proof. We proof this lemma by induction over the structure of the sketch.

Base Case: We show $\vdash_a \langle R \rangle \texttt{com} \langle sp(R, \texttt{com}) \rangle$. For every $r \in R$ we can can show the realizability triple $\vdash_a \langle \{r\} \rangle \texttt{com} \langle \{[\![\texttt{com}]\!](r)\} \rangle$. Using the rule (**GATHER**) we can show $\vdash_a \langle R \rangle \texttt{com} \langle \bigcup_{r \in R} \{[\![\texttt{com}]\!](r)\} \rangle$. By definition, this is the same as $\vdash_a \langle R \rangle \texttt{com} \langle sp(R, \texttt{com}) \rangle$.

Induction Step: First, consider a sequence of sketch. We show the realizability triple

$$\vdash_a \langle R \rangle \texttt{sketch}_1 \,;\, \texttt{sketch}_2 \langle sp(R, \texttt{sketch}_1 \,;\, \texttt{sketch}_2) \rangle \ .$$

Applying the induction hypothesis twice yields the realizability triples:

$$\vdash_a \langle R \rangle \texttt{sketch}_1 \langle sp(R, \texttt{sketch}_1) \rangle \text{ and } \vdash_a \langle sp(R, \texttt{sketch}_1) \rangle \texttt{sketch}_2 \langle sp(sp(R, \texttt{sketch}_1), \texttt{sketch}_2) \rangle \ .$$

Using rule (**SEQ**), we get the realizability triple

$$\vdash_a \langle R \rangle \texttt{sketch}_1 \,;\, \texttt{sketch}_2 \langle sp(sp(R, \texttt{sketch}_1), \texttt{sketch}_2) \rangle \ .$$

This is the same as the realizability triple $\vdash_a \langle R \rangle \texttt{sketch}_1 \,;\, \texttt{sketch}_2 \langle sp(R, \texttt{sketch}_1 \,;\, \texttt{sketch}_2) \rangle$.

Next, consider the sketch is a choice. We show

$$\vdash_a \langle R \rangle \texttt{sketch}_1 + \texttt{sketch}_2 \langle sp(R, \texttt{sketch}_1 + \texttt{sketch}_2) \rangle \ .$$

From the definition of the strongest post, we know that it is the following:

$$sp(R, \texttt{sketch}_1 + \texttt{sketch}_2) = \bigcup_{r \in R} \{s_1 \sqcup s_2 \mid s_1 \in sp(\{r\}, \texttt{sketch}_1) \wedge s_2 \in sp(\{r\}, \texttt{sketch}_2)\} \ .$$

For every r of R, we know from the induction hypothesis that $\vdash_a \langle \{r\} \rangle \texttt{sketch}_1 \langle sp(\{r\}, \texttt{sketch}_1) \rangle$ and $\vdash_a \langle \{r\} \rangle \texttt{sketch}_2 \langle sp(\{r\}, \texttt{sketch}_2) \rangle$ hold. Using the rule (**CSQ**), we can show the realizability triples $\vdash_a \langle \{r\} \rangle \texttt{sketch}_1 \langle \{s_{k1}\} \rangle$ for any s_{k1} of $sp(\{r\}, \texttt{sketch}_1)$ and $\vdash_a \langle \{r\} \rangle \texttt{sketch}_2 \langle \{s_{k2}\} \rangle$ for any

s_{k2} of $sp(\{r\}, \mathtt{sketch}_2)$. Let s_j be the join of any two s_{k1} and s_{k2}. Using the rule (**CSQ**), we can show $\vdash_a \langle\{r\}\rangle \mathtt{sketch}_i \langle\{s_j\}\rangle$. With the rule (**DEM**), we get $\vdash_a \langle\{r\}\rangle \mathtt{sketch}_1 + \mathtt{sketch}_2 \langle\{s_j\}\rangle$. Applying rule (**GATHER**) yields

$$\vdash_a \langle\{r\}\rangle \mathtt{sketch}_1 + \mathtt{sketch}_2 \langle\{s_1 \sqcup s_2 \mid s_i \in sp(\{r\}, \mathtt{sketch}_i)\}\rangle \,.$$

Applying rule (**GATHER**) once more yields

$$\vdash_a \langle R\rangle \mathtt{sketch}_1 + \mathtt{sketch}_2 \langle \bigcup_{r\in R} \{s_1 \sqcup s_2 \mid s_i \in sp(\{r\}, \mathtt{sketch}_i)\}\rangle \,.$$

This is the same as $\vdash_a \langle R\rangle \mathtt{sketch}_1 + \mathtt{sketch}_2 \langle sp(R, \mathtt{sketch}_1 + \mathtt{sketch}_2)\rangle$.

Next, consider the sketch is a loop. Let i be of $sp(R, \mathtt{sketch}^*)$. From soundness of the invariant annotation, we get $\vdash_a \langle\{i\}\rangle \mathtt{sketch} \langle\{i\}\rangle$. With rule loop, we get $\vdash_a \langle\{i\}\rangle \mathtt{sketch}^* \langle\{i\}\rangle$. With gather, we get

$$\vdash_a \langle sp(R, \mathtt{sketch}^*)\rangle \mathtt{sketch}^* \langle sp(R, \mathtt{sketch}^*)\rangle \,.$$

With (**CSQ**), we get $\vdash_a \langle R\rangle \mathtt{sketch}^* \langle sp(R, \mathtt{sketch}^*)\rangle$.

Finally, consider the sketch is a nonterminal. Soundness follows directly from the required soundness of the specification of the nonterminal.

This concludes the proof. □

We show the following corollary:

Corollary 1. *For any R, S of Selections and any sketch* sketch *of* Sketches *with sound annotations, the following implication holds:*

$$sp(R, \mathtt{sketch}) \preceq_a S \implies \vdash_a \langle R\rangle \mathtt{sketch} \langle S\rangle \,.$$

Proof. Lemma A.12 yields $\vdash_a \langle R\rangle \mathtt{sketch} \langle sp(R, \mathtt{sketch})\rangle$. Applying rule (**CSQ**) gives us the sought after result $\vdash_a \langle R\rangle \mathtt{sketch} \langle S\rangle$ □

Proof of Theorem 5.2.

We have already shown soundness above. We prove completeness by an induction over the proof tree of $\vdash_a \langle R\rangle \mathtt{sketch} \langle S\rangle$.

Base Case: We show $\vdash_a \langle\{r\}\rangle \mathtt{com} \langle\{s\}\rangle \implies sp(\{r\}, \mathtt{com}) \preceq_a \{s\}$. From the precondition of rule (**COM**), we know $[\![\mathtt{com}]\!](r) \preceq_d s$. Therefore $sp(\{r\}, \mathtt{com}) = \{[\![\mathtt{com}]\!](r)\} \preceq_a \{s\}$.

Induction Step: Consider the rule (**SEQ**). We show that the validity of $\vdash_a \langle R\rangle \mathtt{sketch}_1; \mathtt{sketch}_2 \langle T\rangle$ implies $sp(R, \mathtt{sketch}_1; \mathtt{sketch}_2) \preceq_a T$. From the preconditions we know that both $\vdash_a \langle R\rangle \mathtt{sketch}_1 \langle S\rangle$ and $\vdash_a \langle S\rangle \mathtt{sketch}_2 \langle T\rangle$ hold. Applying the induction hypothesis yields $sp(R, \mathtt{sketch}_1) \preceq_a S$ and $sp(S, \mathtt{sketch}_2) \preceq_a T$. Because the strongest post function is monotonic we get the sought after inequality $sp(R, \mathtt{sketch}_1; \mathtt{sketch}_2) \preceq_a T$.

Consider the rule (**DEM**). We show

$$\vdash_a \langle\{r\}\rangle \mathtt{sketch}_1 + \mathtt{sketch}_2 \langle S\rangle \implies sp(\{r\}, \mathtt{sketch}_1 + \mathtt{sketch}_2) \preceq_a S \,.$$

From the preconditions we know that $\vdash_a \langle\{r\}\rangle \mathtt{sketch}_1 \langle S\rangle$ and $\vdash_a \langle\{r\}\rangle \mathtt{sketch}_2 \langle S\rangle$ hold. Applying the induction hypothesis yields the inequalities $sp(\{r\}, \mathtt{sketch}_1) \preceq_a S$ and $sp(\{r\}, \mathtt{sketch}_2) \preceq_a S$. Let the predicate s be an element of S. Because the strongest posts are more versatile than S, there is an s_i of each strongest post with $s_i \preceq_d s$. Therefore, s is an upper bound of s_1 and s_2 and thus $s_1 \sqcup s_2 \preceq_d s$ holds. To conclude this case, see that $s_1 \sqcup s_2 \in sp(\{r\}, \mathtt{sketch}_1 + \mathtt{sketch}_2)$, and therefore $sp(\{r\}, \mathtt{sketch}_1 + \mathtt{sketch}_2) \preceq_a S$ holds.

Consider the rule (**ANG**). Completeness follows directly from the requirement that the specification of the nonterminals is complete.

Consider the rule (**CSQ**). We show

$$\vdash_a \langle R\rangle \mathtt{sketch} \langle S'\rangle \implies sp(R, \mathtt{sketch}) \preceq_a S' \ .$$

From the preconditions, we know that the realizability triple $\vdash_a \langle R'\rangle \mathtt{sketch}\langle S\rangle$ holds with $R \preceq_a R'$ and $S \preceq_a S'$. From the induction hypothesis we know that $sp(R', \mathtt{sketch}) \preceq_a S$. The sought after result immediately follows from monotonicity and because the relation $\preceq_a$ is transitive: $sp(R, \mathtt{sketch}) \preceq_a S'$.

Consider the rule (**LOOP**). We show $\vdash_a \langle\{i\}\rangle \mathtt{sketch}^*[I]\langle\{i\}\rangle$ implies $sp(\{i\}, \mathtt{sketch}^*[I]) \preceq_a \{i\}$. Due to the precondition of rule (**LOOP**) and completeness of the annotation, we have $i \in I$. Thus, $i \in sp(\{i\}, \mathtt{sketch}^*[I])$ and therefore the inequality trivially holds.

Lastly, consider the rule (**GATHER**). We show

$$\vdash_a \langle \cup_i R_i\rangle \mathtt{sketch} \langle \cup_i S_i\rangle \implies sp(\cup_i R_i, \mathtt{sketch}) \preceq_a (\cup_i S_i) \ .$$

From the preconditions we know that $\vdash_a \langle R_i\rangle \mathtt{sketch}\langle S_i\rangle$ hold. Applying the induction hypothesis yields $sp(R_i, \mathtt{sketch}) \preceq_a S_i$. Using monotonicity of the strongest post function, we get $sp(\cup_i R_i, \mathtt{sketch}) \preceq_a S_i$ for every i. Together, this means $sp(\cup_i R_i, \mathtt{sketch}) \preceq_a (\cup_i S_i)$ This concludes the proof. □

Proof of Theorem 5.1.

We prove soundness by induction over the structure of the sketch.

Base Case: We show $\models vc(\langle R\rangle \mathtt{com}\langle S\rangle) \implies \vdash_a \langle R\rangle \mathtt{com}\langle S\rangle$. Because the verification conditions hold, we know that the inequality $sp(R, \mathtt{com}) \preceq_a S$ is true. The realizability triple $\vdash_a \langle R\rangle \mathtt{com}\langle S\rangle$ immediately follows from Corollary 1.

Induction Step: In the first case, the sketch is a sequence. We know the following verification conditions hold: $\models vc(\langle R\rangle \mathtt{sketch}_1; \mathtt{sketch}_2\langle S\rangle)$. Thus, we know that the verification conditions $\models vc(\langle R\rangle \mathtt{sketch}_2\langle sp(R, \mathtt{sketch}_1)\rangle)$ and $\models vc(\langle sp(R, \mathtt{sketch}_1)\rangle \mathtt{sketch}_2\langle S\rangle)$ also hold. Applying the induction hypothesis yields

$$\vdash_a \langle R\rangle \mathtt{sketch}_1\langle sp(R, \mathtt{sketch}_1)\rangle \text{ and } \vdash_a \langle sp(R, \mathtt{sketch}_1)\rangle \mathtt{sketch}_2\langle S\rangle \ .$$

Using the rule (**SEQ**), we get $\vdash_a \langle R\rangle \mathtt{sketch}_1; \mathtt{sketch}_2\langle S\rangle$.

If the sketch is a choice, we have $sp(R, \mathtt{sketch}_1 + \mathtt{sketch}_2) \preceq_a S$ and for every $r \in R$ we have $\models vc(\langle\{r\}\rangle \mathtt{sketch}_1\langle sp(\{r\}, \mathtt{sketch}_1)\rangle)$ and $\models vc(\langle\{r\}\rangle \mathtt{sketch}_2\langle sp(\{r\}, \mathtt{sketch}_2)\rangle)$. Applying the induction hypothesis yields the two realizability triples $\vdash_a \langle\{r\}\rangle \mathtt{sketch}_1\langle sp(\{r\}, \mathtt{sketch}_1)\rangle$ and $\vdash_a \langle\{r\}\rangle \mathtt{sketch}_2\langle sp(\{r\}, \mathtt{sketch}_2)\rangle$ for every $r \in R$. Using rule (**CSQ**), we can show the realizability triple $\vdash_a \langle\{r\}\rangle \mathtt{sketch}_1\langle sp(\{r\}, \mathtt{sketch}_1 + \mathtt{sketch}_2)\rangle$ and $\vdash_a \langle\{r\}\rangle \mathtt{sketch}_2\langle sp(\{r\}, \mathtt{sketch}_1 + \mathtt{sketch}_2)\rangle$. Using the rule (**DEM**), we get $\vdash_a \langle\{r\}\rangle \mathtt{sketch}_1 + \mathtt{sketch}_2\langle sp(\{r\}, \mathtt{sketch}_1 + \mathtt{sketch}_2)\rangle$ for every $r \in R$. Using the gather rule, we get $\vdash_a \langle R\rangle \mathtt{sketch}_1 + \mathtt{sketch}_2\langle sp(R, \mathtt{sketch}_1 + \mathtt{sketch}_2)\rangle$. Applying the rule (**CSQ**), we get $\vdash_a \langle R\rangle \mathtt{sketch}_1 + \mathtt{sketch}_2\langle S\rangle$.

If the sketch is a loop, we have the verification conditions $\models (\bigcup_{i\in I} vc(\langle\{i\}\rangle \mathtt{s}\langle\{i\}\rangle))$ from the check annotation function and the inequality $sp(R, \mathtt{s}^*[I]) \preceq_a S$. The induction hypothesis yields the realizability triple $\vdash_a \langle\{i\}\rangle \mathtt{s}\langle\{i\}\rangle$ for every $i \in I$. Applying the rule (**LOOP**) yields $\vdash_a \langle\{i\}\rangle \mathtt{s}^*\langle\{i\}\rangle$ for every $i \in I$. Since $sp(R, \mathtt{s}^*[I])$ is a subset of I we can use multiple applications of the rule (**GATHER**) to get $\vdash_a \langle sp(R, \mathtt{s}^*[I])\rangle \mathtt{s}^*\langle sp(R, \mathtt{s}^*[I])\rangle$. Per definition, we have $R \preceq_a sp(R, \mathtt{s}^*[I])$. Using the rule (**CSQ**), we get that $\vdash_a \langle R\rangle \mathtt{s}^*\langle S\rangle$ holds.

If the sketch is a nonterminal, we know that $sp(R, \mathtt{N}[\Gamma]) \preceq_a S$ holds and from the check annotations function we know there is a j with $\models vc(\langle R\rangle \mathtt{prog}\langle sp(R, \mathtt{prog})\rangle)$ for every prog of the oracle function $\mathbb{O}(\mathtt{N}, j)$. Also, we have $(\bigcup_{\mathtt{prog}\in\mathbb{O}(\mathtt{N},j)} sp(R, \mathtt{prog})) \preceq_a \Gamma(R)$. Using the induction hypothesis and rule (**ANG**), we get $\vdash_a \langle R\rangle \mathtt{N}\langle sp(R, \mathtt{prog})\rangle$ for every prog of $\mathbb{O}(\mathtt{N}, j)$. Using the gather rule, we

get $\vdash_a \langle R\rangle \mathtt{N}\langle \bigcup_{\mathtt{prog}\in\mathbb{O}(\mathtt{N},j)} sp(R,\mathtt{prog})\rangle$. Using the rule (**CSQ**), we get $\vdash_a \langle R\rangle \mathtt{N}\langle \Gamma(R)\rangle$. Since $\Gamma(R)$ is equal to $sp(R,\mathtt{N}[\Gamma])$, we can apply (**CSQ**) one more time to get $\vdash_a \langle R\rangle \mathtt{N}\langle S\rangle$.

Next, we show completeness. First, see that if $S \preceq_a S^{\mathsf{w}}$ and $\models vc(\langle R\rangle \mathtt{sketch}\langle S\rangle)$ then we also have $\models vc(\langle R\rangle \mathtt{sketch}\langle S^{\mathsf{w}}\rangle)$. We proceed by induction over the structure of sketches.

Base Case: We have $\mathtt{sketch} = \mathtt{com}$. Completeness results from completeness of *sp*.

Induction Step: We have $\models_a \langle R\rangle \mathtt{sketch}_1; \mathtt{sketch}_2\langle S\rangle$. From soundness of the strongest post and the induction hypothesis, we get $\models vc(\langle R\rangle \mathtt{sketch}_1\langle sp(R,\mathtt{sketch}_1)\rangle)$ and we also get the verification conditions of $\models vc(\langle sp(R,\mathtt{sketch}_1)\rangle \mathtt{sketch}_2\langle sp(R,\mathtt{sketch}_1;\mathtt{sketch}_2)\rangle)$. Because of completeness of the strongest post and our remark that the post condition in verification conditions may be weakened, we get $\models vc(\langle sp(R,\mathtt{sketch}_1)\rangle \mathtt{sketch}_2\langle S\rangle)$.

The proof for choices directly follows from soundness and completeness of the strongest post function. Then, only an application of the induction hypothesis is left.

In the case of loops we can discharge the first inequality by soundness and completeness from strongest post. The inequalities from the check annotation function are true because the annotations are sound and thus the induction hypothesis can be applied.

The case for non-terminals is more cumbersome. The first inequality follows from completeness of the strongest post function. We now discuss why the inequalities in the check annotations function hold: From $\models_a \langle R\rangle \mathtt{N}[\Gamma]\langle S\rangle$, we get for every $s \in S$ a program prog and an $r \in S$ for which $\models_d \{r\}\mathtt{prog}\{s\}$ holds. This implies that $\models_a \langle\{r\}\rangle \mathtt{prog}\langle\{s\}\rangle$ holds. From completeness of the strongest post, we get $sp(\{r\},\mathtt{prog}) \preceq_a \{s\}$. By monotonicity of the strongest post function, we get $sp(R,\mathtt{prog}) \preceq_a \{s\}$. We collect the programs returned from completeness in the set P. We get that $(\cup_{\mathtt{prog}\in P} sp(R,\mathtt{prog})) \preceq_a S$. Since S is finite, P is finite. Thus, there is a j for which $P \subseteq \mathbb{O}(\mathtt{N},j)$ holds. With this, we also have $(\cup_{\mathtt{prog}\in\mathbb{O}(\mathtt{N},j)} sp(R,\mathtt{prog})) \preceq_a S$. The verification conditions on the individual programs hold because of the soundness of the strongest post function and the induction hypothesis. □

A.4 Proofs for Section 6

Proof of Theorem 6.1

First, we state the theorem using the new notation:

Theorem A.13 (*syn*-Sound-And-Complete). *Consider* $\vdash_p \langle R\rangle \mathtt{po}\langle S\rangle$ *and* $s \in S$ *with* $s \neq fail$. *Then* $syn(\langle R\rangle \mathtt{po}\langle S\rangle, s) = (r,\mathtt{prog})$ *with* $r \in R$, $r \neq fail$, $\mathtt{prog} \in drv(sketch(\mathtt{po}))$, *and* $\models_d \{r\}\mathtt{prog}\{s\}$. *The number of SMT solver calls is at most* $|\mathtt{po}|$.

Proof. We proof this theorem by induction over the shape of $\langle R\rangle \mathtt{po}\langle S\rangle$.

Base Case: We have the valid realizability triple $\langle R\rangle \mathtt{com}\langle S\rangle$. Thus, we know for every $s' \in S$ there is an $r \in R$ with $[\![\mathtt{com}]\!](r) \preceq_d s'$. Since s is of S, the call $syn(\langle R\rangle \mathtt{com}\langle S\rangle, s)$ eventually terminates returning $(r,\mathtt{com})$. The predicate r cannot be *fail* because s is not *fail* and com preserves *fail*. Also $\mathtt{com} \in drv(\mathtt{com})$ holds trivially. Moreover, since $[\![\mathtt{com}]\!](r) \preceq_d s$, the Hoare triple $\models_d \{r\}\mathtt{com}\{s\}$ holds. Since we only have to go through R once to find a suitable r, the number of SMT solver calls is at most $|R| \leq |\mathtt{po}|$.

Induction Step: In the first case, we have $\langle R\rangle \mathtt{po}_1; \mathtt{po}_2\langle S\rangle$. Thus, the call is $syn(\langle R\rangle \mathtt{po}_1; \mathtt{po}_2\langle S\rangle, s)$. Let T be the intermediary selection. Applying the induction hypothesis, the call $syn(\mathtt{po}_2, s)$ returns $(t,\mathtt{prog}_2)$ with $t \in T$, $t \neq fail$, $\mathtt{prog}_2 \in drv(sketch(\mathtt{po}_2))$, and $\models_d \{t\}\mathtt{prog}_2\{s\}$. Also the number of SMT calls is at most $|\mathtt{po}_2|$. Then we call $syn(\mathtt{po}_1, t)$ and by the induction hypothesis we get $(r,\mathtt{prog}_1)$ with $r \in R$, $r \neq fail$, $\mathtt{prog}_1 \in drv(sketch(\mathtt{po}_1))$, and $\models_d \{r\}\mathtt{prog}_1\{t\}$. Also the number of SMT calls is at most $|\mathtt{po}_1|$. Put together, the function returns $(r,\mathtt{prog}_1;\mathtt{prog}_2)$. Since both Hoare triples hold, we get $\models_d \{r\}\mathtt{prog}_1;\mathtt{prog}_2\{s\}$. Also, we have $\mathtt{prog}_1;\mathtt{prog}_2 \in drv(sketch(\mathtt{po}))$. And the number of SMT calls is at most $|\mathtt{po}_1| + |\mathtt{po}_2| \leq |\mathtt{po}|$.

In the next case, we have $\langle R\rangle \mathtt{N(po)} \langle S\rangle$. We have two sub-cases: First, po is only one proof outline as opposed to many separated by the | symbol. Then the pre and post condition of po and N(po) match. We directly get the sought after result by applying the induction hypothesis and seeing that $drv(sketch(\mathtt{po})) \subseteq drv(sketch(\mathtt{N(po)}))$. In the second sub-case, we know that $\mathtt{po} = \mathtt{po}_1 \mid \mathtt{po}_2$. Since S is the union of all posts of po, we know that eventually we try out a subproof with s in its post. Then, the induction hypothesis is applied again directly yielding the required results.

In the next case, we have $\langle R\rangle \mathtt{po}_1 \langle S\rangle + \langle R\rangle \mathtt{po}_2 \langle S\rangle$. Because the proof outline is valid, we know that for every $s \in S$ there is an r of R for which there is a program $\mathtt{prog}_1 + \mathtt{prog}_2$ for which $\models_d \{r\}\mathtt{prog}_1 + \mathtt{prog}_2\{s\}$ holds. This implies that $\models_d \{r\}\mathtt{prog}_i\{s\}$ must hold. This also means, the realizability triple $\models_a \langle r\rangle \mathtt{prog}_i \langle s\rangle$ holds which in turn implies that $\models_a \langle r\rangle drv(\mathtt{po}_i)\langle s\rangle$ holds. By completeness, the outlines $\mathtt{po}'_i$ can be built when supplied with the correct $r \in R$. Then, we use the induction hypothesis to get $(r, \mathtt{prog}_i)$ from the recursive calls with $r \in \{r\} \subseteq R$, and the Hoare triple $\models_d \{r\}\mathtt{prog}_i\{s\}$, and $\mathtt{prog}_i \in drv(sketch(\mathtt{po}'_i))$, and the number of SMT solver calls for each recursion is at most $|\mathtt{po}'_i|$. Since $sketch(\mathtt{po}_i) = sketch(\mathtt{po}'_i)$ we also have that $\mathtt{prog}_1 + \mathtt{prog}_2$ is of $sketch(\mathtt{po}_1 + \mathtt{po}_2)$. Because $\models_d \{r\}\mathtt{prog}_i\{s\}$, we also have $\models_d \{r\}\mathtt{prog}_1 + \mathtt{prog}_2\{s\}$. When the proof outlines $\mathtt{po}'_i$ are properly looked up, there is no need for additional SMT calls. Thus, the SMT calls are at most $|\mathtt{po}'_1 + \mathtt{po}'_2| \leq |\mathtt{po}_1 + \mathtt{po}_2|$.

In the last case consider a loop $\langle R\rangle\langle I\rangle \mathtt{po} \langle I\rangle^* \langle S\rangle$. In the case the realizability triple of the loop was weakened, we keep the original invariant selections. Thus R resp. S are stronger resp. weaker than I. We have $I \preceq_a S$. Therefore, there is an i of I with $i \preceq_d s$. This i will eventually be found by the *syn* function. Since the proof is sound, I is a sound invariant for $sketch(\mathtt{po})$. Thus, for every i of I, the realizability triple $\vdash_a \langle\{i\}\rangle sketch(\mathtt{po})\langle\{i\}\rangle$ is valid. Therefore, the proof outline $\mathtt{po}'$ can be constructed. By the induction hypothesis, $syn(\mathtt{po}', i)$ returns $(i, \mathtt{prog})$ with $\mathtt{prog} \in drv(sketch(\mathtt{po}'))$, and $\models_d \{i\}\mathtt{prog}\{i\}$, and the number of SMT solver calls is at most $|\mathtt{po}'|$. Because $\models_d \{i\}\mathtt{prog}\{i\}$ holds, we also have $\models_d \{i\}\mathtt{prog}^*\{i\}$. Since $R \preceq_a I$ there is an r of R with $r \preceq_d i$. Since $i \preceq_d s$, we can weaken to $\models_d \{r\}\mathtt{prog}^*\{s\}$ and return $r \in R$. Because $\mathtt{prog} \in drv(sketch(\mathtt{po}'))$, we have $\mathtt{prog}^* \in drv(sketch(\mathtt{po}')^*)$. Because $sketch(\mathtt{po}') = sketch(\mathtt{po})$, we have $\mathtt{prog}^* \in drv(sketch(\mathtt{po})^*)$. Because the proof outline $\mathtt{po}'$ can be looked up and does not need recomputing, we need at most $|I|$ additional SMT calls. All in all, we have at most $|\mathtt{po}|$ SMT calls. □

A.5 Proofs for Section 7

Definition 2. *An abstract predicate is fail or maps variables to sets of states of the SMR automaton.*

$$Predicates^{\#} = (Vars \to \mathcal{P}(O)) \cup \{fail\}$$

To substitute the original more precise relation $\preceq_d$ defined on *Predicates*, we introduce a new relation on the abstracted domain.

Definition 3. *We define a stronger relation $\preceq_d^{\#}$ on $Predicates^{\#}$.*

$$a \preceq_d^{\#} b \;\Leftrightarrow\; b = fail \vee \forall v \in Vars : a(v) \subseteq b(v)$$

Ignoring the predicate *fail*, this relation matches the order in Meyer and Wolff's paper.

Definition 4. *We define a predicate abstraction function to abstract the original predicate and a predicate concretisation function to concretize abstract predicates. The predicate abstraction function is a cartesian abstraction:*

$$\alpha_d : Predicates \to Predicates^{\#}$$

Let v be a variable in Vars. Then, the function is defined as follows:

$$\begin{cases} \alpha_d(r) = fail & , r = fail \\ \alpha_d(r)(v) = \bigcup_{q \in r} q(v) & , else \end{cases}$$

We continue with the demon concretisation function:

$$\gamma_d : \mathit{Predicates}^\# \to \mathit{Predicates}$$

$$\gamma_d(a) = \begin{cases} \mathit{fail} & , a = \mathit{fail} \\ \{q \in \mathit{States} \mid \forall v \in \mathit{Vars} : q(v) \subseteq a(v)\} & , \mathit{else} \end{cases}$$

Lemma A.14. $r \preceq_d s \implies \alpha_d(r) \preceq_d^\# \alpha_d(s)$.

Proof. If the predicate s is *fail*, then $\alpha_d(s)$ is also *fail* so the inequality holds. Otherwise, we have

$$\begin{aligned} r \subseteq s &\implies (\bigcup_{q \in r} q(v)) \subseteq (\bigcup_{q \in s} q(v)) \\ &\implies \alpha_d(r) \preceq_d^\# \alpha_d(s) \ . \end{aligned}$$

Here, v is any variable of *Vars*. □

Lemma A.15. $a \preceq_d^\# b \implies \gamma_d(a) \preceq_d \gamma_d(b)$.

Proof. If the abstract predicate b is *fail*, then $\gamma_d(b)$ is also *fail* so the inequality holds. Otherwise, we have

$$\begin{aligned} a(v) \subseteq b(v) &\implies \{p \in \mathit{States} \mid \forall v' \in \mathit{Vars} : p(v) \subseteq a(v)\} \subseteq \\ &\qquad \{p \in \mathit{States} \mid \forall v' \in \mathit{Vars} : p(v) \subseteq b(v)\} \\ &\implies \gamma_d(a) \preceq_d \gamma_d(b) \ . \end{aligned}$$

Here, v is any variable of *Vars*. □

Lemma A.16. *The pair (α_d, γ_d) is a Galois connection between Predicates and Predicates$^\#$:*

$$(\mathit{Predicates}, \preceq_d) \underset{\alpha_d}{\overset{\gamma_d}{\leftrightarrows}} (\mathit{Predicates}^\#, \preceq_d^\#)$$

This means, the following inequalities hold:

$$\forall r \in \mathit{Predicates} : r \preceq_d \ \gamma_d(\alpha_d(r))$$
$$\forall a \in \mathit{Predicates}^\# : \alpha_d(\gamma_d(a)) \preceq_d^\# \ a$$

Proof. We start the proof with the first equation. Let r be of *Predicates*. If $r = \mathit{fail}$ then $\gamma_d(\alpha_d(r)) = \mathit{fail}$. Thus, the inequality $r \preceq_d \gamma_d(\alpha_d(r)) = \mathit{fail}$ holds. If r is not *fail*, then

$$\gamma_d(\alpha_d(r)) = \{p \in \mathit{States} \mid \forall v \in \mathit{Vars} : p(v) \subseteq \bigcup_{q \in r} q(v)\} \ .$$

Let the state q be of r. Then q is also in $\gamma_d(\alpha_d(r))$. Therefore $r \subseteq \gamma_d(\alpha_d(r))$. And that means $r \preceq_d \gamma_d(\alpha_d(r))$.

We continue with the second equation. Let a be of *Predicates*$^\#$. If a is *fail*, then $\alpha_d(\gamma_d(a))$ is trivially stronger. If a is not *fail*, then we show that for any v of *Vars* the inclusion $\alpha_d(\gamma_d(a))(v) \subseteq a(v)$ holds. Let v be of *Vars*. Then

$$\begin{aligned} \alpha_d(\gamma_d(a))(v) &= \cup_{q \in \gamma_d(a)} \ q(v) \\ &= \cup_{q \in \{p \in \mathit{States} \mid \forall v' \in \mathit{Vars} : p(v') \subseteq a(v')\}} \ q(v) \\ &\subseteq a(v) \ . \end{aligned}$$

This concludes the proof. □

Lemma A.17. *The relation $\preceq_d^\#$ is a partial order relation on Predicates$^\#$.*

Proof. We first show reflexivity. Let a be of $Predicates^{\#}$. If $a = fail$, then the inequality $a \preceq_d^{\#} a$ trivially holds. Otherwise, let v be a variable of *Vars*. Then the inclusion $a(v) \subseteq a(v)$ also holds.

We continue with transitivity. Let a, b and c be of $Predicates^{\#}$. We assume that the inequalities $a \preceq_d^{\#} b$ and $b \preceq_d^{\#} c$ are true. If $c = fail$, then the inequality $a \preceq_d^{\#} c$ trivially holds. Otherwise, we get that $b \neq fail$ and also $a \neq fail$. Let v be of *Vars*. We have $a(v) \subseteq b(v)$ and $b(v) \subseteq c(v)$. Through transitivity of the relation $\subseteq$ we also get the inclusion $a(v) \subseteq c(v)$.

Lastly, we show antisymmetry. Let a and b be of $Predicates^{\#}$. We have $a \preceq_d^{\#} b$ and $b \preceq_d^{\#} a$. If a is $fail$ then b must also be $fail$ and vice versa. If neither one is fail, let v be of *Vars*. We know that the inequalities $a(v) \subseteq b(v)$ and $b(v) \subseteq a(v)$ are true and thus we have $a(v) = b(v)$ for all v of *Vars*. That means, a and b are equal. □

Lemma A.18. *The partial order* $(Predicates^{\#}, \preceq_d^{\#})$ *is a complete lattice. In fact, the join and meet can be computed by the following equations: Let* A *be a subset of* $Predicates^{\#}$.

$$\begin{cases} \sqcup(A) = fail & , fail \in A \\ \sqcup(A)(v) = \bigcup_{a \in A} a(v) & , else \end{cases}$$

$$\begin{cases} \sqcap(R) = fail & , R = \{fail\} \\ \sqcap(R) = \bigcap_{a \in (A \setminus \{fail\})} a(v) & , else \end{cases}$$

Proof. Let A be a subset of $Predicates^{\#}$. Let a be an element of A. We first show that the join is an upper bound $a \preceq_d^{\#} \sqcup(A)$. If $fail \in A$, then $\sqcup(A) = fail$ so the inequality $a \preceq_d^{\#} \sqcup(A)$ trivially holds. Otherwise, let v be of *Vars*. The inclusion $a(v) \subseteq \bigcup_{a' \in A} a'(v)$ holds trivially, since $a \in A$. Now, to show that the join is the least upper bound, let u be an upper bound of A. If u is $fail$ the inequality $\sqcup(A) \preceq_d^{\#} u$ holds trivially. Otherwise, if u is not $fail$, we know $fail \notin A$. Let v be of *Vars*. For any abstract demon a of A, $a(v) \subseteq u(v)$ must hold. Therefore the inclusion $\sqcup(A)(v) = \bigcup_{a \in A} a(v) \subseteq u(v)$ holds. Thus, the inequality $\sqcup(A) \preceq_d^{\#} u$ is true.

Moving on with the meet. Let a be an element of A. We first show that the meet is a lower bound: $\sqcap(A) \preceq_d^{\#} a$. If $A = \{fail\}$, then $a = fail$ so the inequality $\sqcap(A) \preceq_d^{\#} a$ trivially holds. Otherwise, let v be of *Vars*. Then, by definition we know that $\sqcap(A)(v) = \bigcap_{a' \in A \setminus \{fail\}} a'(v)$. If a is $fail$, the inequality trivially holds. Otherwise, $\sqcap(A)(v) \subseteq a(v)$ is also true. Now, to show that the meet is the greatest lower bound, let l be a lower bound of A. If l is $fail$, A must be $\{fail\}$ and thus $\sqcap(A) = fail$, so $l \preceq_d^{\#} \sqcap(A)$ trivially holds. Otherwise, let v be of *Vars*. Then for any a of A that is not $fail$, $l(v) \subseteq a(v)$ must hold. Thus, $l(v) \subseteq \bigcap_{a' \in A \setminus \{fail\}} a'(v) = \sqcap(A)(v)$ holds. And therefore the inequality $l \preceq_d^{\#} \sqcap(A)$ is true. □

Next, we lift the abstract predicates to abstract selections. We remind the reader, that selections are sets of predicates, i.e. $Selections = \mathcal{P}(Predicates)$.

Definition 5. *An* Abstract Selection *is a set of abstract predicates.*

$$Selections^{\#} = \mathcal{P}(Predicates^{\#})$$

The more versatile relation on abstract angels is the same as on *Selections*, except that the elements of the abstract selections are compared using the abstract more precise relation defined on abstract predicates.

Definition 6. $A \preceq_a^{\#} B \iff \forall b \in B : \exists\, a \in A : a \preceq_d^{\#} b$

Definition 7. *The selection abstraction function* α_a *relates elements of Selections to their abstract representation in* $Selections^{\#}$.

$$\alpha_a : Selections \longrightarrow Selections^{\#}$$

$$\alpha_a(R) = \{\alpha_d(r) \mid r \in R\}$$

The selection concretisation function γ_a relates elements of Selections$^\#$ to the elements of the original domain Selections.

$$\gamma_a : \mathit{Selections}^\# \to \mathit{Selections}$$

$$\gamma_a(A) = \{\gamma_d(a) \mid a \in A\}$$

Lemma A.19. $R \preceq_a S \implies \alpha_a(R) \preceq_a^\# \alpha_a(S)$.

Proof. Let $\alpha_d(s)$ be of $\alpha_a(S)$. We know there is a predicate r in R more precise than s. Thus, the inequality $\alpha_d(r) \preceq_d^\# \alpha_d(s)$ is true. And therefore the inequality $\alpha_a(R) \preceq_a^\# \alpha_a(S)$ holds. □

Lemma A.20. $A \preceq_a B \implies \gamma_a(A) \preceq_a \gamma_a(B)$.

Proof. Let the predicate $\gamma_d(b)$ be of $\gamma_a(B)$. We know there is an abstract predicate a of A with $a \preceq_d^\# b$. Thus, $\gamma_d(a)$ is more precise than $\gamma_d(b)$. Since $\gamma_d(a) \in \gamma_a(A)$, we know the inequality $\gamma_a(A) \preceq_a \gamma_a(B)$ holds. □

Lemma A.21. *The pair (α_a, γ_a) is a Galois connection between Selections and Selections$^\#$:*

$$(\mathit{Selections}, \preceq_a) \underset{\alpha_a}{\overset{\gamma_a}{\leftrightarrows}} (\mathit{Selections}^\#, \preceq_a^\#)$$

This means, the following inequalities hold:

$$\forall R \in \mathit{Selections} : R \preceq_a \gamma_a(\alpha_a(R))$$

$$\forall A \in \mathit{Selections}^\# : \alpha_a(\gamma_a(A)) \preceq_a^\# A$$

Proof. We start the proof with the first inequality. Let R be of *Selections*. Let the predicate s be of $\gamma_a(\alpha_a(R))$. Then $s \in \{\gamma_d(a) \mid a \in \{\alpha_d(r) \mid r \in R\}\}$. Thus, $s \in \{\gamma_d(\alpha_d(r)) \mid r \in R\}$. Since the inequality $r \preceq_d^\# \gamma_d(\alpha_d(r))$ is true, we get that there is an r in R more precise than s, thus the inequality $R \preceq_a^\# \gamma_a(\alpha_a(R))$ holds.

Moving on with the second inequality, let A be of *Selections*$^\#$. Let the abstract predicate a be of A. We have $\alpha_a(\gamma_a(A)) = \{\alpha_d(r) \mid r \in \{\gamma_d(a) \mid a \in A\}\}$. Thus, $\alpha_d(\gamma_d(a))$ is in $\alpha_a(\gamma_a(A))$. Since we know that the inequality $\alpha_d(\gamma_d(a)) \preceq_d^\# a$ holds, we get that the inequality $\alpha_a(\gamma_a(A)) \preceq_a^\# A$ is true. □

Lemma A.22. *The pair (α_d, γ_d) is a Galois insertion, i.e. the following equation holds for any a of Predicates$^\#$: $\alpha_d(\gamma_d(a)) = a$.*

The pair (α_a, γ_a) also is a Galois insertion, i.e. the following equation holds true for any A of Selections$^\#$: $\alpha_a(\gamma_a(A)) = A$.

Proof. We start with the first equality. Since we know the inequality $\alpha_d(\gamma_d(a)) \preceq_d^\# a$ holds, it is sufficient to show that the inequality $a \preceq_d^\# \alpha_d(\gamma_d(a))$ is true because the relation $\preceq_d^\#$ is antisymmetric. So let a be of *Predicates*$^\#$. If $a = \mathit{fail}$, then $\alpha_d(\gamma_d(a))$ is also *fail*. If a is not *fail*, then let v be any variable of *Vars*. Then

$$\begin{aligned} a(v) &\subseteq \cup_{p \in \mathit{States} \wedge p(v) \subseteq a(v)} p(v) \\ &\subseteq \cup_{p \in \{p' \in \mathit{States} \mid \forall v' \in \mathit{Vars}: p'(v') \subseteq a(v')\}} p(v) \\ &= \alpha_d(\gamma_d(a)) \end{aligned}$$

The second inclusion holds true because of the abundance of available states. Therefore, the inequality $a \preceq_d^\# \alpha_d(\gamma_d(a))$ holds. That means $a = \alpha_d(\gamma_d(a))$ is true.

Because the relation on abstract angels is not antisymmetric, we show equality directly:

$$\begin{aligned}\alpha_a(\gamma_a(A)) &= \{\alpha_d(r) \mid r \in \{\gamma_d(a) \mid a \in A\}\}\\ &= \{\alpha_d(\gamma_d) \mid a \in A\}\\ &= \{a \mid a \in A\}\\ &= A\ .\end{aligned}$$

□

Definition 8. *Meyer and Wolff also provide an interpretation of commands that either maps to another state or to their notation of failing:* $\top$.

$$[\![\mathsf{com}]\!]^\#(a) = \begin{cases} \mathit{fail} & , a = \mathit{fail} \vee [\![\mathsf{com}]\!]_t(a) = \top \\ [\![\mathsf{com}]\!]_t(a) & ,\ \mathit{else} \end{cases}$$

Lemma A.23. *The abstract interpretation of commands* $[\![\mathsf{com}]\!]^\#$ *is a safe abstraction for* $[\![\mathsf{com}]\!]$*, i.e. the following equation holds:*

$$[\![\mathsf{com}]\!]^\#(a) = \alpha_d([\![\mathsf{com}]\!](\gamma_d(a)))$$

Proof. The original interpretation of com in Meyer and Wolff's work is monotonic. Thus, $[\![\mathsf{com}]\!]^\#$ is monotonic. We show equality by showing $\preceq_d$ in both ways. By antisymmetry, equality follows.

We start with $[\![\mathsf{com}]\!]^\#(a) \preceq_d^\# \alpha_d([\![\mathsf{com}]\!](\gamma_d(a)))$: If the right side is *fail* we are done. Otherwise, see that $a \in \gamma_d(a)$. Thus, $[\![\mathsf{com}]\!]^\#(a) \in [\![\mathsf{com}]\!](\gamma_d(a))$. Let v be a variable of *Vars*. Then $[\![\mathsf{com}]\!]^\#(a)(v)$ is a subset of $\alpha_d([\![\mathsf{com}]\!](\gamma_d(a)))(v)$. Therefore, the inequality $[\![\mathsf{com}]\!]^\#(a) \preceq_d^\# \alpha_d([\![\mathsf{com}]\!](\gamma_d(a)))$ holds.

We proceed with the other direction: Again, if the left side is *fail* we are done. Otherwise, see that for all q of $\gamma_d(a)$, $q \preceq_d^\# a$ holds. Now, let b be an element of $[\![\mathsf{com}]\!](\gamma_d(a))$. Then, $b \preceq_d^\# [\![\mathsf{com}]\!]^\#(a)$ holds because the original interpretation of commands is monotonic. Thus, for any variable v of *Vars*, we have $b(v) \subseteq [\![\mathsf{com}]\!]^\#(a)(v)$. Therefore, $\alpha_d([\![\mathsf{com}]\!](\gamma_d(a)))(v)$ is a subset of $[\![\mathsf{com}]\!]^\#(a)(v)$. So, the inequality holds. This concludes the proof. □

Lemma A.24. *The following inequality holds:*

$$\alpha_a(sp(\gamma_a(A), \mathsf{com})) \preceq_a^\# sp^\#(A, \mathsf{com})$$

Proof.

$$\begin{aligned}\alpha_a(sp(\gamma_a(A), \mathsf{com})) &= \{\alpha_d([\![\mathsf{com}]\!](r)) \mid r \in \gamma_a(A)\}\\ &= \{\alpha_d([\![\mathsf{com}]\!](\gamma_d(a))) \mid a \in A\}\\ &= \{[\![\mathsf{com}]\!]^\#(a) \mid a \in A\}\\ &= sp^\#(A, \mathsf{com})\end{aligned}$$

The inequality holds because $\preceq_a^\#$ is reflexive. □

Lemma A.25. *The following equations hold:*

$$\begin{aligned}\alpha_a(R \cup S) &= \alpha_a(R) \cup \alpha_a(S)\\ \gamma_a(A \cup B) &= \gamma_a(A) \cup \gamma_a(B)\ .\end{aligned}$$

Proof. We start with the first equation.

$$\begin{aligned}\alpha_a(R \cup S) &= \{\alpha_d(r) \mid r \in R \cup S\}\\ &= \{\alpha_d(r) \mid r \in R\} \cup \{\alpha_d(s) \mid s \in S\} \cup\\ &= \alpha_a(R) \cup \alpha_a(S)\end{aligned}$$

The proof for the second equation is analogous. □

Lemma A.26. *The following equation holds*

$$sp^{\#}(A, \texttt{com}) \preceq_a^{\#} B \implies sp(\gamma_a(A), \texttt{com}) \preceq_a \gamma_a(B)$$

Proof. Because the abstract strongest post is a safe abstraction, we have

$$\alpha_a(sp(\gamma_a(A), \texttt{com})) \preceq_a^{\#} sp^{\#}(A, \texttt{com}) \preceq_a^{\#} B\ .$$

Because we have a Galois connection, the following inequality holds:

$$sp(\gamma_a(A), \texttt{com}) \preceq_a \gamma_a(\alpha_a(sp(\gamma_a(A), \texttt{com})))\ .$$

Applying the monotonicity of *drv* we get the following two inequalities:

$$sp(\gamma_a(A), \texttt{com}) \preceq_a \gamma_a(sp^{\#}(A, \texttt{com}))$$
$$sp(\gamma_a(A), \texttt{com}) \preceq_a \gamma_a(B)$$

□

Proof of Theorem B.1

We prove the theorem by induction over the structure of sketch.

Base Case: If $\texttt{sketch} = \texttt{com}$, then we know the inequality $sp^{\#}(A, \texttt{com}) \preceq_a^{\#} B$ is true. This then implies that the inequality $sp(\gamma_a(A), \texttt{com}) \preceq_a \gamma_a(B)$ holds. Therefore, all verification conditions of $vc(\langle \gamma_a(A) \rangle \texttt{com} \langle \gamma_a(B) \rangle)$ are valid. Thus, we can show the realizability triple $\vdash_a \langle \gamma_a(A) \rangle \texttt{com} \langle \gamma_a(B) \rangle$.

Induction Step: If the sketch $\texttt{sketch} = \texttt{sketch}_1; \texttt{sketch}_2$, we know that the verification conditions of the functions $vc^{\#}(\langle A \rangle \texttt{sketch}_1 \langle sp^{\#}(A, \texttt{sketch}_1) \rangle)$ and $vc^{\#}(\langle sp^{\#}(\texttt{sketch}_1, A) \rangle \texttt{sketch}_2 \langle B \rangle)$ hold. Applying the induction hypothesis, we get that $\vdash_a \langle \gamma_a(A) \rangle \texttt{sketch}_1 \langle \gamma_a(sp^{\#}(A, \texttt{sketch}_1)) \rangle$ and $\vdash_a \langle \gamma_a(sp^{\#}(A, \texttt{sketch}_1)) \rangle \texttt{sketch}_2 \langle \gamma_a(B) \rangle$ are valid. Using rule (**SEQ**), we prove the realizability triple $\vdash_a \langle \gamma_a(A) \rangle \texttt{sketch}_1; \texttt{sketch}_2 \langle \gamma_a(B) \rangle$.

The other rules are also analogous to the original proof. □

Proof of Theorem B.2

By completeness of Hoare logic, we get $\vdash_d \{\gamma_d(a)\} \texttt{prog} \{\gamma_d(b)\}$. Since the rules are the Same as in Meyer and Wolff's type system, we get $\vdash_t \langle \gamma_d(a) \rangle \texttt{prog} \langle \gamma_d(b) \rangle$. □

B DETAILS ON THE APPLICATION

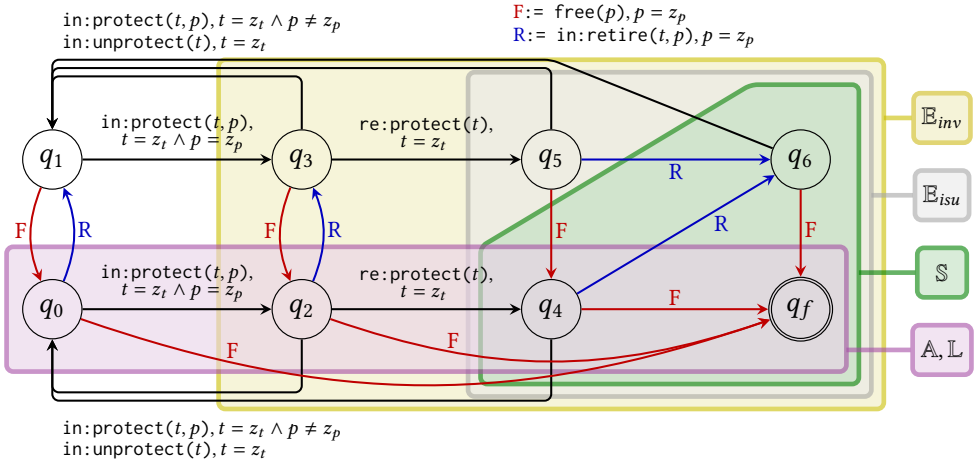
B.1 Background on Safe Memory Reclamation

Our goal is to synthesize code that makes a lock-free data structure memory safe. We recall the basics of safe memory reclamation and outline the approach from [26, 27, 47] to verify that a lock-free data structure properly protects its memory using a safe memory reclamation algorithm. We also demonstrate the approach on an example.

B.2 Background

When writing programs in languages such as C++, memory management is manual. This is opposed to other programming languages, such as Java, which provide *Garbage Collection* (GC) for automatic memory management. Not having GC at hand proves challenging in implementing (and verifying) concurrent, especially lock-free data structures due to use-after-free errors and the ABA problem. For assistance, programmers can resort to Safe Memory Reclamation (SMR) algorithms, e.g. Hazard Pointers (HP) [29]. Instead of freeing memory directly, these algorithms provide a *retire* function that delays freeing memory until it is safe to do so. In order for the SMR algorithm to know when freeing memory is allowed, the programmer has to call SMR specific functions. With HP, the

Fig. 12. SMR automaton for HP, adapted from [47].



programmer has to call a *protect* function on a pointer they want to access. Afterwards, however, the programmer has to check if the protection was successful. In the next chapter, we automatically synthesize the code for such calls and checks.

In [26, 27, 47], Meyer and Wolff proposed a type-based approach to automatically verify lock-free data structures that use SMR algorithms for memory reclamation. We base our synthesis upon their work. The idea is to abstract away the implementation details of the SMR and verify the program with the help of a specification, a so-called SMR automaton. The automata consist of states, and subsets of these states form the types, also called guarantees, in their approach. The states capture the effect that the SMR algorithm has on a pointer. Every SMR automaton comes with at least three guarantees. The local guarantee, $\mathbb{L}$, signals that the pointer is thread local. The active guarantee, $\mathbb{A}$, signals that the pointer is published and was not yet retired. The safe guarantee, $\mathbb{S}$, signals that the pointer is protected by the SMR algorithm. Additionally, SMR automata may bring SMR specific guarantees $\mathbb{E}_L$.

Pointer dereferences are only allowed when the SMR automaton for the pointer is guaranteed to be in a state described by $\mathbb{L}$, $\mathbb{A}$, or $\mathbb{S}$.

The type system tracks the guarantees each pointer has. It annotates the original program commands, SMR commands, and invariant annotations that may have to be added to the code. The invariant annotations provide information about the protection status of a pointer that the type system can rely on, e.g. that a shared pointer is not retired at a certain program location. The invariant annotations need to be verified separately. Importanty, this can be done assuming GC. When the type check is successful and the invariant annotations can be discharged, the program does not unsafely dereference memory and does not suffer from ABAs.

B.3 Example SMR Automaton: Hazard Pointer

We demonstrate the concept of SMR automata on the SMR algorithm Hazard Pointer. In Figure 12, the SMR automaton O for Hazard Pointers is depicted. It consists of eight states, one of them final. The automaton specifies illegal behavior. That means, when using HP, all transitions to the final state will not occur. The automaton takes two parameters: z_p representing the pointer whose state is tracked, and z_t representing the thread whose perspective is taken. The automaton has two extra guarantees: $\mathbb{E}_{inv}$ and $\mathbb{E}_{isu}$. In Figure 12, all guarantees are depicted by colored zones. A pointer is

safe to access when it possesses the guarantees $\mathbb{A}$, $\mathbb{L}$, or $\mathbb{S}$, because all transitions for command `free` lead to the fail state. Therefore, when a pointer is guaranteed to be in one of the states described by the three guarantees, it cannot be freed. Thus, a dereference would be safe. Here, guarantee $\mathbb{S}$ represents the protection by the SMR algorithm. Guarantee $\mathbb{E}_{inv}$ describes the states the automaton is guaranteed to be in when thread z_t invokes a protection on pointer z_p. When that call returns, the automaton is guaranteed to be in the states described by $\mathbb{E}_{isu}$. When thread z_t protects another pointer, or unprotects pointer z_p, the aforementioned guarantees are lost. The transitions for `free` and `retire` do not depend on the executing thread and can be interferences from other threads. The guarantee $\mathbb{A}$ is not closed under interferences. Local pointers cannot experience interference, thus interference can be disregarded for guarantee $\mathbb{L}$. When using HP, after issuing a protection, it must be checked if the protection was successful. This is represented in the automaton by the fact that the guarantees $\mathbb{E}_{isu}$ and $\mathbb{S}$ are not the same. After returning from the protection call, the automaton could be in state q_5 where it is not safe yet. The strategy to protect a pointer therefore is to first call the method `protect` and then compare it to a pointer that possesses the guarantee $\mathbb{A}$. If both are equal, they must be in the state q_4 (q_f is never reached) and thus both automatically get the guarantee $\mathbb{S}$ and are thereby safe to access.

B.4 Example Type Check: Treiber's Stack pop

Fig. 13. Excerpt of pop in Treiber's stack demonstrating Meyer and Wolff's type system using Hazard Pointers (simplified).

```
1  top : O, TOS : O
2  top = TOS;
3  top : O, TOS : O
4  in:protect(top);
5  top : E_inv, TOS : O
6  re:protect();
7  top : E_isu, TOS : O
8  atomic {
9    top : E_isu, TOS : O
10   @inv active(TOS);
11   top : E_isu, TOS : A
12   assume(top == TOS);
13   (top : E_isu ∧ A,
14    TOS : A ∧ E_isu)
15   (top : E_isu ∧ A ∧ S,
16    TOS : A ∧ E_isu ∧ S)
17 }
18 top : E_isu ∧ S, TOS : O
19 d = top.data;
20 top : E_isu ∧ S, TOS : O
```

We demonstrate the type check from [27, 47]. Figure 13 shows an excerpt of the `pop` method from Treiber's stack using Hazard Pointers as the SMR algorithm. There are two pointer variables: the local variable `top` and the shared variable `TOS`. In the beginning, both pointers possess no guarantees. First, the current value of the shared variable is read into `top`. Afterwards a hazard pointer protection for `top` is called. It is invoked in Line 4 and returns in Line 6. After a protection, one has to check if it was successful. This requires two steps. First, we signal the type system that the shared variable `TOS` is not yet retired through the invariant annotation `active(TOS)`. Afterwards, we compare the previously read value stored in the variable `top` with the current value of `TOS`. If both variables point to the same address, we combine the information we have on each variable. Therefore, both pointers now have the guarantees $\mathbb{E}_{isu}$ and $\mathbb{A}$ and by inference also $\mathbb{S}$. Leaving the atomic block, other threads are able to interfere again, so the active guarantee is lost for `top`. Because the pointer `TOS` is shared and other threads can manipulate it in any way, it loses all its guarantees. However, the pointer `top` can now be safely dereferenced in Line 19. We use the theory presented in this paper and leverage the type system discussed above to automatically synthesize the extra code needed to pass the type check, i.e. Lines 4, 6, and 10.

B.5 Assertion Language

Our assertion language for predicates is $Predicates^{\#} = (Vars \to \mathcal{P}(\mathcal{O})) \cup \{fail\}$, and we also call the elements in this domain abstract predicates. We annotate our development by # to indicate that we

Fig. 14. Treiber's Stack Pop with inserted nonterminals. Red nonterminals resolve to `skip`. Purple nonterminals resolve to the gray code beside them.

```
1  BEGIN LOOP {
2  top := TOS; AC;
3  ((assume(top == NULL); AC;
4    result := EMPTY; AC;)
5  +
6  (
7    assume(top != NULL); AC;
8    ((
9      atomic {
10       AC;⊢@inv active(TOS);
11       top := TOS;
12       AC;⊢in:protect(top);
13              re:protect();
14       assume(top != NULL); AC;
15     }
16     AC; next := top.next; AC;
17     atomic {
18       AC;⊢@inv active(TOS);
19       ((assume(CAS(TOS, top, next));
20          AC;
21          flag := true; AC;
22        )
23          +
24        ( flag := false; AC;))
25     }
26     AC;
27     ((assume(flag == true); AC;
28       result := top.data; AC;)
29     +
30     (assume(flag == false); AC;
31      skip; AC;))
32   ))
33   +
34   ( skip; AC;)
35 ))
36 }* END LOOP
```

work with abstract predicates. Functions defined on the abstract domain that are annotated with a # symbol, e.g. $vc^{\#}$. As an abstraction function we have α_d and α_a for predicates and selections, respectively. Abstract selections are a set of abstract predicates. Similarly, the concretisation functions are called γ_d and γ_a. Since it is deterministic, the original interpretation of commands $[\![\texttt{com}]\!]_t$ serves as a safe abstract interpretation of $[\![\texttt{com}]\!]$ when the semantics for *fail* are added. Thus, there is no need to concretize to the non abstract domain. We use the abstract version of verification conditions to proof realizability triples:

Theorem B.1. $\models vc^{\#}(\langle A\rangle \texttt{stmt} \langle B\rangle)$ *implies* $\vdash_a \langle \gamma_a(A)\rangle \texttt{stmt}' \langle \gamma_a(B)\rangle$.

Proofs over programs that start and end in singletons can be replicated in the type system of Meyer and Wolff of which a successful type check is denoted by $\vdash_t$:

Theorem B.2. $\models_d \{\gamma_d(a)\}\texttt{prog}\{\gamma_d(b)\}$ *implies* $\vdash_t \langle a\rangle \texttt{prog} \langle b\rangle$.

C EXAMPLE: POP METHOD IN TREIBER'S STACK

We demonstrate our synthesis approach on the pop method of Treiber's Stack. First, we insert nonterminals that can resolve to calls to the SMR algorithm and invariant annotations into the program code. Then, using verification conditions, it is shown that these nonterminals are sufficient for deriving a memory safe program that passes Meyer and Wolff's type check. Afterwards, using the synthesis function, we derive a memory safe program.

In Figure 14, the pop method of Treiber's stack is depicted. Therein, we already inserted the nonterminal AC. Insertions that will resolve to `skip` are marked in red. Insertions that will resolve to commands other than skip are marked in purple with the commands they will resolve to in gray right beside them. The atomic block in Lines 9 to 15 serves as an atomicity abstraction for `assume(top = TOS)`. The assumption is required to check if a protection was successful. Without diving into details, the atomicity abstraction is required in Meyer and Wolff's type system because

Fig. 15. Insertions for HP in Treiber's Stack.

```
1 AC ::= skip; |
2 atomic {@inv active(TOS);} |
3 (in:protect(top); re:protect(top);)
```

this assumption is at risk of a harmless ABA. A type check does not pass on a harmless ABA. (This is a detail we left out in the overview in Appendix B.2.) The atomicity abstraction circumvents this problem.

In the type system, dereferences and checks for equality can only be performed on pointers that are safe to access, i.e. pointers that possess the active, local, or safe guarantee. Since the `next` pointer is never accessed or compared, it does not need any protection. We therefore omit it in the following. As mentioned before, we assume the shared pointer `TOS` to always be active, therefore a protection is not needed because having the active guarantee $\mathbb{A}$ is sufficient for safe dereferences and comparisons. However, signaling the type system that the pointer is indeed active is still required and is done using invariant annotations. Observe that `TOS` needs to be active at least in Line 19 as it is compared to `top` there. The pointer `top` needs to be protected at multiple locations. It is accessed and compared to another pointer in Lines 16, 19 and 28. Knowing which pointers need to be protected, we restrict nonterminal to the definition depicted in Figure 15. (This optimization is not required.)

We call the resulting program sketch `sketch`. Using verification conditions, the realizability triple $\vdash_a \langle \gamma_d((\texttt{TOS} : O, \texttt{top} : O))\rangle \texttt{sketch} \langle \gamma_d((\texttt{TOS} : O, \texttt{top} : O))\rangle$ is proven. In order to generate verification conditions, the loop of `sketch` needs to be annotated with a loop invariant. We choose the invariant $I = \{(\texttt{TOS} : O, \texttt{top} : O)\}$. In total, the verification conditions function generates 27 unique comparisons. The result is that every comparison holds. Therefore, the above realizability triple is true. Inserting the selections used for generating the verification conditions into the program yields a proof outline `po` with $\mathit{sketch}(\texttt{po}) = \texttt{sketch}$. Having shown that there is a way to concretize this program sketch, we now use the synthesis algorithm to eliminate the nonterminals. Thus, we call the synthesis function with the predicate $(\texttt{TOS} : O, \texttt{top} : O)$ as the target for condition. In Lines 10, 12 and 18 the nonterminals resolve to commands other than `skip`. In the following, we discuss the invocations of the synthesis function where the nonterminal does not resolve to `skip`. In Equation (7), the active invariant annotation for `TOS` is chosen. This insertion can be seen in Line 10. Here, executing the active invariant annotation on the first and second predicate in the pre condition results in the first and second predicate of the post condition, respectively. The third predicate of the post condition can be reached from both of the predicates in the pre condition. From the first predicate, a protection to `top` needs to be issued. From the second predicate, a `skip` is sufficient. The last predicate of the post condition is reached through executing `skip` on the first predicate of the pre condition. Thus, the first, third and fourth predicates of the post condition are reachable from the target pre condition. The first predicate of the post condition matches. Therefore, `@inv active` needs to be inserted and the first predicate of the precondition is returned. Next, in Equation (8), the protections for `top` are synthesized. The result is inserted in Line 12. There are 5 predicates in the post condition of the nonterminal. The first two predicates are results of executing the nonterminal on the second predicate of the precondition. The first predicate of the postcondition stems from calling `skip` or `@inv active(TOS)`. The second predicate is the result of issuing the protection for `top`. The third, fourth and fifth predicate are results of executing the nonterminal on the first predicates of the pre condition. The third predicate is the product of inserting the protection for `top`. The fourth predicate comes from a `skip`. The last predicate

Fig. 16. Recursive calls of decision function

$$
\begin{aligned}
&syn(\langle\gamma_a(\{(\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathcal{O}),(\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathbb{E}_{inv}\wedge\mathbb{E}_{isu})\})\rangle \mathsf{AC}\langle\gamma_a(\{(\mathsf{TOS}:\mathbb{A},\mathsf{top}:\mathcal{O}),\\
&\qquad(\mathsf{TOS}:\mathbb{A},\mathsf{top}:\mathbb{E}_{inv}\wedge\mathbb{E}_{isu}),(\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathbb{E}_{inv}\wedge\mathbb{E}_{isu}),(\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathcal{O})\})\rangle,\\
&\qquad\gamma_d((\mathsf{TOS}:\mathbb{A},\mathsf{top}:\mathcal{O})))\\
&\qquad=(\gamma_d((\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathcal{O})),\texttt{@inv active(TOS);})
\end{aligned}
\tag{7}
$$

$$
\begin{aligned}
&syn(\langle\gamma_a(\{(\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathcal{O}),(\mathsf{TOS}:\mathbb{A},\mathsf{top}:\mathbb{A})\})\rangle \mathsf{AC}\langle\gamma_a(\{(\mathsf{TOS}:\mathbb{A},\mathsf{top}:\mathbb{A}),\\
&\qquad(\mathsf{TOS}:\mathbb{A},\mathsf{top}:\mathbb{E}_{inv}\wedge\mathbb{E}_{isu}\wedge\mathbb{A}\wedge\mathbb{S}),(\mathsf{TOS}:\mathbb{A},\mathsf{top}:\mathcal{O}),\\
&\qquad(\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathcal{O}),(\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathbb{E}_{inv}\wedge\mathbb{E}_{isu})\})\rangle,\\
&\qquad\gamma_d((\mathsf{TOS}:\mathbb{A},\mathsf{top}:\mathbb{E}_{inv}\wedge\mathbb{E}_{isu}\wedge\mathbb{A}\wedge\mathbb{S})))\\
&\qquad=((\mathsf{TOS}:\mathbb{A},\mathsf{top}:\mathbb{A}),\texttt{(in:protect(top); re:protect();)})
\end{aligned}
\tag{8}
$$

$$
\begin{aligned}
&syn(\langle\gamma_a(\{(\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathbb{E}_{inv}\wedge\mathbb{E}_{isu}\wedge\mathbb{S}),\mathit{fail}\})\rangle \mathsf{AC}\langle\gamma_a(\{(\mathsf{TOS}:\mathbb{A},\mathsf{top}:\mathbb{E}_{inv}\wedge\mathbb{E}_{isu}\wedge\mathbb{S}),\\
&\qquad(\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathbb{E}_{inv}\wedge\mathbb{E}_{isu}\wedge\mathbb{S}),\mathit{fail}\})\rangle,\\
&\qquad\gamma_d((\mathsf{TOS}:\mathbb{A},\mathsf{top}:\mathbb{E}_{inv}\wedge\mathbb{E}_{isu}\wedge\mathbb{S})))\\
&\qquad=((\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathbb{E}_{inv}\wedge\mathbb{E}_{isu}\wedge\mathbb{S}),\texttt{@inv active(TOS);})
\end{aligned}
\tag{9}
$$

is the result of calling `@inv active(TOS)`. The second predicate of the post condition matches target post condition. Therefore, the protections for `top` must be synthesized here and the second predicate of the precondition is returned.

Lastly, in Equation (9), the active annotation in Line 18 before the compare and swap is synthesized. There are two predicates in the pre condition, one of which is *fail*. Executing the nonterminal on *fail* leads to *fail* in the post condition. Using the invariant annotation on the first predicate of the pre condition yields the first predicate of the post condition. Inserting `skip` or a protection to `top` leads to the second predicate. The first predicate of the post condition matches the target post condition, thus `@inv active(TOS)` is inserted and the first predicate from the precondition is returned.

We call the version of Treiber's stack with the resolved nonterminals from Figure 14 `prog`. We know that the Hoare triple $\models_d \{\gamma_d((\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathcal{O}))\}\texttt{prog}\{\gamma_d((\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathcal{O}))\}$ holds due to Theorem 6.1. Applying Theorem B.2, we conclude that the resulting program type checks: $\vdash_t \langle(\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathcal{O})\rangle\texttt{prog}\langle(\mathsf{TOS}:\mathcal{O},\mathsf{top}:\mathcal{O})\rangle$ is true. That means, we successfully synthesized a program `prog` that passes Meyer and Wolff's type check and therefore is memory safe (if the invariant annotations can be discharged).