Unrealizability Logic

JINWOO KIM*, Seoul National University, Republic of Korea
LORIS D'ANTONI, University of Wisconsin-Madison, USA
THOMAS REPS, University of Wisconsin-Madison, USA

We consider the problem of establishing that a program-synthesis problem is *unrealizable* (i.e., has no solution in a given search space of programs). Prior work on unrealizability has developed some automatic techniques to establish that a problem is unrealizable; however, these techniques are all *black-box*, meaning that they conceal the reasoning behind *why* a synthesis problem is unrealizable.

In this paper, we present a Hoare-style reasoning system, called *unrealizability logic* for establishing that a program-synthesis problem is unrealizable. To the best of our knowledge, unrealizability logic is the first proof system for overapproximating the execution of an infinite set of imperative programs. The logic provides a general, logical system for building checkable proofs about unrealizability. Similar to how Hoare logic distills the fundamental concepts behind algorithms and tools to prove the correctness of programs, unrealizability logic distills into a single logical system the fundamental concepts that were hidden within prior tools capable of establishing that a program-synthesis problem is unrealizable.

CCS Concepts: • **Theory of computation → Logic and verification**; **Hoare logic**; • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Unrealizability Logic, Unrealizability, Program Synthesis

ACM Reference Format:
Jinwoo Kim, Loris D'Antoni, and Thomas Reps. 2023. Unrealizability Logic. *Proc. ACM Program. Lang.* 7, POPL, Article 23 (January 2023), 30 pages. https://doi.org/10.1145/3571216

1 INTRODUCTION

Program synthesis refers to the task of discovering a program, within a given search space, that satisfies a behavioral specification (e.g., a logical formula, or a set of input-output examples). While there have been many advances in program synthesis, especially in domain-specific settings [Feser et al. 2015; Gulwani 2011; Phothilimthana et al. 2019], program synthesis remains a challenging task with many properties that are not yet well understood.

While tools are becoming better at synthesizing programs, one property that remains difficult to reason about is the *unrealizability* of a synthesis problem, i.e., the non-existence of a solution that satisfies the behavioral specification within the search space of possible programs. One of the ultimate goals behind studying unrealizability is to prune the search space when synthesizing programs, by showing that a certain subset of the search space does not contain the desired solution (see §6 for a further discussion of this idea). Unrealizability also has many applications; for example, one can show that a certain synthesized solution is optimal with respect to some metric by proving

*Part of work done while a student at the University of Wisconsin-Madison.

Authors' addresses: Jinwoo Kim, Seoul National University, Seoul, Republic of Korea, pl@cs.wisc.edu; Loris D'Antoni, University of Wisconsin-Madison, Madison, USA, loris@cs.wisc.edu; Thomas Reps, University of Wisconsin-Madison, Madison, USA, reps@cs.wisc.edu.

2475-1421/2023/1-ART23
https://doi.org/10.1145/3571216

that a better solution to the synthesis problem does not exist—i.e., by proving that the synthesis problem where the search space contains only programs of lower cost is unrealizable [Hu and D'Antoni 2018]. Unrealizability is also used to prune program paths in symbolic-execution engines for program repair [Mechtaev et al. 2018].

Example 1.1. Consider the synthesis problem sy_{first} where the goal is to synthesize a function f that takes as input a state (x, y), and returns a state where $y = 10$. Assume, however, that the search space of possible programs in sy_{first} is defined using the following grammar G_{first}:

$$Start \to y := E \qquad E \to x \mid E + 1$$

Clearly $y := 10 \notin L(Start)$; moreover, all programs in $L(Start)$ are incorrect on at least one input. For example, on the input $x = 15$ every program in the grammar sets y to a value greater than 15. Consequently, sy_{first} is unrealizable.

While it is trivial for a human to establish that sy_{first} is indeed unrealizable, only a small number of known techniques can prove this fact automatically [Hu et al. 2019, 2020; Kim et al. 2021]. However, a common drawback of these techniques is that they do not produce a proof artifact. In particular, Kim et al. [2021] and Hu et al. [2019] rely on external constraint solvers and program verifiers to prove unrealizability. While a solver-based approach has worked well in practice, without extensive knowledge of the internals of the external solvers, it is difficult to understand exactly *why* a synthesis problem is unrealizable.

This paper presents *unrealizability logic*, a proof system for reasoning about the unrealizability of synthesis problems. In addition to the main goal of reasoning about unrealizability, unrealizability logic is designed with the following goals in mind:

- to be a *general* logic, capable of dealing with various synthesis problems;
- to be amenable to *machine reasoning*, as to enable both automatic proof checking and to open future opportunities for automation;
- to *provide insight* into why certain synthesis problems are unrealizable through the process of completing a proof tree.

Via unrealizability logic, one is able to (*i*) reason about unrealizability in a principled, explicit fashion, and (*ii*) produce concrete proofs about unrealizability.

In this paper, unrealizability logic is formulated for synthesis problems over a deterministic, imperative programming language with statements involving Boolean and integer expressions. There are several challenges to creating unrealizability logic, which are illustrated in §2. One such challenge is already illustrated in Example 1.1: because of the recursive definition of nonterminal E, the search space of programs $L(G_{\text{first}})$ is an *infinite* set. In unrealizability logic, one reasons about infinite sets of programs en masse—as opposed to reasoning about each program in the set separately. Proofs in unrealizability logic must thus establish judgements of the following kind:

> For a given a set of input-output examples, no matter which program is chosen (out of a possibly infinite set of programs), there is at least one input-output example that is handled incorrectly.

The proof system for unrealizability logic has sound underpinnings, and provides a way to build proofs of unrealizability similar to the way Hoare logic [Hoare 1969] provides a way to build proofs that a given program cannot reach a set of bad states.

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

- *Unrealizability logic*, the first logical proof system for overapproximating the execution of an infinite set of imperative programs (§3).
- A proof of soundness and relative completeness for unrealizability logic (§4).

- Examples that illustrate the proving power of unrealizability logic: in particular, unrealizability logic can be used to prove unrealizability for problems out of reach for previous methods; e.g., those for which the proof requires reasoning about an infinite number of examples (§5).

§6 discusses related work. §7 concludes. Proofs for theorems in the paper may be found in the full version of this paper, available on arXiv [Kim et al. 2022].

2 MOTIVATING EXAMPLES

In this section, we give several key examples that describe how proof trees in unrealizability logic work, and also illustrate two key challenges behind unrealizability logic (§2.1, §2.2). Unrealizability logic shares much of the intuition behind Hoare logic and its extension toward recursive programs. However, as we will illustrate in §2.3, these concepts alone are insufficient to model unrealizability, which motivated us to develop the new concepts that we introduce in this paper.

Hoare logic is based on triples that overapproximate the set of states that can be reached by a program s; i.e., the Hoare triple

$$\{P\}\ \mathrm{s}\ \{Q\}$$

asserts that Q is an *overapproximation* of all states that may be reached by executing s, starting from a state in P. The intuition in Hoare logic is that one will often attempt to prove a triple like $\{P\}\ \mathrm{s}\ \{\neg X\}$ for a set of bad states X, which ensures that execution of s cannot reach X.

Unrealizability logic operates on the same overapproximation principle, but differs in two main ways from standard Hoare logic. The differences are motivated by how synthesis problems are typically defined, using two components: (*i*) a search space S (i.e., a set of programs), and (*ii*) a (possibly infinite) set of related input-output pairs $\{(i_1, o_1), (i_2, o_2), \cdots\}$.

To reason about *sets* of programs, in unrealizability logic, the central element (i.e., the program s) is changed to a *set* of programs S. The unrealizability-logic triple

$$\{\!|P|\!\}\ S\ \{\!|Q|\!\}$$

thus asserts that Q is an overapproximation of all states that are reachable by executing *any possible combination* of a pre-state $p \in P$ *and* a program $\mathrm{s} \in S$. More formally, the following theorem holds for any unrealizability triple $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$:

Theorem 2.1. *The* unrealizability triple $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ *for a precondition P, postcondition Q, and set of programs S, holds iff for each program* $\mathrm{s} \in S$*, the Hoare triple* $\{P\}\ \mathrm{s}\ \{Q\}$ *holds.*

The second difference concerns *input-output pairs*: in unrealizability logic, we wish to place the input states in the precondition, and overapproximate the set of states reachable from the input states (through a set of programs) as the postcondition. Unfortunately, the input-output pairs of a synthesis problem cannot be tracked using standard pre- and postconditions; nor can they be tracked using auxiliary variables, because of a complication arising from the fact that unrealizability logic must reason about a *set* of programs (see §2.2).

To keep the input-output relations in check, the predicates of unrealizability logic talk instead about (potentially infinite) *vector-states*, which are sets of states in which each individual state is associated with a unique index. Variable x of the state with index i is referred to as x_i.

Example 2.2 (Vector-States). Assume we are given a set of input-output state pairs $\{(x = i_1, x = o_1), (x = i_2, x = o_2), \cdots, (x = i_n, x = o_n)\}$. Then the input vector-state I is denoted by $(x_1 = i_1) \wedge (x_2 = i_2) \wedge \cdots \wedge (x_n = i_n)$. The output vector-state O is denoted by $(x_1 = o_1) \wedge (x_2 = o_2) \wedge \cdots \wedge (x_n = o_n)$. Once I and O have been constructed this way, the proof steps of an unrealizability-logic proof would relate x_i in the precondition to x_i in the postcondition; one has effectively expressed the relation given by the input-output pairs by *renaming* each input-output pair to be unique.

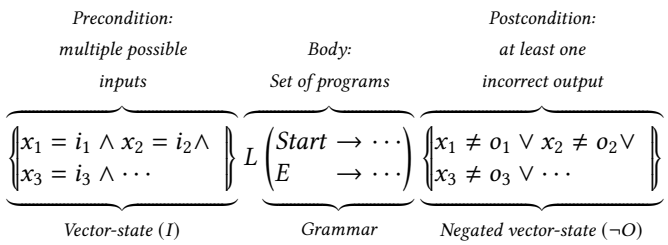

Fig. 1. An unrealizability triple with a negated post-vector-state asserts that a synthesis problem is unrealizable. Observe how the vectorized version of $\neg O$ asserts that there is at *least one* output example that does not satisfy the desired input-output relation.

Keeping these two differences in mind, let us turn to proving unrealizability. Recall that a synthesis problem is given as a search space S and a set of input-output pairs $\mathsf{Ex} = \{(i_1, o_1), \cdots\}$. To prove unrealizability, one must prove that for every program $\mathsf{s} \in S$, there is at least *one* input-output pair $(i_k, o_k) \in \mathsf{Ex}$ such that s does not map i_k to o_k. Note that $\neg O$ denotes the set of all output vector-states for which at least *one* input-output pair does not hold. Then, because of Theorem 2.1, proving $\{I\}\ \mathsf{s}\ \{\neg O\}$ for all $\mathsf{s} \in S$ is equivalent to proving the unrealizability triple

$$\{\!\{I\}\!\}\ S\ \{\!\{\neg O\}\!\}.$$

The goal of unrealizability logic is to prove such triples in a principled logical system, *without* having to descend to the level of individual programs or input-output pairs.

Fig. 1 summarizes what we have discussed so far. In general, in unrealizability logic the input vector-state need not be finite, and the spec need not be functional as well—for example, a synthesis problem that should assign some value $v > x$ to x, for all possible x, can be proved unrealizable via the triple $\{\!\{\forall i.x_i = i\}\!\}\ L(G)\ \{\!\{\exists i.x_i \leq i\}\!\}$.

2.1 Challenge 1: Infinite Sets of Programs

Compared to ordinary Hoare logic, the first challenge is that unrealizability logic needs to reason about infinite sets of programs. Fortunately, the set of programs in a program-synthesis problem is typically formulated as a regular tree grammar (RTG), which defines the set of programs in an inductive manner. Unrealizability logic uses the structure of the RTG to develop proof trees that mimic structural induction.

As illustrated earlier, the unrealizability triple $\{\!\{P\}\!\}\ S\ \{\!\{Q\}\!\}$ captures information about the set of programs S. If S is specified in a recursive manner via an RTG, a triple $\{\!\{P\}\!\}\ S\ \{\!\{Q\}\!\}$ can also be used as an induction hypothesis in a proof tree for unrealizability logic, as we show in Example 2.3.

Example 2.3. Consider a simple synthesis problem sy_e with the following grammar:

$$Start \rightarrow S2 \mid S3 \qquad S2 \rightarrow S2; S2 \mid x := x + 2 \qquad S3 \rightarrow S3; S3 \mid x := x + 3$$

That is, sy_e consists of programs that either (*i*) repeatedly add 2 to x, or (*ii*) repeatedly add 3 to x.

Now suppose that the input specification to sy_e was given as $x \equiv_6 0$ (we use $x \equiv_p r$ as shorthand for $x \equiv r (\mathrm{mod}\ p)$; i.e., that x is equivalent to r modulo p) and the output specification given as $x \equiv_6 1$; that is, the goal is to reach a number of the form $6k' + 1$ when starting from a number $6k$. This problem is unrealizable, because one can only reach $6k'$, $6k' + 2$, $6k' + 3$, or $6k' + 4$ by repeatedly adding 2 or 3 (but not both) to a multiple of 6. We formalize this reasoning as a proof tree in unrealizability logic.

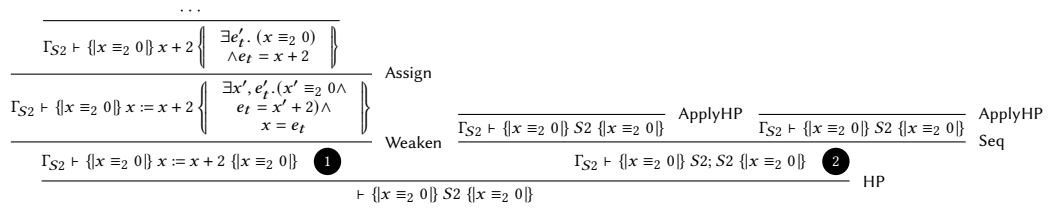

Fig. 2. Simplified proof tree that proves the sub-goal $\{\!|x \equiv_2 0|\!\}\ S2\ \{\!|x \equiv_2 0|\!\}$ in unrealizability logic. Γ_{S2} denotes the context $\{\{\!|x \equiv_2 0|\!\}\ S2\ \{\!|x \equiv_2 0|\!\}\}$. Labels (1) and (2) are names for the triples they are associated with.

Starting the Proof Tree. To prove the synthesis problem sy_e unrealizable, one wishes to prove the following triple, where the output is negated from the specification:

$$\{\!|x \equiv_6 0|\!\}\ \mathit{Start}\ \{\!|x \not\equiv_6 1|\!\}$$

From this point on, a nonterminal as the center element of an unrealizability triple refers to the language of that nonterminal; e.g., $\{\!|x \equiv_6 0|\!\}\ \mathit{Start}\ \{\!|x \not\equiv_6 1|\!\}$ refers to $\{\!|x \equiv_6 0|\!\}\ L(\mathit{Start})\ \{\!|x \not\equiv_6 1|\!\}$.

Note that the postcondition $x \not\equiv_6 1$ is bigger than our previously discussed reachable states of $6k', 6k'+2, 6k'+3$, and $6k'+4$. Hence, the target triple can be proved by proving the following triple and then weakening it (Weaken in Fig. 7):

$$\{\!|x \equiv_6 0|\!\}\ \mathit{Start}\ \{\!|x \equiv_6 0 \vee x \equiv_6 2 \vee x \equiv_6 3 \vee x \equiv_6 4|\!\}$$

To prove this triple in unrealizability logic, we must first introduce the concept of a *context* Γ, which is a set of triples that stores all the induction hypotheses that have been introduced up to some point in a proof tree. Consequently, the judgements of the proof tree have the form:

$$\Gamma \vdash \{\!|\mathcal{P}|\!\}\ S\ \{\!|\mathcal{Q}|\!\}$$

The idea is that instead of reasoning about the provability of triples directly, we wish to reason about the provability of a triple *assuming* that every hypothesis inside Γ is true.

In our case, we wish to prove that $\{\!|x \equiv_6 0|\!\}\ \mathit{Start}\ \{\!|x \equiv_6 0 \vee x \equiv_6 2 \vee x \equiv_6 3 \vee x \equiv_6 4|\!\}$ without assuming anything (i.e., starting from the empty context). This triple can be established by proving that the following judgement holds (where the blank LHS denotes an empty context):

$$\vdash \{\!|x \equiv_6 0|\!\}\ \mathit{Start}\ \{\!|x \equiv_6 0 \vee x \equiv_6 2 \vee x \equiv_6 3 \vee x \equiv_6 4|\!\}$$

The key point in taking the next step is to notice that the language of the nonterminal *Start*, $L(\mathit{Start})$, is the *union* of $L(S2)$ and $L(S3)$. Because unrealizability logic deals with sets of programs, it is equipped with rules for merging triples over different sets of programs. In particular, we can apply the grammar-disjunction rule of unrealizability logic (GrmDisj in Fig. 7), which states that if two program sets satisfy the same pre- and postcondition pair, their union also satisfies the aforementioned pair. In this case, GrmDisj is applied on our target triple as follows:

$$\dfrac{\vdash \{\!|x \equiv_6 0|\!\}\ S2\ \{\!|x \equiv_6 0 \vee x \equiv_6 2 \vee x \equiv_6 3 \vee x \equiv_6 4|\!\} \quad \vdash \{\!|x \equiv_6 0|\!\}\ S3\ \{\!|x \equiv_6 0 \vee x \equiv_6 2 \vee x \equiv_6 3 \vee x \equiv_6 4|\!\}}{\vdash \{\!|x \equiv_6 0|\!\}\ \mathit{Start}\ \{\!|x \equiv_6 0 \vee x \equiv_6 2 \vee x \equiv_6 3 \vee x \equiv_6 4|\!\}}\ \text{GrmDisj}$$

We are now faced with having to prove triples over the nonterminals $S2$ and $S3$. Because $S2$ and $S3$ are defined recursively, taking a naive consideration of the productions from $S2$ and $S3$ will result in an infinite proof tree. To avoid this problem, one must introduce the triples one wishes to prove as *hypotheses* in the context, and validate them in a procedure similar to structural induction.

Introducing Hypotheses. To see how hypotheses are introduced, consider the nonterminal $S2$. The idea is that one wishes to introduce the target triple $\{\!|x \equiv_6 0|\!\}\ S2\ \{\!|x \equiv_6 0 \vee x \equiv_6 2 \vee x \equiv_6 3 \vee x \equiv_6 4|\!\}$ as an *induction hypothesis* about nonterminal $S2$, and prove that this triple holds in a way similar to structural induction. Introducing a new hypothesis can be done using the HP rule in Fig. 7, which splits the proof according to nonterminal N's productions: in this case, nonterminal $S2$ is split into $x := x + 2$ and $S2$; $S2$ (the first application of HP in Fig. 2).

As the hypothesis for $S2$, we introduce the triple $\{\!|x \equiv_2 0|\!\}\ S2\ \{\!|x \equiv_2 0|\!\}$: observe that this triple may be used to prove the target triple $\{\!|x \equiv_6 0|\!\}\ S2\ \{\!|x \equiv_6 0 \vee x \equiv_6 2 \vee x \equiv_6 3 \vee x \equiv_6 4|\!\}$ via an application of Weaken (in fact, $\{\!|x \equiv_6 0|\!\}\ S2\ \{\!|x \equiv_6 0 \vee x \equiv_6 2 \vee x \equiv_6 3 \vee x \equiv_6 4|\!\}$ will not work as a hypothesis directly for $S2$, as we will explain later in this section).

We will use Γ_{S2} to denote the singleton context $\{\{\!|x \equiv_2 0|\!\}\ S2\ \{\!|x \equiv_2 0|\!\}\}$. Fig. 2 depicts the proof tree for $\vdash \{\!|x \equiv_2 0|\!\}\ S2\ \{\!|x \equiv_2 0|\!\}$, where the application of HP at the root of the proof tree introduces the Γ_{S2} context in the two premises.

Proving Hypotheses. If one were proving a property about $S2$ using standard structural induction, one would proceed to show that the property holds on the two sub-cases, $x := x + 2$ and $S2$; $S2$—i.e., the two productions from $S2$—while assuming that the property holds as a hypothesis. The same approach is used here: we assume $\{\!|x \equiv_2 0|\!\}\ S2\ \{\!|x \equiv_2 0|\!\}$ as a hypothesis (in context Γ_{S2}), and attempt to prove that $x \equiv_2 0$ is a valid pre- and postcondition for both $x := x + 2$ and $S2$; $S2$.

First, consider proof goal (1) for $x := x + 2$. The set of programs generated by the right-hand side of this production is completely independent of how $S2$ is defined; thus, for the purpose of performing structural induction on $S2$, it is a *base case*. Here, we invoke the basic rule for assignment to prove (1). Auxiliary variable e_t stores the result of the expression $x + 2$, which is then assigned to x through the Assign rule (in which the postcondition of the conclusion mimics that of the forwards-based assignment rule in Hoare logic). The resulting postcondition can then be weakened to obtain $x \equiv_2 0$, which completes the proof that $\{\!|x \equiv_2 0|\!\}\ x := x + 2\ \{\!|x \equiv_2 0|\!\}$.

Next, consider proof goal (2) for $S2$; $S2$. Here, nonterminal $S2$ appears directly, and provides us with a chance to apply the induction hypothesis via the rule ApplyHP in Fig. 7. In Figure 2, the derivation for (2) is the sub-proof tree on the right. Notice how $S2$; $S2$ is first decomposed using the Seq rule for sequential composition (identical to that in Hoare logic), which requires us to prove two instances of the proof goal (2); in turn, these two instances are proved by directly applying the induction hypothesis through ApplyHP.

Returning to nonterminal *Start*, the second premise $\{\!|x \equiv_6 0|\!\}\ S3\ \{\!|x \equiv_6 0 \vee x \equiv_6 2 \vee x \equiv_6 3 \vee x \equiv_6 4|\!\}$ can be similarly proved by introducing the triple $\{\!|x \equiv_3 0|\!\}\ S3\ \{\!|x \equiv_3 0|\!\}$ as a hypothesis for nonterminal $S3$, and weakening it to get the target triple about $S3$. This step concludes our proof.

One take-away from Example 2.3 is the importance of choosing an appropriate induction hypothesis. For instance, attempting to use the proof goal $\{\!|x \equiv_6 0|\!\}\ S2\ \{\!|x \equiv_6 0 \vee x \equiv_6 2 \vee x \equiv_6 3 \vee x \equiv_6 4|\!\}$ directly as a hypothesis for $S2$ would make the proof fail on the production $S2$; $S2$. This example indicates that, similar to how identifying appropriate invariants for loops is a key component of writing Hoare logic proofs, identifying appropriate hypotheses for nonterminals is an essential part in completing a proof in unrealizability logic.

2.2 Challenge 2: Tracking (Infinite) Input-Output Relations

The second challenge in unrealizability logic is that the specification of a synthesis problem is typically given as a set of input-output *pairs*: a specific input value is associated with a specific output value. (Even if the specification is given as a universally quantified formula, one can understand such a formula as an infinite set of input-output pairs.) The standard way to address this problem in Hoare logic is to introduce auxiliary variables that freeze the values of program variables in the

precondition, and to allow the postcondition to refer to these auxiliary variables. However, this approach alone is unsuitable for unrealizability logic, as illustrated by the following example.

Example 2.4. Consider the identity assignment $x := x$. The Hoare triple for this program is typically given as $\{x = x_{aux}\}\ x := x\ \{x = x_{aux}\}$; that is, starting from $x = x_{aux}$, where x_{aux} is an *auxiliary variable*, one ends up in $x = x_{aux}$. Strictly speaking, the auxiliary variable x_{aux} is quantified *outside* the Hoare triple: $\forall x_{aux}.\{x = x_{aux}\}\, x := x\, \{x = x_{aux}\}$; this position for the quantifier indicates that the triple should hold for *every* value of x_{aux}.

Now suppose that we are given a synthesis problem sy_{id}, where the goal is to synthesize a program equivalent to $x := x$, using the grammar G_{id} below:

$$Start \to x := E \qquad E \to 0 \mid E + 1 \mid E - 1$$

For every integer i, the set $L(E)$ contains a constant expression that evaluates to i, and thus $L(Start)$ consists of exactly the set of all constant assignments. However, synthesizing a statement that is computationally equivalent to $x := x$ is *impossible* because the set only contains constant assignments—i.e., sy_{id} is unrealizable.

Suppose that one tries to specify sy_{id} using an auxiliary variable, so that the goal is to prove $\{\!|x = x_{aux}|\!\}\ Start\ \{\!|x \neq x_{aux}|\!\}$. Unfortunately, this triple is *invalid* in unrealizability logic, in the sense that starting from the precondition $x = x_{aux}$, one can actually reach a state where still, $x = x_{aux}$!

To understand this seemingly counterintuitive fact, we will attempt to prove the triple $\{\!|x = x_{aux}|\!\}\ Start\ \{\!|\exists k.x = k|\!\}$ (where k indicates that x may have any value in the post-state). To do so, let us first characterize the behavior of all programs in the language of E. In doing so, one will generate the following hypothesis (where, again, e_t is an auxiliary variable for storing the value obtained from executing E):

$$\{\!|x = x_{aux}|\!\}\ E\ \{\!|x = x_{aux} \wedge \exists k.e_t = k|\!\}.$$

The best one can say about terms in E is that there exists an integer k whose value is e_t. Applying Assign and Weaken to the triple, one can derive the following (precise) triple:

$$\{\!|x = x_{aux}|\!\}\ Start\ \{\!|\exists k.x = k|\!\}.$$

Like in Hoare logic, the quantification for x_{aux} is *outside* the unrealizability triple, which yields:

$$\forall x_{aux}.\{\!|x = x_{aux}|\!\}\ Start\ \{\!|\exists k.x = k|\!\}.$$

This triple says that for every value of x_{aux}, there is *some* k (and a corresponding program) that works; thus, this triple actually asserts that the problem is realizable (which is clearly not true)!

The problem is that, while for every *individual* value of x_{aux} there does indeed exist a suitable k (corresponding to a specific expression $t_k \in L(E)$ that always evaluates to k), any such value of k (and expression t_k) fails to work for other values of x_{aux}—i.e., the use of a single auxiliary variable fails to capture the fact that the multiple input-output pairs must all be satisfied *simultaneously*. In terms of synthesis, for each input, some program in the search space will work: but this does not guarantee the existence of a program that works for *all* inputs (such a program does not exist).

The issue illustrated in Example 2.4—where one must ensure that *each* input must map to some *specific* output, but all via the *same* program—has appeared in other work on unrealizability. For example, Nay [Hu et al. 2020] uses semi-linear sets over LIA to capture this relation (albeit for a specific class of synthesis problems), while Nope [Hu et al. 2019] constructs a program that executes each example in lockstep.

In this paper, we show that these relations can be elegantly expressed as unrealizability triples by (*i*) merely renaming the set of variables to which each example refers to be unique; (*ii*) conjoining all the renamed states into a single, big state, and (*iii*) modifying the semantics of programs to execute

over the renamed variables. As discussed in Example 2.2, we refer to the renamed, conjoined big states as *vector-states* because the renaming and conjoining process can be intuitively understood as vectorizing the input states, and then executing the program on the single vector. As shown in Example 2.4, and as we will later show in §5, having a simple logical representation allows us to extend the vector-states toward infinite examples, which allows us to prove unrealizability for problems beyond the reach of previous work [Hu et al. 2019, 2020; Kim et al. 2021].

Example 2.5 (Two Input-Output Pairs). Consider again the synthesis problem sy_{id} and grammar G_{id} from Example 2.4. Suppose that the specification for sy_{id} is given as a set of two input-output pairs: $\{[x = 1 \mapsto x = 1], [x = 2 \mapsto x = 2]\}$. Then the input and output specifications can both be expressed as the vector-state $\{x_1 = 1 \wedge x_2 = 2\}$, where x_1 corresponds to the first example and x_2 corresponds to the second example. The following triple is valid in unrealizability logic, i.e., the two examples suffice to demonstrate that sy_{id} is unrealizable:

$$\{\!|x_1 = 1 \wedge x_2 = 2|\!\}\ L(Start)\ \{\!|x_1 \neq 1 \vee x_2 \neq 2|\!\}.$$

In turn, this triple may be derived from the triple:

$$\{\!|x_1 = 1 \wedge x_2 = 2|\!\}\ L(Start)\ \{\!|\exists k.x_1 = k \wedge x_2 = k|\!\},$$

which states that there must exist a k for which both x_1 and x_2 are equivalent to k in the post-state (i.e., that $L(Start)$ is a constant program). This condition implies that x_1 and x_2 are identical, and thus implies that $x_1 \neq 1 \vee x_2 \neq 2$.

Example 2.5 illustrates how unrealizability can be proved using two examples; in §5, we present problems for which proving unrealizability requires infinitely many examples.

The fact that the input-output examples can be packed into a *single* vector-state is important: because the starting precondition contains only a single vector-state, *all* examples inside the single vector-state are guaranteed to be executed on the same program in the grammar. Likewise, because all the output examples are packed into a single vector-state, the negation of this vector-state is guaranteed to contain all vector-states that are wrong on *at least one* input example.

2.3 Why Not Recursive Hoare Logic?

At this point, readers familiar with Hoare logic extended toward recursive procedures [Apt 1981; Nipkow 2002] may notice that the proof structure described in §2.1 is similar to proofs in recursive Hoare logic. In fact, the similarity between program-synthesis problems and nondeterministic, recursive procedures has already been exploited in Nope [Hu et al. 2019], which constructs a recursive program from a synthesis problem, then relies on an external verifier to check whether the problem is unrealizable.

Then what is the problem with applying recursive Hoare logic to programs translated in this way? The answer is that some features of synthesis problems are difficult to model as a nondeterministic recursive program—e.g., loops and infinite examples (both of which Nope cannot support).

With while loops, the problem is that a nondeterministic modelling as described in [Hu et al. 2019] must have a way of *recording* the unbounded choices that were selected when synthesizing the while-loop body, due to the fact that a loop body must stay *fixed* throughout multiple iterations.

Example 2.6 (Multiple Loop Bodies). Consider a variant of sy_{e} from Example 2.3, where the grammar has been modified to contain while loops instead of sequential composition:

$$Start \to \text{while } B \text{ do } S \qquad B \to x < 100 \qquad S \to x := x + 2 \mid x := x + 3$$

sy_{e} still consists of programs that repeatedly add either 2 or 3 to x. However, the repeated addition is performed within a loop (until $x \geq 100$). The search space consists of exactly two programs: one

in which the loop body is $x := x + 2$, and one in which the loop body is $x := x + 3$. Starting from $x \equiv_6 0$, $x \equiv_2 0$ is an invariant of the first program, and $x \equiv_3 0$ is an invariant of the second program.

Nope [Hu et al. 2019] constructs a nondeterministic, recursive program from a synthesis problem, where each nonterminal is translated into a procedure. The basic idea is that executing the translated procedure returns the result of executing that nonterminal—where the nondeterministic choices of an RTG are mimicked by nondeterminsitic choices in the procedure.

```
1  int x, e_t, b_t; // Vars store results of execution
2  void Start() { While(); }
3  void B() { b_t = x < 100; }
4  void S() {
5    int select = nondet(); // Nondeterminsitic choice
6    if (select = 1) x = x + 2; else x = x + 3;
7  }
8  void While() {
9    B();
10   if (b_t) S(); While(); else skip;
11 }
```

Fig. 3. Encoding of sy_e as a nondeterministic program.

A naive translation of sy_e into a nondeterminsitic program following this idea would result in a program like the one in Figure 3. The problem with Figure 3 is that there is no machinery present to ensure that the *same* loop body is repeated for each iteration—thus, an analysis of Figure 3 would report that states in which, e.g., $x \equiv_6 1$ holds are possible! To address this issue, Figure 3 would require some data structure capable of recording the `nondet()` choices in `S()` directly embedded in the program. While a single Int may suffice for Figure 3, as the loop body is decided via a single nondeterministic choice, other synthesis problems in general would require data structures such as lists, as constructing their loop bodies may require the application of multiple productions. In particular, such a list must be of *unbounded* size, because, in general, a term within an RTG may be of unbounded size.

In unrealizability logic, one does not need such complex machinery—instead, one can perform simple invariant-based reasoning. Because unrealizability logic is tailored for synthesis problems, one can first split the search space through GrmDisj, as illustrated in Example 2.3, to consider the two loops "while B do $x := x + 2$" and "while B do $x := x + 3$" separately. At this point, it becomes clear that $\{\!|x \equiv_2 0|\!\}\, x := x + 2 \,\{\!|x \equiv_2 0|\!\}$ and $\{\!|x \equiv_3 0|\!\}\, x := x + 3 \,\{\!|x \equiv_3 0|\!\}$ are invariants for each of the loops. An additional application of Weaken yields the desired triple $\{\!|x \equiv_6 0|\!\}\, Start \,\{\!|x \equiv_2 0 \vee x \equiv_3 0|\!\}$. (While it is possible to reason about both loops via a single invariant, as suggested in §3.3, such an invariant is likely to be too complex; see §3.3 and Lemma 4.3 for details.)

The second problem with trying to apply recursive Hoare logic has to do with dealing with infinitely many examples, or more generally, with vector-states themselves: vector-states provide unrealizability logic a way to ensure that multiple examples are executed along the same program. Nope [Hu et al. 2019] creates copies of sets of variables for each different example when translating the synthesis problem into a program. However, creating copies of variables for each example requires an *infinite* number of variables in the program to support infinite examples.[1] Programming languages, much less program verifiers, typically do not support—or struggle to support—programs with infinite variables, which is why previous attempts such as Nope fail to support cases with infinite examples.

It is true that, if one is willing to develop a logic supporting all necessary features: recursion, nondeterminism, both local and global variables, infinite data structures, and infinite vector-states, then it would be possible to prove unrealizability via an extended Hoare logic. However, to the best of our knowledge, there is no comprehensive study of a system containing *all* of these features at once, whereas this paper proves soundness, relative completeness, and some other decidability results related to unrealizability and unrealizability logic.

[1] Unlike the case with while loops, where an *unbounded* list suffices (e.g., an OCaml list), infinite examples require the use of a truly infinite number of variables.

$$
\begin{array}{llll}
\textit{Stmt} & S & ::= & x := E \mid S; S \mid \text{if } B \text{ then } S \text{ else } S \mid \text{while } B \text{ do } S \\
\textit{IntExpr} & E & ::= & 0 \mid 1 \mid x \mid E + E \mid E - E \mid E \cdot E \mid E/E \mid \text{if } B \text{ then } E \text{ else } E \\
\textit{BoolExpr} & B & ::= & \mathsf{t} \mid \mathsf{f} \mid \neg B \mid B \wedge B \mid E < E \mid E == E
\end{array}
$$

Fig. 4. The base grammar G_{imp}, which defines a universe of terms that we are interested in for this paper.

Moreover, such a logical system would be *more* complex than required to reason about unrealizability—for example, a record-and-replay encoding of while loops as suggested in Example 2.6 completely hides the fact that one is processing a while-loop production, and thus *prevents* simple invariant-based reasoning about loops. Such drawbacks are in direct conflict with the goal of unrealizability logic, which—as stated in §1—is to provide a simple logical system that distills the essence of proving unrealizability. True to this goal, the While rule for loops in unrealizability logic does allow one to perform invariant-based reasoning for loops, and provides similar direct reasoning principles for other constructs as well.

3 UNREALIZABILITY LOGIC

In this section, we formally define unrealizability logic and some necessary preliminaries.

3.1 Preliminary Definitions

In this paper, we consider synthesis problems where the search space is a subset of all terms producible by the imperative grammar G_{imp} (Figure 4)—that is, deterministic, expression-based or imperative synthesis problems defined over integer arithmetic expressions. We assume a function $[\![\cdot]\!]$ that defines the standard semantics of every term $t \in L(G)$. We leave supporting synthesis problems involving more complex operations, e.g., recursive data structures, to future work; this paper lays the foundations for making such extensions possible.

Definition 3.1 (Regular Tree Grammar). A *(typed) regular tree grammar* (RTG) is a tuple $G = (\mathcal{N}, \mathcal{A}, S, a, \delta)$, where $\mathcal{N}$ is a finite set of nonterminals; $\mathcal{A}$ is a ranked alphabet; $S \in \mathcal{N}$ is an initial nonterminal; a is a type assignment that gives types for members of $\mathcal{A} \cup \mathcal{N}$; and δ is a finite set of productions of the form $N_0 \to \alpha^{(i)}(N_1, ..., N_i)$, where for $0 \le j \le i$, each $N_j \in \mathcal{N}$ is a nonterminal such that if $a_{\alpha^{(i)}} = (\tau_0, \tau_1, ..., \tau_i)$ then $a_{N_j} = \tau_j$.

In our setting, the alphabet consists of constructors for each of the constructs of G_{imp} (although for simplicity we write, e.g., $x := 0$, rather than $:=^{(2)}(x^{(0)}, 0^{(0)})$). G_{imp} contains three types of nonterminals: statement nonterminals, expression nonterminals, and Boolean-expression nonterminals.

We assume grammars use productions that differ from the ones in G_{imp} only due to the nonterminal names. We say that a production $N_0 ::= \alpha^{(i)}(N_1, \ldots, N_i)$ is *valid with respect to* G_{imp} iff replacing each nonterminal N_j with the nonterminal of the same type in the set $\{S, E, B\}$ yields a production in G_{imp}, e.g., production $S_1 ::= S_2; S_3$ is valid with respect to G_{imp}.[2]

Definition 3.2 (Synthesis Problem). A synthesis problem is a 4-tuple $sy = \langle G, f, I, \psi \rangle$, where
- G is a regular-tree grammar consisting of productions that are valid with respect to G_{imp},
- f is the name of the function to synthesize,
- I is the set of allowed input states of f (i.e., the domain of f),
- ψ is a Boolean formula that describes a behavioral specification that f must satisfy.

The goal of a synthesis problem is to find $f^* \in L(G)$ such that $\forall r \in I.\ \psi([\![f^*]\!](r), r)$. If such an $f^* \in L(G)$ exists, synthesis problem sy is *realizable*; otherwise, sy is *unrealizable*.

[2]In this paper, we sometimes write a production in which some terminals are in-lined in place of nonterminals—e.g., we write $S ::= x := x$ in place of the two productions $S ::= x := E_x, E_x ::= x$.

In this paper, f may be thought as of a state transformer that has type of State $\rightarrow$ State.

A key point from Definition 3.2 is that the search space is defined as an RTG. As noted in §2.1, this property is used by unrealizability logic to build proof trees that mimic structural induction. Also, note that the specification ψ is given as a Boolean formula, i.e., a logical specification. So far in the paper, we have only considered synthesis problems where the specification is given as a set of input-output examples, which are subsumed by logical specifications. However, unrealizability logic is also capable of dealing with logical specifications directly, because the pre- and postcondition of an unrealizability triple are logical predicates.

3.2 Basic Definitions for Unrealizability Logic

We now introduce some concepts that are specific to unrealizability logic.

In this paper, we will use (and have used) the following notation for different kinds of states:

- Lowercase letters (e.g., p, q, r) for ordinary states,
- Uppercase letters (e.g., P, Q) for sets of ordinary states (and also predicates over states),
- Lowercase Greek letters (e.g., σ, π) for vector-states (as shown in §2, defined in Definition 3.4),
- Calligrahic uppercase letters (e.g., $\mathcal{P}, \mathcal{Q}$) for sets of (and predicates describing) vector-states.

Sometimes, uppercase letters are used also to denote nonterminals, i.e., sets of programs, and lowercase letters used to denote single programs. It should be clear from context, in these cases, what the upper- and lowercase letters denote.

In unrealizability logic, the semantics of expressions are extended such that an expression does not evaluate to a value v, but evaluates to a *state* that stores the value v inside a (reserved) auxiliary variable e_t (or b_t, for Boolean expressions). This non-standard approach is taken because, as we will see in §3.3, simply generating values will lose some information that is required to precisely compute the set of states that a nonterminal can generate.

Definition 3.3 (Semantics of Expressions). Let $[\![\cdot]\!]_{\text{ext}}$ be a semantics for expressions that evaluates an expression to a state instead of a value. Intuitively, we will store the value resulting from computing an expression to a reserved variable e_t in the state.

Given a 'standard' semantics $[\![\cdot]\!]$ that evaluates expressions to values and an arbitrary state r, let $[\![e]\!]_{\text{ext}}(r) = r[e_t \mapsto [\![e]\!](r)]$ if e is an atomic expression (0, 1, or x). (If e is an atomic Boolean expression, the assignment is to b_t instead.)

If e is a n-ary operation, with $e = e_1 + e_2$ as an example, let $[\![e_1 + e_2]\!]_{\text{ext}}(r) = r[e_t \mapsto [\![e_1]\!]_{\text{ext}}(r)[e_t] + [\![e_2]\!]_{\text{ext}}(r)[e_t]]$; that is, $[\![\cdot]\!]_{\text{ext}}$ is defined recursively such that the extended semantics of operators such as +, i.e., $[\![+]\!]_{\text{ext}}$, are state transformers as well. These semantics work by referencing the value of e_t stored in the state that each sub-expression e_1 and e_2 evaluates to (likewise, if e is a n-ary Boolean operation, the final assignment is to b_t instead; Boolean sub-expressions will also reference b_t instead of e_t).

We define the extended semantics recursively, instead of simply stating that $[\![e]\!]_{\text{ext}}(r) = r[e_t \mapsto [\![e]\!](r)]$, because we wish the rules of unrealizability logic to be recursive, and thus need to define operations such as addition on states. In the rest of this paper, we will take $[\![\cdot]\!]_{\text{ext}}$ from Definition 3.3 to be the standard semantics given to a term, and thus drop the subscript ext.

In this paper, we consider a state to be a finite mapping from variables to values, which can also be understood as a finite set of ⟨variable, value⟩ pairs (where the variables are unique).

As we showed in §2, unrealizability logic relies on *vector-states* and a semantics modified to run on vector-states to capture input-output relations. Vector-states may simply be understood as ordinary states defined over an extended set of variables; we formalize this notion below.

Definition 3.4 (Vector-State). Let $r_1, r_2, \cdots$ be a finite or countably infinite collection of states indexed by the natural numbers, that are defined over the same set of variables V. Then the vector-state σ is a state defined over the set of variables $\bigcup_{i \in D}\{v_i \mid v \in V\}$, such that for all $v \in V$ and $i \in D$, we have that $\sigma(v_i) = r_i(v)$.

Intuitively, given a finite collection of states $r_1, \cdots, r_n$, the vector-state $\sigma = \langle r_1[v_1/v], \cdots, r_n[v_n/v]\rangle$ is simply a 'vectorization' of the renamed individual states. Executing a program t on this vector-state produces a vector-state equivalent to the vector obtained by running each individual state through t separately: that is, $[\![t]\!](\sigma) = \langle [\![t]\!](r_1)[v_1/v], \cdots, [\![t]\!](r_n)[v_n/v]\rangle$. This intuition carries over to when the given collection of states is infinite, in which case the resulting vector-state will simply be of infinite length. Def. 3.5 formalizes this intuition, and extends the program semantics to a semantics that works on possibly infinite vector-states, and also to a semantics of a set of programs on a set of possibly infinite vector-states.

Definition 3.5 (Vector-State Semantics). Let $\sigma = \langle r_1[v_1/v], r_2[v_2/v], \cdots\rangle$ be a vector-state indexed by the natural numbers, where $r_1, r_2, \cdots$ are defined over variables $v \in V$.

Then the semantics of a term t on σ is defined as $[\![t]\!](\sigma) = \bigcup_{i \in N, v \in V}\{v_i \mid \sigma(v_i) = [\![t]\!](r_i)(v)\}$. The semantics of t on a *set* of vector-states $\mathcal{P}$ is defined as $[\![t]\!](\mathcal{P}) = \bigcup_{\sigma \in \mathcal{P}}\{[\![t]\!](\sigma)\}$. The semantics of a *set* of programs T on a *set* of vector-states $\mathcal{P}$ is defined as $[\![T]\!](\mathcal{P}) = \bigcup_{t \in T}[\![t]\!](\mathcal{P})$.

The semantics of a set of programs on a set of input states can be understood as producing the set of all states that may arise from taking *any* combination of a program $t \in T$ and a (vector-) state $\sigma \in \mathcal{P}$. This semantics allows us to define the validity of unrealizability triples: a triple $\{\!\{\mathcal{P}\}\!\}\ T\ \{\!\{\mathcal{Q}\}\!\}$ is valid iff $\mathcal{Q}$ overapproximates the states that may result from any combination of $\sigma \in \mathcal{P}$ and $t \in T$.

Definition 3.6 (Validity). Given a precondition over vector-states $\mathcal{P}$, and a postcondition over vector-states $\mathcal{Q}$, an unrealizability triple $\{\!\{\mathcal{P}\}\!\}\ T\ \{\!\{\mathcal{Q}\}\!\}$ is *valid* for a set of programs T, denoted by $\models \{\!\{\mathcal{P}\}\!\}\ T\ \{\!\{\mathcal{Q}\}\!\}$, iff $[\![T]\!](\mathcal{P}) \subseteq \mathcal{Q}$.

As previously stated as Theorem 2.1, $\models \{\!\{\mathcal{P}\}\!\}\ T\ \{\!\{\mathcal{Q}\}\!\}$ iff for all $t \in T$, $\{\mathcal{P}\}$ s $\{\mathcal{Q}\}$ is a valid Hoare triple. Connecting all the definitions, we state that:

A synthesis problem $sy = \langle G, f, I, \psi\rangle$ is unrealizable if $\models \{\!\{I(r)\}\!\}\ N\ \{\!\{\neg\psi(f(r), r)\}\!\}$ is a valid unrealizability triple. Here, N is the initial nonterminal of G, and I and ψ are predicates that describe the vectorized input and output conditions of the synthesis problem.

3.3 The Rules of Unrealizability Logic

In this section, we present the inference rules of unrealizability logic.

The Assertion Language. One issue to discuss before presenting the rules is the choice of *assertion language*—that is, the language used for the pre- and postconditions of unrealizability triples. Unrealizability logic is parametric in the choice of assertion language, as long as the assertion language is as least as expressive as first-order Peano arithmetic (FO-PA). This requirement is due to the fact that the addtional predicates that unrealizability logic adds to the given pre- and postconditions when constructing a proof tree require FO-PA—an assertion language less expressive than FO-PA will fail to encode these additional predicates.

This characteristic makes FO-PA a natural choice for the assertion language in unrealizability-logic proofs, especially as it is well-studied and provides opportunities for automation. However, some synthesis problems will require a stronger assertion language than FO-PA for a proof of unrealizability to be completed in unrealizability logic. For example, proofs that require an infinite number of examples will require a logic capable of supporting an infinite number of variables (such an example is given in §5.1). One example of such a logic is FO-PA extended with infinite arrays.

$$\frac{}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ 0\ \{\!|\exists \mathbf{e}'_\mathbf{t}.\mathcal{P}[\mathbf{e}'_\mathbf{t}/\mathbf{e}_\mathbf{t}] \wedge \mathbf{e}_\mathbf{t} = \mathbf{0}|\!\}}\ \text{Zero} \qquad \frac{}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ 1\ \{\!|\exists \mathbf{e}'_\mathbf{t}.\mathcal{P}[\mathbf{e}'_\mathbf{t}/\mathbf{e}_\mathbf{t}] \wedge \mathbf{e}_\mathbf{t} = \mathbf{1}|\!\}}\ \text{One}$$

$$\frac{}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ \mathrm{t}\ \{\!|\exists \mathbf{b}'_\mathbf{t}.\mathcal{P}[\mathbf{b}'_\mathbf{t}/\mathbf{b}_\mathbf{t}] \wedge \mathbf{b}_\mathbf{t} = \mathrm{t}|\!\}}\ \text{True} \qquad \frac{}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ \mathrm{f}\ \{\!|\exists \mathbf{b}'_\mathbf{t}.\mathcal{P}[\mathbf{b}'_\mathbf{t}/\mathbf{b}_\mathbf{t}] \wedge \mathbf{b}_\mathbf{t} = \mathrm{f}|\!\}}\ \text{False}$$

$$\frac{}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ x\ \{\!|\exists \mathbf{e}'_\mathbf{t}.\mathcal{P}[\mathbf{e}'_\mathbf{t}/\mathbf{e}_\mathbf{t}] \wedge \mathbf{e}_\mathbf{t} = \mathbf{x}|\!\}}\ \text{Var} \qquad \frac{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ B\ \{\!|Q|\!\}}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ !B\ \{\!|\exists \mathbf{b}'_\mathbf{t}.Q[\mathbf{b}'_\mathbf{t}/\mathbf{b}_\mathbf{t}] \wedge \mathbf{b}_\mathbf{t} = \neg \mathbf{b}'_\mathbf{t}|\!\}}\ \text{Not}$$

$$\frac{\begin{array}{ll}\Gamma \vdash \{\!|\mathcal{P} \wedge (\mathbf{v}_1 = \mathbf{v})|\!\}\ E_1\ \{\!|Q_1|\!\} & \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}'_1, \mathbf{v}'_2 \text{ fresh renames of } \mathbf{v} \\ \Gamma \vdash \{\!|\mathcal{P} \wedge (\mathbf{v}_2 = \mathbf{v})|\!\}\ E_2\ \{\!|Q_2|\!\} & \mathbf{v} \text{ refers to full set of vars in } \mathcal{P}\end{array}}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ (E_1 \oplus E_2)\ \{\!|\exists \mathbf{e}'_\mathbf{t}, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}'_1, \mathbf{v}'_2.(\mathcal{P} \wedge Q_1[\mathbf{v}'_1/\mathbf{v}] \wedge Q_2[\mathbf{v}'_2/\mathbf{v}] \wedge (\mathbf{v} = \mathbf{v}_1 = \mathbf{v}_2))[\mathbf{e}'_\mathbf{t}/\mathbf{e}_\mathbf{t}] \wedge \mathbf{e}_\mathbf{t} = \mathbf{e}'_{\mathbf{t}_1} \oplus \mathbf{e}'_{\mathbf{t}_2}|\!\}}\ \text{Bin}$$

$$\frac{\begin{array}{ll}\Gamma \vdash \{\!|\mathcal{P} \wedge (\mathbf{v}_1 = \mathbf{v})|\!\}\ E_1\ \{\!|Q_1|\!\} & \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}'_1, \mathbf{v}'_2 \text{ fresh renames of } \mathbf{v} \\ \Gamma \vdash \{\!|\mathcal{P} \wedge (\mathbf{v}_2 = \mathbf{v})|\!\}\ E_2\ \{\!|Q_2|\!\} & \mathbf{v} \text{ refers to full set of vars in } \mathcal{P}\end{array}}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ (E_1 \odot E_2)\ \{\!|\exists \mathbf{b}'_\mathbf{t}, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}'_1, \mathbf{v}'_2.(\mathcal{P} \wedge Q_1[\mathbf{v}'_1/\mathbf{v}] \wedge Q_2[\mathbf{v}'_2/\mathbf{v}] \wedge (\mathbf{v} = \mathbf{v}_1 = \mathbf{v}_2))[\mathbf{b}'_\mathbf{t}/\mathbf{b}_\mathbf{t}] \wedge \mathbf{b}_\mathbf{t} = \mathbf{e}'_{\mathbf{t}_1} \odot \mathbf{e}'_{\mathbf{t}_2}|\!\}}\ \text{Comp}$$

$$\frac{\begin{array}{ll}\Gamma \vdash \{\!|\mathcal{P} \wedge (\mathbf{v}_1 = \mathbf{v})|\!\}\ B_1\ \{\!|Q_1|\!\} & \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}'_1, \mathbf{v}'_2 \text{ fresh renames of } \mathbf{v} \\ \Gamma \vdash \{\!|\mathcal{P} \wedge (\mathbf{v}_2 = \mathbf{v})|\!\}\ B_2\ \{\!|Q_2|\!\} & \mathbf{v} \text{ refers to full set of vars in } \mathcal{P}\end{array}}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ (B_1 \wedge B_2)\ \{\!|\exists \mathbf{b}'_\mathbf{t}, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}'_1, \mathbf{v}'_2.(\mathcal{P} \wedge Q_1[\mathbf{v}'_1/\mathbf{v}] \wedge Q_2[\mathbf{v}'_2/\mathbf{v}] \wedge (\mathbf{v} = \mathbf{v}_1 = \mathbf{v}_2))[\mathbf{b}'_\mathbf{t}/\mathbf{b}_\mathbf{t}] \wedge \mathbf{b}_\mathbf{t} = \mathbf{b}'_{\mathbf{t}_1} \wedge \mathbf{b}'_{\mathbf{t}_2}|\!\}}\ \text{And}$$

Fig. 5. Inference rules for expressions in unrealizability logic. Bin represents rules for binary expressions, where the operator $\oplus$ is one of + (Plus), − (Minus), · (Mult), or / (Div). Comp represents rules for binary comparators, where the operator $\odot$ is one of < (LT) or == (Eq).

Example 3.7 (FO-PA with Infinite Arrays). Consider the assertion language of FO-PA, extended with axioms from the first-order theory of arrays [McCarthy 1993]. Variables may be arrays ranging over an infinite range of indices (indexed by the natural numbers), and $\mathbf{x}[i]$ indicates reading the array $\mathbf{x}$ at index i.

Then one can encode vector-states of infinite length by introducing an array for each variable: the array $\mathbf{x}$ encodes an infinite-length vector-variable $\langle x_1, x_2, \cdots \rangle$, where $\mathbf{x}[i]$ corresponds to the variable x_i. For example, $\forall i.\mathbf{x}[i] = i$ encodes the condition that $x_i = i$, i.e., that the value of x in the i-th example is i (we will use the two notations x_i and $\mathbf{x}[i]$ interchangably in the rest of this paper).

As stated above, unrealizability logic is parametric in the choice of assertion language; if one wishes to write proofs that necessitate the use of predicates that cannot be expressed in FO-PA, it is entirely possible for one to use an assertion language capable of expressing the required predicates. The examples used in this paper rely on two assertion languages: standard FO-PA for proofs that do not require infinite examples, and FO-PA extended with infinite arrays for cases that do require an infinite number of examples to prove unrealizability (Example 5.1).

The goal of unrealizability logic is to prove unrealizability for synthesis problems defined over an RTG. Therefore, the rules that we state in Figures 5, 6, and 7 are for sets of programs defined as $L(N)$ for some RTG nonterminal N, or as the language of the right-hand side of an RTG production.

3.3.1 Rules for Expressions. Recall that in Definition 3.3 we defined the semantics of expressions to update the value of a reserved auxiliary variable e_t instead of generating a value. The rules for expressions reflect this fact: in every expression rule, the postcondition of the conclusion takes the form $\exists \mathbf{e}'_\mathbf{t}.\mathcal{P}[\mathbf{e}'_\mathbf{t}/\mathbf{e}_\mathbf{t}] \wedge \mathbf{e}_\mathbf{t} = \mathbf{e}$, where $\mathcal{P}$ is some predicate and $\mathbf{e}$ is a vector that encodes the values that may be generated by an expression in the set of expressions considered. Observe that the form of this predicate mimics the postcondition in the forwards-assignment rule of Hoare logic [Floyd 1993] applied to $\mathbf{e}_\mathbf{t} = \mathbf{e}$ (e.g., $\mathbf{e}_\mathbf{t} = \mathbf{0}$ in the case of Zero): the assignment is to a vector-variable because we are working with vector-states (as opposed to single states, as in Definition 3.3).

The intuition that $\mathbf{e_t}$ contains the value generated by the expression nonterminals can be formalized into an invariant about expression nonterminals:

Lemma 3.8. *Given an expression nonterminal E, if an unrealizability triple $\Gamma \vdash \{\!|\mathcal{P}|\!\}\ E\ \{\!|\mathcal{Q}|\!\}$ is derivable using the rules of unrealizability logic under some context Γ, then for every $\mathrm{e} \in L(E)$ and for every $\sigma \in \mathcal{P}$, the formula $\mathcal{Q}[[\![\mathrm{e}]\!](\sigma)/\mathbf{e_t}]$ is true assuming all triples in Γ are true (where in this case, $[\![\mathrm{e}]\!](\sigma)$ refers to the standard semantics that evaluates e to a value).*

Lemma 3.8 shows that, $\mathcal{Q}$ at the very least contains states where $\mathbf{e_t} = [\![\mathrm{e}]\!](\sigma)$: the variable $\mathbf{e_t}$ overapproximates the set of values obtainable by executing an expression $\mathrm{e} \in L(E)$. A similar lemma also holds for Boolean-expression nonterminals, where the variable $\mathbf{e_t}$ is replaced with $\mathbf{b_t}$.

Lemma 3.8 is important, because it allows the postcondition of the conclusion to refer to the values computed by expression nonterminals through use of the variable $\mathbf{e_t}$ (or $\mathbf{b_t}$). In particular, we assume that $\mathbf{e_t}$ and $\mathbf{b_t}$ are *reserved* for the purposes of storing these values. (More precisely, we assume that $\mathbf{e_t}$ and $\mathbf{b_t}$ are included in the set of program variables $vars(N)$, and thus rules such as Sub1 cannot substitute these variables.)

Figure 5 presents the rules for each expression-based operator in unrealizability logic. Having articulated the key intuition behind the rules, let us now look at each rule in detail.

0-ary operators ($0, 1, x, \mathrm{t}, \mathrm{f}$). Rules for 0-ary operators do not rely on the result of another set of programs, and can be computed directly by assigning the correct values ($0, 1, x$, etc.) to the auxiliary (vector-)variable $\mathbf{e_t}$. The symbol $\mathbf{x}$ in the rule Var refers to the vector of values generated by taking x from each sub-state of $\sigma = \langle r_1, r_2, \cdots \rangle$; i.e., $\langle r_1(x_1), r_2(x_2), \cdots \rangle$.

Unary Operators ($!B$). G_{imp} only contains a single instance of a unary operator, Not (!), which takes the value produced by a nonterminal B and negates it. The rule Not mirrors this behavior: it first requires that the behavior of the nonterminal B is captured by the premise $\{\!|\mathcal{P}|\!\}\ B\ \{\!|\mathcal{Q}|\!\}$.

Following Lemma 3.8, the result of executing B is stored in the vector-variable $\mathbf{b_t}$; that is, by referring to the vector-variable $\mathbf{b_t}$ in $\mathcal{Q}$, one can refer to the (set of possible) values created by B. The conclusion of the rule negates the value in $\mathbf{b_t}$ and re-assigns it to $\mathbf{b_t}$ (captured in the postcondition predicate $\exists \mathbf{b_t'}.\mathcal{Q}[\mathbf{b_t'}/\mathbf{b_t}] \wedge \mathbf{b_t} = \neg \mathbf{b_t'}$).

Binary Operators ($E_1 + E_2, E_1 - E_2, E_1 \cdot E_2, E_1/E_2, B_1 \wedge B_2, E_1 < E_2, E_1 == E_2$). Binary operators operate in a similar manner to unary operators in that they rely on the premise triples to refer to the values generated by the sub-nonterminals E_1 and E_2. However, they also pose a unique challenge in that one needs to be careful in how these values are composed, as (partially) described in §2.

Take Plus as an example. A naive version of Plus might be written as following:

$$\frac{\{\!|\mathcal{P}|\!\}\ E_1\ \{\!|\mathcal{Q}_1|\!\} \quad \{\!|\mathcal{P}|\!\}\ E_2\ \{\!|\mathcal{Q}_2|\!\}}{\{\!|\mathcal{P}|\!\}\ E_1 + E_2\ \{\!|\mathcal{Q}_1 + \mathcal{Q}_2|\!\}}\ \text{Plus-bad}$$

The problem arises in resolving the set corresponding to $\mathcal{Q}_1 + \mathcal{Q}_2$: the simplest way is to generate pairs by taking the Cartesian product of the two sets, then add the values in each pair to produce a set of values. However, this approach is imprecise, because it also generates pairs generated by *different* vector-states in the precondition.

Example 3.9 (Plus-bad). Consider an application of the Plus rule where $E_1 ::= x$, $E_2 ::= y$, and the input precondition is $\mathcal{P} = \{\langle (1,3), (2,4) \rangle, \langle (3,1), (4,2) \rangle\}$; i.e., there are two possible two-example vector-states: $(\mathbf{x}[1], \mathbf{y}[1]) = (1,3) \wedge (\mathbf{x}[2], \mathbf{y}[2]) = (2,4)$ and $(\mathbf{x}[1], \mathbf{y}[1]) = (3,1) \wedge (\mathbf{x}[2], \mathbf{y}[2]) = (4,2)$, where $\mathbf{v}[i]$ denotes the i-th entry of $\mathbf{v}$. (We omit e_t in input states; its value is irrelevant.)

Because the only term in $E_1 + E_2$ is $x + y$, the postcondition should only contain states in which $e_t = \langle 4, 6 \rangle$. However, because there are two possible input states in $\mathcal{P}$, the predicate $\mathcal{Q}_1$ contains two states—one in which $e_t = \langle 1, 2 \rangle$ and another in which $e_t = \langle 3, 4 \rangle$. Likewise, the predicate $\mathcal{Q}_2$

contains two states, one in which e_t is $\langle 3, 4\rangle$ and one in which it is $\langle 1, 2\rangle$. If one takes the Cartesian product of the states in $\mathcal{Q}_1$ and $\mathcal{Q}_2$, then the predicate $\mathcal{Q}$ ends up having 3 possible values of e_t: $\langle 4, 6\rangle$, $\langle 2, 4\rangle$, or $\langle 6, 8\rangle$. The problem is that the two possible input states $\langle (1, 3), (2, 4)\rangle$ and $\langle (3, 1), (4, 2)\rangle$ are *mixed* by taking the Cartesian product: for example, $x = \langle 1, 2\rangle$ from the first input vector-state and $y = \langle 1, 2\rangle$ from the second input vector-state are added, these come from different input states!

To remedy this problem, in the Plus rule in unrealizability logic, each original state in the precondition is *appended* with a *copy* of itself, where we refer to the copied part as the '*ghost (vector)-state*', and the original part as the original (vector)-state. This idea is captured in the preconditions of the premises, e.g., $\mathcal{P} \wedge (\mathbf{v_1} = \mathbf{v})$; by taking a set of fresh variables $\mathbf{v_1}$ and setting $\mathbf{v_1} = \mathbf{v}$, one has essentially extended each state in $\mathcal{P}$ with a copy of each variable.

Example 3.10 (Extended States). Consider the precondition $\mathcal{P} = \{\langle (1, 3), (2, 4)\rangle, \langle (3, 1), (4, 2)\rangle\}$, the same set of vector-states used in Example 3.9. Observe that $\mathcal{P} \wedge (\mathbf{v_1} = \mathbf{v})$ is the set of vector-states $(\mathbf{x}[1], \mathbf{y}[1], \mathbf{x}_1[1], \mathbf{y}_1[1], \mathbf{x}[2], \mathbf{y}[2], \mathbf{x}_1[2], \mathbf{y}_1[2]) = \{\langle (1, 3, 1, 3, 2, 4, 2, 4)\rangle, \langle (3, 1, 3, 1, 4, 2, 4, 2)\rangle\}$; note that states such as $\langle (1, 3, 3, 1, 2, 4, 4, 2)\rangle$ are not included in $\mathcal{P} \wedge (\mathbf{v_1} = \mathbf{v})$ because they violate the conjunct "$(\mathbf{v_1} = \mathbf{v})$."

Because the variables of the ghost state have been *renamed*, execution through a program in E will leave these variables *unchanged* according to the semantics of E. Thus, one can say that a state in, e.g., $\mathcal{Q}_1$ is an extended state—where the 'original' part of the state has executed through E, and the 'ghost' part is left unchanged. By this device, one can check *via the ghost parts* whether two extended states from $\mathcal{Q}_1$ and $\mathcal{Q}_2$ should be added. Because the ghost-state portion of each extended state remains unchanged, one can deduce that two extended states must be added if and only if the ghost-state portions of the states are identical.

In the Plus rule, the postcondition of the conclusion illustrates the idea of matching vector-states on ghost-state portions. In particular, the constraint $\mathbf{v_1} = \mathbf{v_2}$ filters out those pairs of extended states in which the ghost-state portions are different! The vocabulary shifts $\mathcal{Q}_1[\mathbf{v}'_1/\mathbf{v}]$ and $\mathcal{Q}_2[\mathbf{v}'_2/\mathbf{v}]$ move the 'original' program variables in $\mathcal{Q}_1$ and $\mathcal{Q}_2$ to $\mathbf{v}'_1$ and $\mathbf{v}'_2$, respectively. The values of the original variables are unneeded, except for the value of $\mathbf{e_t}$ inside these states (available as $\mathbf{e}'_{\mathbf{t}_1}$ and $\mathbf{e}'_{\mathbf{t}_2}$, respectively, after the vocabulary shifts.) The conjunction with $\mathcal{P} \wedge (\mathbf{v} = \mathbf{v_1} = \mathbf{v_2})$ restores the values of the original program variables. The value of $\mathbf{e_t}$ is established by the conjunction with $\mathbf{e_t} = \mathbf{e}'_{\mathbf{t}_1} + \mathbf{e}'_{\mathbf{t}_2}$. Finally, all of the additionally introduced variables are quantified out, leaving (vector-)states over the original variables, plus $\mathbf{e_t}$.

Example 3.11 (Correcting Plus with Extended States). Consider again the grammar from Example 3.9. In this example, we will remove the second example from Examples 3.10 and 3.9 for simplicity: that is, the input precondition $\mathcal{P}$ consists of the length-1 vector-states $\{\langle (1, 3)\rangle, \langle (3, 1)\rangle\}$.

As illustrated in Example 3.10, the extended precondition for E_1 is a set of vector-states of the form (x, y, x_1, y_1), namely, $\{\langle (1, 3, 1, 3)\rangle, \langle (3, 1, 3, 1)\rangle\}$; Similarly, the extended precondition for E_2 is a set of vector-states of the form (x, y, x_2, y_2), namely, $\{\langle (1, 3, 1, 3)\rangle, \langle (3, 1, 3, 1)\rangle\}$.

Because $E_1 ::= x$, the predicate $\mathcal{Q}_1$—the result of executing E_1—has vector-states of the form (x, y, x_1, y_1, e_t), and equals $\{\langle (1, 3, 1, 3, 1)\rangle, \langle (3, 1, 3, 1, 3)\rangle\}$.[3] Similarly, because $E_2 ::= y$, the predicate $\mathcal{Q}_2$—the result of executing E_2—has vector-states of the form (x, y, x_2, y_2, e_t) and equals $\{\langle (1, 3, 1, 3, 3)\rangle, \langle (3, 1, 3, 1, 1)\rangle\}$.

The predicate $\mathcal{Q}_1[\mathbf{v}/\mathbf{v}'_1]$ creates a set of vector-states of the form $(x'_1, y'_1, x_1, y_1, e'_{t_1})$, i.e., $\{\langle (1, 3, 1, 3, 1)\rangle, \langle (3, 1, 3, 1, 3)\rangle\}$ (*original* program variables $\mathbf{v}$ are shifted to $\mathbf{v}'_1$). Similarly, the predicate $\mathcal{Q}_2[\mathbf{v}'_2/\mathbf{v}]$ creates a set of vector-states of the form $(x'_2, y'_2, x_2, y_2, e'_{t_2})$, i.e., $\{\langle (1, 3, 1, 3, 3)\rangle, \langle (3, 1, 3, 1, 1)\rangle\}$.

[3] As in Example 3.9, e_{t_1} is omitted in input vector-states because its value is irrelevant in the example.

$$\frac{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ E\ \{\!|Q|\!\}}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ x := E\ \{\!|\exists \mathbf{x}'.Q[\mathbf{x}'/\mathbf{x}] \wedge \mathbf{x} = \mathbf{e_t}|\!\}}\ \textsf{Assign} \qquad \frac{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ S_1\ \{\!|Q|\!\} \quad \Gamma \vdash \{\!|Q|\!\}\ S_2\ \{\!|\mathcal{R}|\!\}}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ S_1; S_2\ \{\!|\mathcal{R}|\!\}}\ \textsf{Seq}$$

$$\frac{\begin{array}{l}\Gamma \vdash \{\!|\mathcal{P}|\!\}\ B\ \{\!|\mathcal{P}_B|\!\} \\ \Gamma \vdash \{\!|\mathcal{P}_B \wedge (\mathbf{v_1} = \mathbf{v})|\!\}\ S_1\ \{\!|Q_1|\!\} \\ \Gamma \vdash \{\!|\mathcal{P}_B \wedge (\mathbf{v_2} = \mathbf{v})|\!\}\ S_2\ \{\!|Q_2|\!\}\end{array} \qquad \begin{array}{l}\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_1'}, \mathbf{v_2'} \text{ fresh renames of } \mathbf{v} \\ \mathbf{v} \text{ refers to full set of vars in } \mathcal{P}\end{array}}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ \text{if } B \text{ then } S_1 \text{ else } S_2\ \left\{\!\!\left|\exists \mathbf{v_1}, \mathbf{v_2}, \mathbf{v_1'}, \mathbf{v_2'}.\ \begin{array}{l} Q_1[\mathbf{v_1'}[i]/\mathbf{v}[i] \text{ where } \mathbf{b_{t_1}}[i] = \text{false}] \wedge \\ Q_2[\mathbf{v_2'}[i]/\mathbf{v}[i] \text{ where } \mathbf{b_{t_2}}[i] = \text{true}] \end{array} \wedge (\mathbf{v_1} = \mathbf{v_2})\right|\!\!\right\}}\ \textsf{ITE}$$

$$\frac{\begin{array}{l}\Gamma \vdash \{\!|\mathcal{I}|\!\}\ B\ \{\!|\mathcal{I}_B|\!\} \\ \Gamma \vdash \{\!|\mathcal{I}_B \wedge \mathbf{b_{loop}} = \mathbf{b_t}|\!\}\ S\ \{\!|\mathcal{I}_B'|\!\}\end{array} \qquad \begin{array}{l}\mathbf{b_{loop}}, \mathbf{v_1}, \mathbf{v_2} \text{ fresh} \\ \exists \mathbf{v_1}, \mathbf{e_t}, \mathbf{b_t}.\mathcal{I}_B'[\mathbf{v_1}[i]/\mathbf{v}[i] \text{ where } \mathbf{b_{loop}}[i] = \text{false}] \implies \\ \exists \mathbf{v_2}, \mathbf{e_t}, \mathbf{b_t}.\mathcal{I}_B[\mathbf{v_2}[i]/\mathbf{v}[i] \text{ where } \mathbf{b_{loop}}[i] = \text{false}]\end{array}}{\Gamma \vdash \{\!|\mathcal{I}|\!\}\ \text{while } B \text{ do } S\ \{\!|\mathcal{I}_B \wedge \mathbf{b_t} = \mathsf{f}|\!\}}\ \textsf{While}$$

Fig. 6. Inference rules for statements in unrealizability logic.

These are conjoined with $\mathcal{P} \wedge (\mathbf{v} = \mathbf{v_1} = \mathbf{v_2})$, leaving $\mathbf{e_t}$ unconstrained (as $\mathbf{e_t}$ was already unconstrained in $\mathcal{P}$), and produces a set of vector-states of the form $(x, y, x_1', y_1', x_1, y_1, e_{t_1}', x_2', y_2', x_2, y_2, e_{t_2}')$, i.e., $\{\langle(1,3,1,3,1,3,1,1,3,1,3,3)\rangle, \langle(3,1,3,1,3,1,3,3,1,3,1,1)\rangle\}$. Original program variables are colored green, introduced ghost variables are colored cyan, and the results of executing E_1 and E_2 (e_{t_1}' and e_{t_2}', respectively) are colored red.

Once the set of states is conjoined with $\mathbf{e_t} = \mathbf{e_{t_1}'} + \mathbf{e_{t_2}'}$, and the auxiliary variables are quantified out, we obtain exactly the set of states that $E_1 + E_2$ can produce: $(x, y, e_t) = \{\langle(1,3,4)\rangle, \langle(3,1,4)\rangle\}$.

At this point, it becomes possible to answer the question: why do we not simply produce sets of values as the postcondition for expression nonterminals, and instead introduce states with an auxiliary variable $\mathbf{e_t}$? The reason is that, as illustrated in Example 3.9, a scheme that merely returns a set of values is unable to track the originating state, as done in the Plus rule (and other instantiations of the Bin rule of Figure 5), and hence imprecise. Extending the semantics of expressions to return the extended state allows us to keep track of this information by extending the input and output states with ghost variables as illustrated in Example 3.11, and leads to a precise rule—which is why we opt to update the (vector-)variable $\mathbf{e_t}$ in our rules.

3.3.2 Rules for Statements. Unrealizability logic considers four kinds of statements: assignment, sequential composition, branches, and loops, as shown in Figure 6.

Assign and Seq can be explained using the same principles that we used to explain rules for expressions. In Assign, the result of the nonterminal E is stored in the variable $\mathbf{e_t}$ of Q; this value gets referenced and assigned to $\mathbf{x}$. Seq is the same as in Hoare logic, where the sequentiality of the statements are captured by sharing the pre/postcondition Q.

If-Then-Else. Branches pose a similar challenge to operators such as Plus: one requires a mechanism for composing the states generated by two different nonterminals while ensuring that they come from the same 'origin' state. Similarly to what we did for Plus, the solution is to extend the input states with a ghost state: observe that the premise triples are of the same form $\mathcal{P}_B \wedge (\mathbf{v_1} = \mathbf{v})$, where $\mathcal{P}_B$ is identical to $\mathcal{P}$ except that $\mathbf{b_t}$ now stores the result of the branch condition.

The additional complication in the ITE rule is that in a given vector-state, certain examples must pass through the true branch, while others must pass through the false branch. In the ITE rule, we achieve this synchronization by first passing an input state through *both* the true and false

branches (separately)—this step is captured by the two premises about S_1 and S_2, which simply push the ghost-state extended preconditions through the respective branches to obtain Q_1 and Q_2.

When *merging* the states from Q_1 and Q_2, we project out the examples that took the wrong branch—e.g., in Q_1, the postcondition of the true branch, the examples that should go through the false branch are projected out.

This projecting out is captured by the conditional substitution $[\mathbf{v}'_1[i]/\mathbf{v}[i] \text{ where } \mathbf{b}_{\mathbf{t}_1}[i] = \text{false}]$, which substitutes a variable $v[i]$ with a fresh variable $v_1[i]'$ *only if* $\mathbf{b}_{\mathbf{t}_1}[i] = \text{false}$—that is, if the branch condition evaluated to false for that example. (We reference $\mathbf{b}_{\mathbf{t}_1}$, the ghost-version of $\mathbf{b}_\mathbf{t}$, because the value of $\mathbf{b}_\mathbf{t}$ may change through the execution of S_1.)

The quantifier-free segment of the postcondition $Q_1[\mathbf{v}'_\mathbf{i}[i]/\mathbf{v}[i] \text{ where } \mathbf{b}_{\mathbf{t}_1}[i] = \text{false}] \wedge Q_2[\mathbf{v}'_\mathbf{i}[i]/\mathbf{v}[i] \text{ where } \mathbf{b}_{\mathbf{t}_2}[i] = \text{true}] \wedge (\mathbf{v_1} = \mathbf{v_2})$ is thus an extended state with five 'copies' of state, of which only one is retained: the examples that took the correct branches (named via $\mathbf{v}$, and the ones that will be retained), the examples that took the incorrect branches (named via $\mathbf{v}'_1$ and $\mathbf{v}'_2$), and the two ghost states (named via $\mathbf{v_1}$ and $\mathbf{v_2}$). Quantifying out the auxiliary variables $\mathbf{v}'_1, \mathbf{v}'_2, \mathbf{v_1}$, and $\mathbf{v_2}$ leaves just the examples that took the correct branches.

Example 3.12 (If-then-Else with Extended States). Consider the set of states $\mathcal{P}$ given by $(x_1, x_2) = \{\langle(-1, 1)\rangle, \langle(2, -2)\rangle\}$; that is, $\mathcal{P}$ denotes a set of two-example configurations, which could be specified by the formula $(x_1 = -1 \wedge x_2 = 1) \vee (x_1 = 2 \wedge x_2 = -2)$. Consider the If-Then-Else statement if $x > 0$ then $x := x$ else $x := 0 - x$ (we use a single program as it is sufficient to illustrate the ITE rule); this program sets x to its absolute value. Observe that after executing $x > 0$ (corresponding to B), $\mathcal{P}_B$ becomes the set of states $(x_1, x_2, b_{t_1}, b_{t_2}) = \{\langle(-1, 1, \text{false}, \text{true})\rangle, \langle(2, -2, \text{true}, \text{false})\rangle\}$. After conjoining the ghost state and running through S_1, we get the following set (the ghost state is marked in cyan):

$$(x_1, x_2, b_{t_1}, b_{t_2}, x_{1_1}, x_{2_1}, b_{t_{1_1}}, b_{t_{2_1}}) = \{\langle(-1, 1, \text{false}, \text{true}, -1, 1, \text{false}, \text{true})\rangle, \langle(2, -2, \text{true}, \text{false}, 2, -2, \text{true}, \text{false})\rangle\}$$

Applying the conditional substitution to this set yields the following set of states, where non-substituted variables are green, variables substituted with $\mathbf{v}'_1$ are red, and the ghost state is cyan:

$$\{\langle(-1, 1, \text{false}, \text{true}, -1, 1, \text{false}, \text{true})\rangle, \langle(2, -2, \text{true}, \text{false}, 2, -2, \text{true}, \text{false})\rangle\}$$

Similarly, executing S_2 and applying substitution yields the following set:

$$\{\langle(1, -1, \text{false}, \text{true}, -1, 1, \text{false}, \text{true})\rangle, \langle(-2, 2, \text{true}, \text{false}, 2, -2, \text{true}, \text{false})\rangle\}$$

Again, note that the substitution happens for variables highlighted in red, while the green variables are the original program variables. Taking the conjunction and quantifying out all unnecessary variables leaves only the variables highlighted in green, which yields:

$$(x_1, x_2, b_{t_1}, b_{t_2}) = \{\langle(1, 1, \text{false}, \text{true}\rangle, \langle(2, 2, \text{true}, \text{false})\rangle\},$$

Which is exactly the set of states that are computed by the given If-Then-Else statement.

In general, conditionally substituting a variable $\mathbf{x}[i]$ with $\mathbf{x}'[i]$, if a condition $\mathbf{b}[i]$ is false may be performed by replacing all occurences of $\mathbf{v}[i]$ in a formula with if $\mathbf{b}[i]$ then $\mathbf{x}[i]$ else $\mathbf{x}'[i]$: this expression is equivalent to $\mathbf{x}'[i]$ when the branch condition $\mathbf{b}[i]$ is false, and equivalent to $\mathbf{x}[i]$ when the expression is true. Note that this substitution is possible even if the predicate is used to encode infinite vector-states, as long as the predicate itself is finite.

Example 3.13 (Conditional Substitution Over Infinite Vector-States). Consider the infinite vector-state $\forall i.\mathbf{x}[i] = -1 \wedge \mathbf{y}[i] = i$. Although this vector-state is infinite, one can conditionally substitute $\mathbf{x}$ with a new vector-variable $\mathbf{x}'$ for entries where $\mathbf{b}[i]$ is false with the expression $\forall i.(\text{if } \mathbf{b}[i] \text{ then } \mathbf{x}[i] \text{ else } \mathbf{x}'[i]) = -1 \wedge \mathbf{y}[i] = i$. Observe that $\mathbf{x}[i]$ is now unconstrained for entries where $\mathbf{b}[i] = \text{false}$, whereas $\mathbf{x}'[i]$ is unconstrained instead for entries where $\mathbf{b}[i] = \text{true}$.

Loops. The basic intuition behind the While rule is identical to that in Hoare logic—it requires the existence of an *invariant* $\mathcal{I}$. The difference with Hoare logic is that, in our work, $\mathcal{I}$ must work as an invariant for *all* possible completions of the loop body S.

Because the invariant $\mathcal{I}$ spans multiple examples, the way the invariant is checked is also different from in Hoare logic. First, we take a similar approach as in ITE, and push examples through the loop body S, regardless of the loop guard (in the premise $\{\!\{\mathcal{I}_B \wedge \mathbf{b_{loop}} = \mathbf{b_t}\}\!\}\ S\ \{\!\{\mathcal{I}'_B\}\!\}$). Then, the implication checks whether $\mathcal{I}'_B \implies \mathcal{I}_B$ *only on the examples where the loop guard is* true: this is again achieved through conditional substitution, which substitutes variables for which $\mathbf{b_{loop}}[\mathbf{i}] = \text{false}$, and quantifies out the substituted variables again (as well as the auxiliary variables $\mathbf{e_t}$ and $\mathbf{b_t}$, which may change).[4] For examples that are substituted out, $\mathcal{I}_B$ is an invariant because these examples are left unchanged—thus, one only needs to check the invariant condition for examples that pass the loop guard and execute the body, which the implication premise captures.

Finally, the condition $\mathbf{b_t} = \mathsf{f}$ in the postcondition captures vector-states where the loop guard of all examples (stored in $\mathbf{b_t}$ of $\mathcal{I}_B$) evaluate to false: i.e., vector-states for which the loop terminates on all individual examples within the vector-state. In other words, this postcondition checks for *partial correctness*, and does not provide any guarantees about traces that contain an example that fail to terminate.

The reader may find it surprising that a single invariant capturing the behavior of all possible loop bodies suffices for relative completeness. A single invariant does suffice for relative completeness because one can actually write an invariant that talks about the choice of loop body *itself*. However, to pull off this trick, one needs a highly complex encoding whose details are beyond the scope of this paper (see §4.1 for details). In practice, the use of this complex encoding will often result in an invariant too complex for a proof to be completed. Thus, when writing actual proofs in unrealizability logic, it is often beneficial to split the loop bodies into smaller sets, and reason about them separately (as we have done in Example 2.6 of §2) through the use of the structural rules.

3.3.3 Structural Rules. Finally, unrealizability logic has a set of structural rules that operate on conditions, hypotheses, and sets of programs. We list them in Figure 7.

Weaken, Conj. These rules operate like their counterpart in Hoare logic. Weaken can shrink the precondition or enlarge the postcondition following the principle that the postcondition overapproximates the set of all states that the precondition can generate. Weaken can also shrink the set of considered programs. Conj takes the conjunction of the two postconditions.

GrmDisj. GrmDisj allows us to split sets of programs: if two sets of programs, N_1 and N_2, satisfy the same pre- and postcondition, then their union also satisfies those same pre- and postcondition. This rule is not required for the completeness of the logic. However, it helps greatly in simplifying actual unrealizability proofs: a clever division of the search space often results in simpler predicates.

Inv, SubOne, SubTwo. Inv, Sub1, and Sub2 are rules that are necessary for our completeness proof. Inv states that if a precondition does not share any free variables with any program in the set N (i.e., $vars(\mathcal{P}) \cap vars(N) = \emptyset$), then it is an invariant—clearly this holds, because any program N will be unable to access or modify the variables used in $\mathcal{P}$. Sub1 performs α-renaming of variables inside the pre-and postconditions; the side conditions state that one can only rename variables that are not 'captured' by the set of programs N. Sub2 states that variables from the postcondition that do not appear in the set of programs N or the postcondition may be renamed to any new name. Intuitively, this is because a variable that only occurs in the precondition will remain untouched during program execution; see the proof of Sub2 in the full version of the paper for details.

[4]The additional assignment $\mathbf{b_{loop}} = \mathbf{b_t}$ before executing the loop body, as well as referencing $\mathbf{b_{loop}}$ in the postcondition, is again due to the fact that $\mathbf{b_t}$ may change during the execution of S.

$$\frac{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\} \quad \mathcal{P}' \Longrightarrow \mathcal{P} \quad \mathcal{Q} \Longrightarrow \mathcal{Q}' \quad N_1 \subseteq N}{\Gamma \vdash \{\!|\mathcal{P}'|\!\}\ N_1\ \{\!|\mathcal{Q}'|\!\}}\ \text{Weaken} \qquad \frac{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}_1|\!\} \quad \Gamma \vdash \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}_2|\!\}}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}_1 \wedge \mathcal{Q}_2|\!\}}\ \text{Conj}$$

$$\frac{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ N_1\ \{\!|\mathcal{Q}|\!\} \quad \Gamma \vdash \{\!|\mathcal{P}|\!\}\ N_2\ \{\!|\mathcal{Q}|\!\}}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ (N_1 \cup N_2)\ \{\!|\mathcal{Q}|\!\}}\ \text{GrmDisj} \qquad \frac{vars(\mathcal{P}) \cap vars(N) = \emptyset}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{P}|\!\}}\ \text{Inv}$$

$$\frac{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\} \quad \mathbf{y} \cap vars(N) = \emptyset \quad \mathbf{z} \cap vars(N) = \emptyset \quad (\mathbf{y}, \mathbf{z}) \text{ are not } \mathbf{e_t} \text{ or } \mathbf{b_t}}{\Gamma \vdash \{\!|\mathcal{P}[\mathbf{y}/\mathbf{z}]|\!\}\ N\ \{\!|\mathcal{Q}[\mathbf{y}/\mathbf{z}]|\!\}}\ \text{Sub1}$$

$$\frac{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\} \quad \mathbf{z} \cap (vars(N) \cup vars(\mathcal{Q})) = \emptyset}{\Gamma \vdash \{\!|\mathcal{P}[\mathbf{y}/\mathbf{z}]|\!\}\ N\ \{\!|\mathcal{Q}|\!\}}\ \text{Sub2}$$

$$\frac{\begin{array}{ll}\Gamma, \{\!|\mathcal{P}|\!\} N \{\!|\mathcal{Q}|\!\} \vdash \{\!|\mathcal{P}|\!\} RHS_1 \{\!|\mathcal{Q}|\!\} & \\ \cdots & N \to RHS_1 \mid \cdots \mid RHS_n \\ \Gamma, \{\!|\mathcal{P}|\!\} N \{\!|\mathcal{Q}|\!\} \vdash \{\!|\mathcal{P}|\!\} RHS_n \{\!|\mathcal{Q}|\!\} & \text{is exhaustive}\end{array}}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\}}\ \text{HP} \qquad \frac{}{\Gamma, \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\} \vdash \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\}}\ \text{ApplyHP}$$

Fig. 7. Structural rules in unrealizability logic. $vars(N)$ refers to the entire set of variables that may appear in a program $t_N \in N$, with the addition of the reserved auxiliary variables $\mathbf{e_t}$ and $\mathbf{b_t}$.

HP, ApplyHP. HP and ApplyHP allow us to perform structural induction in the proof tree. HP introduces a new hypothesis into the context Γ, while ApplyHP allows us to apply an induction hypothesis. (These rules were explained in §2.1).

Definition 3.14 (Derivability). Given a precondition over vector-states $\mathcal{P}$, a postcondition over vector-states $\mathcal{Q}$, a set of programs S, and a set of hypotheses Γ, we say that a triple $\{\!|\mathcal{P}|\!\}\ S\ \{\!|\mathcal{Q}|\!\}$ is *derivable assuming* Γ, denoted by $\Gamma \vdash \{\!|\mathcal{P}|\!\}\ S\ \{\!|\mathcal{Q}|\!\}$, if there exists a proof tree deriving $\Gamma \vdash \{\!|\mathcal{P}|\!\}\ S\ \{\!|\mathcal{Q}|\!\}$ using the rules of unrealizability logic in Figures 5, 6, and 7.

We say that a triple $\{\!|\mathcal{P}|\!\}\ S\ \{\!|\mathcal{Q}|\!\}$ is *derivable*, if it can be derived with an empty set of hypotheses.

4 SOUNDNESS, COMPLETENESS, AND OTHER THEORETICAL RESULTS

Unrealizability logic is a sound logic in the sense that any derivable triple $\{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\}$ is also valid.

Theorem 4.1 (Soundness). *Given a nonterminal N in a grammar G, the following property holds:*

$$\vdash \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\} \implies \models \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\}$$

Soundness can be proved via structural induction, which shows that the conclusion triple of each rule is sound given that the premises are sound (see the Appendix in the full version of this paper [Kim et al. 2022] for the full proof). While reasoning about some cases is complex due to the vectorized semantics, the overall structure of the proof is a simple structural-induction argument.

Surprisingly, unrealizability logic is also relatively complete, in the sense that all valid triples $\{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\}$ are derivable, assuming that there is an oracle capable of verifying all true sentences in the assertion language.

Theorem 4.2 (Relative Completeness). *Let $\mathcal{P}$ and $\mathcal{Q}$ be predicates in an assertion language L. Assuming the existence of an oracle capable of verifying all true sentences in L, the following property holds for each nonterminal N in a grammar G:*

$$\models \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\} \implies \vdash \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\}$$

Completeness follows from two lemmas that we state later (Lemmata 4.4 and 4.5), which state that (*i*) a *strongest triple* that precisely describes the behavior of a nonterminal is derivable, and (*ii*) that any sound triple about a nonterminal can be derived from the strongest triple.

In the rest of this section, we give a sketch of the proof of completeness, and discuss some other theoretical results. We will present proof sketches for the theorems listed in this section, for the full proofs, we again refer the reader to the full version of the paper [Kim et al. 2022].

4.1 A Sketch of the Proof of Completeness

Completeness of unrealizability logic is a two-step process: first we wish to prove that the non-structural rules (expression and statement rules) of unrealizability logic are complete, in the sense that, given a set of programs generated by a production $op(N_1, \cdots, N_k)$, if all sound premise triples about $N_1, \cdots, N_k$ are derivable, then all sound triples about $op(N_1, \cdots, N_k)$ are also derivable.

Lemma 4.3 (Completeness of Expression and Statement Rules). *Let G be a regular tree grammar and N be a set of programs generated by either a nonterminal or the RHS of a production. Then, the expression and statement rules of unrealizability logic* preserve completeness*, in the sense that, if all sound required premise triples are derivable, then all sound triples about N are derivable in unrealizability logic.*

When creating a proof tree in unrealizability logic, one will encounter triples where the center element is the right-hand side of a production (e.g., ❶ of Figure 2). Consequently, Lemma 4.3 is stated for both nonterminals and right-hand sides of productions. Lemma 4.3 can be proved by showing that the postcondition of the conclusion of each rule *precisely* captures the semantics of the corresponding operator.

One thing to note about the proof of Lemma 4.3 is that it relies on the existence of a predicate that expresses exactly the weakest precondition of a set of while loops for an arbitrary postcondition, a requirement called the *expressibility* of the weakest precondition. Given that the postcondition is encodable in FO-PA (e.g., the postcondition does not contain infinite vector-states), such a weakest precondition can also be constructed in FO-PA similar to how the invariant for while loops in standard Hoare logic is constructed [Winskel 1993], by encoding the term to be synthesized using a pair of integers via the Godel β-function [Davis et al. 1990], and defining a function to interpret the semantics of these integers. In other words, the weakest precondition itself does not necessitate extra expressiveness in the assertion language. The actual encoding of the weakest precondition is highly complex, as stated in §3.3; in this paper, we take what is known as the *extensional* approach and assume that the weakest precondition can be expressed within the assertion language of choice.

While Lemma 4.3 would be sufficient as a proof of completeness in normal Hoare logic, it is insufficient for unrealizability logic because we are working over an RTG; without the use of the HP rule, the proof tree will become infinite. The next step in showing completeness remedies this fact by proving that through use of the HP rule, we can prove the strongest triple for a nonterminal within a finite number of steps.

Lemma 4.4 (Derivability of the Strongest Triple). *Let N be a nonterminal from a RTG G and $\mathbf{z}$ be a set of auxiliary symbolic variables. Let $Q_0 \equiv [\![N]\!](\mathbf{x} = \mathbf{z})$; that is, Q_0 is a formula that precisely captures the behavior of the set of programs $L(N)$ on the symbolic vector-state $x_1 = z_1 \wedge \cdots x_n = z_n$.*[5] *We refer to $\{\!|\mathbf{x} = \mathbf{z}|\!\}\ N\ \{\!|Q_0|\!\}$ as the* strongest triple *for N. Then $\vdash \{\!|\mathbf{x} = \mathbf{z}|\!\}\ N\ \{\!|Q_0|\!\}$ is derivable.*

The proof of Lemma 4.4 performs induction over the number of hypotheses in the context: the proof relies on the fact that one only needs to insert the strongest triple for each nonterminal into

[5]We assume that state is defined over a single variable x; it is trivial to extend this to multiple-variable states.

the context (because the strongest triple can derive any other triple), thus the HP rule need only be applied at most $|G|$ times (the number of nonterminals inside G). When all $|G|$ hypotheses are established, one relies on Lemma 4.3 to show that the strongest hypothesis can actually be derived.

The proof of Lemma 4.4, as well as the final step in arguing completeness, requires a lemma that shows one can derive any triple from the strongest triple.

Lemma 4.5 (Derivability of General Triples). *Let G be a grammar and N be any nonterminal in G. Given the strongest triple H_N for the nonterminal N (as defined in Lemma 4.4), if $\Gamma \vdash H_N$, then $\Gamma \models \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\} \implies \Gamma \vdash \{\!|\mathcal{P}|\!\}\ N\ \{\!|\mathcal{Q}|\!\}$.*

Lemma 4.5 can be proved using the substitution rules Sub1 and Sub2. Relative completeness (Theorem 4.2) then follows from Lemma 4.4 and Lemma 4.5.

4.2 Undecidability of Unrealizability and Decidable Fragments

Although we have proved relative completeness, proving unrealizability is an undecidable problem, even when limited to synthesis problems *without* loops.

Theorem 4.6 (Undecidability of Unrealizability). *Let sy_u be a synthesis problem with a grammar G_u that does* not *contain productions of the form $S ::= \mathsf{while}\ B\ \mathsf{do}\ S_1$. Checking whether sy_u is unrealizable is an undecidable problem.*

The proof of Theorem 4.6 relies on translating an arbitrary program for a two-counter machine into a program-synthesis problem that is realizable iff the original program halts. This translation is performed by taking each instruction of the original program and translating it into a production that performs the same computation. The goal of the resultant synthesis problem is to set a special variable h to 1: by translating only the final Halt instruction into a production that sets h to 1, the translated synthesis problem is realizable iff the original program halts. As the halting problem for two-counter machines is undecidable, it follows that unrealizability is undecidable as well.

Theorem 4.6 shows that proving unrealizability of synthesis problems is undecidable, even when while loops are out of the question. On the other hand, when one is limited to synthesis over finite domains, proving unrealizability becomes decidable—even when the synthesis problem may contain an unbound number of loops.

Theorem 4.7 (Decidability of Unrealizability over Finite Domains). *Determining whether a synthesis problem sy_{fin} is unrealizable, where the grammar G_{fin} is valid with respect to G_{impv}, is* decidable *if the semantics of programs in G_{fin} is defined over a finite domain.*

The proof of Theorem 4.7 relies on the fact that if the domain is finite, one can perform grammar-flow analysis [Möncke and Wilhelm 1991] to obtain the greatest-fixed point of values that a nonterminal may generate. In this paper, because both expressions and statements are of type State $\to$ State, the values that nonterminals generate are pairs of states, representing input-output relations; this computation is guaranteed to terminate because the domain is finite.

Furthermore, if sy_{fin} is unrealizable, then there exists a proof of this fact in unrealizability logic.

5 THE POWER OF UNREALIZABILITY LOGIC

In this section, we give some example proofs that highlight the capabilities of unrealizability logic.

5.1 Dealing with an Infinite Number of Examples

As stated in §2 and §3, one of the major features of unrealizabilty logic is that it is capable of dealing with synthesis problems which require an infinite number of examples to prove unrealizability. We illustrate one such example in this section.

Example 5.1 (Proof with Infinitely Many Examples). Consider a synthesis problem $sy_{\text{id\_ite}}$, with the grammar $G_{\text{id\_ite}}$ given below:

$$\begin{array}{lcl} \mathit{Start} & \to & \text{if } B \text{ then } A \text{ else } \mathit{Start} \mid A \\ B & \to & y == N \\ N & \to & 0 \mid N + 1 \\ A & \to & x := N \end{array}$$

$sy_{\text{id\_ite}}$ aims to synthesize a function f which takes as input a state (x, y), and return a state where x is equivalent to y.

$sy_{\text{id\_ite}}$ is unrealizable, and one needs an infinite number of examples to prove this fact. This is because a specification over n examples for $sy_{\text{id\_ite}}$ will be realizable by a term that contains n If-then-Elses. However, a term of finite size in $G_{\text{id\_ite}}$ can only assign to x a finite number of constants; thus it is impossible to have a function in $G_{\text{id\_ite}}$ that sets x to y in the general case. $sy_{\text{id\_ite}}$ is, in fact, an imperative variant of an example often used to show that there are synthesis problems that cannot be proven unrealizable using only a finite number of examples [Hu et al. 2019]; previous approaches [Hu et al. 2019, 2020; Kim et al. 2021] thus cannot prove $sy_{\text{id\_ite}}$ unrealizable. We show that a proof tree for $sy_{\text{id\_ite}}$ can be constructed in unrealizability logic, using FO-PA extended with infinite arrays as the assertion language.

Consider a set of inputs (x, y) given by the predicate $\forall i \in \mathbb{N}.x_i = -1 \wedge y_i = i$. That is, the input is an infinite set of examples where y spans over the positive integers, and x is assigned the fixed value -1 (this implies $x_i \neq y_i$ for every $i \in \mathbb{N}$). The output specification for this input is given as $\forall i.x_i = i \wedge y_i = i$; the infinite vector-state where $x_i = y_i = i$ for the i-th example. This infinite set of examples does *not* encompass the entire input state; however, it is sufficient to prove unrealizability.

To prove unrealizability, first negate the postcondition to obtain the triple that we wish to prove:

$$\{\!|\forall i.x_i = -1 \wedge y_i = i|\!\}\ \mathit{Start}\ \{\!|\exists i.x_i \neq i \vee y_i \neq i|\!\}$$

Instead of proving this triple directly, we will prove the following triple (from which we can obtain the target triple via Weaken):

$$\left\{\!\!\left|\forall i.y_i = i \wedge \begin{array}{l}\text{only a finite no. of } i \\ \text{such that } x_i = y_i\end{array}\right|\!\!\right\}\ \mathit{Start}\ \left\{\!\!\left|\forall i.y_i = i \wedge \begin{array}{l}\text{only a finite no. of } i \\ \text{such that } x_i = y_i\end{array}\right|\!\!\right\}$$

Let $\mathcal{I}$ denote the condition '$\forall i.y_i = i \wedge$ only a finite no. of i such that $x_i = y_i$'. To see the implication, observe that: (i) $\forall i.x_i = -1 \wedge y_i = i \implies \mathcal{I}$ as there are 0 (a finite number) is for which $x_i = y_i$; (ii) $\mathcal{I} \implies \exists i.x_i \neq i \vee y_i \neq i$, because if $x_i = y_i$ for only a finite number of i, then there must exist some i for which $x_i \neq i$.

We will prove that $\{\!|\mathcal{I}|\!\}\ \mathit{Start}\ \{\!|\mathcal{I}|\!\}$ holds by introducing this triple as a hypothesis for *Start* via the HP rule. Let $\Gamma_{\mathcal{I}}$ denote the context containing only the triple $\{\!|\mathcal{I}|\!\}\ \mathit{Start}\ \{\!|\mathcal{I}|\!\}$.

$$\frac{\Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ A\ \{\!|\mathcal{I}|\!\} \quad \Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ \text{if } B \text{ then } A \text{ else } \mathit{Start}\ \{\!|\mathcal{I}|\!\}}{\vdash \{\!|\mathcal{I}|\!\}\ \mathit{Start}\ \{\!|\mathcal{I}|\!\}}\ \text{HP}$$

Consider first the base case $\{\!|\mathcal{I}|\!\}\ A\ \{\!|\mathcal{I}|\!\}$, where A is the simple assignment $x := N$. The hypothesis in this case can be proved simply via Assign and Weaken:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ N\ \{\!|\exists \mathbf{e}_t'.\mathcal{I}[\mathbf{e}_t'/\mathbf{e}_t] \wedge \exists k.k \geq 0 \wedge \mathbf{e}_t = k|\!\}}\ \text{HP}}{\Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ N\ \{\!|\mathcal{I} \wedge \exists k.k \geq 0 \wedge \mathbf{e}_t = k|\!\}}\ \text{Weaken}}{\Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ A\ \{\!|\exists \mathbf{x}'.\mathcal{I}[\mathbf{x}'/\mathbf{x}] \wedge \exists k.k \geq 0 \wedge \mathbf{e}_t = k \wedge \mathbf{x} = \mathbf{e}_t|\!\}}\ \text{Assign}}{\Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ A\ \{\!|\exists \mathbf{x}'.\mathcal{I}[\mathbf{x}'/\mathbf{x}] \wedge \exists k.k \geq 0 \wedge \forall i.x_i = k|\!\}}\ \text{Weaken}}{\Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ A\ \{\!|\mathcal{I}|\!\}}\ \text{Weaken}$$

The ellipsis over the top-level HP rule indicates that we can prove that N results in a value above 0 via the HP rule, through the induction hypothesis $\{\!|\mathcal{I}|\!\}\ N\ \{\!|\exists \mathbf{e}'_\mathbf{t}.\mathcal{I}[\mathbf{e}'_\mathbf{t}/\mathbf{e}_\mathbf{t}] \wedge \exists k.k \geq 0 \wedge \mathbf{e}_\mathbf{t} = k|\!\}$ (the exact reasoning is omitted). The final application of Weaken works as, if all $x_i = k$ for some $k \geq 0$, then there is at most one (a finite number) i for which $x_i = y_i$.

The inductive case requires an application of the ITE rule. We first write:

$$\dfrac{\dfrac{\Gamma_\mathcal{I} \vdash \{\!|\mathcal{I}|\!\}\ B\ \{\!|\mathcal{I}|\!\} \quad \Gamma_\mathcal{I} \vdash \{\!|\mathcal{I} \wedge \mathbf{v_1} = \mathbf{v}|\!\}\ A\ \{\!|\mathcal{I}|\!\} \quad \Gamma_\mathcal{I} \vdash \{\!|\mathcal{I} \wedge \mathbf{v_2} = \mathbf{v}|\!\}\ \mathit{Start}\ \{\!|\mathcal{I}|\!\}}{\Gamma_\mathcal{I} \vdash \{\!|\mathcal{I}|\!\}\ \text{if } B \text{ then } A \text{ else } \mathit{Start}\ \left\{\!\middle|\exists \mathbf{v_1}, \mathbf{v_2}, \mathbf{v'_1}, \mathbf{v'_2}.\ \begin{array}{l}\mathcal{I}[\mathbf{v'_1}[i]/\mathbf{v}[i] \text{ where } \mathbf{b_{t_1}}[i] = \text{false}] \wedge \\ \mathcal{I}[\mathbf{v'_2}[i]/\mathbf{v}[i] \text{ where } \mathbf{b_{t_2}}[i] = \text{true}]\end{array} \wedge (\mathbf{v_1} = \mathbf{v_2}) \middle|\!\right\}}\ \text{ITE}}{\Gamma_\mathcal{I} \vdash \{\!|\mathcal{I}|\!\}\ \text{if } B \text{ then } A \text{ else } \mathit{Start}\ \{\!|\mathcal{I}|\!\}}\ \text{Weaken}$$

Observe that the first premise $\{\!|\mathcal{I}|\!\}\ B\ \{\!|\mathcal{I}|\!\}$ seems to do nothing. This is the result of weakening the postcondition generated by B, and effectively 'forgetting' the value of the branch condition as done in the following proof tree, where the postcondition of the premise represents the exact postcondition one would have obtained through a precise reasoning of Eq:

$$\dfrac{\dfrac{\cdots}{\Gamma_\mathcal{I} \vdash \{\!|\mathcal{I}|\!\}\ B\ \left\{\!\middle|\exists \mathbf{b'_t}, \mathbf{v_1}, \mathbf{v_2}, \mathbf{v'_1}, \mathbf{v'_2}.\mathcal{I} \wedge \begin{array}{l}\mathcal{I}[\mathbf{v'_1}/\mathbf{v}] \wedge \mathbf{e'_{t_1}} = \mathbf{y} \wedge \\ \mathcal{I}[\mathbf{v'_2}/\mathbf{v}] \wedge \exists k.\mathbf{e'_{t_2}} = k\end{array} \wedge \begin{array}{l}(\mathbf{v} = \mathbf{v_1} = \mathbf{v_2}) \wedge \\ \mathbf{b_t} = (\mathbf{e'_{t_1}} == \mathbf{e'_{t_2}})\end{array} \middle|\!\right\}}\ \text{Eq}}{\Gamma_\mathcal{I} \vdash \{\!|\mathcal{I}|\!\}\ B\ \{\!|\mathcal{I}|\!\}}\ \text{Weaken}$$

We take such an approach to take advantage of the fact that the branch condition is actually irrelevant to the proof of unrealizability; this example shows that sometimes triples inside the proof tree need not be precise to prove unrealizability (which can, as shown, greatly simplify predicates).

The second premise is an instance of A; one can prove this premise by using the same proof tree as we used to prove $\Gamma_\mathcal{I} \vdash \{\!|\mathcal{I}|\!\}\ A\ \{\!|\mathcal{I}|\!\}$ in the base case, and apply an additional Weaken to it.

$$\dfrac{\dfrac{\cdots}{\Gamma_\mathcal{I} \vdash \{\!|\mathcal{I}|\!\}\ A\ \{\!|\mathcal{I}|\!\}}}{\Gamma_\mathcal{I} \vdash \{\!|\mathcal{I} \wedge \mathbf{v_1} = \mathbf{v}|\!\}\ A\ \{\!|\mathcal{I}|\!\}}\ \text{Weaken}$$

Finally, the third premise can be derived with a simple combination of ApplyHP and Weaken, where ApplyHP applies the *induction hypothesis* that $\{\!|\mathcal{I}|\!\}\ \mathit{Start}\ \{\!|\mathcal{I}|\!\}$.

$$\dfrac{\dfrac{}{\Gamma_\mathcal{I} \vdash \{\!|\mathcal{I}|\!\}\ \mathit{Start}\ \{\!|\mathcal{I}|\!\}}\ \text{ApplyHP}}{\Gamma_\mathcal{I} \vdash \{\!|\mathcal{I} \wedge \mathbf{v_2} = \mathbf{v}|\!\}\ \mathit{Start}\ \{\!|\mathcal{I}|\!\}}\ \text{Weaken}$$

Returning to the application of ITE, observe that $\mathcal{I}$ lacks any occurence of $\mathbf{v_1}$ and $\mathbf{v_2}$. This is again because our triples for A and *Start* in the premises were not exact; $\mathcal{I}$ is an *overapproximation* of the set of states that may occur when, e.g., executing *Start* over the input precondition $\mathcal{I} \wedge \mathbf{v_2} = \mathbf{v}$. Although the derived postcondition is thus also not exact, it is still a sound overapproximation that is *precise enough* for the proof. To see this, observe that mixing examples from two states where '$\forall i.y_i = i \wedge$ only a finite no. of i such that $x_i = y_i$' holds (the postcondition of the conclusion of ITE) still results in a state where '$\forall i.y_i = i \wedge$ only a finite no. of i such that $x_i = y_i$' holds. Thus the final application of Weaken that derives $\{\!|\mathcal{I}|\!\}$ if B then A else *Start* $\{\!|\mathcal{I}|\!\}$ is a valid application.

By proving that $\Gamma \vdash \{\!|\mathcal{I}|\!\}\ A\ \{\!|\mathcal{I}|\!\}$ and $\Gamma \vdash \{\!|\mathcal{I}|\!\}$ if B then A else *Start* $\{\!|\mathcal{I}|\!\}$, we have proven the induction hypothesis; thus $\{\!|\mathcal{I}|\!\}\ \mathit{Start}\ \{\!|\mathcal{I}|\!\}$ holds, which can further be weakened down into the target triple $\{\!|\forall i.x_i = -1 \wedge y_i = i|\!\}\ \mathit{Start}\ \{\!|\exists i.x_i \neq i \vee y_i \neq i|\!\}$. Hence $sy_{\text{id\_ite}}$ is *unrealizable*, and we have proved this fact quite simply via some imprecise reasoning.

In Example 5.1, we used predicates such as $\forall i.x_i = i \wedge y_i = i$, or 'only a finite no. of i such that $x_i = y_i$'; whether or not such predicates are supported is dictated by the choice of the assertion language. As mentioned in §3, the choice of assertion language is parametric in unrealizability logic, as long as it contains the operators used in the rules.

5.2 Loops and Expressing Proof Strategies from Other Frameworks

In this section, we give an example of how loops are dealt with in unrealizability logic, and also show that the reasoning behind other frameworks for proving unrealizability (e.g., Nay [Hu et al. 2020] or MESSY [Kim et al. 2021]) can be captured as a *proof strategy* for completing a unrealizability logic proof tree.

Example 5.2 (Proof with Loops). Consider a synthesis problem sy_{sum} where the goal is to synthesize a function f that takes as input a state (x, y), and assigns to y the sum of all integers between 1 and x. Let as assume that sy_{sum} is given the following grammar G_{sum}:

$$\begin{array}{lcl} \mathit{Start} & \to & \text{while } B \text{ do } S \\ B & \to & E < E \\ S & \to & x := E \mid y := E \mid S; S \\ E & \to & x \mid y \mid E + E \mid E - E \end{array}$$

Then, the problem sy_{sum} is unrealizable because if x and y are both even, then E can only produce even values (note that E does not contain productions such as $E + 1$). This fact conflicts with cases where, e.g., $x = 2$, in which case y must be assigned 3. sy_{sum} is introduced as an unrealizable problem in Kim et al. [2021], where the solver MESSY exploits this fact to prove that sy_{sum} is indeed unrealizable when given the single input example $(x, y) = (2, 0)$. We show that this argument can be mimicked directly as a proof tree for unrealizability logic.

Let $\mathcal{I}$ denote the condition $x \equiv_2 0 \wedge y \equiv_2 0$, i.e., that x and y are both even (because we use a single example, we temporarily drop the vector notation). We wish to use $\mathcal{I}$ as an invariant for the loop while B do S. Begin with an application of While for *Start*, where, similar to Example 5.1, the loop condition is irrelevant to the proof and thus can be skipped:

$$\frac{\vdash \{\!|\mathcal{I}|\!\}\ B\ \{\!|\mathcal{I}|\!\} \qquad \vdash \{\!|\mathcal{I}|\!\}\ S\ \{\!|\mathcal{I}|\!\}}{\vdash \{\!|\mathcal{I}|\!\}\ \text{while } B \text{ do } S\ \{\!|\mathcal{I} \wedge \mathbf{b_t} = \text{false}|\!\}}\ \text{While}$$

The first premise can be proved via Weaken, as we did in Example 5.1. The implication condition of the While rule has been omitted, as in this case $\mathcal{I}'_B \equiv \mathcal{I}_B \equiv \mathcal{I}$ and thus the implication is trivial.

We wish to prove that $\vdash \{\!|\mathcal{I}|\!\}\ S\ \{\!|\mathcal{I}|\!\}$; let us introduce $\{\!|\mathcal{I}|\!\}\ S\ \{\!|\mathcal{I}|\!\}$ as a hypothesis. We denote the context containing only this triple as Γ_S. We then have the proof obligation:

$$\frac{\Gamma_S \vdash \{\!|\mathcal{I}|\!\}\ x := E\ \{\!|\mathcal{I}|\!\} \qquad \Gamma_S \vdash \{\!|\mathcal{I}|\!\}\ y := E\ \{\!|\mathcal{I}|\!\} \qquad \Gamma_S \vdash \{\!|\mathcal{I}|\!\}\ S; S\ \{\!|\mathcal{I}|\!\}}{\vdash \{\!|\mathcal{I}|\!\}\ S\ \{\!|\mathcal{I}|\!\}}\ \text{HP}$$

The first two premises can be proved by introducing a hypothesis $\{\!|x \equiv_2 0 \wedge y \equiv_2 0|\!\}\ E\ \{\!|e_t \equiv_2 0|\!\}$. The third premise can be proved via two applications of ApplyHP. This completes that $\{\!|\mathcal{I}|\!\}\ S\ \{\!|\mathcal{I}|\!\}$, and therefore that $\{\!|\mathcal{I}|\!\}$ while B do S $\{\!|\mathcal{I} \wedge \mathbf{b_t} = \text{false}|\!\}$.

Finally, we apply Weaken to $\{\!|\mathcal{I}|\!\}$ while B do S $\{\!|\mathcal{I} \wedge \mathbf{b_t} = \text{false}|\!\}$ to obtain the triple $\{\!|x = 2 \wedge y = 0|\!\}\ \mathit{Start}\ \{\!|y \neq 3|\!\}$. We have thus proved that sy_{sum} is unrealizable, mainly by using an invariant that is valid across the entire set of possible loop bodies.

In §2, specifically Example 2.3, we highlighted the importance of finding good hypotheses and invariants for completing a proof in unrealizability logic. In Example 5.2 above, knowing the invariant $x \equiv_2 0 \wedge y \equiv_2 0$ resulted in a very simple proof tree. The algorithms implemented by external solvers (such as Spacer [Komuravelli et al. 2016] for MESSY, or the semilinear-set approach from Nay [Hu et al. 2020]) may be thought of as *proof strategies* for finding these hypotheses or invariants: they remain parametric of the logic itself, while providing critical information to complete the proof trees. Although not the focus of this paper, we hope that this view will allow future work in automating unrealizability logic to draw from a significant body of work in grammar-flow analysis [Möncke and Wilhelm 1991] and other program/constraint-solving techniques.

5.3 Working with Symbolic Examples and Auxiliary Variables

In §2.2, we saw that auxiliary variables are insufficient to model unrealizability. However, auxiliary variables can still be used to set the input examples of an unrealizability triple to be *symbolic*. A triple derived in unrealizability logic using symbolic examples indicates that the said triple must hold for *any instantiation* of the symbolic examples. In unrealizability logic, this can be used to avoid having to search for hard-to-find examples, instead completing a symbolic proof tree and checking at the end whether there are examples that can be used to show unrealizability.

Example 5.3 (Proof with Symbolic Variables). Recall the synthesis problem $sy_{\text{id\_ite}}$ from Example 5.1, where the goal is to synthesize a function that assigns to x the (initial) value of y. We will consider a variant of $sy_{\text{id\_ite}}$, named $sy_{\text{id\_const}}$, where the goal is the same but the supplied grammar $G_{\text{id\_const}}$ is different:

$$\begin{array}{lcl} \mathit{Start} & \to & \mathsf{if}\ B\ \mathsf{then}\ A\ \mathsf{else}\ \mathit{Start} \mid A \\ B & \to & y == E \\ E & \to & 0 \mid E + 1 \\ N & \to & 1 \mid 2 \mid \cdots \mid 999 \\ A & \to & x := N \end{array}$$

In $G_{\text{id\_const}}$, the nonterminal N may generate only a fixed set of integers from 1 to 999 (as opposed to G_{inf}, which could generate any positive integer). $sy_{\text{id\_const}}$ is unrealizable, and this time only requires one example to prove so: for example, $(x, y) = (-1, 1000)$. However, this specific example may be difficult to find, because it uses large constants. We show that one can avoid having to explicitly find this example by completing a proof tree in unrealizability logic using a single *symbolic example*—which may then be instantiated (perhaps using a constraint solver)—to find a concrete example for which $sy_{\text{id\_const}}$ is unrealizable.

Our goal this time is to prove the following triple (we again drop the vector-subscripts here as we only have one example):

$$\{\!|x = -1 \wedge y = y_{aux}|\!\}\ \mathit{Start}\ \{\!|x < 1000 \wedge y = y_{aux}|\!\}$$

This triple states that: starting from a *single* example $(x, y) = (-1, y_{aux})$, we will only be able to reach states in which $x < 1000$. Note that the given single example is made symbolic through the use of the auxiliary variable y_{aux}. This time, we will use the following hypothesis for *Start*:

$$\{\!|x < 1000 \wedge y = y_{aux}|\!\}\ \mathit{Start}\ \{\!|x < 1000 \wedge y = y_{aux}|\!\}$$

Denote the triple above as $\mathcal{I}$. In a similar process to the one used in Example 5.1, one can see that the hypothesis holds for the base case (assignment) (we omit the application of the HP rule):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ N\ \{\!|\exists e_t'.\mathcal{I}[e_t'/e_t] \wedge \exists k.0 < k < 1000 \wedge e_t = k|\!\}}\ \text{HP}}{\Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ N\ \{\!|\mathcal{I} \wedge \exists k.k < 1000 \wedge e_t = k|\!\}}\ \text{Weaken}}{\Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ A\ \{\!|\exists x'.\mathcal{I}[x'/x] \wedge \exists k.k < 1000 \wedge e_t = k \wedge x = e_t|\!\}}\ \text{Assign}}{\Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ A\ \{\!|\exists x'.\mathcal{I}[x'/x] \wedge x < 1000|\!\}}\ \text{Weaken}}{\Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ A\ \{\!|\mathcal{I}|\!\}}\ \text{Weaken}$$

And also for the inductive case, If-Then-Else (where we omit the reasoning for the premises):

$$\dfrac{\Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ B\ \{\!|\mathcal{I}|\!\} \quad \Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I} \wedge v_1 = v|\!\}\ A\ \{\!|\mathcal{I}|\!\} \quad \Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I} \wedge v_2 = v|\!\}\ \mathit{Start}\ \{\!|\mathcal{I}|\!\}}{\Gamma_{\mathcal{I}} \vdash \{\!|\mathcal{I}|\!\}\ \mathsf{if}\ B\ \mathsf{then}\ A\ \mathsf{else}\ \mathit{Start}\ \left\{\!\middle|\exists v_1, v_2, v_1', v_2'.\ \begin{array}{l}\mathcal{I}[v_1'[i]/v[i]\ \text{where}\ b_{t_1}[i] = \text{false}] \wedge \\ \mathcal{I}[v_2'[i]/v[i]\ \text{where}\ b_{t_2}[i] = \text{true}]\end{array} \wedge (v_1 = v_2)\middle|\!\right\}}\ \begin{array}{l}\text{ITE} \\ \\ \text{Weaken}\end{array}$$

Thus $\{\!|x < 1000 \wedge y = y_{aux}|\!\}\ \mathit{Start}\ \{\!|x < 1000 \wedge y = y_{aux}|\!\}$ is a valid triple.

At this point, observe that $x < 1000 \wedge y = y_{aux}$ does not immediately imply the negation of the specification, i.e., $x \neq y_{aux} \vee y \neq y_{aux}$. However, when setting $y_{aux} = 1000$, it does become clear that $x < 1000 \wedge y = y_{aux} \implies x \neq 1000 \vee y \neq 1000$.

To understand this more formally, recall that as shown in Example 2.4, the quantification for auxiliary variables happens *outside* the triple:

$$\forall y_{aux}.\{\!|x < 1000 \wedge y = y_{aux}|\!\}\ Start\ \{\!|x < 1000 \wedge y = y_{aux}|\!\}$$

This, in turn, means that the triple holds for all configurations of y_{aux}, *each of which constitutes a different example*. For synthesis, the program must work for *all* examples: thus it is sufficient that there *exists* an example for which the synthesis problem is unrealizable, i.e., the derived postcondition implies the negation of the specification. Thus to check unrealizability in this scenario where we have used an auxiliary variable, we can check whether the following formula is valid:

$$\exists y_{aux}.(x < 1000 \wedge y = y_{aux} \implies x \neq 1000 \vee y \neq 1000)$$

And this formula is certainly valid, as witnessed by $y_{aux} = 1000$. Thus $sy_{\text{id\_const}}$ is unrealizable.

In general, when using a specification with symbolic examples denoted using the variables $\mathbf{v}_{aux}$, one can check whether $\exists \mathbf{v}_{aux}.\, Q \implies \neg O$ is a valid formula, where Q is the derived postcondition and O denotes the desired postcondition of the synthesis problem. As shown in Example 5.3, the existential quantifier over $\mathbf{v}_{aux}$ asks whether there *exists* a concrete instantiation of the symbolic examples such that $Q \implies \neg O$. Although this approach will not be able to prove unrealizability if the supplied number of symbolic examples is *less* than the number of examples required to show unrealizability (as in Example 2.4), it will succeed in proving unrealizability if the supplied number of examples is sufficient. This can be very useful if the examples required to prove unrealizability are difficult to find, as in Example 5.3.

Note that it is still *sound* if one decides to drop the existential quantifier and simply ask whether $Q \implies \neg O$; adding the existential simply makes the final query more precise.

6 RELATED AND FUTURE WORK

Unrealizability. There has been limited work that focuses on proving the unrealizability of synthesis problems. Nay [Hu et al. 2020] and Nope [Hu et al. 2019] can prove unrealizability for syntax-guided synthesis (SyGuS) problems where the input grammar only contains expressions. Both Nay and Nope reduce an unrealizability problem to a program-verification problem, and present techniques for solving the reduced problem. Kamp and Philippsen [2021] use some of the ideas presented in Nope to design specialized unrealizability-checking algorithms for problems involving bit-vector arithmetic. When a grammar is not given as part of the input, CVC4 [Reynolds et al. 2015] is also capable of detecting unrealizability. These works only focus on expression-synthesis problems. MESSY [Kim et al. 2021] proposes a general algorithm for proving whether a given SemGuS problem is unrealizable. SemGuS is a general framework for specifying synthesis problems, which also allows one to define synthesis problems involving imperative constructs. MESSY is currently the only tool that can prove unrealizability for problems involving imperative programs. Farzan et al. [2022] have a technique for proving unrealizability, which they use as part of a synthesis technique; however, their technique is limited to a very specific class of functional programs and specifications.

While previous approaches all provide ways to solve variants of unrealizability problems, these approaches are embedded within a specific system that employs a fixed solver or algorithm. With the exception of Nay and its use of grammar-flow analysis, these tools do not produce a proof artifact that can be separately verified. For example, in MESSY, one is at the mercy of an external constraint solver, which makes it difficult for researchers to develop new solvers tailored towards

unrealizability, or even understand why certain problems can be proved unrealizable while others cannot. In contrast, unrealizability logic provides a general, logical system for (both human and machine-based) reasoning about unrealizability. While Nay can produce proof artifacts, it is limited to SyGuS problems over expressions; similar to what we showed in Example 5.2, the GFA algorithm of Nay may be understood as a proof strategy to discover hypotheses over nonterminals (where in this case, the assertion language is over semilinear sets).

Hyperproperties. The way we synchronize between multiple examples in unrealizability logic, using vector-states, is similar to the concept of *hyperproperties*, which are, in essence, sets of properties [Clarkson and Schneider 2010]. Hyperproperties are used to model, for example, k-safety properties, which are properties that should hold over k separate runs of a program [Sousa and Dillig 2016] (for example, transitivity is a 3-safety property).

There is a subtle but fundamental difference between properties in unrealizability logic and standard hyperproperties: properties in unrealizability logic are 'properties over vector-states'; that is, 'properties over (sets of states)', whereas standard hyperproperties are 'sets of (properties over states)'. The difference between the two is highlighted when considering nondeterminism: when verifying k-safety properties for a nondeterministic program, one will want to let different states execute on different paths. In contrast, unrealizability logic introduces vector-states to synchronize the paths—corresponding to one specific program within the set S—that each constituent state in a vector-state follows. For example, given a grammar $E \to x := x + 1 \mid x := x + 2$, the unrealizability triple $\{\!|x_1 = 0 \wedge x_2 = 10|\!\}\ E\ \{\!|(x_1 = 1 \wedge x_2 = 11) \vee (x_1 = 2 \wedge x_2 = 12)|\!\}$ is derivable. However, a standard hyperproperty approach would likely wish to treat the above triple as invalid (taking a nondeterministic-program interpretation of the different productions of E).

Hoare Logic for Recursive Procedures. The rules of unrealizability logic have much in common with the rules of Hoare logic extended towards recursive procedures. However, as discussed in §2.3, one requires many features to fully support the range of features in a synthesis problem; for example, nondeterminism, mutual recursion, both local and global variables, and infinite data structures. Combinations of some of these features have been studied previously, such as local variables and mutual recursion [Oheimb 1999], or nondeterminism and recursion [Nipkow 2002]. There is a vast amount of work on variants of Hoare logic; Apt and Olderog [2019] provides a survey of how the original Hoare logic [Hoare 1969] has evolved throughout the years.

Despite this body of work, we are unaware of a study of a system that contains *all* of the features listed above, and proves soundness, completeness, and decidability results as we have. Also as discussed in §2.3, even though one does have an extended Hoare logic, such a logic would hide the fact that we are dealing with synthesis problems and trying to prove unrealizability—whereas unrealizability logic takes advantage of this fact through rules like GrmDisj.

Supporting Nondeterministic Statements. As previously discussed in this section and §2.3, there is a similarity between program-synthesis problems and nondeterministic, recursive procedures that has been exploited in previous approaches to proving unrealizability [Hu et al. 2019]. A natural question that this similarity leads to is: can nondeterministic statements be supported in unrealizability logic as well?

In general, nondeterminism in program synthesis varies according to whether one takes an *angelic* interpretation, where the specification is considered met if there exists a nondeterministic execution of the synthesized program that satisfies the specification, or a *demonic* interpretation, where all executions of the synthesized program must meet the specification.

Supporting nondeterministic statements in unrealizability logic is simpler for angelic nondeterminism, in which case one can merely add a rule that generates the set of all possible vector-states generated by the nondeterministic statement. For example, given a statement $x := \mathsf{nondet}(V)$, which

nondeterministically assigns a value from the set V to x, a (simplified) rule for nondeterminism might be (where $|\mathbf{x}|$ denotes the length of the vector-state $\mathbf{x}$):

$$\frac{}{\Gamma \vdash \{\!|\mathcal{P}|\!\}\ x := \mathsf{nondet}(V)\ \{\!|\exists \mathbf{x}'.\mathcal{P}[\mathbf{x}'/\mathbf{x}] \wedge \mathbf{x} \in V^{|\mathbf{x}|}|\!\}}\ \textsc{Nondet}$$

The postcondition of the Nondet rule is an overapproximation of all states that may be generated by nondet. Thus, using Nondet in a proof tree in tandem with other unrealizability-logic rules will also derive an overapproximation of the set of reachable states; by showing that this set does not intersect with the desired specification, one can guarantee that the synthesis problem is unrealizable under the angelic interpretation.

However, Nondet is incomplete when used to prove unrealizability under a demonic interpretation: the demonic interpretation only requires that there *exists* a nondeterministic choice that fails to meet the specification, but a proof generated using Nondet will report unrealizability only when *all* possible nondeterministic choices fail to meet the specification. To prove unrealizability in the demonic sense, one would require a mechanism for reasoning about the set of produced vector-states simultaneously—for example, another level of vectors.

Unrealizability and Program Synthesis. As briefly discussed in §1, one of the ultimate goals of studying unrealizability is to *prune the search space* of a synthesis problem, by showing that certain subsets of the search space do not contain a desired solution. Although this work is the first to distill the fundamental concepts behind unrealizability into a logical system, there exist synthesizers that have already utilized the concept of pruning unrealizable parts of the search space to some extent: for example, Neo [Feng et al. 2018] discovers unrealizable subsets of the search space by analyzing conflicts, and avoids traversing these subsets during the search procedure.

We hope that the logical characterization of unrealizability provided in this paper will lay the foundation for future attempts to utilize unrealizability towards synthesizing programs.

7 CONCLUSION

We presented *unrealizability logic*, the first proof system for overapproximating the execution of an infinite set of programs. Unrealizability logic is both sound and relatively complete; it is also the first approach that allows one to prove unrealizability for synthesis problems that require infinitely many inputs to be proved unrealizable. We believe unrealizability logic will prove to be essential in further developments having to do with unrealizability.

A natural question that follows from this paper is the design of a *realizability logic*: "Can a similar logic be constructed for program synthesis, i.e., for proving *realizability*?" Because program synthesis requires a guarantee that a certain (vector-)state is *reachable*, one must devise suitable principles of *underapproximation* (like those discussed in reverse-Hoare (aka incorrectness) logic [de Vries and Koutavas 2011; O'Hearn 2019]) instead of overapproximation as used in this paper. We believe the results presented in this paper will also be useful in designing a *realizability logic*.

ACKNOWLEDGMENTS

Supported, in part, by a gift from Rajiv and Ritu Batra; by ONR under grant N00014-17-1-2889; by NSF under grants CCF-{1750965, 1763871, 1918211, 2023222, 2211968, 2212558}; by a Facebook Research Faculty Fellowship, by a Microsoft Research Faculty Fellowship, and a grant from the Korea Foundation of Advanced Studies. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

REFERENCES

Krzysztof R Apt. 1981. Ten years of Hoare's logic: A survey—Part I. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 3, 4 (1981), 431–483. https://doi.org/10.1145/357146.357150

Krzysztof R Apt and Ernst-Rüdiger Olderog. 2019. Fifty years of Hoare's logic. *Formal Aspects of Computing* 31, 6 (2019), 751–807. https://doi.org/10.1007/s00165-019-00501-3

Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210. https://doi.org/10.1109/CSF.2008.7

Martin Davis, Kurt Godel, and Stephen C Kleene. 1990. On Undecidable Propositions of Formal Mathematical Systems. PostscriptumIntroductory Note to 1934. *Journal of Symbolic Logic* 55, 1 (1990).

Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–171. https://doi.org/10.1007/978-3-642-24690-6_12

Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion synthesis with unrealizability witnesses. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 244–259. https://doi.org/10.1145/3519939.3523726

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices* 53, 4 (2018), 420–435. https://doi.org/10.1145/3192366.3192382

John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239. https://doi.org/10.1145/2737924.2737977

Robert W Floyd. 1993. Assigning meanings to programs. In *Program Verification*. Springer, 65–81. https://doi.org/10.1007/978-94-011-1793-7_4

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330. https://doi.org/10.1145/1926385.1926423

Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. 2019. Proving unrealizability for syntax-guided synthesis. In *International Conference on Computer Aided Verification*. Springer, 335–352. https://doi.org/10.1007/978-3-030-25540-4_18

Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. 2020. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1128–1142. https://doi.org/10.1145/3385412.3385979

Qinheping Hu and Loris D'Antoni. 2018. Syntax-guided synthesis with quantitative syntactic objectives. In *International Conference on Computer Aided Verification*. Springer, 386–403. https://doi.org/10.1007/978-3-319-96145-3_21

Marius Kamp and Michael Philippsen. 2021. Approximate Bit Dependency Analysis to Identify Program Synthesis Problems as Infeasible. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12597)*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer, 353–375. https://doi.org/10.1007/978-3-030-67067-2_16

Jinwoo Kim, Loris D'Antoni, and Thomas Reps. 2022. Unrealizability Logic. *arXiv preprint arXiv:2211.07117* (2022).

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. 2021. Semantics-guided synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–32. https://doi.org/10.1145/3410258

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods in System Design* 48, 3 (2016), 175–205. https://doi.org/10.1007/978-3-319-08867-9_2

John McCarthy. 1993. Towards a mathematical science of computation. In *Program Verification*. Springer, 35–56. https://doi.org/10.1007/978-94-011-1793-7_2

Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic execution with existential second-order constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 389–399. https://doi.org/10.1145/3236024.3236049

Ulrich Möncke and Reinhard Wilhelm. 1991. Grammar Flow Analysis. In *Attribute Grammars, Applications and Systems, International Summer School SAGA, Prague, Czechoslovakia, June 4-13, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 545)*, Henk Alblas and Borivoj Melichar (Eds.). Springer, 151–186. https://doi.org/10.1007/3-540-54572-7_6

Tobias Nipkow. 2002. Hoare logics for recursive procedures and unbounded nondeterminism. In *International Workshop on Computer Science Logic*. Springer, 103–119. https://doi.org/10.1007/3-540-45793-3_8

Peter W O'Hearn. 2019. Incorrectness logic. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32. https://doi.org/10.1145/3371078

David von Oheimb. 1999. Hoare logic for mutual recursion and local variables. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 168–180. https://doi.org/10.1007/3-540-46691-6_13

Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. Swizzle inventor: data movement synthesis for GPU kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 65–78. https://doi.org/10.1145/3297858.3304059

Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. 2015. Counterexample-guided quantifier instantiation for synthesis in SMT. In *International Conference on Computer Aided Verification*. Springer, 198–216. https://doi.org/10.1007/978-3-319-21668-3_12

Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 57–69. https://doi.org/10.1145/2908080.2908092

Glynn Winskel. 1993. *The formal semantics of programming languages: an introduction*. MIT press.

Received 2022-07-07; accepted 2022-11-07