Automating Unrealizability Logic: Hoare-Style Proof Synthesis for Infinite Sets of Programs

SHAAN NAGY, University of Wisconsin-Madison, USA
JINWOO KIM, Seoul National University, Republic of Korea
THOMAS REPS, University of Wisconsin-Madison, USA
LORIS D'ANTONI, University of California San Diego, USA

Automated verification of all members of a (potentially infinite) set of programs has the potential to be useful in program synthesis, as well as in verification of dynamically loaded code, concurrent code, and language properties. Existing techniques for verification of sets of programs are limited in scope and unable to create or use interpretable or reusable information about sets of programs. The consequence is that one cannot learn anything from one verification problem that can be used in another. Unrealizability Logic (UL), proposed by Kim et al. as the first Hoare-style proof system to prove properties over sets of programs (defined by a regular tree grammar), presents a theoretical framework that can express and use reusable insight. In particular, UL features *nonterminal summaries*—inductive facts that characterize recursive nonterminals (analogous to procedure summaries in Hoare logic). In this work, we design the first UL proof synthesis algorithm, implemented as Wuldo. Specifically, we decouple the problem of deciding how to apply UL rules from the problem of synthesizing/checking nonterminal summaries by computing proof structure in a fully syntax-directed fashion. We show that Wuldo, when provided nonterminal summaries, can express and prove verification problems beyond the reach of existing tools, including establishing how infinitely many programs behave on infinitely many inputs. In some cases, Wuldo can even synthesize the necessary nonterminal summaries. Moreover, Wuldo can reuse previously proven nonterminal summaries across verification queries, making verification 1.96 times as fast as when summaries are instead proven from scratch.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Program verification**; **Logic and verification**; **Automated reasoning**; **Hoare logic**.

Additional Key Words and Phrases: Unrealizability logic, automated reasoning, infinite sets of programs

ACM Reference Format:
Shaan Nagy, Jinwoo Kim, Thomas Reps, and Loris D'Antoni. 2024. Automating Unrealizability Logic: Hoare-Style Proof Synthesis for Infinite Sets of Programs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 275 (October 2024), 27 pages. https://doi.org/10.1145/3689715

1 Introduction

In program verification, one typically verifies a *single program*. However, there are settings in which one would wish to verify a possibly *infinite set* of programs (i.e., a language). That is, given some program property ϕ and a set of programs S, one wants to show $\forall s \in S.\ \phi(s)$. When reasoning about dynamically-loaded code (e.g., code run via `Eval` in JavaScript and Python), analyzing a single program is insufficient because the program changes as it executes [4]—the problem is better

Authors' Contact Information: Shaan Nagy, University of Wisconsin-Madison, USA, sanagy@wisc.edu; Jinwoo Kim, Seoul National University, Republic of Korea, jinwoo.kim@sf.snu.ac.kr; Thomas Reps, University of Wisconsin-Madison, USA, reps@cs.wisc.edu; Loris D'Antoni, University of California San Diego, USA, ldantoni@ucsd.edu.

ACM 2475-1421/2024/10-ART275
https://doi.org/10.1145/3689715

treated as analysis of a set of programs. Verification of safety properties of *all* programs in a given programming language (or subset of a language) also falls into the same paradigm.

Pruning in Enumerative Synthesis. As an illustrative example, consider the task of pruning in enumerative synthesis. Enumerative synthesis is an approach to program synthesis in which one enumerates programs in a search space (represented by a grammar) until a satisfying program is found. Because the search space is often large, one hopes to identify and discard/prune regions in which no solution can be found. Consider the following simple synthesis problem:

Example 1.1. Given the following grammar, which describes $L(S)$, the set of programs derivable from S, synthesize a program $s \in L(S)$ that always sets x to 5 (i.e., satisfies the specification property $\phi(s) = \forall\sigma.\ [\![s]\!](\sigma)[x] = 5$):

$$S \mapsto x := E \mid x := E + 1 \mid x := E + 2 \qquad\qquad E \mapsto 0 \mid E + 3$$

To find such a program s, an enumerative synthesizer considers the programs that can be generated from S, one by one. An efficient synthesizer will observe that no program generated from $x := E$ or $x := E + 1$ will satisfy the specification. This observation holds because E only generates natural numbers divisible by 3 ($\forall e \in L(E).\ e = 0 \pmod 3$). We say that $e = 0 \pmod 3$ is a "nonterminal summary" of E, denoted by Q_E, because it is a formula true of every $e \in E$ (and captures important semantic information about E). Thus, $x := E$ sets x to a multiple of 3, and $x := E + 1$ sets x to a natural number equal to 1 modulo 3. However, our specification demands that x be set to 5, which equals 2 modulo 3. So, an efficient synthesizer can immediately discard/prune the two productions $x := E$ and $x := E + 1$, reducing the search space substantially.

To prune these productions, one must prove that the specification is false on every program in $L(x := E)$ or $L(x := E + 1)$—i.e. one must prove that (*i*) $\forall s \in L(x := E).\ \exists\sigma.\ [\![s]\!](\sigma)[x] \neq 5$, and (*ii*) $\forall s \in L(x := E + 1).\ \exists\sigma.\ [\![s]\!](\sigma)[x] \neq 5$ hold.

When we prove such properties, we say that we are proving that a synthesis (sub-)problem is *unrealizable*. Such unrealizability verification problems ($\forall s \in S.\ \neg\phi(s)$) are expressible in a logic for verification of sets of programs (addressing verification problems of the form $\forall s \in S.\ \psi(s)$) as long as the logic in which properties such as ϕ and ψ are expressed is closed under complement.

This paper presents a technique for automatically verifying a property of a set of programs. The main vein of prior related work is on proofs of unrealizability; thus, our benchmarks and comparisons—and many of our examples—are posed as unrealizability problems.

Limitations of Existing Work. Existing unrealizability techniques [12, 13, 17] would reduce the query $\forall s \in L(x := E).\neg\phi(s)$ directly to a problem for a third-party verifier (e.g., a CHC solver), assuming ϕ and $L(x := E)$ are suited to the technique's capabilities. However, this closed-box use of external verifiers means that, beyond a yes/no answer to the question of unrealizability, it is very difficult to learn anything about the structure of $L(x := E)$ or $L(E)$. Moreover, we have no way to apply knowledge of $x := E$ or E we may have on hand. In other words, when the query $\forall s \in L(x := E + 1).\ \exists\sigma.\ [\![s]\!](\sigma)[x] \neq 5$ is posed, we cannot use anything we have learned from the previous query to help us answer it, even though the two queries share a common element.

Instead, a more efficient technique would discover the summary Q_E when checking unrealizability of the first template $x := E$ (i.e., $\forall s \in L(x := E).\ \exists\sigma.\ [\![s]\!](\sigma)[x] \neq 5$). Then, when proving $\forall s \in L(x := E + 1).\ \exists\sigma.\ [\![s]\!](\sigma)[x] \neq 5$, the summary Q_E could be reused without having to rediscover or reprove it. The net effect could be a faster decision for checking unrealizability of the second template $x := E + 1$ (i.e., $\forall s \in L(x := E + 1).\ \exists\sigma.\ [\![s]\!](\sigma)[x] \neq 5$), resulting in faster overall synthesis. As an added benefit, discovering Q_E would provide insight into the cause of a query's truth/falsehood. Furthermore, users would be able to manually provide summaries like Q_E to help the technique answer hard queries on which existing techniques fail.

There are many applications beyond program synthesis for which such reusable summaries would be useful. If one is verifying properties of programs with a common function that dynamically loads external code, having a summary over the set of possible external functions would save the work of reverifying properties of the function for each program. Similarly, if one has a language for which one would like to verify multiple safety properties (e.g., x cannot fall below a certain value and x cannot exceed a certain value), then it can be helpful to reuse summaries learned while verifying one property to verify the next. Finally, in any context in which one must verify a set of programs, it is helpful to be able to change some small feature of that set (e.g., fixing a bug in a program featuring dynamically-loaded code) but reuse summaries from the original set.

Summaries via Unrealizability Logic. In this paper, we propose an automated proof synthesizer for verification of sets of programs that is based on a preexisting proof system, Unrealizability Logic (UL), and can take advantage of the aforementioned summaries [16]. UL is a *sound and* (for loop-free programs) *relatively-complete*[1] Hoare-style *strongest-postcondition* proof system for reasoning about sets of programs, where the set of programs we want to verify is expressed with a regular-tree grammar (RTG)—i.e., the most common formalism for describing a set of programs.

UL is based on the unfortunately named *unrealizability triple* $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$, which states that for *every* individual program $s \in S$, the Hoare triple $\{P\}\ s\ \{Q\}$ holds—essentially lifting Hoare logic to sets of programs. These triples capture verification-style properties of sets of programs, including "infinite" hyperproperties (those which require infinitely many examples to falsify) and properties $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ where infinitely many states satisfy P, both of which are beyond the reach of existing automated unrealizability techniques. With some cleverness, these triples can express unrealizability properties as well (§ 2.2). For example, when P admits only one state, $\nexists s \in S.\{P\}s\{Q\}$ is exactly $\{\!|P|\!\}\ S\ \{\!|\neg Q|\!\}$.

The key ingredient of UL that allows reasoning about infinite sets of programs is the notion of *nonterminal summaries*—inductive facts of the form $\{\!|P_N|\!\}\ N\ \{\!|Q_N|\!\}$ that characterize a recursively-defined nonterminal N in the grammar defining the set of programs S. Nonterminal summaries are analogous to procedure summaries in Hoare logic for recursive procedures. These summaries let us capture facts like $\{\!|\mathit{true}|\!\}\ E\ \{\!|e_t = 0 \pmod 3|\!\}$ in our running example.

Proof Derivation in Unrealizability Logic. While UL seems well-suited to taking advantage of reusable summaries, there is no existing automation of proof construction in the logic. The stumbling block seems to be that there is no obvious, efficient way to attain either of the following objectives: given a target triple $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ and summaries, construct a UL proof of $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ (i.e., *proof derivation from summaries*), or given only a target triple $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$, synthesize the necessary summaries and construct a UL proof of $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ (i.e., *whole cloth proof derivation*). The key difficulty is the entangling of proof search and nonterminal-summary/loop-invariant search. To construct a valid proof, one needs to decide both the order in which proof rules should be applied and guess the summaries/invariants to be used. These two factors are interdependent, so the total search space is the product of the two spaces separately. Even for proof derivation from summaries, the number of possible proofs and the cost of verification makes search expensive.

Our key insight to solve these proof derivation objectives is that (for a variant of UL), given a set of programs S, a feasible "proof skeleton" (i.e., the order in which inference rules are applied) can be computed in a fully syntax-directed fashion (§ 4). This insight eliminates the complexity of searching for proof structures altogether. From the proof skeleton, we can extract parametric verification

[1] In the course of this work, we discovered an error in the original paper that makes S-UL incomplete for program containing loops. In this paper, we observe that the complexity of a "relatively complete" While rule would likely be quite high, and we give an alternative, sound but incomplete rule instead. A highly complex, "relatively complete" noncompositional rule has been submitted as a separate corrigendum [24], but this detail is irrelevant to the present paper.

conditions (PVCs)—verification conditions parametric in P, Q, nonterminal summaries, and loop invariants. With these PVCs in hand, proof derivation from summaries and whole cloth proof derivation are reduced to summary/invariant checking and summary/invariant synthesis, respectively. For proof derivation from summaries, we can plug P, Q, and the provided summaries/invariants into our PVCs to yield checkable verification conditions—a task that can be offloaded to existing constraint solvers. For whole cloth proof derivation, we instead synthesize summaries/invariants that will satisfy our PVCs for the given P and Q—a task that can be offloaded to existing SyGuS synthesizers [1]. In short, the novelty in this work is that the insight about syntax-directed proof structure and proof skeletons let us create the first ever UL proof synthesizer, a demonstrably more general and transparent technique for unrealizability and verification of sets of programs than all existing techniques of which we are aware.

Contributions. The contributions of our work are as follows:

- A weakest-precondition version of UL (W-UL) that is amenable to our strategy for syntax-directed creation of a proof skeleton and preserves the loop-free proving power (see Footnote 1) of UL (§3). (The version of UL proposed by Kim et al. [16] is a strongest-postcondition logic, which we refer to as S-UL.)
- A syntax-directed algorithm that, given a set of programs S, constructs a "proof skeleton," which can prove any true triple $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ given sufficient nonterminal summaries and loop invariants. If summaries and invariants are provided, the ability to create a proof skeleton reduces proof derivation from summaries to checking verification conditions (§4).
- An algorithm that, given an unrealizability triple $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ and the proof skeleton of the previous contribution, generates a Syntax-Guided Synthesis (SyGuS) problem to find summaries/invariants (i.e., whole cloth proof derivation). We prove that the SyGuS problem is satisfiable if and only if a proof of $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ exists (§5).
- We implement our algorithms in an automated proof deriver Wuldo. Wuldo can derive proofs from summaries for many problems beyond existing tools and can synthesize summaries in some cases (§6).

§7 discusses related work. §8 concludes.

2 Overview

In this section, we illustrate via a simple example (§2.1) how one can write a proof of unrealizability in our new logic W-UL (§2.2 and §2.3); how W-UL allows verification conditions to be generated automatically from nonterminal summaries (§2.4); and how such conditions can be used to phrase the problem of synthesizing nonterminal summaries as a syntax-guided-synthesis problem (§2.5).

2.1 An Unrealizable Synthesis Problem

Suppose that our goal is to synthesize an imperative program that operates over two variables x and y, such that when the program terminates, both variables hold the same value. Formally, we want a program s that satisfies the Hoare triple $\{true\}\ s\ \{x = y\}$.

It is trivial to write such a program in general (e.g., $x := y$), but let us assume for the sake of illustrating unrealizability that we are bound to the rules of the following example.

Example 2.1. Synthesize a program in the following grammar BD from the start symbol S that has the same semantics as $x := y$:[2]

$$S \mapsto \text{if } B \text{ then } A \text{ else } S \mid A \qquad B \mapsto y == N$$
$$N \mapsto 0 \mid N + 1 \qquad A \mapsto x := N$$

The grammar BD describes programs that contain arbitrarily many if-then-else operations, but where each subprogram is only allowed to assign a constant to x—e.g., programs such as "(if $y == 5$ then $x := 5$ else (if $y == 3$ then $x := 3$ else $x := 2$))." Because of the restricted form of this grammar, any program in $L(S)$ (the set of programs derivable from S, sometimes abbreviated as just S) can only set the variable x to a finite set of values—e.g., the set $\{5, 3, 2\}$ for our example. Equivalently, for a given program in S, the final value of x is bounded. This property implies that no program in S satisfies our target specification; that is, this synthesis problem is unrealizable.

Existing automated tools [12, 13, 17] cannot prove that this particular synthesis problem is unrealizable because a proof requires infinitely many examples. For any specific finite set of inputs $\{i_1, \ldots, i_n\}$, one can construct a program in S which hardcodes the correct outputs on the input set.

2.2 Unrealizability Logic

An unrealizability triple takes the form $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ where P and Q are logical formulas and S denotes a set of programs encoded by a regular tree grammar. Such a triple is said to hold if, for all programs $s \in S$, the Hoare triple $\{P\}\ s\ \{Q\}$ holds in the sense of partial correctness (i.e., when s executes any input state satisfying P, if the program terminates, then the output state satisfies Q).

In UL, the pre- and postconditions P and Q are often formulas over vectors of states —i.e., $\sigma[]$—rather than individual states—i.e., σ. Rather than write such vector states directly, we often represent $\sigma[]$ with vectors giving the value of each relevant variable in each state —i.e., $x[]$ and $y[]$ where, for example, $x[5]$ gives the value of x in the state $\sigma[5]$.

This "vectorized" semantics lets programs act on a vector of states simultaneously (i.e., by executing each program on all states in the vector at once), allowing one to reason about the behavior of programs on an infinite collection of examples (e.g., hyperproperties).

Putting this all together, we can write the following UL triple that contradicts realizability of the synthesis problem in Example 2.1. Thus, proving the following triple will prove that Example 2.1 is unrealizable.

$$\{\!|\forall i.x[i] = 0 \wedge y[i] = i|\!\}\ S\ \{\!|\exists i.x[i] \neq y[i]|\!\} \tag{1}$$

The triple states that every program s in S is incorrect for at least one input state—i.e., there exists an input i such that executing s on $x = 0$ and $y = i$ results in a state where $x \neq y$. Vector states capture this condition by considering all possible inputs i at once.

In general, an unrealizability query $(\forall s \in S.\neg\phi(s))$ can always be converted into a corresponding UL triple by using vector-states to represent the state space. A relational specification ϕ can be written as a predicate on the input-output relation induced by the program—i.e., $\phi(s) = Q(\{(\sigma, [\![s]\!](\sigma)) \mid \sigma \in \mathit{State}\})$. Then $\neg\phi(s) = \neg Q(\{(\sigma, [\![s]\!](\sigma)) \mid \sigma \in \mathit{State}\})$. To write the right-hand side as a UL triple, we first encode the relation as a (countably) infinite vector: writing the relation $\{(\sigma, [\![s]\!](\sigma)) \mid \sigma \in \mathit{State}\}$ as the vector $[(\sigma_1, [\![s]\!](\sigma_1)), (\sigma_2, [\![s]\!](\sigma_2)), \cdots]$ for a fixed enumeration $\sigma_1, \sigma_2, \cdots$ of State. Now, because the σ_j are constants, Q depends only on $[[\![s]\!](\sigma_1), [\![s]\!](\sigma_2), \cdots]$. Unrealizability of ϕ over S is exactly the triple over infinite vector-states $\{\!|P|\!\}\ S\ \{\!|\neg Q'(x)|\!\}$, where P is the predicate $\forall j.x_j = \sigma_j$ and Q' is the aforementioned

[2]Restricted grammars are often used to restrict the search space of a synthesis problem [18] and can also emerge when optimizing syntactic quantitative objectives in a synthesis problem [14].

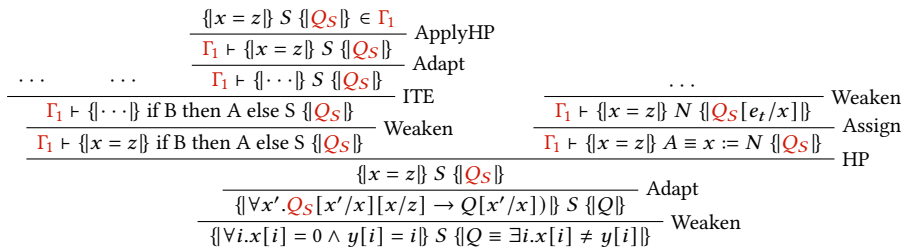

Fig. 1. Proof tree for Example 2.1 where $Q_S = \exists n.\forall i.x[i] < n$, and $\Gamma_1 = \{\{\!|x = z|\!\}\ S\ \{\!|Q_S|\!\}\}$ is a *context* that contains our inductive fact about S for later use. Our automation algorithms can derive all terms in black when the pre and postconditions in red—i.e., the nonterminal summaries—are provided. In some cases, even the summaries in red can be synthesized.

predicate on $[[\![s]\!](\sigma_1), [\![s]\!](\sigma_2), \cdots]$. For many specifications ϕ (e.g., input-output examples), Q only looks at a small part of the relation. For input-output examples specifically (which encompasses all benchmarks from prior work, as discussed in §6), we follow this general strategy over only the part of the relation containing our examples. Specifically, given examples $(in_1, out_1), \cdots, (in_n, out_n)$, we can say that no program in S satisfies all examples by writing $\{\!|x = [in_1, \cdots, in_n]|\!\}\ S\ \{\!|x \neq [out_1, \cdots, out_n]|\!\}$, as in Kim et al. [16].

2.3 Writing Proofs in Unrealizability Logic

Before describing proofs in W-UL, our weakest-precondition UL variant, we introduce some notation. We write judgements of the form $\Gamma \vdash \{\!|P|\!\}\ S\ \{\!|Q|\!\}$, where the *context* Γ is a set of unrealizability triples, to denote that, assuming all triples in Γ hold, $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ holds as well.

The concept of a context is essential when performing an inductive argument in UL to show that all programs in a grammar satisfy some triple.

Inference rules in W-UL, such as Seq below, yield conclusions from 0 or more hypotheses:

$$\frac{\Gamma \vdash \{\!|P|\!\}\ S_1\ \{\!|R|\!\} \qquad \Gamma \vdash \{\!|R|\!\}\ S_2\ \{\!|Q|\!\}}{\Gamma \vdash \{\!|P|\!\}\ S_1; S_2\ \{\!|Q|\!\}}\ \text{Seq}$$

A proof is a tree of inference rules in which all hypotheses are proven. Figure 1 illustrates (a sketch of) a W-UL proof tree for the triple in Eqn. (1). The meaning of each rule is covered in §3 but is not important here.

To prove hypotheses about recursive nonterminals—i.e., properties that are true for all programs derivable from a nonterminal—we need formulas called *nonterminal summaries*, which describe the nonterminals' behaviors. Like loop invariants, these formulas typically require clever insights. In our example, the summary $Q_s \triangleq \exists n.\forall i.x[i] < n$ appearing in $\{\!|x = z|\!\}\ S\ \{\!|\exists n.\forall i.x[i] < n|\!\}$ will be handy. This summary captures a very general property of S that can be used across many proofs over the set. One can see that proving the summary Q_S constitutes the bulk of the proof. This example further illustrates the utility of reusing summaries as described in §1.

2.4 Deriving Proof Structure Without Nonterminal Summaries

In the previous section, we showed a complete proof tree in Unrealizability Logic. We now discuss the main contribution of the paper—how such trees can be constructed *automatically* in W-UL. Specifically, in this section, we will focus on the problem of first constructing a *proof skeleton*—e.g., the derivation tree in Figure 1 but where all the summaries in red are left unspecified, replaced instead with function symbols over an appropriate set of variables (e.g., $Q_S(a, b, c, d)$). Combining

such a proof skeleton with a method for synthesizing summaries (described in §2.5) then yields an algorithm for synthesizing proof trees such as Figure 1. This algorithm is relatively complete (i.e., complete up to completeness of the underlying logic) in the sense of Corollary 4.5.

The ability to consider the structure of the proof tree *separately* from the nonterminal summaries is a key part of proof automation using W-UL. In W-UL, the derivation of a proof structure is fully syntax-directed and *independent* of the specific summaries—-i.e., by simply looking at the set of programs in the center of the UL triple, we can determine what inference rules must be applied to build a valid proof, as noted in §2.3 and discussed in detail in §4. To construct a proof skeleton, we adopt a weakest-precondition approach, substituting Q in the skeleton and deriving conditions during an Euler tour that visits right-children before left children, based on syntax-directed applications of W-UL rules. The key rule that makes this derivation possible is a revised version of the little-known rule of adaptation used in procedural program verification [11, 22].

The only rule in the proof tree that requires us to reason about predicates that are not syntactically derivable is the Weaken inference rule—which allows tightening of preconditions and loosening of postconditions. The Weaken rule lets one establish that certain conclusions hold when we have proven a stronger claim. Specifically, Weaken is the only rule that imposes logical constraints on its conclusion and hypothesis; all other rules in W-UL impose purely syntactic constraints. These Weaken applications give constraints on our invariants and summaries. In the example from §2.3, the constraint on Q_S from the lowest Weaken in the tree would be $(\forall i.x[i] = 0 \wedge y[i] = i) \rightarrow (\forall x'.Q_S(x', x, y) \rightarrow Q[x'/x])$. If these verification conditions are valid, the summaries are powerful enough to complete this proof. Otherwise, the proof fails.

For this problem, our tool Wuldo generates a complete proof in under 80 seconds using the theorem prover Vampire [3] when Q_S is given as above.

2.5 Synthesizing Summaries in W-UL

Our work allows the automatic construction of W-UL proofs from invariants and summaries.

However, because our work provides us with an explicit set of constraints on the elements that are missing from verification conditions, we can move beyond checking proofs and actually synthesize nonterminal summaries and loop invariants—thereby synthesizing a complete W-UL proof! In particular, discovering the nonterminal summaries and loop invariant that are needed can be formulated as a SyGuS problem [1] (as discussed in §5). All we need to do is to specify, for each invariant/summary, a grammar over which the search is to be carried out.

For our running example, we might specify a grammar G for $Q_S(x, y, z)$ that generates all existentially quantified formulas over the language of integer arithmetic with infinite arrays. If we define $\psi(Q_S)$ to be the conjunction of our verification conditions, then the SyGuS problem is simply written as $(G, \psi(Q_S))$.

To the best of the authors' knowledge, no SyGuS synthesizers over the theory of infinite arrays exist. However, for triples that do not require infinite vector-states, one can dispatch the corresponding SyGuS problem to a solver and hope to obtain a summary. In practice, this approach is feasible when one can give a tight enough grammar. Our experiments suggest that, under ideal conditions, synthesis can be completed within a couple of seconds (see Table 2 in §6).

3 Weakest-Precondition Unrealizability Logic

Recall that our goal is to prove triples of the form $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ where S is a partial program over a regular tree grammar (RTG). An RTG is a grammar with constructors of various arities. A partial program over G is a set of programs denoted by a program in which some terms are nonterminals from G. Moving forward, all sets of programs will be assumed to be partial programs over an RTG as formalized in [16]. Intuitively, an RTG is like a context-free grammar (CFG) except in that it

Statement	$S ::= x := E \mid S; S \mid \text{if } B \text{ then } S \text{ else } S \mid \text{while } B \text{ do } S$
IntExpr	$E ::= 0 \mid 1 \mid 2 \mid \cdots \mid x \mid E + E$
BoolExpr	$B ::= \text{true} \mid \text{false} \mid \neg B \mid B \wedge B \mid E < E \mid E == E$

Fig. 2. The base grammar G_{imp} which defines the terms over which our inference rules operate.

explicitly accepts parse trees rather than strings, making compositional reasoning of the sort UL requires much easier.

Definition 3.1 (Unrealizability Triple). An *unrealizability triple* is a triple $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ where P and Q are predicates, and S is a partial program in some regular tree grammar (RTG) over G_{imp} (Figure 2) describing a set of programs. Such a triple is said to hold if, for all programs $s \in S$, the Hoare triple $\{P\}\ s\ \{Q\}$ holds in the sense of partial correctness (i.e., $\{\!|P|\!\}\ S\ \{\!|Q|\!\} \triangleq \forall s \in S.\{P\}\ s\ \{Q\}$).[3]

We write the claim that a triple $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ is derivable in W-UL from a *context* (i.e., a set of triples we assume to be true) Γ as $\Gamma \vdash \{\!|P|\!\}\ S\ \{\!|Q|\!\}$. Contexts typically contain inductive facts we require for reasoning about the behavior of self-referential (recursive) nonterminals.

To prove such claims, W-UL provides the set of inference rules given in Figure 3. W-UL has four kinds of inference rules: (*i*) rules for expressions, (*ii*) rules for statements, (*iii*) rules for nonterminals (sets of programs), and (*iv*) structural rules, which do not correspond to a specific program or grammar construct but allow one to manipulate the pre- and postconditions of a triple. With these inference rules, W-UL is *sound* in general and *relatively complete* over sets of loop-free programs.

Theorem 3.2 (Soundness). *For any predicate P, Q, and set of programs S, $\emptyset \vdash \{\!|P|\!\}\ S\ \{\!|Q|\!\} \implies \forall s \in S.\{P\}\ s\ \{Q\}$ in the sense of partial correctness for every $s \in S$.*

Theorem 3.3 (Relative Completeness for Sets of Loop-Free Programs). *For any predicate P, Q, and set of programs S, if S is loop-free, then $\forall s \in S.\{P\}\ s\ \{Q\} \implies \emptyset \vdash \{\!|P|\!\}\ S\ \{\!|Q|\!\}$.*

The proofs of Theorems 3.2 and 3.3 can be found in Appendix A of the extended version of the paper [20]. In this section, we present an intuitive reading of the inference rules for our weakest-precondition formulation of Unrealizability Logic W-UL. An example proof is given in Figure 1.

Remark. *The precondition P and postcondition Q of a triple $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ are sometimes formulas over vectors of states—i.e., $x[]$ and $y[]$—rather than individual states—i.e., x and y, in which case the semantics of each program s are adjusted so that s is applied to each state simultaneously. S-UL, the original Unrealizability Logic, introduced these* vector-states *to synchronize between multiple examples given to a synthesis problem. The rules listed in Figure 3 are for the single-state situation only. In practice (e.g., our implementation in §6), W-UL also has a set of rules extended to support vector-states. Since supporting vector-states is not the main contribution of this paper, we present only the single-state form of the rules. We give vectorized versions of the inference rules in Appendix B of the extended version of the paper [20].*

3.1 Rules for Expressions and Statements

3.1.1 Rules for Expressions. Our rules for expressions—that is, constants, right-hand-side variables, and binary operations, are listed in the top box Rules for Expressions in Figure 3.

[3]We slightly modify the meaning of a Hoare triple here so that the value of the last evaluated integer expression is stored in a variable e_t and the value of the last evaluated Boolean expression is stored in a variable b_t (see §3.1 for a more detailed explanation). Thus, Hoare triples can reason about the evaluation of expressions with the semantics introduced in [16].

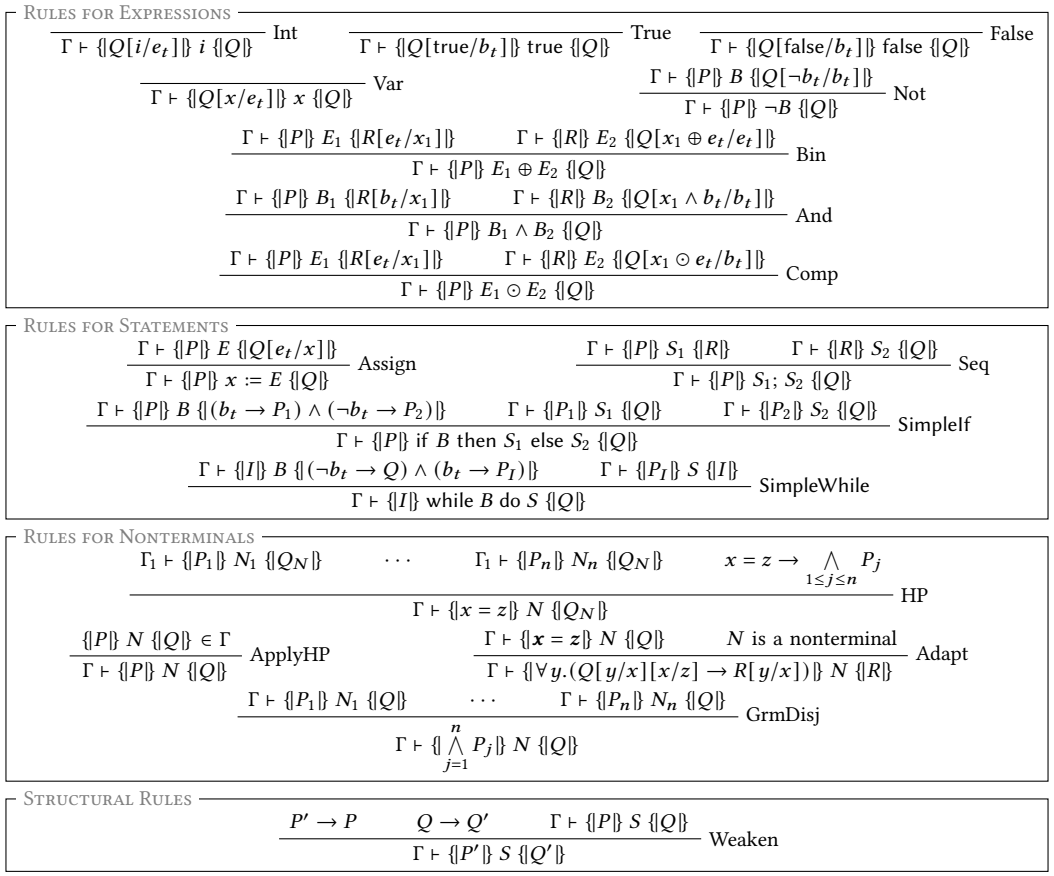

Fig. 3. W-UL Inference Rules. Note that the context Γ is a collection of triples used as inductive hypotheses. In the HP rule, N is a nonterminal with productions $N_1, \cdots, N_n$.

Before discussing the rules for expressions, it is useful to understand what happens when expressions appear in assignments. In Hoare logic, in the rule for assignments, expressions are directly substituted into formulas, so for example $\{Q[e/x]\}x := e\{Q\}$ is taken as an axiom for all e. In Unrealizability Logic, an assignment $x := E$ corresponds to a (potentially infinite) set of assignments $x := e$ for each $e \in E$. To represent these expressions in the language of our logical conditions, we introduce two reserved variables (e_t for integer-valued expressions and b_t for Boolean-valued expressions) into which expressions' values are stored when they are evaluated. For example, a program $x := 5$ can be thought of instead as $e_t := 5; x := e_t$. This follows the convention of [16].

The rules to handle constants and right-hand-side occurrences of variables simply substitute the constant/variable for e_t or b_t in the postcondition to derive the precondition.

The inference rules for unary operators transform the postcondition of the conclusion into the postcondition of the hypothesis by applying the unary operation to all occurrences of the appropriate expression variable (e_t or b_t)—e.g., see Not in Figure 3, where b_t is replaced with $\neg b_t$.

Binary operators are more complicated. To avoid managing multiple copies of e_t, we consider an expression $E_1 \oplus E_2$ as the partial program $x_1 := E_1; e_t := x_1 \oplus E_2$, where x_1 is a fresh variable. This approach lets us hold the value of the left term in an unreferenced variable when we evaluate the

right term. Suppose we specify E_1, E_2, and Q and want to derive a triple of the form $\{\!|P|\!\}\ E_1 \oplus E_2\ \{\!|Q|\!\}$. We first derive R as the weakest precondition of $Q[x_1 \oplus e_t/e_t]$ with respect to E_2. Then R gives exactly the necessary constraints on x_1 for Q to hold on $x_1 \oplus e_2$ for any $e_2 \in E_2$. Having obtained R, we find P as the weakest precondition of $R[e_t/x_1]$ with respect to E_1. Then P is exactly the necessary constraint so that Q holds on $e_1 \oplus e_2$ for any $e_1 \in E_1$ and $e_2 \in E_2$.

3.1.2 Rules for Statements. The rules for statements in G_{imp} are listed in the second box of Figure 3. Rules for Assign and Seq mirror those from traditional Hoare logic. However, the rules for statements with control flow (conditionals and while loops) diverge from their Hoare-logic counterparts because the guards may be sets of expressions.

If-Then-Else is straightforward when handling only a single state, as in the rule SimpleIf presented in Figure 3. If the poststate is to satisfy Q, then it must do so after passing through either the then branch or the else branch. We generate preconditions P_1 and P_2 sufficient to guarantee Q on each branch. Then we demand that, for every acceptable input, the guard sends it to a branch through which it can satisfy Q. When handling vector-states, one must account for the possibility that states run through different branches; we show how to do so in Appendix B of the extended version of the paper [20].

For loops, the single-state SimpleWhile rule is given in Figure 3. The intuition is that, for I to be a loop invariant, whenever B evaluates to true ($b_t = \mathit{true}$) on a prestate in I, we should get a poststate satisfying a condition P_I sufficient to recover I after running any program in S. When B is false on such a prestate, we must recover the desired postcondition Q.

Though straightforward, SimpleWhile is imprecise, overapproximating the semantics of sets of loops. Like S-UL, W-UL is relatively complete only over loop-free programs[1] (a proof is given in Appendix A of the extended version of the paper [20]).

In fact, one cannot write a relatively complete single-state While rule without some way to track programs. Single-state unrealizability triples do not carry enough information to track the behavior of individual programs across multiple inputs. For example, if $s_1, s_2 \in S$ and $\sigma_1 \neq \sigma_2$ so that $[\![s_1]\!](\sigma_1) = \sigma_1'$ and $[\![s_2]\!](\sigma_2) = \sigma_2'$, then there is no triple $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ that can tell whether $\exists s \in S. [\![s]\!](\sigma_1) = \sigma_1' \wedge [\![s]\!](\sigma_2) = \sigma_2'$. We cannot tell whether or not two possible, independent input-output behaviors can be produced by a single program. As a result, we must overapproximate sets of loop bodies by assuming that such s always exist. In practice, this means that a set of loops is overapproximated as a nondeterministic loop in which the body can be a different $s \in S$ on every iteration. This imprecision is called the "shifting-sands problem".

Suppose that you have two sets of potential guards, $B_1 \mapsto x = 1 \mid x \neq 1$ and $B_2 \mapsto x = 2 \mid x \neq 2$. Every single-state unrealizability triple true on B_1 also holds on B_2 because, for each input, one possible guard of each set yields true and the other yields false. However, we can write loops with the same bodies using these sets of guards that have vastly different effects. For example, the triple $\{\!|x = 0|\!\}$ while B do $x := x + 1$ $\{\!|x \neq 2|\!\}$ holds, but $\{\!|x = 0|\!\}$ while B' do $x := x + 1$ $\{\!|x \neq 2|\!\}$ does not hold, even though B and B' cannot be distinguished by their valid unrealizability triples.

Although the SimpleWhile rule is incomplete in general, if a fact can be proven with the same loop invariant for all possible loops, then the rule given in Figure 3 suffices to prove it. Furthermore, if finitely many distinct invariants are needed, one can split the set of loops into pieces and prove each separately, as in S-UL [16]. If infinitely many invariants are needed, neither strategy works.

3.2 Rules for Nonterminals and Other Structural Rules

Reasoning about nonterminals (i.e., the constructs that allow us to define infinite sets of programs) is the key challenge of Unrealizability Logic (even for loop-free programs). To reason about the fact that nonterminals are used recursively in a grammar—i.e., a nonterminal N may be defined in

terms of itself—we introduce nonterminal summaries of the form $\{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ N\ \{\!|Q_N|\!\}$, which may be introduced as *inductive facts* in some context Γ and then proved via *induction* with the HP rule.

We define some sets of variables associated to sets of programs for convenience:

Definition 3.4 ($\boldsymbol{x}$, $\boldsymbol{z}$, and $\boldsymbol{y}$ for S). Let S be a set of programs. (*i*) $\boldsymbol{x}$ is the set of program variables appearing in programs of S, containing e_t if the programs in S are integer expressions and b_t if the programs in S are Boolean expressions. (*ii*) $\boldsymbol{z}$ is a set of fresh ghost variables that save the values of program variables in $\boldsymbol{x}$ that programs in S can mutate. When the programs in S are integer/Boolean expressions, $\boldsymbol{z}$ is empty. These ghost variables allow us to recall the *initial* values of program variables in our postcondition. (*iii*) $\boldsymbol{y}$ is a set of fresh variables renaming the program variables in $\boldsymbol{x}$ whose values can be changed by a program in S. When the programs in S are integer/Boolean expressions, $\boldsymbol{y}$ contains only a renaming of e_t/b_t. These variables are used to represent unknown poststates in the precondition. They are effectively the opposite of ghost variables. For relations between $\boldsymbol{x}$ and $\boldsymbol{z}$ like $\boldsymbol{x} = \boldsymbol{z}$, the relation is applied to the matching variables of each set. Variables unmatched on either side (e.g., e_t and b_t) are omitted.

Thus, for S in Example 2.1, $\boldsymbol{x} = \{x, y\}$, $\boldsymbol{z} = \{x_z\}$, and $\boldsymbol{y} = \{x_y\}$. Then writing $\boldsymbol{x} = \boldsymbol{z}$ (i.e., $x = x_z$) saves the value of x in x_z for later reference in the summary Q_S.

Nonterminal summaries contain the key information we need about recursive nonterminals. Whenever we must discuss a property of N, we use its summary Q_N. The next 3 rules HP, ApplyHP, and Adapt relate to proving correctness of such summaries (see Figure 3). The tightest Q_N possible for a nonterminal N and the precondition $\boldsymbol{x} = \boldsymbol{z}$ is called the "most general formula" of N (also called the strongest triple in [16]). The most general formula of N can prove all true claims about the set of programs in N [16] (because Q_N can code exactly $\bigcup_{s \in L(N)} (x = [\![s]\!](z))$ as explained in Appendix A of the extended version of the paper [20]). Nonterminal summaries are proved in W-UL by (structural) induction as follows:

3.2.1 HP and ApplyHP. To prove a nonterminal summary, we first add the summary we want to prove to our context as an inductive fact. We then show that the nonterminal's production rules satisfy the triple as well, assuming the inductive fact in the context.

Given a nonterminal $N \mapsto N_1 \mid \cdots \mid N_n$ and writing $\Gamma' \equiv (\Gamma \cup \{\{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ N\ \{\!|Q_N|\!\}\})$, we give the rule HP in Figure 3. As a shorthand, we often write

$$\dfrac{\dfrac{\Gamma' \vdash \{\!|P_1|\!\}\ N_1\ \{\!|Q_N|\!\} \qquad \cdots \qquad \Gamma' \vdash \{\!|P_n|\!\}\ N_n\ \{\!|Q_N|\!\}}{\Gamma \vdash \{\!|\bigwedge_{1 \le j \le n} P_j|\!\}\ N\ \{\!|Q_N|\!\}}\ \text{HP}}{\Gamma \vdash \{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ N\ \{\!|Q_N|\!\}}\ \text{Weaken}$$

Of course, one can generalize this rule for arbitrary preconditions beyond $\boldsymbol{x} = \boldsymbol{z}$. We will not need to do so because one can precisely encode the semantics of the programs in N as a formula Q_N relating prestates $\boldsymbol{z}$ to poststates $\boldsymbol{x}$ [16].

The ApplyHP rule lets us recall/invoke inductive facts from the context Γ'.

3.2.2 Adapt. To make use of summaries of the form $\{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ N\ \{\!|Q_N|\!\}$, we need to connect them to the premises we actually want to prove.

In W-UL, this role is fulfilled by a version of the *rule of adaptation* Adapt in Figure 3. This rule and several variants originally appeared in the context of Hoare logic for recursive programs [11, 22], but we have appropriated and restricted it to match the semantics of W-UL. The Adapt rule computes approximate weakest preconditions from nonterminal summaries $\{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ N\ \{\!|Q|\!\}$.

A general summary $\{\!|P|\!\}\ N\ \{\!|Q|\!\}$ tells us that a prestate σ is mapped to a post-state σ' by a program in N only if either σ does not satisfy P or σ satisfies P and σ' satisfies Q (i.e., $P(\sigma) \rightarrow Q(\sigma')$). So,

$$\dfrac{\dfrac{}{\Gamma' \vdash \{\!|Q_N(2)|\!\}\ 2\ \{\!|Q_N(e_t)|\!\}}\ \text{Int} \quad \dfrac{\dfrac{}{\Gamma' \vdash \{\!|R[e_t/x_1][2/e_t]|\!\}\ 2\ \{\!|R[e_t/x_1]|\!\}}\ \text{Int} \quad \dfrac{\dfrac{}{\Gamma' \vdash \{\!|true|\!\}\ N\ \{\!|Q_N(e_t)|\!\}}\ \text{ApplyHP}}{\Gamma' \vdash \{\!|R|\!\}\ N\ \{\!|Q_N(x_1 + e_t)|\!\}}\ \text{Adapt}}{\Gamma' \vdash \{\!|R[e_t/x_1][2/e_t]|\!\}\ 2 + N\ \{\!|Q_N(e_t)|\!\}}\ \text{Bin-Plus}}{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \{\!|Q_N(2) \wedge R[e_t/x_1][2/e_t]|\!\}\ N\ \{\!|Q_N(e_t)|\!\}}\ \text{HP}}{\emptyset \vdash \{\!|true|\!\}\ N\ \{\!|Q_N(e_t)|\!\}}\ \text{Weaken}}{\emptyset \vdash \{\!|\forall e_{t_y}.Q_N(e_{t_y}) \rightarrow e_{t_y} \neq 3|\!\}\ N\ \{\!|e_t \neq 3|\!\}}\ \text{Adapt}}{\emptyset \vdash \{\!|true|\!\}\ N\ \{\!|e_t \neq 3|\!\}}\ \text{Weaken}}$$

Fig. 4. Following Example 3.5, we give a proof/proof skeleton for a triple $\{\!|true|\!\}\, N\, \{\!|e_t \neq 3|\!\}$ where $N \mapsto 2 \mid 2{+}N$. Above, R is an abbreviation for $\forall e_{t_y}.(Q_N(e_{t_y}) \rightarrow Q_N(x_1 + e_{t_y}))$. When $Q_N(a)$ is taken to be $a = 0 \pmod 2$, we get a valid proof tree for $\{\!|true|\!\}\ N\ \{\!|e_t \neq 3|\!\}$. When $Q_N(a)$ (marked in red) is left as a parameter, we get a proof skeleton (§4.1). Substitution of a second order parameter is taken to be a substitution into each of its inputs. Note, $\Gamma' = \{\{\!|true|\!\}\ N\ \{\!|Q_N(e_t)|\!\}\}$.

given a prestate σ, we can only be certain a condition R will hold on the poststate if all poststates satisfying the above relation also satisfy R. Quantifying out auxiliary variables ($\boldsymbol{u}$) used only by P and Q, yields $\forall \boldsymbol{y}.((\forall \boldsymbol{u}.(P \rightarrow Q[\boldsymbol{y}/\boldsymbol{x}])) \rightarrow R[\boldsymbol{y}/\boldsymbol{x}])$. Here $\boldsymbol{x}$ plays the role of σ, and $\boldsymbol{y}$ plays that of σ' (as per Definition 3.4). In a sense, this formula is the weakest precondition we can derive for R knowing nothing about N besides the truth of $\Gamma \vdash \{\!|P|\!\}\ N\ \{\!|Q|\!\}$. Plugging in $P \equiv \boldsymbol{x} = \boldsymbol{z}$, this simplifies to $\forall \boldsymbol{y}.(Q_N[\boldsymbol{y}/\boldsymbol{x}][\boldsymbol{x}/\boldsymbol{z}] \rightarrow R[\boldsymbol{y}/\boldsymbol{x}])$.

The Adapt rule is well-suited for the automation of W-UL for two reasons: (*i*) applying this single, unambiguous rule for using nonterminal summaries in W-UL is more straightforward than carefully combining the 4 corresponding rules in S-UL [16], and (*ii*) Adapt contains only a single universal quantifier in the precondition, which pulls out to the front of our verification conditions in the single-state setting (§6), adding very limited complexity to proof checking.

For any recursive nonterminal N, if $Q_N = \bigcup_{p \in N}(\boldsymbol{z}, [\![p]\!](\boldsymbol{z}))$ (i.e., the most general formula, specifying the precise relation between end states $\boldsymbol{x}$ and start states $\boldsymbol{z}$), then *any* valid triple about N can be derived from $\{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ N\ \{\!|Q_N|\!\}$ by applying Adapt. The set of $\boldsymbol{y}$ on which $Q_N[y/x][x/z]$ holds are exactly the possible poststates programs in N can reach starting from the state $\boldsymbol{x}$.

Together, these three rules let us prove properties of sets of programs.

Example 3.5. Let $N \mapsto 2 \mid 2 + N$, with N's nonterminal summary Q_N defined to be $(e_t \equiv_2 0)$ (i.e., the output of any program derivable from N is an even number). Note that $\boldsymbol{x}$ is only e_t in this example because we have only integer expressions with no program variables, and there are no $\boldsymbol{z}$ variables. We might want to show $\{\!|true|\!\}\ N\ \{\!|e_t \neq 3|\!\}$; i.e., no expression in $L(N)$ yields 3. Then the proof tree for this triple is exactly the structure in Figure 4 where $Q_N(a) \triangleq a \equiv_2 0$.

The proof tree in Figure 4 centers around proving the summary Q_N, where we load the triple $\{\!|true|\!\}\ N\ \{\!|Q_N(e_t)|\!\}$ into the context as an inductive fact via the application of the HP rule and attempt to prove this triple on each of the possible productions. Observe that whenever this inductive fact is invoked (via ApplyHP), it is immediately followed by an application of Adapt, which ensures that the induction fact can be adapted to discharge the current proof obligation.

3.2.3 Grammar Disjunction. W-UL contains one additional rule to handle nonrecursive nonterminals (i.e., nonterminals N where N is not reachable from itself by following the productions of the grammar). The GrmDisj rule lets one split a nonterminal into its productions. It is analogous to HP, except that no inductive fact is introduced. One could always use HP instead and ignore the inductive fact, but using GrmDisj limits the size of the context.

3.2.4 Weaken. Weaken is the only structural rule in W-UL, i.e., a rule with semantic, rather than syntactic, constraints on the pre- and postconditions of the triples it is applied to. (Note, we ignore

HP by applying our shorthand.) The Weaken rule of W-UL is as one would expect, allowing strengthening the precondition and weakening the postcondition. It is most useful when relaxing a hypothesis into a desired form (e.g., with loops and recursive nonterminals).

3.3 Extending to Vector-States

Recall that a *vector-state* is a (potentially infinite) vector of states on which a program executes simultaneously. When handling vector-states, we extend traditional single-state program semantics $[\![p]\!]_{sing}(\sigma)$ to the semantics for vectors states $[\![p]\!]_{vs}(\boldsymbol{\sigma}) = \langle [\![p]\!]_{sing}(\boldsymbol{\sigma}[1]), \cdots \rangle$ as in [16]. Vector-states allow us to reason about multiple input-output examples as well as hyperproperties.

Inference rules for vector-states are presented in Appendix B of the extended version of the paper [20], but we summarize the important points here. The rules for non-branching expressions and statements are nearly identical to the single-state rules. We simply need to substitute all entries in a vector state and match indices appropriately. The rules for handling nonterminals are the same, as is the Weaken rule. The rules for branching programs become more complicated. When handling multiple examples with vector-states, one must be careful to collect the results of evaluation along both branches. For If-Then-Else statements, we handle this with fresh variables similar to the approach used for binary operators. For While loops, we provide an overapproximate rule that proves invariants using the If-Then-Else rule. This While rule suffers from the "shifting sands" problem described in § 3.1.2.

Identifying a While rule that gives relative completeness is an open problem for future work.

3.4 The Utility of W-UL in Automation

The advantages of W-UL over S-UL are mostly various kinds of simplifications that make the handling of proofs easier and unambiguous. The algorithms we propose later can be implemented over S-UL, but they are much less clean.

First, W-UL eliminates the need to handle the extraneous variables introduced by S-UL. The (loop-free) rules of both S-UL and W-UL are designed to avoid introducing quantifier alternation when applied intelligently, thus bounding the difficulty of verification by the complexity of the invariants/summaries (§6). However, the strongest-postcondition S-UL introduces many quantified variables which are tedious to remove.

Second, W-UL features more easily synthesizable nonterminal summaries. W-UL restricts nonterminal summaries to be of the form $\{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ S\ \{\!|Q_S|\!\}$. Here, the $\boldsymbol{x}$ variables are program variables whose value may change in S, and the $\boldsymbol{z}$ variables are fresh renamings (ghost variables) that Q_S can reference. As remarked in [16], triples of this form are no less expressive than those of the form $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ used in S-UL but require one fewer predicate and are thus easier to search for.

Third, given a postcondition R and summary triple $\{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\, S\, \{\!|Q_S|\!\}$, W-UL fixes the precondition of the triple that Adapt derives over S—i.e., the precondition of $\{\!|\forall y.(Q_S[y/x][x/z] \rightarrow R[y/x])|\!\}\, S\, \{\!|R|\!\}$ is fixed. In the strongest-postcondition logic S-UL, when one is given a precondition P and summary triple $\{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ S\ \{\!|Q_S|\!\}$, the derivable postcondition R in $\{\!|P|\!\}\ S\ \{\!|R|\!\}$ is not fixed. The choice of R adds an extra degree of freedom to proof synthesis, although one can eliminate this degree of freedom by setting R to be a precise function of P and Q_S, as described in §4.1.1.

Most crucially, the *rule of adaptation* (Adapt) in W-UL subsumes four rules of S-UL (Inv, Sub1, Sub2, Conj) [16]. With the rule of adaptation in hand, we can automatically, unambiguously determine the shape of the proof tree of any triple, up to its summaries and invariants. When one plugs nonterminal summaries into this "proof skeleton," the rules of W-UL yield verification conditions for checking whether the summaries complete the proof. By eliminating the need for proof search, proof verification and derivation from summaries become equivalent (i.e., both can be solved by simply checking verification conditions). Although it is possible in principle to construct such

proof skeletons in S-UL by simulating Adapt with the four other rules, W-UL offers a much simpler interface for automation by eliminating the ambiguity on when these four rules should be applied.

4 Generating W-UL Proof Skeletons and Verification Conditions

The main result of this paper is an algorithm for generating proofs in W-UL. The W-UL proof-generation algorithm is similar to classical program verification for Hoare logic. Classically, one specifies a target triple along with loop invariants, and the program verifier generates a set of assertions called verification conditions (VCs). These VCs are discharged to an automated theorem prover to determine the validity of the proof.

In our algorithm, we do not require summaries/invariants to be specified immediately. Instead, we introduce second-order relational variables called *parameters* that stand in for them (e.g., $Q_N(\cdot,\cdot,\cdot)$). Similarly, we define *parametrized formulas* to be formulas that may contain such relational variables (e.g., $Q_N(v, e_t + 7, v_z) \wedge v = 5$). This approach gives us the option to either plug in summaries/invariants (e.g., obtained from some other tool) or attempt to synthesize them later.

In our first step (§4.1), given an unrealizability triple $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$, our algorithm generates a "proof skeleton"—a proof tree whose pre and postconditions are parametrized formulas—that forms the outline of our proof (see Fig 4). Our construction is agnostic with respect to loop invariants and nonterminal summaries. It is even independent of the specific target conditions P and Q used in the final proof (up to the auxiliary variables appearing in Q), although we will specify P and Q in practice. The ability to separate the proof skeleton from nonterminal summaries and target conditions is a key feature of W-UL, which contains a version of the rule of adaptation tailored to Unrealizability Logic.

In the second step (§4.2), the algorithm generates a set of parametrized verification conditions (PVCs)—VCs that are parametrized formulas. Then, if one substitutes (candidate) loop invariants and nonterminal summaries for the parameters, the PVCs can be converted to VCs. An automated theorem prover can discharge the VCs, determining whether the provided invariants and summaries are strong enough to prove $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$.

For sets of loop-free programs S, the generated PVCs are satisfiable exactly when $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ is true (Corollary 4.5). Solutions to the PVCs can be plugged into the proof skeleton as nonterminal summaries and loop invariants to complete the proof. If some programs in S contain loops, failure to satisfy the VCs does not necessarily imply nonexistence of a proof.

We outline the proof generation algorithm with specified summaries in the pseudocode below. When summaries are not provided, the algorithm is unchanged except that we do not take summaries as input but rather synthesize summaries satisfying the PVCs on Line 3 as in §5.

Algorithm 1 ULProve(P, S, Q, summaries)

Requires: Precondition P, Postcondition Q, RTG with start symbol S, Collection of invariants/summaries *summaries*.

```
1: skel ← P-SKEL(∅, P, S, Q)                    ▷ Compute proof skeleton and PVCs
2: pvcs ← EXTRACTPVCS(skel)
3: proved ← CHECKSUMMARIES(pvcs, summaries)      ▷ Check summaries against PVCs
4: if proved then          ▷ Plug summaries into proof skeleton if check succeeded
5:     return PLUGIN(skel, summaries)
6: else
7:     return "unproven"
```

4.1 Computing a Proof Skeleton

Our first step in proving a triple $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ is to compute a proof skeleton based on S. Recall that a proof skeleton is a proof tree whose pre- and postconditions are parametrized formulas—formulas containing symbols representing the target pre- and postcondition, loop invariants, and nonterminal summaries. A proof skeleton gives a template for a complete proof. Moreover, we can easily extract parametrized verification conditions from the proof skeletons that are created (§4.2).

An example proof skeleton for Example 3.5 is given in Figure 4, where $Q_N(e_t)$ is a parameter.

To generate a proof skeleton, we recurse on the structure of S. Specifically, we define a function w-skel(Γ, S, Q) that takes as input a set of programs S defined via an RTG as in §3, a context Γ whose triples' postconditions are parametrized formulas, and a postcondition Q as a parametrized formula. The function w-skel returns a skeleton in the form of a proof tree in which each node n is labeled with a derivation rule $rule(n)$, and each pre and postcondition is a parametrized formula.

Note that w-skel does not take the precondition P as input—a parametrized precondition is generated by constructing the skeleton. The output is essentially a proof skeleton for proving the weakest precondition of S with respect to Q. When a proof skeleton is returned, we can simply apply Weaken to recover $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$.

To simplify matters, the version of the algorithm that we present operates over single-state inference rules. The extension to vector-states is straightforward: the vector-state rules for handling nonterminals are identical to the single-state rules, so very little of the algorithm changes.

4.1.1 Weakest Skeleton. We define a function w-skel (standing for weakest skeleton) that operates recursively over S. We describe next what happens in each possible case. We encourage readers to follow along in Figure 4.

Case 1: $S = op(S_1, \ldots, S_k)$ where op is a constructor from G_{Imp} other than While (e.g., +, true, if $-$ then $-$ else *).* We apply the inference rule corresponding to op. We call w-skel recursively on its hypotheses, working right to left. Each hypothesis takes parametrized postconditions as dictated by the inference rule, where substitutions $[b/a]$ on a parameter are applied to each of the parameter's inputs. The hypotheses' skeletons are the children of the application of the inference rule op. The precondition of the conclusion is as dictated by the inference rule.

In Figure 4, the use of inference rule Bin-Plus is covered by this case. Here, the set of programs in the middle position of the triple in the conclusion is given by $2 + N$. We apply Bin-Plus, computing proof skeletons for the hypotheses from right to left and transforming parameterized conditions as specified by the inference rules. Letting R stand for the parametric precondition of the skeleton for N, the parametric formula $R[e_t/x_1]$ becomes the parametric postcondition for the skeleton for 2.

$$\frac{\text{w-skel}(\Gamma_1, 2, R[e_t/x_1]) \qquad \text{w-skel}(\Gamma_1, N, Q_N(x_1 + e_t))}{\Gamma_1 \vdash \{\!|R[e_t/x_1][2/e_t]|\!\}\ 2 + N\ \{\!|Q_N(e_t)|\!\}}\ \text{Bin-Plus}$$

Case 2: $S =$ while B do S_1. If S is a while loop, then apply the While rule with a fresh invariant parameter $I_S(\boldsymbol{x}, \boldsymbol{v})$. Derive the skeleton of the body hypothesis by invoking w-skel recursively with parametric postcondition $I_S(\boldsymbol{x}, \boldsymbol{v})$, where $\boldsymbol{x}$ are the variables appearing in the loop (as per Definition 3.4) and $\boldsymbol{v}$ are the free variables of Q which are not in $\boldsymbol{x}$. For the guard hypothesis, substitute the parametric precondition of the body hypothesis for P_{I_S} in $(\neg b_t \to Q) \wedge (b_t \to P_{I_S})$ and recursively generate a skeleton for B with this postcondition. Then apply Weaken to prove $\{\!|I_S(\boldsymbol{x}, \boldsymbol{v})|\!\}\ B\ \{\!|(\neg b_t \to Q) \wedge (b_t \to P_{I_S})|\!\}$, the guard hypothesis of the While application.

$$\frac{\dfrac{\text{w-skel}(\Gamma, B, (\neg b_t \to Q) \wedge (b_t \to P_{I_S}))}{\Gamma \vdash \{|I_S(\boldsymbol{x}, \boldsymbol{v})|\}B\{|(\neg b_t \to Q) \wedge (b_t \to P_{I_S})|\}}\ \text{Weaken} \qquad \text{w-skel}(\Gamma, S, I_S(\boldsymbol{x}, \boldsymbol{v}))}{\Gamma \vdash \{\!|I_S(\boldsymbol{x}, \boldsymbol{v})|\!\}\ \text{while } B \text{ do } S\ \{\!|Q|\!\}}\ \text{Single-While}$$

Case 3: $S \mapsto S_1 \mid \cdots \mid S_n$, where S is a Non-Recursive Nonterminal. If S is non-recursive (i.e., it does not appear in any of its own productions nor in their expansions), apply GrmDisj. Construct proof skeletons recursively for each production of S with the same postcondition. The precondition of the conclusion is $\bigwedge_{j=1}^{n} P_n$, the conjunction of the parametrized preconditions derived for the hypotheses.

$$\frac{\textsc{w-skel}(\Gamma, S_1, Q) \qquad \cdots \qquad \textsc{w-skel}(\Gamma, S_n, Q)}{\Gamma \vdash \{\!|\bigwedge_{j=1}^{n} P_j|\!\}\ S\ \{\!|Q|\!\}}\ \text{GrmDisj}$$

Case 4: S is a Recursive Nonterminal. If S is recursive, we have two subcases.

First, if a triple $\{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ S\ \{\!|Q_S(\boldsymbol{x}, \boldsymbol{z})|\!\}$ exists in the context Γ, close off this branch of the proof skeleton by using ApplyHP and Adapt as shown in the following proof-skeleton fragment:

$$\dfrac{\dfrac{}{\Gamma \vdash \{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ S\ \{\!|Q_S(\boldsymbol{x}, \boldsymbol{z})|\!\}}\ \text{ApplyHP}}{\Gamma \vdash \{\!|\forall \boldsymbol{y}.Q_S(\boldsymbol{y}, \boldsymbol{x}) \rightarrow Q[\boldsymbol{y}/\boldsymbol{x}]|\!\}\ S\ \{\!|Q|\!\}}\ \text{Adapt}$$

Otherwise, we hit our second subcase and apply the HP rule with a postcondition Q_S and construct proof skeletons of its hypotheses. Define $\boldsymbol{x}$ and $\boldsymbol{z}$ of S as in Definition 3.4 so that $\boldsymbol{x}$ captures the program variables in S (plus e_t/b_t if S is a set of integer/Boolean expression) and $\boldsymbol{z}$ gives fresh ghost variables for all program variables mutated by programs in S. Take the inductive fact to be $\{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ S\ \{\!|Q_S(\boldsymbol{x}, \boldsymbol{z})|\!\}$. This construction follows the convention from §3.2.

Now apply Adapt to $\{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ S\ \{\!|Q_S(\boldsymbol{x}, \boldsymbol{z})|\!\}$ to complete the skeleton. Note that Γ_1 denotes $\Gamma \cup \{\{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ S_1\ \{\!|Q_S(\boldsymbol{x}, \boldsymbol{z})|\!\}\}$ below. The parametrized preconditions of HPs hypotheses are denoted by $P_1, \cdots, P_n$. See the first application of Adapt in Figure 4 for a further example.

$$\dfrac{\dfrac{\dfrac{\textsc{w-skel}(\Gamma_1, S_1, Q_S(\boldsymbol{x}, \boldsymbol{z})) \qquad \cdots \qquad \textsc{w-skel}(\Gamma_1, S_n, Q_S(\boldsymbol{x}, \boldsymbol{z}))}{\Gamma \vdash \{\!|\bigwedge_{j=1}^{n} P_j|\!\}\ S\ \{\!|Q_S(\boldsymbol{x}, \boldsymbol{z})|\!\}}\ \text{HP}}{\Gamma \vdash \{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ S\ \{\!|Q_S(\boldsymbol{x}, \boldsymbol{z})|\!\}}\ \text{Weaken}}{\Gamma \vdash \{\!|\forall \boldsymbol{y}.(Q_S(\boldsymbol{y}, \boldsymbol{x}) \rightarrow Q[\boldsymbol{y}/\boldsymbol{x}])|\!\}\ S\ \{\!|Q|\!\}}\ \text{Adapt}$$

With some care, it is possible to construct similar unambiguous proof skeletons in S-UL, although they are longer and more involved than in W-UL because one must apply a series of five smaller structural rules in place of Adapt to derive a target triple $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ from a summary triple $\{\!|\boldsymbol{x} = \boldsymbol{z}|\!\}\ S\ \{\!|Q_S(\boldsymbol{x}, \boldsymbol{z})|\!\}$. Moreover, the rule of adaptation Adapt explicitly fixes a concrete target precondition P—namely, $\forall \boldsymbol{y}.(Q_S(\boldsymbol{y}, \boldsymbol{x}) \rightarrow Q[\boldsymbol{y}/\boldsymbol{x}])$—whereas the strongest-postcondition logic S-UL requires one to specify a target postcondition Q in addition to Q_S.[4] One may understand our formulation of the rule of adaptation and the associated precondition as a key step that makes automated proofs in W-UL shorter and clearer than those in S-UL.

4.1.2 Connecting the Weakest Skeleton to the Precondition P. Our proof skeleton is almost complete, but it does not yet mention the precondition P. What we have given is a proof skeleton for proving the weakest precondition of Q with respect to S. The final step to produce a complete proof skeleton for $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ is to Weaken the conclusion of the skeleton produced by $\textsc{w-skel}(\emptyset, S, Q)$ to connect it to precondition P. After applying this final Weaken, the proof skeleton is complete.

$$\frac{\textsc{w-skel}(\emptyset, S, Q)}{\emptyset \vdash \{\!|P|\!\}\ S\ \{\!|Q|\!\}}\ \text{Weaken}$$

[4] One can choose Q to be as strong as possible, similar to the weakest precondition given by the rule of adaptation. For example, one could take Q to be $\exists \boldsymbol{z}.(P[\boldsymbol{x}/\boldsymbol{z}] \wedge Q_S(\boldsymbol{z}, \boldsymbol{x}) \rightarrow Q)$.

We call this final skeleton p-skel$(\emptyset, P, S, Q)$.

By Theorem 4.1 below, any true fact over sets of loop-free programs has a proof with the same structure as the skeleton we generate:

Theorem 4.1. *If $\emptyset \vdash \{\!|P|\!\}\ S\ \{\!|Q|\!\}$ for conditions P and Q, and a set of loop-free programs S, then there exists a proof of $\emptyset \vdash \{\!|P|\!\}\ S\ \{\!|Q|\!\}$ which adheres to the proof skeleton p-skel$(\emptyset, P, S, Q)$.*

This result follows from the constructive proof of relative completeness for loop-free programs given in Appendix A of the extended version of the paper [20]. An analogous result holds for nonempty contexts Γ.

4.2 Computing Verification Conditions

After running the algorithm in §4.1, we have a proof skeleton p-skel(Γ, P, S, Q), corresponding to p-skel$(\Gamma, \mathit{true}, N, e_t \neq 3)$ in Figure 4.

In this section, we show how to generate (parametrized) verification conditions (PVCs) from a proof skeleton. PVCs give exactly the constraints on our summaries and invariants needed to construct a valid proof. In terms of the example from Figure 4, the PVCs are the constraints a supplied predicate Q_N must satisfy to make the proof skeleton into a valid concrete proof tree.

For a proof tree to be valid, all nodes in the tree must correctly follow the syntactic and semantic restrictions of the corresponding W-UL inference rules. Because of how we generate proof skeletons, all syntactic restrictions the rules impose are satisfied. (In fact, the proof-skeleton algorithm uses these syntactic relations to propagate parametrized postconditions through the tree.)

The only rule applications that are not guaranteed to be valid by construction are the applications of Weaken. Weaken is the only rule with semantic restrictions on its logical conditions. Because no Weaken applications in our proof skeleton change the postcondition from the conclusion to the hypothesis, all our PVCs check whether Weaken preconditions are implied (see Figure 3).

Definition 4.2 (Parametrized Verification Conditions). Given an application A of the Weaken rule of the following form:

$$\frac{\Gamma \vdash \{\!|P'|\!\}\ S\ \{\!|Q|\!\}}{\Gamma \vdash \{\!|P|\!\}\ S\ \{\!|Q|\!\}}\ \text{Weaken}$$

$PVC(A)$ denotes the parameterized formula $\forall \boldsymbol{v}.P \rightarrow P'$, where $\boldsymbol{v}$ are the variables free in $P \rightarrow P'$.

Given a set $\alpha = \{A_1, \ldots, A_n\}$ of Weaken applications, we use $PVC(\alpha) = \{PVC(A_1), \ldots, PVC(A_n)\}$ to denote the set of PVCs for all Weaken applications in α.

Example 4.3. For the proof tree in Figure 4, which contains two instances of Weaken, $PVC(\{Wkn_1, Wkn_2\})$ contains the following two formulas, where P is *true* and Q is $e_t \neq 3$ as in Example 3.5:

- $\mathit{true} \rightarrow (\forall e_{t_y}.Q_N(e_{t_y}) \rightarrow e_{t_y} \neq 3)$
- $\mathit{true} \rightarrow (Q_N(2) \wedge R[e_t/x_1][2/e_t])$

Definition 4.4 (Satisfiability). A collection C of PVCs is *satisfiable* if there exists an assignment of relations to parameters that makes each verification condition in C true as a first-order formula.

The following corollary follows from Theorem 4.1:

Corollary 4.5. *Let S be a set of loop-free programs, and α be the set of Weaken applications in the proof skeleton for $\emptyset \vdash \{\!|P|\!\}\ S\ \{\!|Q|\!\}$. If $PVC(\alpha)$ is unsatisfiable, then $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ is not provable in W-UL. In particular, this condition means that $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ is false.*

In other words, once formulas are proposed for each parameter, we may substitute them into our PVCs and check the resulting verification conditions to complete the proof. If the check on the verification conditions succeeds, the proof is correct. Otherwise, the proof is incorrect (either because the supplied formulas are insufficient, the claim is not true, or loops are present) or the solver used is not powerful enough to prove correctness of the verification conditions.

5 Synthesizing Summaries and Invariants with SyGuS

The procedure described in §4 gives us a set of parametrized verification conditions that characterize the set of summaries and loop invariants for which a completion of the proof tree is valid. In this section, we show that just as such verification conditions can be used as specifications to synthesize loop invariants in Hoare logic, they can be used to synthesize summaries and invariants in W-UL.

We translate the problem of synthesizing summaries for parameterized verification conditions into synthesis problems in the SyGuS framework [1]. The goal of a SyGuS problem is to find terms in a user-given grammar that agree with a given formula over a fixed background theory.

Definition 5.1 (SyGuS [1]). A *Syntax-Guided Synthesis* (SyGuS) *problem* over a background theory $\mathcal{T}$ (e.g., LIA) is a tuple $sy = (G_1, \cdots, G_n, \psi(f_1, \cdots, f_n))$. Here $G_1, \cdots, G_n$ are a regular tree grammars that only generate terms in the background theory $\mathcal{T}$, and $\psi(f_1, \cdots, f_n)$ is a sentence in the language of $\mathcal{T}$ with additional function symbols $f_1, \cdots, f_n$, which stand for the terms to be synthesized. A *solution* to the problem sy is a sequence of ground terms $e_1, \cdots, e_n$ so that each e_j is in the language described by the grammar G_j and the formula $\psi(e_1, \cdots, e_n)$ is valid.

Customizing the grammar lets one choose what form the sought-for invariants and summaries should have. Grammars can also restrict search spaces of synthesis problems to increase scalability.

Definition 5.2 ($SY(G_1, \cdots, G_n, C_{PVC})$). Let C_{PVC} be a (finite) collection of PVCs with parameters $Q_1(\boldsymbol{x_1}), \cdots, Q_n(\boldsymbol{x_n})$. For each Q_j, G_j is a grammar deriving some first-order formulas in the language of integer arithmetic with free variables $\boldsymbol{x_j}$. Then define the SyGuS problem $SY(G_1, \cdots, G_n, C_{PVC})$ to be $(G_1, \cdots, G_n, \psi(Q_1, \cdots, Q_n))$ where ψ is the conjunction of the conditions in C_{PVC}.

When $PVC(\alpha)$ of a triple has a solution, there are sufficiently expressive grammars $G_1, \cdots, G_n$ so that $SY(G_1, \cdots, G_n, PVC(\alpha))$ has a solution. In particular, if S is a set of loop-free programs and each G_j derives all first-order formulas in its allowed language, then there is a proof of $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ if and only if $SY(G_1, \cdots, G_n, PVC(\alpha))$ has a solution.

As an example, we encode the parameterized verification conditions from Example 4.3 as a SyGuS problem. We take G to be the grammar deriving all formulas of the form $e_t \equiv_n 0$ for some n. Then $SY(G, PVC(\alpha)) = (G, \psi(Q_N))$ where

$$\begin{aligned}\psi(Q_N) = \mathit{true} &\rightarrow (\forall e_{t_y}.Q_N(e_{t_y}) \rightarrow e_{t_y} \neq 3) \\ \wedge\ \mathit{true} &\rightarrow (Q_N(2) \wedge (\forall e_{t_y}.Q_N(e_{t_y}) \rightarrow Q_N(2 + e_{t_y})))\end{aligned}$$

The SyGuS problem $(G, \psi(Q_N))$ can be discharged to a SyGuS solver. For our example, it is clear that taking $Q_N(e_t)$ to be the formula $(e_t \equiv_2 0) \in L(G)$ satisfies the verification conditions. Indeed, this formula was the summary used in Example 3.5.

Although this approach enables summary synthesis, readers interested in applying this technique to pruning for program synthesis may be dissatisfied. Program-synthesis problems are often specified as SyGuS queries, and using our technique to perform pruning for program synthesis requires solving SyGuS queries to synthesize summaries. Moreover, synthesizing invariants for pruning can be harder than solving the synthesis problem itself (see Guria [10]). However, one can surmount this difficulty by considering invariants and summaries that are easy to synthesize by,

for example, considering restrictive grammars of possible invariants/summaries (see RQ2 in §6), or by establishing and proving useful summaries in advance if one plans to ask many synthesis problems over the same grammar (see RQ4 in §6). In addition, readers should bear in mind that pruning is only one potential application of our technique.

6 Implementation and Experimental Results

We implemented proof-skeleton generation (§4.1), verification-condition extraction (§4.2), and reduction to SyGuS (§5) in OCaml in a tool called Wuldo. Wuldo discharges the verification conditions to z3alpha [25] when these conditions involve single states or vector-states of finite lengths, and to Vampire [3] when the vector-states are infinite (because z3alpha cannot handle infinite arrays). Wuldo uses cvc5 [2] for summary synthesis and to implement part of the optimization to discharge verification conditions discussed in the next four paragraphs.

In practice, many of the queries that Wuldo generates are difficult for SMT solvers to handle due to extractable quantifiers from the Adapt rule scattered throughout the verification conditions. To remedy this issue, we implemented an optimization that (*i*) simplifies the appearances of quantifiers by pulling them to the front of the VCs and (*ii*) uses a SyGuS solver (in parallel with the other constraint solver) to attempt to solve complex formulas involving existential quantifiers. The optimization starts by simplifying nested implications, moving universal quantifiers that appear on the right-hand sides of VCs to the front of the formula, and existential quantifiers that appear on the left-hand sides of VCs to the front of the formula (by turning them into universal quantifiers). We also split VCs of the form $A \rightarrow (B_1 \wedge B_2)$ in two: $A \rightarrow B_1$ and $A \rightarrow B_2$.

This optimization simplifies the PVCs in Example 4.3 to

$$\begin{aligned} \forall e_{t_y}.((\mathit{true} \wedge Q_N(e_{t_y})) &\rightarrow e_{t_y} \neq 3) \\ \mathit{true} &\rightarrow Q_N(2) \\ \forall e_{t_y}.((\mathit{true} \wedge Q_N(e_{t_y})) &\rightarrow Q_N(2 + e_{t_y})) \end{aligned}$$

The resulting PVCs are guaranteed to have no existential quantifiers (excluding quantifiers in P, Q, and any invariants/summaries we may fill in). When program variables exist, these quantifiers introduce no additional degree of alternation (because the program variables are already universally quantified at the front of the PVC). If P or another term on the left-hand side were to contain an existential quantifier, such a quantifier would also be moved to the front of the formula as a universal quantifier.

Our optimization, so far, cannot extract any existential quantifiers that appear in the rightmost implicand (e.g., $e_{t_y} \neq 3$ and $Q_N(2 + e_{t_y})$ above). To address this challenge, we replace any such existentially quantified variables with second-order function variables. These function variables take as input the variables extracted from the left-hand side of the VC. We can then encode the problem of discharging this modified VC as a SyGuS problem where we ask a solver (cvc5) to find an instantiation for the second-order variable that makes the verification condition hold.

When attempting this operation, we run z3alpha in parallel, on the original quantified VC, and return the result of whichever tool terminates first. Because cvc5 cannot handle infinite vectors, we do not issue a SyGuS query for formulas that use them.

When this optimization is used, we call our tool Wuldo; without the optimization, we call it Wuldo$^-$.

We evaluated the effectiveness of Wuldo at generating proofs with varying degrees of guidance, the benefits of the compositionality of our technique, and the potential for reuse of nonterminal summaries to solve collections of similar problems. Thus, our evaluation is designed to answer the following research questions:

RQ1 How effective is Wuldo at constructing proofs when invariants and summaries are provided (i.e., proof derivation from summaries)?
RQ2 How effective is Wuldo at constructing proofs when invariants and summaries are *not* provided and must be synthesized instead (i.e., whole cloth proof derivation)?
RQ3 How effective is the optimized tool Wuldo compared to Wuldo$^-$?
RQ4 Does reuse of proven nonterminal summaries allow Wuldo to prove many properties over similar sets of programs quickly?

All experiments were run three times on a 4-core i7-7700HQ CPU processor with 16GB RAM running Windows Home 10 version 22H2, with a 5-minute timeout. Tables 1 and 2, as well as Figure 5, report median times.

Benchmarks for RQ1-3. We evaluated Wuldo (and Wuldo$^-$) on a set of benchmarks that may be grouped into four categories, drawn from three previous papers that study unrealizability.

The first category consists of unrealizable Linear Integer Arithmetic (LIA) SyGuS problems, created by Hu et al. to evaluate Nope [12] (these benchmarks contain only expressions and no statements). Hu et al. considered 132 unrealizable benchmarks, each of which are a variant of the 60 realizable benchmarks from the LIA SyGuS competition track. The original SyGuS benchmarks are made unrealizable by either limiting the number of times a certain operator (e.g., if-then-else) may occur, or disallowing certain constants in the grammar (e.g., the grammar may not contain odd numbers), reflecting restrictions one may impose when trying to synthesize an optimal program with respect to some metric [14].

Of these 132 benchmarks, we identified 9 representatives (one from each of type of grammar we identified in the benchmarks). For 4 of these 9 benchmarks (distinguished with the suffix _hard), we also introduced a variant with a simplified search space to produce an equivalent set of programs that is easier for W-UL to verify (distinguished with the suffix _easy). These representative benchmarks contain between 1 to 5 nonterminals (each with between 2 and 10 production rules) and 1 to 9 input-output examples as the specification. These 13 representative benchmarks were supplied with hand-specified summaries to test the efficacy of Wuldo at proof derivation from summaries (RQ1). We considered the full set of 132 benchmarks for evaluating the efficacy of whole cloth proof generation (RQ2).

The second category of benchmarks consists of unrealizable synthesis problems over an imperative language with bitvector variables, drawn from Kim et al.'s evaluation of Messy [17]. These benchmarks represent synthesis problems that require tricky bitvector-arithmetic reasoning, e.g., synthesizing bitwise-xor using only bitwise-and and bitwise-or. Kim et al. [17] provide a total of 14 such bitvector benchmarks, 8 of which we could verify as unrealizable; these benchmarks contain between 1 to 4 nonterminals (each with between 1 and 5 production rules) and 1 to 4 examples.[5] We used all 8 benchmarks for RQ1 and 7 for RQ2 (omitting one that needs no summaries).

The third set of benchmarks consists of the 8 examples described in the original UL paper [16] and our own Example 3.5. These 9 benchmarks' grammars contain between 1 and 3 nonterminals (each with between 1 and 4 production rules; except one outlier with 100 production rules that derive the constants 0 through 99) and are intended to demonstrate coverage of the constructors of G_{imp}.

Finally, we wrote 3 "limit-point" benchmarks for which proving unrealizability requires infinitely many examples (and therefore infinite vector states). These benchmarks ask one to prove that a program s is excluded from a set that contains arbitrarily close approximations to s (i.e., the proof requires separating a set of programs from a limit point). One of the 3 benchmarks is the running

[5]The benchmarks in Kim et al. [17] contain a mix of realizable and unrealizable benchmarks because their tool Messy targets both synthesis and proving unrealizability. We thus use only the unrealizable subset for this paper.

example from §2, which appeared in [16] as well (these grammars have 1 to 2 nonterminals, each with between 1 and 2 production rules).

RQ1: How effective is Wuldo when invariants and summaries are provided?

Table 1 shows the results of running Wuldo, Messy, and Nay on all benchmarks when summaries are provided. Wuldo can generate valid W-UL proofs when invariants and summaries are specified for 29/33 benchmarks, taking between 0.20s and 79.33s (avg. 13.45s).

Wuldo fails on four Nope benchmarks. The summaries used in these benchmarks describe semi-linear sets to exactly capture the behavior of all the expressions in the input grammar, inspired by Hu et al. [12]. For example, the possible outputs of any program in the grammar $E \mapsto E + 2 \mid x$ on the inputs $x = 2$ and $x = 3$ can be captured by the formula $(2, 3) + a(2, 2)$, where all operations are defined elementwise and a is a free integer variable. A summary capturing this property must quantify over a (i.e., $Q_E \equiv \exists a.(e_t[1], e_t[2]) = (2, 3) + a(2, 2)$). Proving correctness of these summaries is challenging for the SMT solver because it requires checking implications in which a quantifier appears on both sides (e.g., $Q_E \rightarrow Q_E[(e_t + 2)/e_t]$). The benchmarks for which Wuldo failed (benchmarks with *_hard* in the name) all contained recursive nonterminals with a production rule of the form "if B then E else E" with nonterminal E similar to the above. We were able to manually simplify these benchmarks' grammars to eliminate recursion in these nonterminals, producing the corresponding *_easy* benchmarks.

Although the state-of-the-art tool for SyGuS unrealizability Nay [13] can solve the Nope benchmarks on average 9.85 times as fast as Wuldo (without requiring a summary), it implements a domain-specific solution for LIA SyGuS problems and does not produce an interpretable proof artifact. Messy, the only tool not specialized for LIA, solves only 2 of the 9 benchmarks that Wuldo can solve.

Table 1. Times to generate and verify proofs when invariants and summaries are specified. An X denotes a timeout. In each row, the fastest time is shown in **bold**.

		Wuldo⁻	Wuldo	Messy [17]	Nay [13]
Nope [12]	const_example1	2.375	3.035	1.304	**0.669**
	if_mpg_example1	X	5.466	X	**0.784**
	if_guard	X	9.684	X	**0.678**
	if_max3_hard	X	X	X	**0.771**
	if_max3_easy	X	14.208	X	**0.596**
	if_ite1_hard	X	X	X	**0.982**
	if_ite1_easy	X	14.447	X	**0.580**
	if_sum_3_5_hard	X	X	X	**0.905**
	if_sum_3_5_easy	X	15.568	X	**0.598**
	if_search_2_hard	X	X	X	**0.628**
	if_search_2_easy	X	17.343	X	**0.565**
	plus_array_search2	5.985	19.486	X	**1.019**
	plus_mpg_ite1	2.801	4.150	1.491	**0.749**
Messy Bitvector [17]	AND	**0.383**	0.776	X	N/A
	ADD	**0.235**	0.845	18.547	N/A
	MAX	**0.518**	0.786	X	N/A
	NAND	**0.192**	0.375	2.688	N/A
	PLUS	**0.397**	0.608	1.728	N/A
	SWAP_XOR	**0.204**	5.883	2.017	N/A
	XOR	**0.478**	1.881	1.967	N/A
	WHL_XOR	**0.812**	0.912	X	N/A
S-UL [16]	ex1.1_simp	**0.384**	0.512	1.246	1.208
	ex2.3_simp	**0.829**	4.544	N/A	N/A
	ex2.6_simp	**0.349**	6.502	N/A	N/A
	ex3.8_simp	**0.208**	0.296	1.069	1.239
	ex5.2_simp	**1.384**	9.722	X	N/A
	ex5.3_simp	**0.332**	19.508	N/A	N/A
	ex2.5_finVS	**0.543**	0.751	X	0.624
	ex3.11_finVS	**0.249**	0.300	N/A	N/A
	ex3.5_ulw	**0.230**	1.327	1.054	1.087
Limit Pt	id_not_in_ite	83.147	**79.330**	N/A	N/A
	t_Linterval_union	84.614	**76.515**	N/A	N/A
	y_0_decr	**74.559**	75.261	N/A	N/A

The Messy bitvector benchmarks were easier for Wuldo to verify than the Nope benchmarks because the specified summaries were quantifier-free. The summaries we wrote were mostly from a grammar G_{BV} of formulas of the form "if the input(s) are set to v in the n^{th} bit, the outputs must be set to v' in the n^{th} bit" for the finitely many possible values of v, n, v' (i.e., $(in \wedge 2^n) = v \rightarrow (out \wedge 2^n) = v'$ and $((in_1 \wedge 2^n) = v_1 \wedge (in_2 \wedge 2^n) = v_2) \rightarrow out \wedge 2^n = v'$). (We take advantage of this structure

in the next section when we answer RQ2.) When summaries are provided, Wuldo produces proof certificates on average 2.76 times as fast as Messy decides unrealizability for the bitvector benchmarks when both tools terminate, and Wuldo solves 3 benchmarks Messy cannot solve. Of course, specifying summaries gives Wuldo an advantage, but Wuldo is the only tool that provides an interpretable proof artifact and thus performs a strictly harder task.

By solving all S-UL benchmarks, Wuldo demonstrates that it can handle all language features of G_{imp} (though loops remain challenging as in ex5.2_simp). In addition, several of these benchmarks are of the form $\{\!|P|\!\}\ S\ \{\!|Q|\!\}$ where infinitely many states satisfy P. Such problems are beyond the reach of any existing automated solver, but Wuldo solves them handily. The 3 limit-point benchmarks are also problems that *no prior automated tool* for proving unrealizability can express. Wuldo solves these benchmarks as well (albeit slowly due to the solver Vampire used for infinite arrays).

Findings. Wuldo can discharge the verification conditions for 29/33 benchmarks (50% more than any other tool) when summaries and invariants are provided (i.e., proof derivation from summaries). Furthermore, compared to Messy, the only other tool not restricted to expressions, Wuldo is 1.79 times as fast as and solves 18 more benchmarks. This advantage suggests that the use of summaries confers a substantial advantage for verification of sets of programs.

RQ2: How effective is Wuldo when invariants and summaries must be synthesized?

Because existing SyGuS solvers provide little to no support for synthesizing formulas with quantifiers and infinite arrays, we could not evaluate this research question on the limit-point benchmarks, nor could we specify a reasonable grammar to guide synthesis of the Nope benchmarks.

The major challenge of this mode of proof synthesis is generating summaries and invariants that satisfy the PVCs. Our first approach to summary synthesis was to let cvc5 search over *all* possible quantifier-free formulas; we call this grammar *Unconstrained*. While this solved 5 examples from the S-UL paper with trivial grammars (see Unconstrained columns in Table 2), it failed on all 7 Messy and 132 Nope benchmarks. On the S-UL examples, Wuldo verified 5/9 benchmarks, taking 0.19s on average, which is 24.36 times as fast as when summaries were specified. The synthesized summaries are very similar to those manually specified in RQ1; we believe summary synthesis is faster than summary verification on some of our benchmarks because z3alpha, the SMT solver that we use to check verification conditions, constructs a strategy (i.e., an algorithm for choosing how to apply symbolic reasoning steps) based on the input query before solving. The SyGuS solver that we use, cvc5, does not generate strategies and instead solves queries directly. The performance difference suggests that, on easy problems, the time spent constructing a strategy outweighs the time needed to verify or synthesize a solution.

Table 2. Time taken to generate proofs when nonterminal summaries and loop invariants are synthesized from (*i*) an Unconstrained grammar and (*ii*) the grammar G_{BV} for bitvectors. An X denotes a timeout. In each row, the fastest time is shown in **bold**.

		Unconstrained		G_{BV}	
		Wuldo$^-$	Wuldo	Wuldo$^-$	Wuldo
Messy Bitvectors [17]	AND	X	X	**0.18s**	0.44s
	ADD	X	X	2.57s	**0.44s**
	MAX	X	X	X	X
	NAND	X	X	X	X
	PLUS	X	X	**0.23s**	0.31s
	XOR	X	X	**0.20s**	0.52s
	WHL_XOR	X	X	X	X
S-UL [16]	ex1.1_simp	0.38s	**0.21s**	N/A	N/A
	ex2.3_simp	X	X	N/A	N/A
	ex2.6_simp	X	X	N/A	N/A
	ex3.8_simp	0.16s	**0.15s**	N/A	N/A
	ex5.2_simp	X	X	N/A	N/A
	ex5.3_simp	X	X	N/A	N/A
	ex2.5_finVS	**0.16s**	0.17s	N/A	N/A
	ex3.11_finVS	**0.16s**	0.19s	N/A	N/A
	ex3.5_ulw	X	**0.21s**	N/A	N/A

We then supplied the grammar G_{BV} (from RQ1) to restrict the search space for the bitvector benchmarks. When given the grammar G_{BV}, Wuldo solved 4/6 benchmarks with summaries to synthesize, taking on average 0.43s (see the G_{BV} columns of Table 2). Note that SWAP_XOR requires no summaries or invariants, so we excluded it from this study. For the 4 solved benchmarks, Wuldo took on average 53% less time than when summaries were manually provided.

Findings. When no grammar for summaries is specified, Wuldo can find summaries only on benchmarks with simple, quantifier-free summaries. Given a summary/invariant grammar, Wuldo can synthesize these components, but its effectiveness relies on the power of the underlying SyGuS solver—i.e., improvements in SyGuS solvers will improve Wuldo's ability to generate W-UL proofs when summaries and invariants are not provided. These experiments suggest our approach for synthesizing summaries/invariants is at least feasible, but given the cost of summary verification in RQ1, a more sophisticated summary-synthesis approach may be necessary in practice.

RQ3: How effective is the optimization in Wuldo?

When summaries were provided (Table 1), Wuldo could solve 4 Nope benchmarks that the unoptimized solver Wuldo$^-$ could not. For these 4 benchmarks, Wuldo could, for example, prove that the sum of two linear combinations over some variables must itself be a linear combination over the variables. Our encoding of existential quantifiers as second-order functions allowed cvc5 to determine that the coefficients of the sum should be the sum of the summands' coefficients. On the benchmarks Wuldo$^-$ could solve, Wuldo$^-$ remained 2.79 times as fast as Wuldo.

When summaries were not provided (Table 2), Wuldo solved 1 benchmark that the unoptimized solver Wuldo$^-$ could not. On the benchmarks Wuldo$^-$ solved, Wuldo was on average 0.5% faster (geomean).

Findings. Our optimization enables Wuldo to solve 5 more benchmarks than Wuldo$^-$. However, on the benchmarks that both solve, when summaries are provided Wuldo$^-$ is approximately 2.79 times as fast as Wuldo. Thus, our optimization Wuldo trades speed for proving power.

RQ4: Does Nonterminal Summary Reuse Improve the Speed of Wuldo?

RQ4 Design and Benchmarks. To assess the benefit of reusing pre-proven summaries, we designed two sets of program-synthesis problems and proved unrealizability over various templates that an enumerative synthesizer might consider.

Our first set of four synthesis problems are unrealizable synthesis problems derived from the Messy bitvector benchmarks. We took four benchmarks (ADD, MAX, NAND, XOR) with grammars of the form $E ::= x \mid y \mid f(x,y) \mid g(x,y) \mid \ldots$. We transformed these into grammars of the form $E' ::= x \mid y \mid g(x,y) \mid \ldots$; $S ::= E' \mid f(E',S)$. We then considered the smallest 100 templates of the form E', $f(E',E')$, $f(E',f(E',E'))$, and so on to generate 400 total benchmarks. Because the initial Messy problems were unrealizable, the specifications remain unrealizable over each template. We took the summaries described and used in RQ1 as summaries for E'.

To demonstrate that summary reuse is beneficial irrespective of non-looping G_{imp} constructs, and to demonstrate that reusable summaries need not be tuned to the verification problem at hand, we hand wrote three additional realizable synthesis problems. These synthesis problems require determining necessary constants (bitvec-and), if-then-else nesting (alternating-half-intervals), and sequence operations (frobenius). We gave Wuldo the most general formula for each recursive nonterminal, meaning that the summaries are not specific to the synthesis specification. Because enumerative synthesizers typically enumerate programs in the grammar by increasing AST size [9], we then proved unrealizability of the smallest 7 (unrealizable) templates—i.e., the first 7 templates that a synthesizer would consider.

For each synthesis problem, we enumerated program templates that we would want a synthesizer to prune and used Wuldo to prove that the templates were unrealizable. We ran Wuldo using two modes: *ctx*, where summaries of recursive nonterminals are provided and can be assumed to have already been verified, and *no_ctx*, where summaries of recursive nonterminals are provided but must be verified again for each template. We then compare the runtimes of each mode over the templates to understand how much time can be saved by reusing pre-proven summaries.

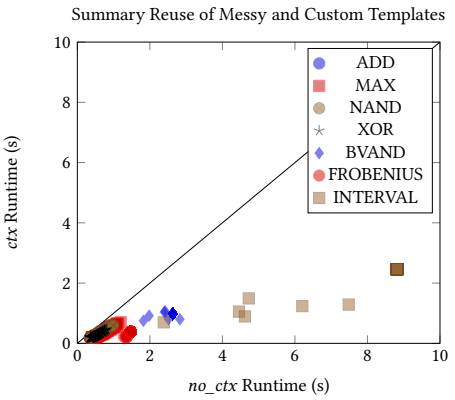

Fig. 5. Summary reuse experiments. The time to verify each template with summary verification (*no_ctx*) is plotted against the time to verify with pre-proven summaries (i.e., without summary verification) (*ctx*).

RQ4 Results. The results of our experiments are given in Figure 5. On all templates, verifying the provided summaries took over 2.1 times as long as applying pre-proven summaries to prove unrealizability.

On the hand-designed problems, verifying summaries took on average about 73% of the total verification time (*no_ctx*). Relative to re-proving summaries, by assuming that summaries were already proven, proofs were derived 2.61 times as fast for the bitvec-and templates, 5.24 times as fast for the alternating-single-interval templates, and 5.35 times as fast for the frobenius templates. On average, summary verification took about 72% of the total verification time (*no_ctx*). Assuming summaries were pre-proven made proof derivation on average 3.90 times as fast as proof derivation with summary reproving (*ctx*).

On the Messy-derived templates, verifying summaries took on average about 47% of the total verification time (*no_ctx*). Relative to re-proving summaries, by assuming that summaries were already proven, proofs were derived 2.05 times as fast for the MAX templates, 1.83 times as fast for the ADD templates, 1.83 times as fast for the NAND templates, and 1.94 times as fast for the XOR templates. On average, summary verification took about 47% of the total verification time (*no_ctx*), so assuming pre-proven summaries made proof derivation 1.91 times as fast on average.

Findings. Reuse of nonterminal summaries whose correctness is already established substantially speeds up proof construction. Although too slow for pruning in simple enumerative synthesis tasks (whose total runtime is often on the order of our verification step here [23]), the observed speedup conferred by summary reuse suggests that Wuldo can be useful in other contexts where many similar sets of programs must be analyzed (e.g., repair of programs with `Eval`). Moreover, Wuldo may be suitable for enumerative synthesis pruning over substantially larger grammars, or in contexts where program execution is expensive.

7 Related Work

Techniques for verifying sets of programs can be used for proving that a program is optimal with respect to a given quantitative objective [14], pruning parts of the search space explored during program synthesis [7, 15], verifying incomplete programs [19], and in other settings [5].

Tools for Proving Unrealizability. Nope [12] and Nay [13] prove unrealizability for SyGuS problems by encoding the unrealizability problem into queries dischargeable to external solvers (e.g., a program-reachability query in Nope, and a satisfiability query over a complex logic in Nay). As discussed in §6, while such encodings may be efficient in practice, they cannot produce proof

certificates. In addition, both Nope and Nay are limited to supporting SyGuS problems, while our work is capable of supporting any synthesis problem over an imperative programming language.

Similar to Nope and Nay, Messy [17] can verify whether a synthesis problem expressed in the SemGuS framework is unrealizable by encoding it as a satisfiability query over a set of Constrained Horn Clauses (CHCs)—i.e., it also cannot produce a proof certificate. However, Messy is also capable of identifying synthesis problems as realizable, whereas Wuldo can only show unrealizability in practice because SyGuS solvers fail to recognize unsatisfiability of PVCs. Moreover, Messy is also defined over the SemGuS framework, which lets Messy identify unrealizability over a wide variety of languages (e.g., regular expressions).

The last key distinction is that Messy, Nope, and Nay cannot reason about properties inexpressible over finitely many examples (e.g., monotonicity, boundedness, constant-valuedness, etc.). Moreover, W-UL and S-UL are both relatively complete proof systems on sets of loop-free programs.

8 Conclusion

Our work presents a sound and relatively complete algorithm for synthesizing W-UL proofs, along with an implementation Wuldo that, when nonterminal summaries are provided, demonstrates speed and breadth beyond existing techniques. Moreover, Wuldo's ability to reuse already proven summaries recommends its use for applications like pruning in enumerative synthesis, verification and repair of dynamically-loaded code, and verification of DSLs' properties. The principal limitation of Wuldo is the power of underlying solvers; as these solvers improve, so too will Wuldo.

Because our work is the first attempt at proof synthesis of properties of sets of programs, it opens a number of research questions and opportunities. First, the reach of Wuldo is hampered by limited solver support for LIA with infinite arrays and Π_2^0 formulas (i.e., $\forall\exists$-formulas). These logical fragments are understudied, but our work provides a benchmark suite that can focus research efforts on building solvers for these theories. Second, our work raises the question of whether summary synthesis can be made more tractable by more clever/bespoke algorithms to solve PVCs, in a similar way to how loop-invariant synthesis is done in Hoare logic (e.g., by ICE learning [8]). In this vein, can strong summaries be obtained by gradually strengthening weaker ones, as done by Dillig et al. [6]? Finally, can G_{imp} and W-UL be extended to support additional language features (e.g., nondeterminism)? If so, can syntax-directed proof skeletons still be constructed?

Acknowledgments

This work was supported, in part, by a Microsoft Faculty Fellowship, a gift from Rajiv and Ritu Batra, and NSF under grants CCF-1750965, CCF-1918211, CCF-2023222, CCF-2211968, and CCF-2212558. 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.

Data-Availability Statement

We provide a comprehensive Docker image on Zenodo containing the source code of Wuldo, all benchmarks, and all dependencies and third-party tools [21].

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE, 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[2] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. Cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European*

Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I (Munich, Germany). Springer-Verlag, Berlin, Heidelberg, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[3] Régis Blanc, Ashutosh Gupta, Laura Kovács, and Bernhard Kragl. 2013. Tree interpolation in vampire. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 173–181. https://doi.org/10.1007/978-3-642-45221-5_13

[4] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. 2007. Certified self-modifying code. *SIGPLAN Not.* 42, 6 (jun 2007), 66–77. https://doi.org/10.1145/1273442.1250743

[5] Loris D'Antoni. 2023. Verifying Infinitely Many Programs at Once. In *Static Analysis*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer Nature Switzerland, Cham, 3–9. https://doi.org/10.1007/978-3-031-44245-2_1

[6] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 443–456. https://doi.org/10.1145/2509136.2509511

[7] 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

[8] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *International Conference on Computer Aided Verification*. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-319-08867-9_5

[9] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[10] Sankha Narayan Guria. 2023. *Program Synthesis with Lightweight Abstractions*. Ph. D. Dissertation. University of Maryland, College Park. https://doi.org/10.13016/dspace/txaj-5df6

[11] Charles Antony Richard Hoare. 2006. Procedures and parameters: An axiomatic approach. In *Symposium on semantics of algorithmic languages*. Springer, 102–116. https://doi.org/10.5555/63445.C1104361

[12] 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

[13] 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

[14] 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

[15] 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

[16] Jinwoo Kim, Loris D'Antoni, and Thomas Reps. 2023. Unrealizability logic. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 659–688. https://doi.org/10.1145/3571216

[17] 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/10.1145/3434311

[18] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 436–449. https://doi.org/10.1145/3192366.3192410

[19] 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

[20] Shaan Nagy, Jinwoo Kim, Loris D'Antoni, and Thomas Reps. 2024. Automating Unrealizability Logic: Hoare-style Proof Synthesis for Infinite Sets of Programs. https://doi.org/10.48550/arXiv.2401.13244 arXiv:2401.13244 [cs.PL]

[21] Shaan Nagy, Jinwoo Kim, Thomas Reps, and Loris D'Antoni. 2024. Wuldo Unrealizability Logic Proof Synthesizer. https://doi.org/10.5281/zenodo.12627576

[22] Ernst-Rüdiger Olderog. 1983. On the Notion of Expressiveness and the Rule of Adaption. *Theor. Comput. Sci.* 24 (1983), 337–347. https://api.semanticscholar.org/CorpusID:32890639

[23] Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 364–381. https://doi.org/10.1007/978-

3-319-66706-5_18
[24] Unrealizability Logic Corrigendum [n. d.]. Pending.
[25] Lu Zhengyang. 2023. Z3-Alpha: A Reinforcement Learning Guided Smt Solver. https://smt-comp.github.io/2023/system-descriptions/z3-alpha.pdf

Received 2024-04-04; accepted 2024-08-18