Synthesizing Specifications

KANGHEE PARK, University of Wisconsin-Madison, USA
LORIS D'ANTONI, University of Wisconsin-Madison, USA
THOMAS REPS, University of Wisconsin-Madison, USA

Every program should be accompanied by a specification that describes important aspects of the code's behavior, but writing good specifications is often harder than writing the code itself. This paper addresses the problem of synthesizing specifications automatically, guided by user-supplied inputs of two kinds: (*i*) a query Φ posed about a set of function definitions, and (*ii*) a domain-specific language $\mathcal{L}$ in which each extracted property φ_i is to be expressed (we call properties in the language $\mathcal{L}$-properties). Each of the φ_i is a *best* $\mathcal{L}$-property for Φ: there is no other $\mathcal{L}$-property for Φ that is strictly more precise than φ_i. Furthermore, the set $\{\varphi_i\}$ is exhaustive: no more $\mathcal{L}$-properties can be added to it to make the conjunction $\bigwedge_i \varphi_i$ more precise.

We implemented our method in a tool, SPYRO. The ability to modify both Φ and $\mathcal{L}$ provides a SPYRO user with ways to customize the kind of specification to be synthesized. We use this ability to show that SPYRO can be used in a variety of applications, such as mining program specifications, performing abstract-domain operations, and synthesizing algebraic properties of program modules.

CCS Concepts: • **Theory of computation** → **Program specifications**; **Abstraction**; • **Software and its engineering** → **Automated static analysis**; **Automatic programming**.

Additional Key Words and Phrases: Program Specifications, Program Synthesis

ACM Reference Format:
Kanghee Park, Loris D'Antoni, and Thomas Reps. 2023. Synthesizing Specifications. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 285 (October 2023), 30 pages. https://doi.org/10.1145/3622861

1 INTRODUCTION

Specifications make us understand how code behaves. They also have many uses in testing, verifying, repairing, and synthesizing code. Because programmers iteratively refine their code to meet a desired intent (that often changes along the way), writing and maintaining specifications is often harder than writing and maintaining the code itself. A number of approaches have been proposed for automatically generating specifications, but these approaches are restricted to certain types of specifications, limited types of properties, and are based on dynamic testing—i.e., they yield likely specifications that though correct on the observed test cases might be unsound in general.

In this paper, we present the first customizable framework for synthesizing provably sound, most-precise (i.e., "best") specifications from a given set of function definitions. Our framework can be used to mine specifications from code, but also to enable several applications where obtaining precise specifications is crucial—e.g., generating algebraic specifications for modular program synthesis [Mariano et al. 2019], automating sensitivity analysis of programs [D'Antoni et al. 2013], and enabling abstract interpretation for new abstract domains [Yao et al. 2021]. The engine/primitive that drives the framework is an algorithm for the following problem: ***Given a query*** Φ ***posed***

Authors' addresses: Kanghee Park, khpark@cs.wisc.edu, University of Wisconsin-Madison, , USA; Loris D'Antoni, loris@cs.wisc.edu, University of Wisconsin-Madison, , USA; Thomas Reps, reps@cs.wisc.edu, University of Wisconsin-Madison, , USA.

2475-1421/2023/10-ART285
https://doi.org/10.1145/3622861

about a set of function definitions, find a most-precise conjunctive formula—expressed in the user-supplied, domain-specific language (DSL) $\mathcal{L}$—that is implied by Φ.

Our algorithm synthesizes a set of properties $\{\varphi_i\}$—each expressed in $\mathcal{L}$—that are consequences of Φ. Each φ_i is a most-precise $\mathcal{L}$-property for Φ: there is no $\mathcal{L}$-property for Φ that is strictly more precise than φ_i. Furthermore, the set $\{\varphi_i\}$ is exhaustive: the conjunction of the properties $\bigwedge_i \varphi_i$ is a best $\mathcal{L}$-conjunction—i.e., no $\mathcal{L}$-properties can be added to make the conjunction more precise.

This primitive is quite flexible. For instance, if the user desires a specification of the *input-output behavior* of a function `foo`, then Φ is "$out = \texttt{foo}(in)$." Here the objective of specification synthesis is to perform a kind of "projection" operation that provides information about the behavior of `foo` solely in terms of the variables visible on entry and exit (i.e., the ones visible to a client of `foo`). On the other hand, if the user wants a specification of the properties of *combinations* of the stack operations `push` and `pop`, then Φ is $os_1{=}\texttt{push}(s_1, x_1) \wedge (os_2, x_2){=}\texttt{pop}(s_2)$. In this case, when given an appropriate DSL $\mathcal{L}$, our tool SPYRO synthesizes the $\mathcal{L}$-properties (*i*) $\texttt{eq}(os_1, s_2) \Rightarrow x_1{=}x_2$ (the element ox obtained after $(os, ox) = \texttt{pop}(\texttt{push}(s, x))$ is the pushed element x), (*ii*) $\texttt{eq}(os_1, s_2) \Rightarrow \texttt{eq}(s_1, os_2)$ (the stack os obtained after $(os, ox) = \texttt{pop}(\texttt{push}(s, x))$ is the original stack s), and (*iii*) $\texttt{eq}(os_2, s_1) \wedge x_1{=}x_2 \Rightarrow \texttt{eq}(os_1, s_2)$ (after $\texttt{pop}(s)$, a `push` of the popped element restores stack s).

Our approach is different from existing specification-mining algorithms in that it meets three important objectives: (*i*) **Expressiveness:** The synthesized specifications are not limited to a fixed type, but are customizable through the user-provided DSL. (*ii*) **Soundness:** The synthesized specifications are sound for all inputs to the function definitions, not just for a specified set of test cases.[1]. (*iii*) **Precision:** The synthesized specifications are precise in that no specification in the DSL $\mathcal{L}$ is more precise than the synthesized ones—both at the level of each individual $\mathcal{L}$-property φ_i and at the level of the synthesized best $\mathcal{L}$-conjunction $\bigwedge_i \varphi_i$.[2]

The Key Challenge. Finding a sound $\mathcal{L}$-property for Φ is trivial: "*true*" is always one. However, finding a *best* $\mathcal{L}$-property is difficult because it is a kind of optimization problem, requiring SPYRO to find a *most-precise* solution. Proving most-preciseness of an $\mathcal{L}$-property requires showing that synthesizing a more precise $\mathcal{L}$-property is an *unrealizable* problem—i.e., no such a property exists.

Key Ideas. Our algorithm is based on a form of counterexample-guided inductive synthesis (CEGIS) that iteratively accumulates positive and negative examples of Φ. We build upon the algorithm for synthesizing abstract transformers proposed by Kalita et al. [2022] and generalize it to the problem of synthesizing specifications. Our algorithm uses three key ideas (the first and third of which differ from Kalita et al.). First, although the overall goal of the framework is to obtain a specification as a conjunctive formula, the core algorithm that underlies the synthesis process "dives below" this goal and focuses on synthesizing a most-precise *individual conjunct* (i.e., a most-precise $\mathcal{L}$-property for Φ). The smaller size of individual $\mathcal{L}$-properties compared to a full conjunctive $\mathcal{L}$-conjunction is better aligned with the capabilities of synthesis tools. Second, to handle competing objectives of soundness and precision, the algorithm treats negative examples specially. Some negative examples *must* be rejected by the $\mathcal{L}$-property we are synthesizing and some *may* be rejected. Third, to speed up progress, the algorithm accumulates *must* negative examples *monotonically*, so that once a sound $\mathcal{L}$-property is identified, *may* negative examples change status to *must* negative examples, and the algorithm only searches for (better) $\mathcal{L}$-properties that also reject those examples.

[1]As explained in §2, the framework requires a definition of the semantics of the function symbols that appear in query Φ (e.g., `push`, `pop`, `reverse`, etc.) and DSL $\mathcal{L}$. The specification obtained with our framework is *sound with respect to the supplied semantics*, but our implementation sometimes uses bounded or approximate semantics

[2]Readers familiar with symbolic methods for abstraction interpretation [Reps et al. 2004] will recognize that our problem is an instance of the *strongest-consequence problem*. Given a formula Φ in logic $\mathcal{L}_1$ (with meaning function $[\![\cdot]\!]_1$), find the strongest formula Ψ in a different logic $\mathcal{L}_2$ (with meaning function $[\![\cdot]\!]_2$) such that $[\![\Phi]\!]_1 \subseteq [\![\Psi]\!]_2$.

Our Framework. The core algorithm is a CEGIS loop that handles some negative-example classifications as "maybe" constraints, and guarantees progress via monotonic constraint hardening until an $\mathcal{L}$-property is found that is both sound and precise. By repeatedly calling the core algorithm to synthesize incomparable $\mathcal{L}$-properties, a most-precise $\mathcal{L}$-conjunction is created.

The core algorithm relies on three simple primitives.

SYNTHESIZE: synthesizes an $\mathcal{L}$-property that accepts a set of positive examples and rejects a set of negative examples; it returns $\perp$ if no such a property exists.

CHECKSOUNDNESS: checks if the current $\mathcal{L}$-property is sound; if is not, CHECKSOUNDNESS returns a new positive example that the property fails to accept.

CHECKPRECISION: checks if the current $\mathcal{L}$-property is precise; if it is not, CHECKPRECISION returns a new $\mathcal{L}$-property that accepts all positive examples, rejects all negative examples, and rejects one more negative example (which is also returned).

Our current implementation of each primitive relies on satisfiability modulo theory (SMT) solvers, limiting the scope of our framework. Nevertheless, as long as one has implementations for such primitives, the algorithm is sound. When the DSL $\mathcal{L}$ is finite, the algorithm is also complete.

Contributions. Our work makes the following contributions:

- A formal framework for the problem of synthesizing best $\mathcal{L}$-conjunctions (§2).
- An algorithm for synthesizing best $\mathcal{L}$-conjunctions (§3).
- A tool that we implemented to support our framework, called SPYRO. There are two instantiations of SPYRO: SPYRO[SKETCH] and SPYRO[SMT], which have different capabilities (see §4).
- An evaluation of SPYRO on a variety of benchmarks, showcasing four different applications of SPYRO (§5): mining specifications [Lo et al. 2017], generating algebraic specifications for modular program synthesis [Mariano et al. 2019], automating sensitivity analysis of programs [D'Antoni et al. 2013], and enabling abstract interpretation for new abstract domains [Yao et al. 2021].

§6 discusses related work. §7 concludes. In the extended paper [Park et al. 2023a], §A contains proofs; §B contains implementation details; and §C contains further details about the evaluation.

2 PROBLEM DEFINITION

In this section, we define the problem addressed by our framework. Throughout the paper, we use a running example in which the goal is to synthesize interesting consequences of the following query, which allows obtaining properties of up to two calls of the list-reversal function.

$$ol_1 = \texttt{reverse}(l_1) \wedge ol_2 = \texttt{reverse}(l_2), \tag{1}$$

In particular, we are interested in identifying properties that are consequences of query formula (1) and are expressible in the DSL defined by the following grammar $\mathcal{L}_{list}$:

$$\begin{array}{rcl} D & := & \top \mid AP \mid AP \vee AP \mid AP \vee AP \vee AP \\ AP & := & \texttt{isEmpty}(L) \mid \neg\texttt{isEmpty}(L) \mid \texttt{eq}(L,L) \mid \neg\texttt{eq}(L,L) \mid S\ \{\leq\mid<\mid\geq\mid>\mid=\mid\neq\}\ S + \{0 \mid 1\} \\ S & := & 0 \mid \texttt{len}(L) \\ L & := & l_1 \mid l_2 \mid ol_1 \mid ol_2 \end{array} \tag{2}$$

An $\mathcal{L}$*-property* is a property expressible in a DSL $\mathcal{L}$. We say that an $\mathcal{L}$-property is *sound* if it is a consequence of—i.e., implied by—the given query formula Φ. The goal of our framework is to synthesize a set of incomparable *sound and most-precise* $\mathcal{L}$-properties (i.e., a conjunctive specification), not just any $\mathcal{L}$-properties. For example, the $\mathcal{L}_{list}$-property $\texttt{len}(l_1) \leq \texttt{len}(ol_1)$ is sound but not most-precise ($\texttt{len}(l_1) = \texttt{len}(ol_1)$ is a more precise sound $\mathcal{L}_{list}$-property).

In the rest of this section, we describe what a user of the framework has to provide to solve this problem, and what they obtain as output. The user needs to provide the following inputs:

Input 1: Query. The query formula consists of a finite set of atomic formulae $\Phi = \{\sigma_1, \ldots, \sigma_n\}$ (denoting their conjunction). Each atomic formula σ_i is of the form $o^i = f^i(x^i_1, \ldots, x^i_n)$, where o^i is an output variable, each x^i_j is an input variable, and f^i is a function symbol. In our running example, the query is given in Eq. (1).
Input 2: Grammar of $\mathcal{L}$-properties. The grammar of the DSL $\mathcal{L}$ in which the synthesizer is to express properties. In our example, the DSL $\mathcal{L}_{list}$ is defined in Eq. (2).
Input 3: Semantics of function symbols. A specification of the semantics of the function symbols that appear in query Φ (e.g., `reverse`) and in the DSL $\mathcal{L}$ (e.g., `len`).

We assume semantic definitions are given in—or can be translated to—formulas in some logic fragment. For pragmatic reasons, in our implementation semantic definitions are given as code that is then automatically transformed into first-order formulas. For instance, the semantics of `reverse` is given as a program in the SKETCH programming language [Solar-Lezama 2013] from which we automatically extract the following bounded semantics $\varphi^k_{\texttt{reverse}}(l, ol)$ for a given bound $k > 0$:

$$\begin{aligned} &\varphi^0_{\texttt{reverse}}(l, ol) := \bot \\ &\varphi^n_{\texttt{reverse}}(l, ol) := [\varphi_{\texttt{eq}}(l, []) \Rightarrow \varphi_{\texttt{eq}}(ol, [])] \wedge \\ &\qquad\qquad \exists v_{hd}, l_{tl}, l' [\varphi_{\texttt{eq}}(l, v_{hd} :: l_{tl}) \Rightarrow \varphi^{n-1}_{\texttt{reverse}}(l_{tl}, l') \wedge \varphi_{\texttt{snoc}}(l', v_{hd}, ol)] \ \textit{if}\ n > 0 \end{aligned} \tag{3}$$

We discuss this limitation—i.e., that the semantics is bounded—and how we mitigate it in Section 5.

Let V_Φ be the set of all variables in Φ. We use φ_Φ to denote the formula that exactly characterizes the space of valid models over the variables V_Φ in Φ. For example, let $\varphi_{\texttt{reverse}}(l, ol)$ be the formula that exactly characterizes the result of reversing a list l and storing the result in ol—e.g., $(l, ol) = ([1, 2], [2, 1])$ is a valid model of $\varphi_{\texttt{reverse}}$. We use $[\![\varphi]\!]$ to denote the set of models of a formula φ. Then, in our example $[\![\varphi_\Phi(l_1, ol_1, l_2, ol_2)]\!] = [\![\varphi_{\texttt{reverse}}(l_1, ol_1) \wedge \varphi_{\texttt{reverse}}(l_2, ol_2)]\!]$. Henceforth, we omit variables in formulas when no confusion should result, and merely write φ_Φ.
Output: Best $\mathcal{L}$-properties. The goal of our method is to synthesize a set of incomparable *sound and most precise* $\mathcal{L}$-properties that are consequences of query Φ. Ideally, the best $\mathcal{L}$-property would be one that exactly describes φ_Φ, but in general the language $\mathcal{L}$ might not be expressive enough to do so. We argue that this feature is actually a desirable one![3] The customizability of our approach via a DSL is what allows our work to focus on identifying small and readable properties (rather than complex first-order formulas), and to apply our method to different use cases (see §5).

Because in general there might not be an $\mathcal{L}$-property that is equivalent to φ_Φ, the goal becomes instead to find $\mathcal{L}$-properties that tightly approximate φ_Φ.

Definition 2.1 (A best $\mathcal{L}$-property)**.** *An $\mathcal{L}$-property φ is* a best $\mathcal{L}$-property *for a query Φ if and only if* (i) *φ is* sound *with respect to Φ: $[\![\varphi_\Phi]\!] \subseteq [\![\varphi]\!]$.* (ii) *$\varphi$ is* precise *with respect to Φ and $\mathcal{L}$: $\neg\exists\varphi' \in \mathcal{L}.\ [\![\varphi_\Phi]\!] \subseteq [\![\varphi']\!] \subset [\![\varphi]\!]$. We use $\mathcal{P}(\Phi)$ to denote the set of all best $\mathcal{L}$-properties for Φ.*

When we refer to "a sound $\mathcal{L}$-property," soundness is always relative to some φ_Φ. Strictly speaking, we should say "a φ_Φ-sound $\mathcal{L}$-property," but φ_Φ should always be clear from context.

A best $\mathcal{L}$-property is *a strongest consequence* of φ_Φ that is expressible in $\mathcal{L}$. Because $\mathcal{L}$ is constrained, there may be multiple, incomparable $\mathcal{L}$-properties that are all strongest consequences of φ_Φ—thus, we speak of *a* best $\mathcal{L}$-property and not *the* best $\mathcal{L}$-property. In our running example, $\texttt{len}(l_1) = \texttt{len}(ol_1)$ is a best $\mathcal{L}$-property and so is $\neg\texttt{eq}(ol_1, l_2) \vee \texttt{eq}(l_1, ol_2)$. (Stated as an implication: $\texttt{eq}(ol_1, l_2) \Rightarrow \texttt{eq}(l_1, ol_2)$.) The former states that the sizes of the input and output of reverse are the same, while the latter states that applying reverse twice to a list yields the same list.

The goal of this paper is to find semantically minimal sets of incomparable best $\mathcal{L}$-properties.

[3]The idea of imposing a limit on the expressibility of the language in which properties can be stated is related to the concept of *inductive bias* in machine learning [Mitchell 1997, §2.7]: When there is no inductive bias, "[a] concept learning algorithm is ... completely unable to generalize beyond the observed examples."

Definition 2.2 (Best $\mathcal{L}$-conjunction). *A potentially infinite set of $\mathcal{L}$-properties $\Pi = \{\varphi_i\}$ forms a* best $\mathcal{L}$-conjunction $\varphi_\wedge = \bigwedge_i \varphi_i$ *for query Φ if and only if (*i*) every $\varphi \in \Pi$ is a best $\mathcal{L}$-property for Φ; (*ii*) every two distinct $\varphi_i, \varphi_j \in \Pi$ are* incomparable*—i.e., $[\![\varphi_i]\!] \setminus [\![\varphi_j]\!] \neq \emptyset$ and $[\![\varphi_j]\!] \setminus [\![\varphi_i]\!] \neq \emptyset$; (*iii*) the set is* semantically minimal*—i.e., for every best $\mathcal{L}$-property $\varphi \in \mathcal{P}(\Phi)$ we have $[\![\varphi_\wedge]\!] \subseteq [\![\varphi]\!]$.*

While there can be multiple best $\mathcal{L}$-conjunctions, they are all logically equivalent and they are all equivalent to a strongest $\mathcal{L}$-conjunction. However, note that a strongest $\mathcal{L}$-conjunction is not necessarily a best $\mathcal{L}$-conjunction; it could contain $\mathcal{L}$-properties that are not best and could potentially have repeated or redundant $\mathcal{L}$-properties.

Theorem 2.1. *If $\varphi_\wedge$ is a best $\mathcal{L}$-conjunction, then its interpretation coincides with the conjunction of all possible best properties: $[\![\varphi_\wedge]\!] = [\![\bigwedge_{\varphi \in \mathcal{P}(\Phi)} \varphi]\!]$.*

We are now ready to state our problem definition:

Definition 2.3 (Problem definition). *Given query Φ, the concrete semantics φ_Φ for the function symbols in Φ, and a domain-specific language $\mathcal{L}$ with its corresponding semantic definition, synthesize a best $\mathcal{L}$-conjunction for Φ.*

As illustrated in Section 3, given query (1), the DSL in Eq. (2), and the semantic definitions of `reverse`, `isEmpty`, `len`, etc., our tool SPYRO synthesizes the set of $\mathcal{L}$-properties shown below in Eq. (4), and establishes that the conjunction of these properties is a best $\mathcal{L}$-conjunction. (For clarity, we write properties of the form $\neg a \vee b$ as $a \Rightarrow b$.)

$$\begin{array}{ccc} \mathtt{len}(l_1) = \mathtt{len}(ol_1) & \mathtt{len}(l_2) = \mathtt{len}(ol_2) & \mathtt{eq}(l_2, ol_2) \vee \mathtt{len}(l_2) > 1 \\ \mathtt{len}(l_1) > 1 \vee \mathtt{eq}(l_1, ol_1) & \mathtt{eq}(ol_2, l_1) \Rightarrow \mathtt{eq}(l_2, ol_1) & \mathtt{eq}(ol_1, ol_2) \Rightarrow \mathtt{eq}(l_1, l_2) \\ \mathtt{eq}(ol_1, l_2) \Rightarrow \mathtt{eq}(l_1, ol_2) & \mathtt{eq}(l_1, l_2) \Rightarrow \mathtt{eq}(ol_1, ol_2) & \end{array} \tag{4}$$

Even though `reverse` is a simple function, its corresponding best $\mathcal{L}$-conjunction (w.r.t. the DSL $\mathcal{L}$) for query (1) is non-trivial. For example, our approach can discover properties involving single function calls (e.g., `reverse` behaves like the identity function on a list of length 0 or 1), but also hyperproperties, i.e., properties involving multiple calls to the same function. For example, the property $\mathtt{eq}(ol_1, l_2) \Rightarrow \mathtt{eq}(l_1, ol_2)$ states that applying the `reverse` function twice to an input returns the same input, while the property $\mathtt{eq}(ol_1, ol_2) \Rightarrow \mathtt{eq}(l_1, l_2)$ shows that `reverse` is injective!

Moreover, because the user has control over the DSL $\mathcal{L}$, they can change the language in which properties are to be expressed. In particular, if the formulas returned by SPYRO are too complicated for the user's taste, they can modify $\mathcal{L}$ and reinvoke SPYRO until they are satisfied with the results.

Depending on the DSL, a best $\mathcal{L}$-conjunction may need to be an infinite formula.

Example 2.1 (Infinite $\mathcal{L}$-conjunction). *Consider again the running example, and assume we change the DSL to the one defined by the following grammar $\mathcal{L}_{inf}$:*

$$Root := l_1 = CL \wedge l_2 = ol_1 \Rightarrow ol_2 = CL \qquad CL := [\,] \mid 1 :: CL \mid 2 :: CL$$

where "::" denotes the infix `cons` *operator. There exists only one best $\mathcal{L}_{inf}$-conjunction, which has an infinite number of conjuncts.*

$$\begin{array}{cc} l_1 = [\,] \wedge l_2 = ol_1 \Rightarrow ol_2 = [\,] & l_1 = 1 :: [\,] \wedge l_2 = ol_1 \Rightarrow ol_2 = 1 :: [\,] \\ l_1 = 2 :: [\,] \wedge l_2 = ol_1 \Rightarrow ol_2 = 2 :: [\,] & l_1 = 1 :: 1 :: [\,] \wedge l_2 = ol_1 \Rightarrow ol_2 = 1 :: 1 :: [\,] \\ l_1 = 1 :: 2 :: [\,] \wedge l_2 = ol_1 \Rightarrow ol_2 = 1 :: 2 :: [\,] & \ldots \end{array}$$

Our implementation focuses on DSLs for which this problem does not arise. Assumptions on DSLs are formally discussed in Section 3.

All the inputs to the framework are reusable. To synthesize a best $\mathcal{L}$-conjunction for a different query Φ that still operates over lists, one only needs to supply the semantic definition of the functions in Φ, and (if needed) modify the variable names generated by nonterminal L of Eq. (2).

For example, for the function that takes a list and duplicates its entries $ol = \texttt{stutter}(l)$—e.g., $\texttt{stutter}([1,2]) = [1,1,2,2]$)—SPYRO synthesizes the following $\mathcal{L}$-conjunction using the DSL $\mathcal{L}_{list}$.

$$\begin{array}{c} \texttt{len}(ol){=}\texttt{len}(l){+}1 \lor \texttt{len}(l){>}1 \lor \texttt{isEmpty}(ol) \\ \texttt{len}(ol){\le}0 \lor \texttt{len}(l){=}1 \lor \texttt{len}(ol){>}\texttt{len}(l){+}1 \qquad \texttt{len}(ol){\le}0 \lor \texttt{len}(ol){>}\texttt{len}(l) \end{array} \tag{5}$$

Because our DSL does not contain multiplication by 2, SPYRO could not synthesize the property that states that the length of the output list is twice the length of the input list. If we modify the DSL $\mathcal{L}_{list}$ to contain multiplication by 2 and the ability to describe when an element appears both in the input and output list, SPYRO successfully synthesizes the following new best $\mathcal{L}$-properties:

$$\begin{array}{c} \texttt{len}(ol) = 2 \cdot \texttt{len}(l) \\ \forall v.\,(\exists x{\in}l.\,x{=}v) \Rightarrow (\exists x{\in}ol.\,x{=}v) \quad \forall v.\,(\exists x{\in}ol.\,x{=}v) \Rightarrow (\exists x{\in}l.\,x{=}v) \\ \forall v.\,(\forall x{\in}l.\,x{\le}v) \Rightarrow (\forall x{\in}ol.\,x{\le}v) \quad \forall v.\,(\forall x{\in}ol.\,x{\le}v) \Rightarrow (\forall x{\in}l.\,x{\le}v) \\ \forall v.\,(\forall x{\in}l.\,x{\ge}v) \Rightarrow (\forall x{\in}ol.\,x{\ge}v) \quad \forall v.\,(\forall x{\in}ol.\,x{\ge}v) \Rightarrow (\forall x{\in}l.\,x{\ge}v) \end{array} \tag{6}$$

The ability to modify the DSL empowers the user of SPYRO with ways to customize the type of properties they are interested in synthesizing. As we will show in Section 5, customizing the DSL also allows us to use SPYRO for different applications and case studies—e.g., synthesizing abstract transformers and algebraic properties of programs.

3 AN ALGORITHM FOR SYNTHESIZING BEST $\mathcal{L}$-CONJUNCTIONS

In this section, we present the main contribution of the paper: an algorithm for synthesizing a best $\mathcal{L}$-conjunction. The algorithm synthesizes one most-precise $\mathcal{L}$-property at a time. It keeps track of the $\mathcal{L}$-properties it has synthesized and uses this information to synthesize a new most-precise $\mathcal{L}$-property that is incomparable to all the ones synthesized so far.

3.1 Positive and Negative Examples

Given query Φ, the concrete semantics φ_Φ for the function symbols in Φ, and a domain-specific language $\mathcal{L}$ with its corresponding semantic definition, our algorithm synthesizes best $\mathcal{L}$-properties and a best $\mathcal{L}$-conjunction for Φ using an example-guided approach.

Definition 3.1 (Examples)**.** *Given a model e, we say that e is a* positive example *if $e \in [\![\varphi_\Phi]\!]$ and a* negative example *otherwise.*

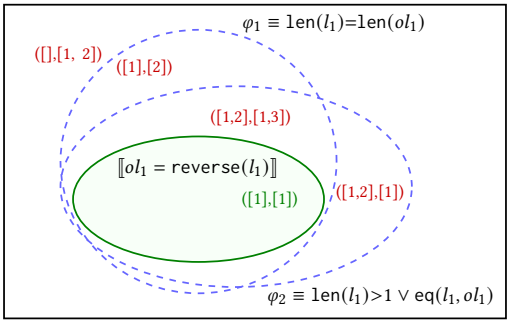
Given a formula φ and an example e, we write $\varphi(e)$ to denote $e \in [\![\varphi]\!]$ and $\neg\varphi(e)$ to denote $e \notin [\![\varphi]\!]$. Given a set of examples E, we write $\varphi(E)$ to denote $\wedge_{e\in E}\varphi_\Phi(e)$ and $\neg\varphi(E)$ to denote $\wedge_{e\in E}\neg\varphi_\Phi(e)$.

EXAMPLE 3.1. *Given the query $ol_1 = \texttt{reverse}(l_1)$, the model that assigns l_1 to the list $[1,2]$ and ol_1 to the list $[2,1]$ is a positive example. For brevity, we use the notation $([1,2],[2,1])$ to denote such an example. The following examples are negative ones: $([1],[2])$, $([1,2],[1,3])$, $([1,2],[1])$, $([],[1])$.*

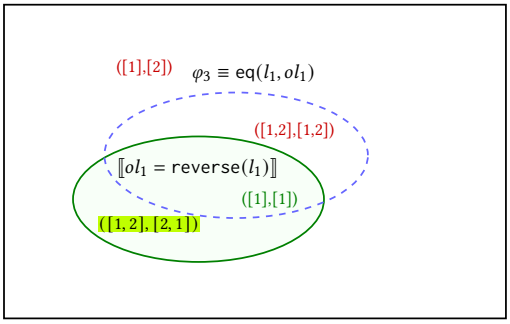
When considering the query from Eq. (1), which contains two calls on `reverse`*, $([1,2],[2,1],[1],[1])$ is a positive example (where the values denote l_1, ol_1, l_2, and ol_2, respectively), while $([1,2],[1,3],[1],[1])$ and $([1,2],[1,3],[1],[0])$ are negative examples.*

Intuitively, a best $\mathcal{L}$-property must accept all positive examples while also excluding as many negative examples as possible.

Positive examples can be treated as ground truth—i.e., a best $\mathcal{L}$-property should always accept all positive examples—but negative examples are more subtle. First, there can be multiple, incomparable,



(a) Positive and Negative examples. Different negative examples are rejected by different best $\mathcal{L}$-properties φ_1 and φ_2



(b) A possible soundness counterexample ([1, 2], [2, 1]) produced when performing CheckSoundness(φ_3, φ_Φ)

Fig. 1. The role of examples and CheckSoundness.

best $\mathcal{L}$-properties, each of which rejects a different set of negative examples. Second, there may be negative examples that no best $\mathcal{L}$-property can reject—they are accepted by *every* best $\mathcal{L}$-property.

Example 3.2. *Consider again the query* $ol_1 = \texttt{reverse}(l_1)$ *and the diagram shown in Figure 1a. The* $\mathcal{L}$*-properties* $\varphi_1 \equiv \texttt{len}(l_1) = \texttt{len}(ol_1)$ *and* $\varphi_2 \equiv \texttt{len}(l_1)>1 \vee \texttt{eq}(l_1, ol_1)$ *are both best* $\mathcal{L}$*-properties. While they both accept the positive example* $([1], [1])$ *and reject the negative example* $([], [1, 2])$*, we can see that* φ_1 *rejects the negative example* $([1, 2], [1])$ *whereas* φ_2 *accepts it. Similarly* φ_2 *rejects the negative example* $([1], [2])$ *whereas* φ_1 *accepts it. Finally, neither property rejects the negative example* $([1, 2], [1, 3])$*. In fact, no best* $\mathcal{L}$*-property in the given DSL can reject this example, and thus any best* $\mathcal{L}$*-conjunction will accept this negative example.*

In most cases, we use common datatypes—integers or lists—as the domain of our examples; however, more complicated definitions are sometimes required. If we are interested in queries involving binary search trees (BSTs), a tree datatype can describe the syntactic structure of the examples, but cannot capture BST invariants, e.g., for every node n, all values in the left (resp. right) subtree of n must be $\leq$ (resp. $\geq$) n's value. In our implementation, the set of valid BSTs is defined using the following sketch program—called a generator—that uses BST insertion operations to generate valid binary search trees:

```
generate_BST() := if (??) then emptyBST() else insert(generate_BST(), ??)
```

The code is then transformed into a bounded formula similar to the one in Eq. 3, but where the holes (i.e., ??) at each recursive call are replaced with unknown variables. In this case, different values for the holes result in different BSTs. In the rest of the paper, we assume that examples are only drawn from their valid domains regardless of how this domain is expressed.

3.2 Soundness and Precision

Now that we have established how examples relate to best $\mathcal{L}$-properties, we can introduce the two key operations that make our algorithm work: CheckSoundness and CheckPrecision. These two operations are similar to the ones used by Kalita et al. [2022] to synthesize abstract transformers, and are used by the inductive-synthesis algorithm to determine whether an $\mathcal{L}$-property is sound (i.e., a valid $\mathcal{L}$-property) and precise (i.e., a best $\mathcal{L}$-property). We modify the definition of CheckPrecision proposed by Kalita et al. [2022] to account for already synthesized best $\mathcal{L}$-properties and thus facilitate the synthesis of distinct best $\mathcal{L}$-properties.

3.2.1 Checking Soundness. Given an $\mathcal{L}$-property φ, CheckSoundness(φ, φ_Φ) checks whether φ is an overapproximation of φ_Φ. In other words, CheckSoundness checks if there exists a positive example $e^+ \in [\![\varphi_\Phi]\!]$ that is not accepted by φ; it returns that example if it exists, and $\perp$ otherwise. The soundness check can be expressed as $\exists e^+. \neg\varphi(e^+) \wedge \varphi_\Phi(e^+)$.

Example 3.3 (CheckSoundness). *Consider again the query $ol_1 = \texttt{reverse}(l_1)$. We describe how* CheckSoundness *operates using the example depicted in Figure 1b. The property $\varphi_3 \equiv \texttt{eq}(l_1, ol_1)$ is unsound because it does not overapproximate φ_Φ.* CheckSoundness$(\varphi_3, \varphi_\Phi)$ *would return a positive example that is incorrectly rejected by φ_3—e.g.,* $([1,2],[2,1])$. CheckSoundness$(\varphi_1, \varphi_\Phi)$ *on the property $\varphi_1 \equiv \texttt{len}(l_1) = \texttt{len}(ol_1)$ would instead return $\perp$ because the property φ_1 is sound (see Figure 1a).*

3.2.2 Checking Precision. Given an $\mathcal{L}$-property φ, a set of positive examples E^+ accepted by φ, a set of negative examples E^- rejected by φ, and a Boolean formula ψ denoting the set from which examples can be drawn, CheckPrecision$(\varphi, \psi, E^+, E^-)$, checks whether there exist an $\mathcal{L}$-property φ' and a negative example e^- such that: (*i*) φ' accepts all the positive examples in E^+ and rejects all the negative examples in E^-; (*ii*) $\psi(e^-)$ and φ' rejects e^-, whereas φ accepts e^-. Formally,

$$\exists \varphi', e^-.\ \psi(e^-) \wedge \varphi(e^-) \wedge \neg\varphi'(e^-) \wedge \varphi'(E^+) \wedge \neg\varphi'(E^-)$$

CheckPrecision can be thought of as a primitive that synthesizes a negative example and a formula that can reject such an example at the same time (or proves whether the synthesis problem does not admit a solution). The formula φ' is a witness that the negative example e^- can be rejected by some $\mathcal{L}$-property. In our algorithm, the set ψ is used to ensure that the negative example produced by CheckPrecision is not already rejected by best $\mathcal{L}$-properties we already synthesized.

Example 3.4 (CheckPrecision). *Consider again the query $ol_1 = \texttt{reverse}(l_1)$. We describe how* CheckPrecision *operates using the example depicted in Figure 2. The property $\varphi_4 \equiv \texttt{len}(l_1) > 0 \vee \texttt{eq}(l_1, ol_1)$ is sound but imprecise.*

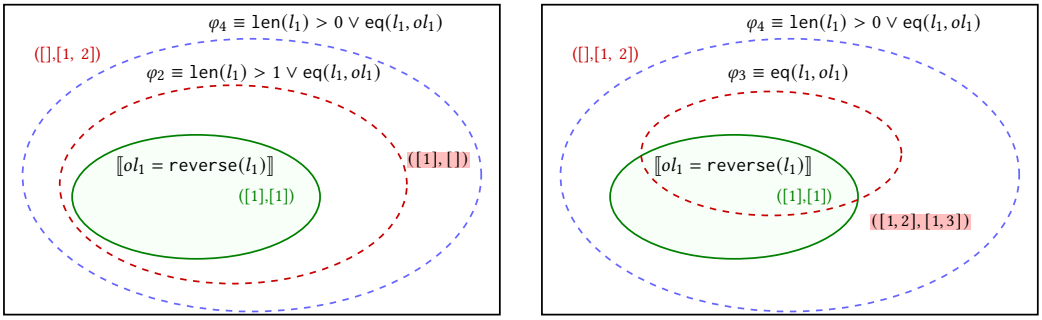
Figure 2a shows how running CheckPrecision$(\varphi_4, \neg\varphi_\Phi, \{([1], [1])\}, \{([], [1, 2])\})$ *could for example return $\varphi_2 \equiv \texttt{len}(l_1) > 1 \vee \texttt{eq}(l_1, ol_1)$ and the negative example $e^- = ([1], [])$.*

Figure 2b shows how running CheckPrecision$(\varphi_4, \neg\varphi_\Phi, \{([1], [1])\}, \{([], [1, 2])\})$ *could alternatively return $\varphi_3 \equiv \texttt{eq}(l_1, ol_1)$ and the negative example $e^- = ([1, 2], [1, 3])$. While φ_3 satisfies all the requirements of* CheckPrecision*—i.e., it correctly classifies the current positive and negative examples—the formula φ_3 is unsound because it incorrectly rejects, among others, the positive example $([1, 2], [2, 1])$. Moreover, in this case* CheckPrecision *has returned the negative example $([1, 2], [1, 3])$, which—as observed in Example 3.2—is not rejected by any sound $\mathcal{L}$-property!*

3.2.3 Monotonicity of Sound $\mathcal{L}$-properties. The fact that CheckPrecision can return (i) an unsound $\mathcal{L}$-property, and (ii) a negative example that cannot be rejected by any sound $\mathcal{L}$-property, makes designing a synthesis algorithm that can solve our problem challenging.

A key property that we exploit in our algorithm is that, under some assumptions on the language $\mathcal{L}$, once a sound $\mathcal{L}$-property is found, there must exist a best $\mathcal{L}$-property that implies it. This property allows searching for sound $\mathcal{L}$-properties in a monotonic manner. Once a sound $\mathcal{L}$-property is found, the search space can be narrowed down to a smaller set that includes all $\mathcal{L}$-properties that are more precise than the one already found. We will show in Theorem 3.4 that this narrowing is guaranteed to be finite under some assumptions about the language $\mathcal{L}$.

We say that a relation $\preceq$ on a set X is a *well-quasi order* if $\preceq$ is a reflexive and transitive relation such that any infinite sequence of elements $x_0, x_1, x_2, \ldots$ from X contains an increasing pair $x_i \preceq x_j$ with $i < j$. If the consequence relation $\Rightarrow$ for language $\mathcal{L}$ is a well-quasi order, we have that any descending sequence of sound $\mathcal{L}$-properties cannot be infinite—i.e., for any $\mathcal{L}$-property φ, there exist only finitely many $\mathcal{L}$-properties that imply φ. Moreover, a well-quasi order has no infinite



(a) Possible result (φ_2, ([1], [])) obtained when calling CheckPrecision on φ_4 and the current examples. φ_2 is *sound*.

(b) Possible result (φ_3, ([1, 2], [1, 3])) obtained when calling CheckPrecision on φ_4 and the current examples. φ_3 is *unsound*.

Fig. 2. Possible results produced by CheckPrecision

anti-chains (i.e., there are no infinite sequences of pairwise incomparable elements). Clearly, if $\mathcal{L}$ is *finite*, then $\Rightarrow$ is a well-quasi order, but finiteness is not a necessary condition. For example, consider the absolute-value function $o = \texttt{abs}(x)$, and a grammar $\mathcal{L}_{inf}$ that defines properties of the form $-20 \leq x \leq 10 \Rightarrow o \leq N$ (for any natural number N). The set of properties is infinite, but $\Rightarrow$ is a well-quasi order on the set of sound $\mathcal{L}_{inf}$-properties—i.e., for any concrete value of N, it is only possible to decrease the value of N and strengthen the property a finite number of times.

Lemma 3.1 (Monotonicity of Sound $\mathcal{L}$-properties). *If $\Rightarrow$ is a well-quasi order on the set of $\mathcal{L}$-properties, for every sound $\mathcal{L}$-property φ, there exists a best $\mathcal{L}$-property φ' such that $\varphi' \Rightarrow \varphi$.*

Consequently, if φ is a sound $\mathcal{L}$-property that rejects a set of negative examples E^-, there must exist a best $\mathcal{L}$-property that also rejects the examples in E^-. This property lets us infer when a set of negative examples *must* be rejected by a best $\mathcal{L}$-property (i.e., after a sound property is found).

3.3 Synthesizing One Most Precise $\mathcal{L}$-property

We are now ready to describe the method used to synthesize an individual best $\mathcal{L}$-property (Algorithm 1). The procedure SynthesizeStrongestConjunct takes as input

- φ_Φ: a formula describing the behavior of the program,
- ψ: a formula describing the domain from which examples can be drawn,
- φ_{init}: an initial sound $\mathcal{L}$-property that rejects all of the examples in E^-_{must},
- E^+: a set of positive examples that *must* be accepted by the returned $\mathcal{L}$-property,
- E^-_{must}: a set of negative examples that *must* be rejected by the returned $\mathcal{L}$-property,

and produces as output

- a sound $\mathcal{L}$-property φ that accepts all the examples in E^+, rejects all the examples in E^-_{must},
- an updated set of positive examples E^+ that the synthesized $\mathcal{L}$-property accepts,
- an updated set of negative examples E^-_{must} that the synthesized $\mathcal{L}$-property rejects.

We next discuss the procedure Synthesize, which SynthesizeStrongestConjunct uses to identify $\mathcal{L}$-properties that behave correctly on a finite set of examples.

Synthesis from examples. Besides CheckSoundness and CheckPrecision, which we have already discussed, SynthesizeStrongestConjunct uses the procedure Synthesize(E^+, E^-), which returns—when one exists—an $\mathcal{L}$-property φ that accepts all of the examples in E^+ (i.e., $\varphi(E^+)$) and

Algorithm 1: SynthesizeStrongestConjunct $(\varphi_\Phi, \psi, \varphi_{init}, E^+, E^-_{must})$

```
1  φ, φ_last ← φ_init; E⁻_may ← ∅                    // initial sound ℒ-property that rejects E⁻_must
2  while true do
3      e⁺ ← CheckSoundness(φ, φ_Φ)                                  // check soundness first
4      if e⁺ ≠ ⊥ then
5          E⁺ ← E⁺ ∪ {e⁺}                                  // unsound, update positive examples
6          φ′ ← Synthesize(E⁺, E⁻_must ∪ E⁻_may)                      // learn a new ℒ-property
7          if φ′ ≠ ⊥ then
8              φ ← φ′                                                // new candidate ℒ-property
9          else
10             φ ← φ_last; E⁻_may ← ∅        // no sound ℒ-property rejects E⁻_may, revert to φ_last
11     else
12         E⁻_must ← E⁻_must ∪ E⁻_may;   E⁻_may ← ∅   // sound, so E⁻_may example is added to E⁻_must
13         φ_last ← φ                                              // remember sound ℒ-property
14         e⁻, φ′ ← CheckPrecision(φ, ¬φ_Φ ∧ ψ, E⁺, E⁻_must)          // sound, check precision
15         if e⁻ ≠ ⊥ then
16             E⁻_may ← E⁻_may ∪ {e⁻}                              // update negative examples
17             φ ← φ′                                                  // new candidate formula
18         else
19             return φ, E⁺, E⁻_must ∪ E⁻_may                              // sound and precise
```

rejects all of the examples in E^- (i.e., $\neg\varphi(E^-)$). If no such a property exists—i.e., the synthesis problem is *unrealizable*—$\textsc{Synthesize}(E^+, E^-) = \bot$.

Example 3.5 (Example-based Synthesis). *Figure 1a showed that there may be negative examples that no sound $\mathcal{L}$-property can reject. Our algorithm uses E^-_{may} to handle such examples and make sure that they never end up in E^-_{must}. With $\mathcal{L}_{list}$ from Eq. (2), if $E^+ = \{([1],[1])\}$, $E^-_{must} = \{\}$, and $E^-_{may} = \{([1,2],[1,3])\}$, then $\textsc{Synthesize}(E^+, E^-_{must} \cup E^-_{may})$ can return the $\mathcal{L}$-property $\mathtt{len}(l_1) = 1$, which is clearly unsound. In this case, CheckSoundness will repeatedly add positive E^+ (without changing E^-_{must} and E^-_{may}) until $\textsc{Synthesize}(E^+, E^-_{must} \cup E^-_{may})$ returns $\bot$—e.g., when $E^+ = \{([1],[1]),([1,2],[2,1])\}$. Even when each member of a set of negative examples can be rejected (individually) by a sound $\mathcal{L}$-property, there may not exist a single sound $\mathcal{L}$-property that rejects all members of the set. For example, the negative examples $\{([1],[2]),([1,2],[1])\}$ in Figure 1a cannot both be rejected by a single best $\mathcal{L}$-property. If $E^+ = \{([1,2],[2,1])\}$, $E^-_{must} = \{([1],[]),([1],[2])\}$, and $E^-_{may} = \{([1,2],[1])\}$, then $\textsc{Synthesize}(E^+, E^-_{must} \cup E^-_{may})$ will return $\bot$.*

Preserved Invariants. We describe SynthesizeStrongestConjunct and the invariants it maintains.

Invariant 1: At the beginning of each loop iteration, the $\mathcal{L}$-property φ accepts all the examples in the current set E^+ and rejects all the examples in the current set $E^-_{must} \cup E^-_{may}$.

In each iteration, SynthesizeStrongestConjunct checks if the current property is sound using CheckSoundness (line 3). If the property is sound, it is then checked for precision using CheckPrecision (line 14). The algorithm terminates once the property is sound and precise (line 19).

After a new candidate is found by CheckPrecision, the set E^-_{may} stores one negative example that might be rejected by a sound $\mathcal{L}$-property. If CheckSoundness determines that φ is sound, the example in E^-_{may} is added to E^-_{must} (line 12) in accordance with Lemma 3.1. If a new negative example is returned by CheckPrecision, together with a new property φ', the set E^-_{may} is reinitialized, and φ' becomes the current property to check in the next iteration (lines 16 and 17).

The next invariant is guaranteed by Lemma 3.1 and by the fact that negative examples are added to E^-_{must} only when a sound property is found (line 12).

Invariant 2: There exists a sound $\mathcal{L}$-property that rejects all the negative examples in E^-_{must}.

When E^-_{must} is augmented with E^-_{may} in line 12, SynthesizeStrongestConjunct stores in φ_{last} the current sound $\mathcal{L}$-property that rejects all of the negative examples in E^-_{must}. In the next iteration, if a new positive example is returned by CheckSoundness, the positive examples are updated, and a new property is synthesized using Synthesize (line 6). If Synthesize cannot synthesize a property that is consistent with the new set of examples, spyro discards the conflicting negative example in E^-_{may} and reverts to the last sound property it found (line 10).

Invariant 3: At the beginning of each loop iteration, the $\mathcal{L}$-property φ_{last} accepts all of the examples in the current set E^+ and rejects all of the examples in the current set E^-_{must}.

Monotonically increasing the set of negative examples E^-_{must} when a sound $\mathcal{L}$-property is found is one of the contributions of our algorithm. While the algorithm is sound even without line 12, this step prevents the algorithm from often oscillating between multiple best $\mathcal{L}$-properties throughout its execution. We found that this optimization gives a 3.06% speedup to the algorithm (see §5.5).

Example 3.6 (Algorithm 1 Run). *Consider again the query* $ol_1 = \texttt{reverse}(l_1)$, *and a call to* SynthesizeStrongestConjunct $(\varphi_\Phi, \top, \top, \emptyset, \emptyset)$*—i.e.,* $\varphi_{init} = \psi = \top$ *and* $E^+ = E^-_{must} = \emptyset$.
Iteration 1. The run starts with CheckSoundness$(\top, \varphi_\Phi)$ *(line 3), which returns* $\bot$ *because the property* $\top$ *is sound. Then all negative examples in* E^-_{may} *are added to* E^-_{must} *(line 12), but because* $E^-_{may} = E^-_{must} = \emptyset$, *both sets remain empty.* CheckPrecision $(\top, \neg\varphi_\Phi, \emptyset, \emptyset)$ *(line 14) returns a new candidate* $\varphi_1 \equiv \texttt{eq}(l_1, ol_1)$ *with a negative example* $([], [1, 2])$ *(line 14). This negative example is added to* E^-_{may} *(line 16), and the code goes back to line 3.*
Iteration 2. CheckSoundness$(\varphi_1, \varphi_\Phi)$ *returns a positive example* $([1, 2], [2, 1])$, E^+ *is updated to* $\{([1, 2], [2, 1])\}$ *(line 5), and* Synthesize$(\{([1, 2], [2, 1])\}, \{([], [1, 2])\})$ *(line 6) returns a new candidate* $\varphi_2 \equiv \texttt{len}(l_1) \neq 0$.
Iteration 3. CheckSoundness$(\varphi_2, \varphi_\Phi)$ *returns a positive example* $([], [])$, E^+ *is updated to* $\{([1, 2], [2, 1]), ([], [])\}$ *(line 5).* Synthesize$(E^+, \{([], [1, 2])\})$ *(line 6) returns a new candidate* $\varphi_3 \equiv \texttt{len}(l_1) > 1 \vee \texttt{eq}(l_1, ol_1)$. E^-_{may} *is unchanged, and the code goes to line 3.*
Iteration 4. CheckSoundness$(\varphi_3, \varphi_\Phi)$ *returns* $\bot$ *because* φ_3 *is sound. Because the synthesizer found a sound property, the negative example in* E^-_{may} *is added to* E^-_{must} *(line 12)—i.e.,* $E^-_{must} = \{([], [1, 2])\}$. *Although* φ_3 *is a best* $\mathcal{L}$*-property,* CheckPrecision$(\varphi_3, \neg\varphi_\Phi, E^+, \{([], [1, 2])\})$ *(line 14) returns* $\varphi_4 \equiv \texttt{len}(l_1) = \texttt{len}(ol_1)$ *with a negative example* $([1, 2], [1])$. E^-_{may} *is set to* $\{([1, 2], [1])\}$, *and the code goes to line 3. The current set of examples is not enough for* CheckPrecision *to prove that* φ_3 *is indeed a best* $\mathcal{L}$*-property—i.e.,* CheckPrecision *was able to satisfy all requirements from §3.2.2 by finding a different,* incomparable $\mathcal{L}$*-property* φ_4 *and an additional negative example.*
Iteration 5. CheckSoundness$(\varphi_4, \varphi_\Phi)$ *returns* $\bot$, *and the negative example* $([1, 2], [1])$ *in* E^-_{may} *is added to* E^-_{must} *(line 12).* CheckPrecision$(\varphi_4, \neg\varphi_\Phi, E^+, E^-_{must})$ *(line 14) returns* $\bot$, *which means that we found a sound and precise* $\mathcal{L}$*-property.* SynthesizeStrongestConjunct *terminates with* $E^-_\delta = \emptyset$, $\varphi \equiv \varphi_4 \equiv \texttt{len}(l_1) = \texttt{len}(ol_1)$, $E^+ = \{([1, 2], [2, 1]), ([], [])\}$, *and* $E^-_{must} = \{([], [1, 2]), ([1, 2], [1])\}$.

The last invariant merely states that all the examples in E^-_{may} are elements of ψ.

Invariant 4: For every example $e \in E^-_{may}$, we have $\psi(e)$.

We say that a sound $\mathcal{L}$-property φ is *precise for φ_Φ with respect to ψ* if there does not exists a negative example $e^- \in [\![\psi]\!]$ and $\mathcal{L}$-property φ' such that $\varphi_\Phi \Rightarrow \varphi' \Rightarrow \varphi$ and φ' rejects e^-, whereas φ accepts e^-. The following lemma characterizes the behavior of SynthesizeStrongestConjunct.

Lemma 3.2 (Soundness and Relative Precision of SynthesizeStrongestConjunct). *If* SynthesizeStrongestConjunct *terminates, it returns a sound $\mathcal{L}$-property φ that accepts all the examples in E^+, rejects all the examples in $E^-_{must} \cup E^-_{may}$, and is precise for φ_Φ with respect to ψ.*

3.4 Synthesizing a Most-Precise $\mathcal{L}$-conjunction

In this section, we present SynthesizeStrongestConjunction (Algorithm 2), which uses SynthesizeStrongestConjunct to synthesize a best $\mathcal{L}$-conjunction of $\mathcal{L}$-properties.

On each iteration, SynthesizeStrongestConjunction maintains a conjunction of best $\mathcal{L}$-properties $\varphi_\wedge$, and uses SynthesizeStrongestConjunct to synthesize a best $\mathcal{L}$-property that rejects some negative examples that are still accepted by $\varphi_\wedge$ (i.e., negative examples in $\varphi_\wedge \wedge \neg\varphi_\Phi$). It also maintains the set of positive examples E^+ that have been observed so far.

Each iteration performs three steps: First, it uses SynthesizeStrongestConjunct to try to find an $\mathcal{L}$-property φ that rejects new negative examples E^-_{must} that no $\mathcal{L}$-property synthesized so far could reject—i.e, by calling SynthesizeStrongestConjunct with $\psi = \varphi_\wedge \wedge \neg\varphi_\Phi$ (line 5).

Second, it checks whether φ rejects some example that was not rejected by $\varphi_\wedge$ (line 7). If it does not, the algorithm terminates, and returns the $\mathcal{L}$-properties in Π synthesized so far. They are all best $\mathcal{L}$-properties and their conjunction is a best $\mathcal{L}$-conjunction.

Finally, if we reach line 12, we know that φ rejects negative examples in E^-_{must} that $\varphi_\wedge$ did not reject. Furthermore, because of the guarantees of SynthesizeStrongestConjunct, φ is precise with respect to $\varphi_\wedge$—i.e., no sound $\mathcal{L}$-property φ' exists that can reject more negative examples in $\varphi_\wedge$ than φ could reject. However, there may be a more precise $\mathcal{L}$-property that rejects more negative examples outside of $\varphi_\wedge$ that φ does not reject, while still rejecting all the negative examples in E^-_{must}. The call to SynthesizeStrongestConjunct in line 12 addresses this issue; it computes a best $\mathcal{L}$-property starting from φ and makes sure that the $\mathcal{L}$-property obtained rejects everything in E^-_{must} while allowing negative examples to be computed anywhere—i.e., $\psi = \top$. (Compare this call with the one on line 5, which only allows negative examples to be drawn from $\varphi_\wedge$.) Because precision with respect to $\top$ implies actual precision, we have that when $\psi = \top$, if SynthesizeStrongestConjunct terminates, it returns the best $\mathcal{L}$-property φ for φ_Φ by Lemma 3.2.

Example 3.7 (Algorithm 2 Run). *Revisiting the query $ol_1 = \texttt{reverse}(l_1)$, assume that $\varphi_\wedge$ has been updated with the recently synthesized property $\texttt{len}(l_1) = \texttt{len}(ol_1)$. We now describe the execution of* SynthesizeStrongestConjunct *(φ_Φ, $\varphi_\wedge$, $\top$, $\{([1,2],[2,1]),([],[])\}$, $\emptyset$)—i.e., $\psi = \varphi_\wedge$, $\varphi_{init} = \top$, $E^+ = \{([1,2],[2,1]),([],[])\}$ and $E^-_{must} = \emptyset$.*
Iteration 1. The run starts with CheckSoundness$(\top, \varphi_\Phi)$ *(line 3), which returns $\bot$ because the property $\top$ is sound. Then all negative examples in E^-_{may} are added to E^-_{must} (line 12), but because $E^-_{may} = E^-_{must} = \emptyset$, both sets remain empty.* CheckPrecision *($\top$, $\varphi_\wedge \wedge \neg\varphi_\Phi$, E^+, $\emptyset$) (line 14) returns a new candidate $\varphi_1 \equiv \texttt{len}(l_1) = 0 \vee \neg\texttt{eq}(l_1, ol_1)$ with a negative example $([1,2],[1,2])$ (line 14). Note that this negative example satisfies $\psi = \varphi_\wedge$. This negative example is added to E^-_{may} (line 16), and the code goes back to line 3.*

Algorithm 2: SynthesizeStrongestConjunction(φ_Φ)

```
1  φ_∧ ← ⊤                                                  // conjunction of best ℒ-properties
2  Π ← ∅                                                    // set of best ℒ-properties
3  E⁺, E⁻_may ← ∅                                           // initialize examples
4  while ⊤ do
       // find sound φ that rejects examples in E⁻_must that are still in φ_∧
5      φ, E⁺, E⁻_must ← SynthesizeStrongestConjunct(φ_Φ, φ_∧, ⊤, E⁺, ∅)
6      if E⁻_must = ∅ then
7          e⁻ ← IsSat(φ_∧ ∧ ¬φ)                             // check if φ improves φ_∧
8          if e⁻ ≠ ⊥ then
9              E⁻_must ← {e⁻}
10         else
11             return Π                                     // return best ℒ-conjunction
       // refine φ to reject more examples outside φ_∧ if possible
12     φ, E⁺, _ ← SynthesizeStrongestConjunct(φ_Φ, ⊤, φ, E⁺, E⁻_must)
13     Π ← Π ∪ {φ}                                          // φ is a best ℒ-property
14     φ_∧ ← φ_∧ ∧ φ                                        // Update conjunction
```

Iteration 2. CheckSoundness(φ_1, φ_Φ) *returns a positive example* ([1], [1]), E^+ *is updated to* $\{([1,2],[2,1]),([],[]),([1],[1])\}$ *(line 5), and* Synthesize($E^+, E^-_{must} \cup E^-_{may}$) *(line 6) returns a new candidate* $\varphi_2 \equiv \texttt{len}(l_1) > 1 \vee \texttt{eq}(l_1, ol_1)$.
Iteration 3. CheckSoundness(φ_2, φ_Φ) *returns* $\bot$*, and the negative example* E^-_{may} *is added to* E^-_{must} *(line 12), but because* $E^-_{may} = \emptyset$*, the set* E^-_{must} *remains the same.* CheckPrecision($\varphi_2, \varphi_\wedge \wedge \neg\varphi_\Phi, E^+, E^-_{must}$) *(line 14) returns* $\bot$*, which means that we found a sound and precise* $\mathcal{L}$*-property. So* SynthesizeStrongestConjunct *terminates with* $\varphi \equiv \varphi_2 \equiv \texttt{len}(l_1) > 1 \vee \texttt{eq}(l_1, ol_1)$, $E^+ = \{([1,2],[2,1]),([],[]),([1],[1])\}$, $E^-_{must} = \{([1,2],[1,2])\}$ *and* $E^-_\delta = \emptyset$.

Every iteration of the loop computes a best $\mathcal{L}$-property, which is conjoined onto $\varphi_\wedge$ (line 14).

Invariant 5: Π is a set of incomparable best $\mathcal{L}$-properties.

We are now ready to show that SynthesizeStrongestConjunction is sound.

Theorem 3.3 (Soundness of SynthesizeStrongestConjunction). *If* SynthesizeStrongestConjunction *terminates, it returns a best* $\mathcal{L}$*-conjunction for* φ_Φ.

Note: In some settings, we might know a priori that certain $\mathcal{L}$-properties hold, and we would not want to waste time synthesizing them. In such a situation, the formula $\varphi_\wedge$ in Algorithm 2 can be initialized to hold those properties, in which case Algorithm 2 would synthesize only best $\mathcal{L}$-properties that are not subsumed by $\varphi_\wedge$. For example, consider synthesizing a specification for two calls of the list-reversal function, as shown in Section 2. We can initialize $\varphi_\wedge$ with trivial properties—such as $eq(l_1, l_2) \Rightarrow eq(ol_1, ol_2)$—that are true of every function definition. Furthermore, after synthesizing a property like $eq(l_2, ol_1) \Rightarrow eq(ol_2, l_1)$, we can also include the symmetric property $eq(l_1, ol_2) \Rightarrow (ol_1, l_2)$ in the conjunction. This approach enables us to effectively filter out redundant and trivial specifications during the synthesis process.

3.5 Completeness

We observe that in SynthesizeStrongestConjunct, because positive examples in E^+ are never removed, any property stronger than a property that fails CheckSoundness at line 3 is never considered again. Consequently, the sequence of unsound $\mathcal{L}$-properties in an execution of SynthesizeStrongestConjunct is non-strengthening. Thus, if a non-strengthening sequence of unsound $\mathcal{L}$-properties can only be finite, SynthesizeStrongestConjunct can only find finitely many unsound $\mathcal{L}$-properties.

Another key observation about SynthesizeStrongestConjunct is that at line 14 if CheckPrecision$(\varphi, \ldots)$ returns a sound property φ' with a negative example e^-, CheckSoundness will return $\perp$ in the next iteration, and the negative example e^- will be added to E^-_{must} (from E^-_{may} in line 12). Therefore, any property weaker than φ—i.e., φ on the current iteration— will never be considered during this execution of SynthesizeStrongestConjunct. (Recall from the definition of CheckPrecision$(\varphi, \ldots)$ that e^- satisfies φ.) Thus, if a non-weakening sequence of sound $\mathcal{L}$-properties can only be finite, SynthesizeStrongestConjunct can only find finitely many sound $\mathcal{L}$-properties.

Based on the above two observations, Theorem 3.4 provides a sufficient condition for our algorithm to terminate when DSL $\mathcal{L}$ generates an infinite set of formulas.

Theorem 3.4 (Relative Completeness). *Suppose that $\Rightarrow$ is a well-quasi order on the set of sound $\mathcal{L}$-properties. Let $\Leftarrow$ denote the inverse of $\Rightarrow$, and suppose that $\Leftarrow$ is a well-quasi order on the set of unsound $\mathcal{L}$-properties. If* Synthesize, CheckSoundness *and* CheckPrecision *are decidable on $\mathcal{L}$, then* SynthesizeStrongestConjunct *and* SynthesizeStrongestConjunction *always terminate.*

Above, our argument about the second observation involved line 12 of SynthesizeStrongestConjunct. Nevertheless, Theorem 3.4 remains valid even if we eliminate line 12 from SynthesizeStrongestConjunct—i.e., line 12 is an optimization.

During each iteration, SynthesizeStrongestConjunct either adds a new positive example to E^+ or adds a new negative example to E^-_{must}. As a result, the number of iterations is also limited by the size of the example domain.

Corollary 3.5. *Suppose that either $\mathcal{L}$ contains finitely many formulas, or the example domain is finite. If* Synthesize, CheckSoundness *and* CheckPrecision *are decidable on $\mathcal{L}$, then* SynthesizeStrongestConjunct *and* SynthesizeStrongestConjunction *always terminate.*

4 IMPLEMENTATION

We implemented our framework in a tool called spyro. Following §2, spyro takes the following inputs: (*i*) A query Φ for which spyro is to find a best $\mathcal{L}$-conjunction. (*ii*) The context-free grammar of the DSL $\mathcal{L}$ in which properties are to be expressed. (*iii*) A specification, as a logical formula, of the concrete semantics of the function symbols in Φ and $\mathcal{L}$. Synthesize and CheckPrecision may be undecidable synthesis problems in general, but we show that these primitives can be implemented in practice using program-synthesis tools that are capable of both finding solutions to synthesis problems and establishing that a problem is unrealizable (i.e., it has no solution).

We implemented two versions of spyro: spyro[smt] supports problems in which semantics are definable as SMT formulas, and spyro[sketch] supports arbitrary problems but relies on the bounded/underapproximated encoding of program semantics of the sketch language [Solar-Lezama 2013]. For the current implementations of spyro[smt] and spyro[sketch], it is necessary to give the inputs in slightly different forms. In particular, input (iii) is provided to spyro[smt] in SMT-Lib format, whereas it is provided to spyro[sketch] as a piece of code in the sketch programming language.

In SPYRO[SMT], CHECKSOUNDNESS is just an SMT query, and SYNTHESIZE and CHECKPRECISION can be expressed as SYGUS problems. For the latter two primitives, SPYRO runs two SYGUS solvers in parallel and returns the result of whichever terminates first: (i) CVC5 (v. commit `b500e9d`) [Barbosa et al. 2022], which is optimized for finding solutions to SYGUS synthesis queries, and (ii) a re-implementation of the constraint-based unrealizability-checking technique from [Hu et al. 2020] that is specialized for finding whether the output of SYNTHESIZE and CHECKPRECISION is $\perp$.

In SPYRO[SKETCH], SYNTHESIZE, CHECKSOUNDNESS, and CHECKPRECISION are all implemented by calling the SKETCH synthesizer (v. 1.7.6) [Solar-Lezama 2013]. We describe how each primitive is encoded in SKETCH in Appendix B and how SKETCH's encoding affects soundness in Section 5.

Timeouts. We use a timeout threshold of 300 seconds for each call to SYNTHESIZE, CHECKSOUNDNESS, and CHECKPRECISION. If any such call times out, SYNTHESIZESTRONGESTCONJUNCTION returns the current $\mathcal{L}$-conjunction, together with an indication that it might not be a best $\mathcal{L}$-conjunction. (However, each of the individual conjuncts in the returned $\mathcal{L}$-conjunction is a best $\mathcal{L}$-property.)

Additional Tooling. In our evaluation, we used Dafny [Leino and Wüstholz 2014] to verify that the properties obtained by SPYRO[SKETCH] were sound for inputs beyond the bounds considered by SKETCH. Furthermore, for the SYGUS benchmarks only, we invoked CVC5 to verify whether the properties obtained by SPYRO[SKETCH] *exactly* characterized the function (which is a sufficient condition for an answer to be a "best" answer).

5 EVALUATION

We evaluated the effectiveness of SPYRO through four case studies: specification mining (§5.1), synthesis of algebraic specifications for modular synthesis (§5.2), automating sensitivity analysis (§5.3), and enabling new abstract domains (§5.4). For each case study, we describe how we collected the benchmarks, present a *quantitative analysis* of the running time and effectiveness of SPYRO, and a *qualitative analysis* of the synthesised $\mathcal{L}$-conjunctions. In §5.5, we describe additional experiments to identify what parameters affect SPYRO's algorithm.

We ran all experiments on an Apple M1 8-core CPU with 8GB RAM. All results in this section are for the median run, selected from the results of three runs ranked by their overall synthesis time.

5.1 Application 1: Specification Mining

We considered a total of 45 general *specification-mining* problems to evaluate SPYRO: 7 syntax-guided synthesis (SYGUS) problems from the SYGUS competition [Alur et al. 2019], where the semantics of operations is expressed using SMT formulas; 24 type-directed synthesis problems from SYNQUID [Polikarpova et al. 2016], where the semantics of operations is expressed using SKETCH; and 14 problems we designed to cover missing interesting types of properties (11 had their semantics expressed using SKETCH and 3 had semantics expressed using SMT formulas). Cumulatively, we have 10 benchmarks for which the semantics of operations is expressed using SMT formulas, and 35 benchmarks for which the semantics of operations is expressed using SKETCH. For the SYGUS and SYNQUID benchmarks, we "inverted" the roles from the original benchmarks: given the reference implementation, SPYRO synthesized a specification. Each input problem consists of a set of functions (1 to 14 functions per problem, and the size of each function ranges from 1 to 30 lines of code per function). The largest problem contains 14 functions (8 list functions and 6 queue functions) and the file contains a total of 140 lines of code. Most functions are recursive and can call each other—e.g., `dequeue` calls `reverse`, which calls `snoc`, which calls `cons`.

For each set of similar benchmarks, we designed a DSL that contained operations that could describe interesting properties for the given set of problems. The construction of each DSL depended on syntactic information from the code: the number, types, and names of input and output variables,

constants, and function symbols used in the code. We included operations that are commonly used in each category, such as equality, size primitives, and emptiness checking, but avoided problem-specific information. For benchmarks involving data structures with structural invariants (e.g., stacks, queues, and binary search trees), we provided data-structure constructors that guaranteed that functions were only invoked with data-structure instances that satisfied the invariants. The exact grammars are described in Appendix C.1. A DSL designed for a specific problem domain was often reused by modifying what function symbols could appear in the DSL. Overall, we created 7 distinct grammars for 14 different SyGuS and arithmetic problems; 10 grammars for 72 Synquid problems; and 3 grammars for 7 Stack and Queue problems. Although all but one of the DSLs are finite, they are still large languages; our finite DSLs can express between 4 thousand and 14.8 trillion properties, thus making the problem of synthesizing specifications challenging.

5.1.1 Quantitative Analysis, Part 1: Performance. spyro[smt] synthesized best $\mathcal{L}$-conjunctions for 6/10 benchmarks for which the semantics was expressed using SMT formulas. It took less than 6 minutes each for it to solve the successful examples, and timed out on the remaining 4 benchmarks (`max4`, `arrSearch3`, `abs` with the grammar from Eq. (22), and hyperproperties of `diff`)—spyro[smt] typically times out when the synthesis algorithm requires many examples.

Although we did not consider this option in our initial set of benchmarks, for the 4 benchmarks on which spyro[smt] failed, we also encoded the semantics of the function symbols in Φ and $\mathcal{L}$ using sketch. spyro[sketch] could synthesize properties for all 4 benchmarks, and guaranteed that 1/4 were best $\mathcal{L}$-conjunctions (with respect to sketch's bounded semantics), but for the other 3 benchmarks (`max4`, `arrSearch3` and `diff`) spyro[sketch] timed out on a call to CheckPrecision. However, the 3 $\mathcal{L}$-conjunctions obtained by the time CheckPrecision timed out were indeed best $\mathcal{L}$-conjunctions: although most-preciseness was not shown by spyro[sketch] within the timeout threshold, we found—using an SMT solver—that the $\mathcal{L}$-conjunctions in hand on the synthesis round on which the timeout occurred defined the *exact* semantics of the functions of interest, which implies they were best $\mathcal{L}$-conjunctions. The 1 problem for which spyro[sketch] established most-preciseness terminated within 5 minutes. For the other 3 problems, if we disregard the last iteration—the one on which most-preciseness of the $\mathcal{L}$-conjunction was to be established—spyro[sketch] found a best $\mathcal{L}$-conjunction within 10 minutes.

spyro[sketch] could synthesize properties for 35/35 benchmarks for which the semantics was expressed using sketch, and guaranteed that 34/35 were best $\mathcal{L}$-conjunctions. It took less than 10 minutes to solve each List and Tree benchmark, except for the `branch` problem—spyro[sketch] took about 30 minutes to find the best $\mathcal{L}$-conjunction, but failed to show most-preciseness. It took less than 15 minutes to solve each Stack, Queue, and Integer-Arithmetic benchmark. For `nonlinSum`, spyro[sketch] was able to synthesize in 900 seconds a best $\mathcal{L}$-conjunction from a grammar that contains ≈14.8 trillion properties (see Eq. (23)).

As a baseline, we compared the running time of spyro to an estimate of the running time of an algorithm that enumerates all sound properties in $\mathcal{L}$. For each benchmark, we estimated the cost of enumerating all terms in the grammar and checking for their soundness by multiplying the size $|\mathcal{L}|$ of the language generated by the grammar by the average running time of each call to CheckSoundness observed when running spyro on the same benchmark. As shown in Table 1, while spyro demonstrated a small estimated speedup for smaller problems like `emptyQueue` (i.e., 3.5×, with $|\mathcal{L}| = 64$), spyro was 2-5 orders of magnitude faster than the baseline for problems with large languages—$10^4 \leq |\mathcal{L}| \leq 6 \cdot 10^7$—and 8-10 orders of magnitude faster than the baseline for the two problems with very large languages—$10^{10} \leq |\mathcal{L}| \leq 1.5 \cdot 10^{13}$.

Together, spyro[sketch] and spyro[smt] synthesized properties for 41/45 benchmarks (45/45 if we consider the 4 benchmarks rewritten using a sketch semantics), and guaranteed that 40 were

best $\mathcal{L}$-conjunctions (44 if we consider the 4 benchmarks rewritten using a SKETCH semantics and our further analysis using an SMT solver).

5.1.2 Quantitative Analysis, Part 2: Soundness. To assess whether the properties synthesized by SPYRO[SKETCH] were indeed sound beyond the given input bound considered by SKETCH, we used an external verifier: Dafny [Leino and Wüstholz 2014] (a general purpose semi-automatic verifier). Dafny successfully verified that 23/35 $\mathcal{L}$-conjunctions synthesized by SPYRO[SKETCH] on non-SYGUS benchmarks were sound without any manual input from us. We could increase this number to 33/35 by providing invariants or some logical axioms to Dafny—e.g., $(\forall l.\, \texttt{len}(l) \geq 0)$. Dafny failed to verify properties synthesized from `enqueue` and `reverse`, which require a more expressive $\mathcal{L}$ to describe the order of elements.

5.1.3 Qualitative Analysis. Fig. 3 shows the properties synthesized by SPYRO on one of our three runs. In the SYGUS benchmarks, "[SKETCH]" denotes cases in which SPYRO[SKETCH] terminated with a semantics defined using SKETCH, but SPYRO[SMT] did not with a semantics defined using SMT formulas. Due to space constraints, we omit `max4` and `arrSearch3`. They are similar to `max3` and `arrSearch2`, respectively, but result in many properties.

SYGUS Benchmarks. The $\mathcal{L}$-conjunctions synthesized by SPYRO[SMT] and SPYRO[SKETCH] are more precise or equivalent to the original specifications given in the SYGUS problems themselves. In fact, SPYRO found $\mathcal{L}$-conjunctions that define the exact semantics of the given queries. Inspired by this equivalence, we attempted to use a SYGUS solver (CVC5) on the SYGUS benchmarks to synthesize an exact formula: we used a grammar of conjunctive properties (including the "and" operator, unlike the grammars used by SPYRO), and the specification was the semantics of the function. For 6/8 cases, CVC5 timed out, thus showing that our approach (of synthesizing one $\mathcal{L}$-property at a time) is beneficial even in the artificial situation in which an oracle supplies the semantics of the best $\mathcal{L}$-conjunction. Moreover, directly synthesizing an $\mathcal{L}$-conjunction—as CVC5 attempts—can yield a set of conjuncts of which some are not most-precise $\mathcal{L}$-properties.

Synquid Benchmarks. To evaluate the synthesized properties, we provided the synthesized $\mathcal{L}$-conjunctions to SYNQUID and asked it to re-synthesize the reference implementation from which we extracted the properties. In 12/16 cases, SYNQUID could re-synthesize the reference implementation. In 4/16 cases—`elemIndex`, `ith`, `reverse`, and `stutter`—the synthesized properties were not precise enough to re-synthesize the reference implementation. For example, as stated at the end of §2, for the `stutter` benchmark our DSL did not contain multiplication by 2 and SPYRO could not synthesize a property stating that the length of the output list is twice the length of the input list. After modifying the DSL to contain multiplication by 2 and the ability to describe when an element appears in both the input and output lists, SPYRO successfully synthesized 7 properties in 154.71 seconds (see Eq. 6). From the augmented set of properties, SYNQUID could synthesize the reference implementation of `stutter`. This experiment shows how the ability to modify the DSL empowers the user of SPYRO with ways to customize the type of properties they are interested in synthesizing.

Other Benchmarks. A core property of Stack is the principle of Last-In First-Out (LIFO). SPYRO was able to synthesize a simple formula that captures LIFO by looking at the relationship between `push` and `pop`: given the query $os_1{=}\texttt{push}(s_1, x_1)$ and $(os_2, x_2){=}\texttt{pop}(s_2)$, SPYRO synthesized the properties $\texttt{eq}(os_1, s_2) \Rightarrow x_1{=}x_2$ and $\texttt{eq}(os_1, s_2) \Rightarrow \texttt{eq}(s_1, os_2)$.

A Queue is a data structure whose formal behavior is somewhat hard to describe. Unlike Stack, the behavior of a Queue is not expressible by a simple combination of input and output variables. SPYRO could synthesize formulas describing the behavior of each Queue operation by providing a conversion function from a Queue consisting of two Lists into a List. For the query $(oq, x) = \texttt{dequeue}(q)$, SPYRO synthesized the property $\texttt{eq}(\texttt{toList}(q), \texttt{cons}(x, \texttt{toList}(oq)))$.

SyGuS

o=max2(x_1, x_2)
$o{=}x_1 \Rightarrow o{=}x_2$
$x_2{=}x_1 \lor o{>}x_1 \lor o{>}x_2$

o=max3(x_1, x_2, x_3)
$(x_3{\geq}x_1 \land x_3{\geq}x_2) \Rightarrow o{=}x_3$
$x_2{=}x_3 \lor o{>}x_2 \lor o{>}x_3$
$x_1{<}x_2 \lor o{=}x_1 \lor o{=}x_3$
$x_2{=}x_1 \lor o{>}x_2 \lor o{>}x_1$
$x_1{<}x_3 \lor o{=}x_1 \lor o{=}x_2$

o=diff(x, y)
$x{=}y{+}o \lor y{=}x{+}o$
$x{\geq}y \Rightarrow x{=}o{+}y$
$x{>}y{+}y \lor y{=}x{+}o \lor 0{<}x{+}o$
$x{\leq}y \Rightarrow y{=}o{+}x$

o_1=diff(x_1, y_1) [SKETCH]
o_2=diff(x_2, y_2)
$(y_2{=}x_2 \land x_1{=}y_1) \Rightarrow o_2{=}o_1$
$(x_2{=}x_1 \land y_1{=}y_2) \Rightarrow o_1{=}o_2$
$(x_1{=}y_1 \land o_1{=}o_2) \Rightarrow y_2{=}x_2$
$(o_1{=}o_2 \land x_2{=}y_2) \Rightarrow y_1{=}x_1$
$(x_2{=}y_1 \land y_2{=}x_1) \Rightarrow o_2{=}o_1$

o=arrSearch2(x_1, x_2, k)
$x_2{\leq}k \lor o{=}0 \lor o{=}1$
$k{<}x_2 \lor o{=}0 \lor o{=}2$
$(k{\geq}x_1 \land k{<}x_2) \Rightarrow o{=}1$
$(k{\geq}x_1 \land k{\geq}x_2) \Rightarrow o{=}2$
$k{<}x_1 \Rightarrow o{=}0$

o=abs(x) (Eq. ??)
$-x{\geq}x \Rightarrow -o{=}x \qquad x{\geq}-x \Rightarrow o{=}x$

o=abs(x) (Eq. ??) [SKETCH]
$-x{+}2o{=}0 \lor x{-}o{+}2{=}0$
$\lor x{+}2o{-}1{>}0$
$-x{+}o{=}0 \lor 2x{+}2o{=}0$

o=abs(x) (Eq. ??)
$-20{\leq}x{\leq}10 \Rightarrow o{\leq}20$

Other: integer arithmetic

o=linSum(x) (Eq. ??)
$o{=}x \lor -o{=}0$
$-x{=}x \lor 1{\leq}o \lor o{>}x$

o=linSum(x) (Eq. ??)
$4o{=}0 \lor 3x{-}3o{=}0$
$-3x{-}o{+}4{=}0 \lor -3x{+}4o{-}1{>}0$
$\lor 4x{-}o{=}0$

o=nonlinSum(x) (Eq. ??)
$x{>}1 \lor o{=}1 \lor o{=}0$
$-o{<}-x \lor o{=}1 \lor x{=}0$
$-o{=}o \lor x{=}1 \lor o{>}0$

o=nonlinSum(x) (Eq. ??)
$2o{=}0 \lor -x{+}2o{-}x^2{=}0$
$-4x{+}2o{-}x^2{=}0 \lor 3x{-}4o{+}x^2{\leq}0$
$-x{+}2o > 4$
$3x{+}o{>}4 \lor -3o{=}0 \lor -x{+}2o{-}4x^2{=}-3$

Queue

oq=emptyQueue()
isEmpty(toList(oq))

oq=enqueue(q, x)
eq(toList(oq), snoc(toList(q), x))

(oq, x)=dequeue(q)
eq(toList(q), cons(x, toList(oq)))

List

ol=append(l_1, l_2)
len(ol)=len(l_1)+len(l_2)
eq(l_2, ol) $\lor$ eq(l_1, ol) $\lor$ len(ol)>1
eq(l_2, ol) $\lor$ len(ol)=len(l_2)+1 $\lor$ len(l_1)>1
eq(l_1, ol) $\lor$ len(ol)>len(l_1)+1 $\lor$ len(l_2)=1

ol=deleteFirst(l, v)
eq(l, ol) $\lor$ len(l)=len(ol)+1
$(\exists x{\in}l.\, x{<}v) \Rightarrow (\exists x{\in}ol.\, x{<}v)$
$(\forall x{\in}l.\, x{\neq}v) \Rightarrow$ eq(l, ol)
$(\forall x{\in}l.\, x{\geq}v) \Rightarrow (\forall x{\in}ol.\, x{\geq}v)$
$(\exists x{\in}l.\, x{<}v) \lor$ len(l)=len(ol)+1
$\lor(\forall x{\in}ol.\, x{\neq}v)$
$(\exists x{\in}l.\, x{>}v) \Rightarrow (\exists x{\in}ol.\, x{>}v)$
$(\exists x{\in}l.\, x{=}v) \Rightarrow$ len(l)=len(ol)+1
$(\forall x{\in}l.\, x{\leq}v) \Rightarrow (\forall x{\in}ol.\, x{\leq}v)$

ol=delete(l, v)
$(\forall x{\in}l.\, x{\geq}v) \Rightarrow (\forall x{\in}ol.\, x{>}v)$
$(\forall x{\in}l.\, x{\neq}v) \Rightarrow$ eq(l, ol)
$(\exists x{\in}l.\, x{>}v) \Rightarrow (\exists x{\in}ol.\, x{>}v)$
eq(l, ol) $\lor$ len(l)$\geq$len(ol) + 1
$\forall x{\in}ol.\, x{\neq}v$
$(\exists x{\in}l.\, x{<}v) \Rightarrow (\exists x{\in}ol.\, x{<}v)$
$(\forall x{\in}l.\, x{\leq}v) \Rightarrow (\forall x{\in}ol.\, x{<}v)$

b=drop(l, v)
len(ol)=n+len(l) $\lor$ n>1
$\lor$len(l)=len(ol)+1
len(l)=n+len(ol)
eq(ol, l) $\lor$ n=1 $\lor$ len(l)>len(ol)+1

b=elem(l, v)
$b \Leftrightarrow (\exists x{\in}l.\, x{=}v)$

idx=elemIndex(l, v)
idx<len(l)
$idx{=}-1 \lor (\exists x{\in}l.\, x{=}v)$
$idx{>}-1 \lor (\forall x{\in}l.\, x{\neq}v)$
$(\exists x{\in}l.\, x{\neq}v) \lor 0{=}idx{+}1 \lor 0{=}idx$

o=min(l)
$\exists x{\in}l.\, x{=}o \qquad \forall x{\in}l.\, x{\geq}o$

ol=replicate(n, v)
len(ol)=n $\qquad \forall x{\in}ol.\, x{=}v$

ol=reverse(l)
eq(l, ol) $\lor$ len(l)>1 $\qquad$ len(l)=len(ol)

ol_1=reverse(l_1)
ol_2=reverse(l_2)
eq(ol_1, l_2) $\Rightarrow$ eq(ol_2, l_1)
eq(ol_2, l_1) $\Rightarrow$ eq(ol_1, l_2)
eq(ol_2, ol_1) $\Rightarrow$ eq(l_1, l_2)
eq(l_1, l_2) $\Rightarrow$ eq(ol_1, ol_2)

v=ith(l, idx)
$0{<}idx{+}1 \qquad$ len(l)=idx+1 $\lor$ len(l)>1
idx<len(l) $\qquad \exists x{\in}l.\, x{=}v$

ol=snoc(l, v)
len(ol)=len(l)+1 $\qquad \exists x{\in}l.\, x{=}v$
$(\forall x{\in}l.\, x{\geq}v) \Rightarrow (\forall x{\in}ol.\, x{\geq}v)$
$(\exists x{\in}l.\, x{<}v) \Rightarrow (\exists x{\in}ol.\, x{<}v)$
$(\exists x{\in}l.\, x{\neq}v) \Rightarrow (\exists x{\in}ol.\, x{\neq}v)$
$(\exists x{\in}l.\, x{>}v) \Rightarrow (\exists x{\in}ol.\, x{>}v)$
$(\forall x{\in}l.\, x{\leq}v) \Rightarrow (\forall x{\in}ol.\, x{\leq}v)$

ol=stutter(l)
len(ol)=len(l)+1 $\lor$ len(l)>1 $\lor$ eq(l, ol)
isEmpty(l) $\lor$ len(ol)>len(l)
eq(l, ol) $\lor$ len(ol)>len(l)+1 $\lor$ len(l)=1

ol=take(l, n)
n=len(ol) $\quad n$=len(l)$\lor$len(l)$\geq$len(ol)+1
eq(l, ol)$\lor$len(l)=n+1$\lor$len(l)>len(ol)+1

Binary Tree

ot=emptyTree()
isEmpty(ot)

ot=branch(v, t_1, t_2)
len(ot)=len(t_1)+len(t_2)+1
$(\exists x{\in}t_1.\, x{>}v) \Rightarrow (\exists x{\in}ot.\, x{>}v)$
$\exists x{\in}ot.\, x{=}v$
$(\exists x{\in}t_1.\, x{\neq}v) \Rightarrow (\exists x{\in}ot.\, x{\neq}v)$
$(\forall x{\in}t_1.\, x{\leq}v) \land (\forall x{\in}t_2.\, x{\leq}v)$
$\Rightarrow (\forall x{\in}ot.\, x{\leq}v)$
$(\exists x{\in}t_1.\, x{<}v) \Rightarrow (\exists x{\in}ot.\, x{<}v)$
$(\exists x{\in}t_2.\, x{\neq}v) \Rightarrow (\exists x{\in}ot.\, x{\neq}v)$
$(\forall x{\in}t_1.\, x{\geq}v) \land (\forall x{\in}t_2.\, x{\geq}v)$
$\Rightarrow (\forall x{\in}ot.\, x{\geq}v)$
$(\exists x{\in}t_2.\, x{<}v) \Rightarrow (\exists x{\in}ot.\, x{<}v)$
$(\exists x{\in}t_2.\, x{>}v) \Rightarrow (\exists x{\in}ot.\, x{>}v)$

b=elem(t, v)
$b \Leftrightarrow (\exists x{\in}t.\, x{=}v)$

ot_1=branch(v, t_1, t_2)
ot_2=left(t)
eq(t_2, t) $\Rightarrow \neg$eq(ot_1, ot_2)
eq(ot_1, t) $\Rightarrow$ eq(t_1, ot_2)
eq(t_1, t) $\Rightarrow \neg$eq(ot_1, ot_2)

ot_1=branch(v, t_1, t_2)
ot_2=right(t)
eq(t_2, t) $\Rightarrow \neg$eq(ot_1, ot_2)
eq(ot_1, t) $\Rightarrow$ eq(t_2, ot_2)
eq(t_1, t) $\Rightarrow \neg$eq(ot_1, ot_2)

ot_1=branch(v, t_1, t_2)
ov=rootval(t)
eq(ot_1, t) $\Rightarrow$ eq(v, ov)

Binary Search Tree

ot=emptyBST()
isEmpty(ot)

ot=insert(t, v)
len(ot)=len(t)+1 $\lor$ eq(t, ot)
$(\exists x{\in}ot.\, x{=}v)$
$(\forall x{\in}t.\, x{\geq}v) \Rightarrow (\forall x{\in}ot.\, x{\geq}v)$
$(\forall x{\in}t.\, x{<}v)\lor(\exists x{\in}ot.\, x{>}v)\lor$eq($t, ot$)
$(\forall x{\in}t.\, x{\leq}v) \Rightarrow (\forall x{\in}ot.\, x{\leq}v)$
$(\forall x{\in}t.\, x{>}v)\lor(\exists x{\in}ot.\, x{<}v)\lor$eq($t, ot$)

ot=delete(t, v)
$(\forall x{\in}t.\, x{\neq}v) \Rightarrow$ eq(t, ot)
$(\exists x{\in}t.\, x{=}v) \Rightarrow$ len(t)=len(ot)+1
$(\forall x{\in}t.\, x{\geq}v) \Rightarrow (\forall x{\in}ot.\, x{>}v)$
$\forall x{\in}ot.\, x{\neq}v$
$(\forall x{\in}t.\, x{\leq}v) \Rightarrow (\forall x{\in}ot.\, x{<}v)$
$(\exists x{\in}t.\, x{<}v) \Rightarrow (\exists x{\in}ot.\, x{<}v)$
$(\exists x{\in}t.\, x{>}v) \Rightarrow (\exists x{\in}ot.\, x{>}v)$

b=elem(t, v)
$b \Rightarrow (\exists x{\in}t.\, x{=}v)$
$(\exists x{\in}t.\, x{=}v) \Rightarrow b$

Stack

os=emptyStack()
isEmpty(os)

os=push(s, x)
len(os)=len(s)+1

(os, x)=pop(s)
len(s)=len(os)+1

os_1=push(s_1, x_1)
(os_2, x_2)=pop(s_2)
eq(os_1, s_2) $\Rightarrow x_1{=}x_2$
eq(os_1, s_2) $\Rightarrow$ eq(s_1, os_2)
eq(os_2, s_1) $\land x_1{=}x_2 \Rightarrow$ eq(s_1, s_2)

Fig. 3. $\mathcal{L}$-properties synthesized by SPYRO. Some properties are rewritten as implications for readability.

Finding: SPYRO can synthesize sound best $\mathcal{L}$-properties and mine specifications for most of the programs in our three categories. Furthermore, SPYRO synthesizes desirable properties that can be easily inspected by a user, who can then modify the DSL $\mathcal{L}$ to obtain other properties if desired.

5.2 Application 2: Synthesizing Algebraic Specifications for Modular Synthesis

In many applications of program synthesis, one has to synthesize a function that uses an existing implementation of certain external functions such as data-structure operations—i.e., synthesis is to be carried out in a modular fashion. Even if one has to synthesize a small function implementation, the synthesizer will need to reason about the large amount of code required to represent the external functions, which can hamper performance. Mariano et al. [2019] recently proposed a new approach to modular synthesis—i.e., functions are arranged in modules—where instead of providing the synthesizer with an explicit implementation of the external functions, one provides an algebraic specification—i.e., one that does not reveal the internals of the module—of how the functions in a module operate. For example, to describe the semantics of the functions `emptySet`, `add`, `remove`, `contains` and `size` in a `HashSet` module, one would provide the algebraic properties in Eq. 7, which describe appropriate data-structure invariants, such as handling of duplicate elements.

$$\begin{gathered}
\texttt{contains}(\texttt{emptySet}, x) = \bot \qquad x_1 = x_2 \Rightarrow \texttt{contains}(\texttt{add}(s, x_1), x_2) = \top \\
x_1 \neq x_2 \Rightarrow \texttt{contains}(\texttt{add}(s, x_1), x_2) = \texttt{contains}(s, x_2) \\
\texttt{remove}(\texttt{emptySet}, x) = \texttt{emptySet} \qquad x_1 = x_2 \Rightarrow \texttt{remove}(\texttt{add}(s, x_1), x_2) = \texttt{remove}(s, x_2) \\
x_1 \neq x_2 \Rightarrow \texttt{remove}(\texttt{add}(s, x_1), x_2) = \texttt{add}(\texttt{remove}(s, x_2), x_1)
\end{gathered} \tag{7}$$

While their approach has shown promise in terms of scalability, to use this idea in practice one has to provide the algebraic specifications to the synthesizer *manually*, a tricky task because these specifications typically define how multiple functions interact with each other.

In our case study, we used SPYRO to synthesize algebraic specifications for benchmarks used in the evaluation of JLIBSKETCH, an extension of the SKETCH tool that supports algebraic specifications [Mariano et al. 2019]. We considered the 3 modules—`ArrayList`, `HashSet`, and `HashMap`—that provided algebraic specifications, did not use `string` operations (our current implementation does not support strings), and did not require auxiliary functions that were not present in the implementation to describe the algebraic properties. For each module, JLIBSKETCH contained both the algebraic specification of the module and its mock implementation—i.e., a simplified implementation that mimics the intended library's behavior (e.g., `HashSet` is implemented using an array). Given the mock implementation of the module, we asked SPYRO to synthesize most-precise algebraic specifications.

For this case study, designing a grammar that accepted all possible algebraic specifications but avoided search-space explosion proved to be challenging. Instead, we opted to create multiple grammars for each module to target different parts of the algebraic specifications, and called SPYRO separately for each grammar. For example, if the JLIBSKETCH benchmark contained an algebraic specification $\texttt{size}(\texttt{add}(s, x)) = \texttt{size}(s) + 1$, we considered the grammar to contain properties of the form $\mathit{guard} \Rightarrow \texttt{size}(\texttt{add}(s, x)) = \mathit{exp}$. All the DSLs designed for algebraic specifications synthesis were reused by modifying what function symbols could appear in the DSL. The detailed grammars are presented in Appendix C.2.

SPYRO terminated with a best $\mathcal{L}$-conjunction for all the benchmarks (and grammars) in less than 800 seconds per benchmark. SPYRO was slower than the enumerative baseline presented in Section 5.1.1 for very small languages ($|\mathcal{L}| < 5$) but faster in all other cases. The speedups were not as prominent as for Application 1.

For all but one benchmark, the $\mathcal{L}$-conjunctions synthesized by SPYRO were equivalent to the algebraic properties manually designed by the authors of JLIBSKETCH. For the implementation

Table 1. Evaluation results of SPYRO. A few representatives benchmarks are selected from each application. A (*) indicates a timeout when attempting to prove precision in the last iteration. In that case, we report as total time the time at which SPYRO timed out. The Enum. column reports the estimated time required to run CHECKSOUNDNESS for all formulas in the DSL $\mathcal{L}$. This estimation is achieved by multiplying the size of the grammar by the average running time of the CHECKSOUNDNESS.

		Problem	$\lvert\mathcal{L}\rvert$	SYNTHESIZE		CHECKSOUNDNESS		CHECKPRECISION		Last Iter.	Enum.	Total
				Num	T(sec)	Num	T(sec)	Num	T(sec)	T(sec)	T(sec)	T(sec)
Application 1	SyGuS	max2	$1.57 \cdot 10^5$	6	0.13	12	0.06	9	1.37	0.05	787.32	1.55
		max3	$8.85 \cdot 10^5$	19	13.83	48	3.53	36	131.44	0.46	$6.51 \cdot 10^4$	148.79
		diff	$5.66 \cdot 10^7$	18	19.41	41	1.06	28	236.61	0.48	$1.46 \cdot 10^6$	257.08
		arrSearch2	$3.37 \cdot 10^6$	16	3.44	41	1.44	31	60.97	0.28	$1.19 \cdot 10^5$	65.84
	LIA	abs (Eq. 22)	$3.37 \cdot 10^6$	6	1.14	14	0.76	11	10.16	0.38	$1.83 \cdot 10^5$	12.05
		abs (Eq. 24)	∞	22	3.98	23	0.15	3	0.67	0.70	∞	4.80
	List	append	$4.97 \cdot 10^8$	29	46.62	68	81.50	43	91.46	57.59	$5.95 \cdot 10^8$	219.58
		delete	$2.99 \cdot 10^6$	24	39.49	66	87.15	49	107.07	18.41	$3.94 \cdot 10^6$	233.71
		min	$2.38 \cdot 10^5$	5	6.02	14	13.71	11	14.42	2.53	$2.38 \cdot 10^5$	34.15
		reverse	$1.73 \cdot 10^6$	6	7.28	16	18.11	12	16.84	7.83	$1.96 \cdot 10^6$	42.23
		reverse, reverse	$6.40 \cdot 10^4$	20	26.64	52	96.65	40	76.37	32.21	$1.19 \cdot 10^5$	199.66
	Stack	emptyStack	$1.25 \cdot 10^5$	2	2.12	6	5.61	5	5.89	2.05	$1.17 \cdot 10^5$	13.63
		push	$1.73 \cdot 10^6$	4	4.9	10	11.16	7	9.73	2.50	$1.91 \cdot 10^6$	25.79
		pop	$1.73 \cdot 10^6$	4	4.84	11	12.16	8	10.97	12.02	$1.93 \cdot 10^6$	27.98
		push, pop	$2.62 \cdot 10^5$	32	51.12	77	95.72	53	107.13	27.66	$3.26 \cdot 10^5$	253.97
	Queue	emptyQueue	64	2	2.18	6	5.86	5	9.81	2.07	62.51	17.85
		enqueue	$5.93 \cdot 10^5$	4	5.4	11	16.82	8	270.26	200.15	$9.06 \cdot 10^5$	292.48
		dequeue	$5.93 \cdot 10^5$	4	5.51	9	12.51	6	290.28	192.02	$8.24 \cdot 10^5$	308.30
	Arithmetic	linSum (Eq. 22)	$1.01 \cdot 10^7$	7	6.69	20	14.50	15	15.33	5.33	$7.31 \cdot 10^6$	36.52
		linSum (Eq. 23)	$2.90 \cdot 10^{10}$	15	15.31	99	71.72	88	112.87	2.70	$2.10 \cdot 10^{10}$	199.90
		nonlinSum (Eq. 22)	$1.01 \cdot 10^7$	8	7.95	30	21.58	25	26.48	2.53	$7.25 \cdot 10^6$	56.01
		nonlinSum (Eq. 23)	$1.48 \cdot 10^{13}$	17	16.82	121	102.71	107	734.23	48.21	$1.26 \cdot 10^{13}$	853.77
Application 2	HashSet	size, emptySet	3	2	0.62	6	1.80	5	1.62	0.62	0.90	4.04
		size, add	1,315	7	2.39	14	4.56	9	3.40	0.70	428.31	10.36
		contains, emptySet	3	2	0.60	6	1.80	5	1.64	0.64	0.90	4.04
		contains, add	55	5	1.64	12	3.94	9	3.28	0.69	18.06	8.86
		remove, emptySet	5	1	0.34	5	1.05	5	1.78	0.55	1.05	3.17
		remove, add	181	7	3.38	15	6.20	10	13.23	3.86	74.81	22.81
Application 3	Sensitivity Edit Dist.	append	16,385	43	29.18	82	118.52	43	55.27	7.95	$2.37 \cdot 10^4$	202.97
		cons	769	6	2.49	10	4.18	4	2.02	8.69	321.44	8.69
		cons_delete	769	19	9.00	36	15.80	19	12.70	11.75	337.51	37.50
		deleteFirst	769	33	17.9	68	115.43	42	31.53	1.31	1385.38	164.86
		delete	769	26	13.42	55	190.48	33	23.55	3.64	2663.26	227.45
		reverse	257	13	6.00	27	61.43	16	10.11	3.32	594.24	77.53
		snoc	769	29	15.44	61	37.21	36	25.72	5.80	700.07	78.38
		stutter	257	14	6.29	26	13.66	14	8.51	3.03	135.02	28.47
		tail	257	14	6.17	29	12.83	17	10.01	9.66	113.70	29.01
Application 4	BV Polyhedra	square	$1.68 \cdot 10^7$	20	14.79	76	24.65	60	136.76	68.99	$5.44 \cdot 10^6$	176.19
		half	$1.68 \cdot 10^7$	19	16.17	56	19.27	40	353.14	*	$5.77 \cdot 10^6$	388.57
		squareIneq	$1.68 \cdot 10^7$	45	105.24	70	20.24	28	95.41	98.74	$4.85 \cdot 10^6$	220.89
		conjunction	$1.68 \cdot 10^7$	46	73.08	67	20.49	23	102.32	117.45	$5.13 \cdot 10^6$	195.89
		disjunction	$1.68 \cdot 10^7$	48	178.05	62	22.69	15	102.73	105.71	$6.14 \cdot 10^6$	303.47

of HashMap provided in JLibSketch, for one specific grammar, spyro synthesized an empty $\mathcal{L}$-conjunction (i.e., the predicate *true*) instead of the algebraic specification provided by the authors of JLibSketch—i.e., $k_1 = k_2 \Rightarrow \texttt{get}(\texttt{put}(m, k_1, v), k_2) = v$. Upon further inspection, we discovered that the implementation of HashMap used in JLibSketch was incorrect and did not satisfy the specification the authors provided, due to an incorrect handling of hash collision! After fixing the bug in the implementation of HashMap, we were able to synthesize the algebraic specification. Because algebraic properties often involve multiple functions, we were not able to separately verify their correctness on all inputs using the Dafny verifier, but the fact that we obtained the same properties that the authors of JLibSketch specified in their benchmarks is a strong signal that our properties are indeed sound.

Finding: spyro can help automate modular synthesis by synthesizing precise algebraic specifications that existing synthesis tools can use to speed up modular synthesis. Thanks to spyro's provable guarantees, we were able to uncover a bug in one of JLibSketch's module implementations.

5.3 Application 3: Automating Sensitivity Analysis

Automatically reasoning about quantitative properties such as differential privacy [D'Antoni et al. 2013] in programs requires one to analyze how changes to a program input affect the program output—e.g., differential privacy typically requires that bounded changes to a function's input cause bounded changes to its output. A common and inexpensive approach to tackle this kind of problem is to use a compositional *sensitivity analysis* (either in the form of an abstract interpretation or of a type system [D'Antoni et al. 2013]) in which one tracks how sensitive each operation in a program is to changes in its input. For example, one can say that the function $f(x) = \texttt{abs}(2x)$, when given two inputs x_1 and x_2 that differ by k, produces two outputs that differ by at most $2k$.

While for the previous function, it was pretty easy to identify a precise sensitivity property, it is generally tricky to do so for functions involving data structures, such as lists, which are of interest in differential privacy when the list represents a database of individuals [Wang et al. 2016]. In this case study, we considered 9 list-manipulating functions (`append`, `cons`, `deleteFirst`, `delete`, `reverse`, `snoc`, `stutter`, `tail`, and `cons_delete`) and used spyro[sketch] to synthesize precise sensitivity properties describing how changes to the input lists affect the outputs. The `cons_delete` benchmark uses the query $\texttt{deleteFirst}(\texttt{cons}(x, l), x)$ involving the composition of two functions.

For each function f, we used spyro to synthesize a property of the form

$$guard(x_1, x_2) \wedge dist(l_1, l_2) \leq d \Rightarrow dist(f(l_1, x_1), f(l_2, x_2)) \leq exp$$

where *guard* can be the predicate true or an equality/inequality between x_1 and x_2 (the grammars vary across benchamrks), *dist* is the function computing the distance between two lists (we run experiments using both edit and Hamming distance), and the expression *exp* (the part to synthesize) can be any linear combinations of $\texttt{len}(x)$, $\texttt{len}(y)$, d, and constants in the range -1 to 2. When considering all combinations of functions, guards, and distances, we obtained 18 benchmarks. All the DSLs designed for algebraic specifications synthesis were reused by modifying what function symbols could appear in the DSL. The complete grammars are shown in Appendix C.3.

spyro terminated with a best $\mathcal{L}$-conjunction for all the benchmarks (and grammars) in less than 250 seconds per benchmark. spyro outperformed the enumerative baseline presented in Section 5.1.1 for every problem (3.14× speedup for Hamming-distance sensitivity problems and 11.93× speedup for edit-distance sensitivity problems—geometric mean).

We observed that even for simple functions, sensitivity properties are fairly complicated and hard to reason about manually. For example, spyro synthesizes the following sensitivity $\mathcal{L}$-property for

the function `deleteFirst` (D_e denotes the edit distance):

$$D_e(l_1, l_2) \leq d \Rightarrow D_e(\texttt{deleteFirst}(l_1, x_1), \texttt{deleteFirst}(l_2, x_2)) \leq d + 2$$

However, if we add a condition that the element removed from the two lists is the same, SPYRO can synthesize the following $\mathcal{L}$-property that further bounds the edit distance on the output:

$$x_1 = x_2 \wedge D_e(l_1, l_2) \leq d \Rightarrow D_e(\texttt{deleteFirst}(l_1, x_1), \texttt{deleteFirst}(l_2, x_2)) \leq d + 1$$

When inspecting this property, we were initially confused because we had thought that the edit distance should not increase at all if identical elements are removed. However, that is false as illustrated by the following tricky counterexample $l_1 = [1; 2; 3]$, $l_2 = [3; 2; 3]$ and $x_1 = x_2 = 3$. Besides the $\mathcal{L}$-property shown above with bound $d + 1$, SPYRO also synthesized (incomparable) best $\mathcal{L}$-properties with bounds $\texttt{len}(l_1) - \texttt{len}(l_2) + 2d$ and $\texttt{len}(l_2) - \texttt{len}(l_1) + 2d$ for the same query. All combined, these $\mathcal{L}$-properties imply that the edit distance should not increase when $d = 0$.

Because of the complexity added by the programs that compute the edit and Hamming distances, by the use of unbounded data structures, and by the fact that sensitivity properties are hyperproperties, we were not able to separately verify the soundness of they synthesized $\mathcal{L}$-conjunctions on all inputs using the Dafny verifier. However, we believe that the synthesized properties are indeed sound given that they hold for lists up to length 7—i.e., the bound imposed by SKETCH.

Finding: SPYRO can synthesize precise sensitivity properties for functions involving lists; the synthesized function would be challenging for a human to handcraft.

5.4 Application 4: Enabling New Abstract Domains

One of the most powerful relational abstract domains is the domain of *convex polyhedra* [Bagnara et al. 2008; Cousot and Halbwachs 1978]. While programs typically operate over `int`-valued program variables for which arithmetic is performed modulo a power of 2, such as 2^{16} or 2^{32}, existing implementations of polyhedra are based on conjunctions of linear inequalities with rational coefficients over rational-valued variables. This disconnect prevents polyhedra from precisely modeling how values wrap around in `int`/bit-vector arithmetic when arithmetic operations overflow.

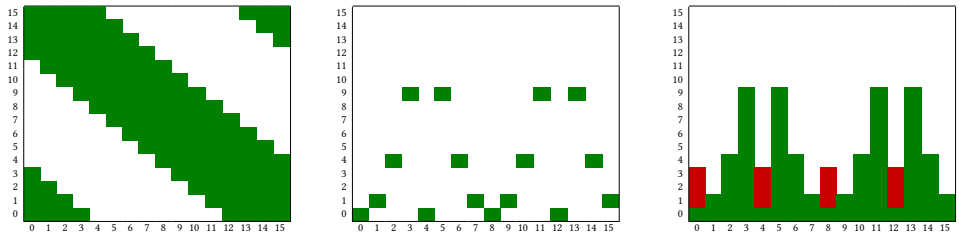
Heretofore, it has not been known how to create an analog of polyhedra that is appropriate for bit-vector arithmetic. Yao et al. [Yao et al. 2021] recently defined two domains (Version *1* and Version *2* below), but have only devised algorithms to support Version *2*.

Version *1* (bit-vector-polyhedra domain): conjunctions of linear bit-vector inequalities
Version *2* (integral-polyhedra domain): conjunctions of linear integer inequalities

The case study described in this section shows that SPYRO provides a way to enable precise polyhedra operations for Version *1*. The main reason why operations for bit-vector polyhedra have not been proposed previously is that it is challenging to work with relations over bit-vector-valued variables. For example, let x and y be 4-bit bit-vectors. Fig. 4(a) depicts the satisfying assignments of the inequality $x + y + 4 \leq 7$ interpreted over 4-bit unsigned modular arithmetic. As seen in the plot, the set of points that satisfy a single bit-vector inequality can be a non-contiguous region.

In general, a bit-vector inequality over unsigned bit-vector variables $X = \{x_1, \ldots, x_n\}$ has the form $\sum_{i=1}^{n} p_i x_i + q \leq \sum_{j=1}^{n} r_j x_j + s$, where $\{p_i\} \cup \{q\} \cup \{r_j\} \cup \{s\}$ are unsigned bit-vector constants, and $\leq$ is unsigned comparison. Conjunctions of inequalities are a fragment of quantifier-free bit-vector logic ($\mathcal{L}_{BV}$). Let $[\![\varphi]\!]_{BV}$ denote the set of assignments to X that satisfy formula $\varphi \in \mathcal{L}_{BV}$.

We instantiated SPYRO to take as input a formula $\varphi \in \mathcal{L}_{BV}$ and return a conjunction ψ of bit-vector inequalities—i.e., the *symbolic abstraction* [Reps and Thakur 2016, §5] of φ in the conjunctive fragment $\mathcal{L}_{BV(\wedge)}$. Because SPYRO computes best $\mathcal{L}$-properties, in this setting it computes the most-precise symbolic abstraction—i.e., the formula $\widehat{\alpha}$ computed by SPYRO is one representation of the *most-precise abstraction* of φ that is expressible as a conjunction of bit-vector inequalities.



$x + y + 4 \leq 7$

(a) $x + y + 4 \leq 7$

$$\begin{aligned} 10x + 11y + 0 &\leq 2x + 3y + 7 \\ 12x + 5y + 7 &\leq 2x + 8y + 15 \\ 0x + 5y + 3 &\leq 8x + 2y + 4 \\ 12x + 7y + 1 &\leq 12x + 3y + 8 \end{aligned}$$

(b) $y = x * x$

$$\begin{aligned} 2x + 15y + 3 &\leq 0x + 15y + 15 \\ 10x + 0y + 4 &\leq 0x + 15y + 15 \\ 8x + 15y + 4 &\leq 8x + 0y + 4 \end{aligned}$$

(c) $y \leq x * x$

Fig. 4. Each subfigure illustrates a bit-vector formula (in green) and the most precise bit-vector polyhedron computed by SPYRO (i.e., the inequalities above the formulas). Each colored cell in the plots represents a solution in 4-bit unsigned modular arithmetic of the conjunction of the inequalities found by SPYRO: green cells represent solutions to the original formula, whereas red cells are points that are solution to the inequalities, but do not satisfy the original formula. In (a) and (b), the conjunctive formula represents the original formula exactly (there are only green cells). In (b), the twelve occurrences of red cells are points that do not satisfy the original formula, but are needed for a conjunctive formula to over-approximate the original formula.

As known from the literature ([Reps et al. 2004; Thakur et al. 2012; Thakur and Reps 2012] and [Reps and Thakur 2016, §5]), operations needed for abstract interpretation, such as (i) the creation and/or application of abstract transformers, and (ii) taking the join of two abstract-domain elements, can be performed via an algorithm for symbolic abstraction. For instance, if ψ_a and ψ_b are two formulas in $\mathcal{L}_{BV(\wedge)}$, we can perform the join $\psi_a \sqcup \psi_b$ by $\widehat{\alpha}(\psi_a \vee \psi_b)$. (Note that $\psi_a \vee \psi_b$ is not a formula of $\mathcal{L}_{BV}$.) For this reason, we say that SPYRO *enables* this new abstract domain.

In our experiments, we limited inequalities to two variables x and y on each side, and used 4-bit unsigned arithmetic. Our benchmarks were taken from an earlier study conducted by one of the authors, which on each example used brute force to consider all 16,762,320 non-tautologies of the 16,777,216 4-bit inequalities of the form $ax + by + c \leq dx + ey + f$. That study found that some example formulas had hundreds of thousands of inequalities as consequences. We selected 9 interesting-looking formulas to use as benchmarks, including linear/nonlinear operations, equalities and inequalities, Boolean combinations, and one pair of formulas on which to perform the join operation. (See §5.4.2 and Appendix C.4.)

5.4.1 Quantitative Analysis. SPYRO[SKETCH] computed a sound best $\mathcal{L}$-conjunction for 9/9 formulas, and guaranteed that 8/9 were best $\mathcal{L}$-conjunctions. For the query $y = x/2$, SPYRO[SKETCH] timed out on a call to CHECKPRECISION, but the obtained $\mathcal{L}$-conjunction was indeed a best $\mathcal{L}$-conjunction because it defined the exact semantics of the query. SPYRO computed an $\mathcal{L}$-conjunction for all the 9 benchmarks in less than 400 seconds per benchmark, which is 2-5 orders of magnitude faster than the enumerative baseline presentd in Section 5.1.1. Each output $\mathcal{L}$-conjunction contained between 1 and 6 $\mathcal{L}$-properties. For this domain, SKETCH—and hence SPYRO[SKETCH]—is sound and precise because we are working with bit-vector arithmetic of fixed bit-width.

SPYRO[SKETCH] could not terminate for most of our benchmarks when considering 8-bit arithmetic. Because our examples contain several multiplications, this limitation is not surprising because multiplication is one of the known weaknesses of SKETCH and its underlying SAT solver. Recently, there have been promising advances in SAT solving for multiplication circuits [Kaufmann et al. 2022] that, if integrated with SKETCH, we believe would help SPYRO scale to larger bit-vectors.

5.4.2 Qualitative Analysis. Examples of results obtained by SPYRO are shown in Fig. 4. The result that the most-precise abstraction of these formulas could be expressed using only a small number of inequalities was surprising to the authors. In Fig. 4(c), the formula we are abstracting is $\varphi =_{df} y \leq x * x$). Fig. 4(c) shows that, in addition to the green points that satisfy the non-linear inequality $y \leq x * x$, the bit-vector-polyhedral abstraction found for φ includes twelve "extra" points, indicated by the red cells. Because SPYRO finds a *most-precise* sound bit-vector-polyhedral abstraction of φ, every sound bit-vector-polyhedral abstraction of φ must also include those twelve points. In an earlier study conducted by one of the authors, they used brute force to consider all 16,762,320 non-tautologies of the 16,777,216 4-bit inequalities of the form $ax + by + c \leq dx + ey + f$. That study found that the following numbers of inequalities over-approximated the original formula: 564 for Fig. 4(a), 109,008 for Fig. 4(b), and 456 for Fig. 4(c). Thus, SPYRO showed that a most-precise abstraction could be 2-4 orders of magnitude smaller than one obtained by brute force.

Finding: SPYRO can synthesize the most-precise sound bit-vector-polyhedral abstraction of a given bit-vector formula φ over 4-bit arithmetic. Furthermore, SPYRO surprised the authors by showing that the most-precise sound bit-vector-polyhedral abstraction for the presented examples could be precisely expressed with only a handful of bit-vector inequalities.

5.5 Further Analysis of SPYRO's Performance

In the previous sections, we have shown that SPYRO can synthesize best $\mathcal{L}$-conjunctions for a variety of case studies. In this section, we analyze what parameters affect SPYRO's running time.

Q1: How do Different Primitives of the Algorithm Contribute to the Running Time? On average, SPYRO spends 13.69 % of the time performing SYNTHESIZE, 26.78 % performing CHECKSOUNDNESS, and 42.33 % performing CHECKPRECISION (details in Tables 2 and 3 in App. C.5).

It usually takes longer for CHECKSOUNDNESS and CHECKPRECISION to show the nonexistence of an example—i.e., to return $\perp$—than to find an example. CHECKSOUNDNESS is one of the simplest queries, but occupies a large portion of the running time because it is expected to return $\perp$ many times, whereas CHECKPRECISION needs to return $\perp$ only once for each call to SYNTHESIZE-STRONGESTCONJUNCT. The last call to CHECKPRECISION (i.e., the one that returns $\perp$) often takes a significant amount of time to complete (on average 19.61 % of the time spent on each run of SYNTHESIZESTRONGESTCONJUNCT).

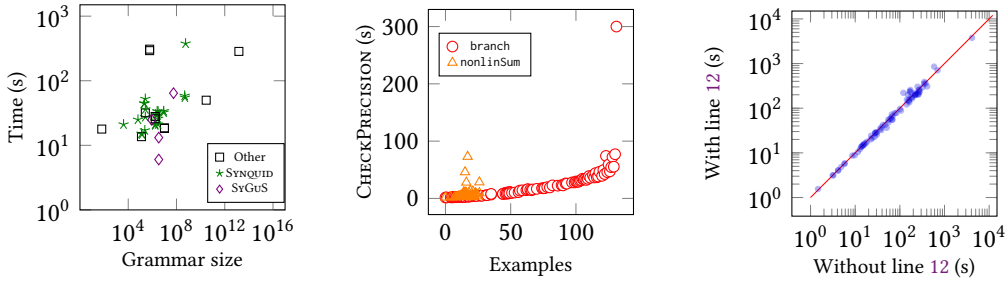
Finding: SPYRO spends most of the time checking soundness and precision.

Q2: What Parts of the Input Affect the Running Time? The number of $\mathcal{L}$-properties in the language $\mathcal{L}$ has a large impact on the time taken by SYNTHESIZE (Fig. 5a) and CHECKPRECISION.

The complexity of the code defining the semantics of various operators has a large impact on how long CHECKSOUNDNESS takes. `insert`, `delete` of BST and edit distance have relatively complicated implementations, and CHECKSOUNDNESS takes longer for these problems.

The size and complexity of the example space also affect the running time. The biggest factor contributing to the size of the example space is the number of input and output variables used. The number of possible examples, i.e., variable assignments, increases exponentially with the number of variables. The size of the example space affects not only the number of total queries but also the time that each query takes. Specifically, the number of positive and negative examples affects the time taken by SYNTHESIZE or CHECKPRECISION (notice that CHECKSOUNDNESS does not take the examples as input), as shown in Figure 5b.

Finding: The running time of SPYRO is affected by the sizes of (i) the property search space, (ii) the programs that describe the semantics of the operators, and (iii) the example search space.



(a) Time to synthesize a best $\mathcal{L}$-property vs. grammar size

(b) CheckPrecision time vs. number of examples

(c) Time with/without line 12.

Fig. 5. Evaluation of the running time of spyro for different input sizes and optimizations.

Q3: How effective is line 12 in Algorithm 1? We compared the running times of spyro with and without line 12 (Figure 5c). When line 12 is present, spyro is 3.06% faster (geometric mean) than when line 12 is absent, but both versions can solve the same problems. The optimization is only effective when the language has many incomparable properties that do not imply each other and cause Synthesize to often return $\perp$, thus triggering Line 10 in Algorithm 1—e.g., in all SyGuS benchmarks the language $\mathcal{L}$ is such that the optimization is not used.

Finding: Freezing negative examples is slightly effective.

6 RELATED WORK

Abstract-interpretation techniques. Many static program-analysis techniques are pitched as tools for checking safety properties, but behind the scenes they construct an artifact that abstracts the behavior of a program (in the sense of abstract interpretation [Cousot and Cousot 1977]). One such kind of artifact is a procedure summary [Cousot and Cousot 1978; Cousot and Halbwachs 1978; Gopan and Reps 2007; Sharir and Pnueli 1981], which abstracts a procedure's transition relation with an abstract value from an abstract domain, the elements of which denote transition relations.

Our problem is an instance of the *strongest-consequence problem* [Reps and Thakur 2016]. Existing techniques for solving this problem rely on properties of the language $\mathcal{L}$ that are typical of abstract interpretation. Some techniques work from "below," identifying a chain of successively weaker implicants, until one is a consequence of φ [Reps et al. 2004]. Other techniques work from "above," identifying a chain of successively stronger implicates, until no further strengthening is possible [Thakur et al. 2012; Thakur and Reps 2012]. Ozeri et al. [2017] explored a different approach, which works from above by repeatedly applying a semantic-reduction operation [Cousot and Cousot 1977]. (A semantic reduction operation finds a less-complicated description of a given set of states if one exists.) Our work differs from methods that use abstract interpretation in several aspects. First, our algorithm is the first to use both positive and negative examples to achieve precision. Second, while our work supports a variety of DSLs specified via a grammar, existing methods require that certain operations can be performed on concrete states and elements of the language $\mathcal{L}$ (e.g., joins [Thakur and Reps 2012]), thus limiting the language that can serve as the DSL.

Type inference. Liquid type inference [Hashimoto and Unno 2015; Rondon et al. 2008; Vazou et al. 2014] can infer a weakest precondition from a given postcondition or a strongest postcondition from a given precondition. To make the problem tractable, properties must be specified in a user-given restricted set of predicates that are closed under Boolean operations. Although our work shares some similarities—e.g., looking for properties over a restricted DSL—we tackled a fundamentally

different problem because no pre- or post-condition is given as an input, and our algorithm instead looks for best $\mathcal{L}$-properties. Furthermore, our work is not restricted to functional languages.

Invariant inference. Several data-driven, CEGIS-style algorithms can synthesize program invariants. These techniques look for any invariant that is satisfactory for a client verification problem, whereas we do not assume there is a client for whom the properties are synthesized. Without a client, "true" is a sound (and also weakest) but useless specification. The absence of a client requires synthesized specifications to be precise, therefore requiring our new CheckPrecision primitive.

A closely related system is Elrond [Zhou et al. 2021], which synthesizes weakest library specifications that make verification possible in a client program. Elrond allows one to specify a set of target predicates of interest, and finds quantified Boolean formulas with equalities over the variables and the predicates. Because soundness (with respect to the library function) is only checked on a set of inputs, Elrond tries to synthesize weakest specifications using an iterative weakening approach. Their algorithm takes advantage of the structure of supported formulas (they can contain disjunctions), but has some limitations (they can only contain equalities).

The key differences between Elrond and our work are: (*i*) Our work supports a user-supplied DSL, which enables more generality, but prevents the use of techniques that rely on access to arbitrary Boolean operations. (*ii*) Our work uses a parametric DSL that can contain complex user-given functions, whereas Elrond only allows parametric atoms (i.e., user-defined Boolean function), equality over variables, and Boolean combinations of them. Our tool spyro can synthesize the $\mathcal{L}$-property "$2o = 0 \vee -x + 2o - x^2 = 0$," whereas Elrond does not consider arithmetic predicates. (*iii*) The specifications generated by spyro are most precise with respect to the DSL $\mathcal{L}$, allowing for their reuse in multiple problem instances that use the same function. Such reuse is possible because spyro ensures soundness of the synthesized properties. In contrast, Elrond operates in a closed-box setting and uses a random sampler for soundness, intentionally weakening the synthesized specifications to enhance the likelihood of soundness. (*iv*) Our work can synthesize arithmetic properties efficiently (e.g., the ones considered in Sections 5.3 and 5.4) as well as complex algebraic properties (e.g., the ones in Section 5.2). In theory, Elrond can describe all the necessary components to express algebraic properties such as the property $(s, x) = \mathsf{pop}(\mathsf{push}(s, x))$, but one would need to provide a function that combines pop and push. This approach is feasible if one is interested in only a few possible ways of combining functions, but becomes infeasible once more combinations are possible (e.g., for an arithmetic formula). In short, the two approaches have different goals.

The presence of a user-supplied DSL and absence of a client distinguish our work from other prior work, e.g., abductive inference [Dillig et al. 2012], ICE-learning [Garg et al. 2014], LoopInvGen [Padhi et al. 2016], Hanoi [Miltner et al. 2020], and Data-Driven CHC Solving [Zhu et al. 2018].

Dynamic techniques. Daikon [Ernst et al. 2001, 2007] is a system for identifying likely invariants by inspecting program traces. Invariants generally involve at most two program quantities and are checked at procedure entry and exit points (i.e., invariants form precondition/postcondition pairs). In Daikon, the default is to check 75 different forms of invariants, instantiated for the program variables of interest. The language of invariants can be extended by the user.

spyro differs from Daikon (and follow-up work, e.g., [Beckman et al. 2010]) in two ways: (*i*) The language $\mathcal{L}$ is not limited to a set of predicates, and spyro scales to languages containing millions of properties; (*ii*) the properties that spyro synthesizes are sound and provably *best* $\mathcal{L}$-properties. Furthermore, while Daikon's dynamic approach can scale to large programs, we could not find a way to encode our case studies as instances that Daikon could receive as input.

A similar tool to Daikon is QuickSpec [Smallbone et al. 2017], which generates equational properties of Haskell programs from random tests. spyro differs from QuickSpec in two ways: (*i*)

The language $\mathcal{L}$ is not limited to equational properties; (*ii*) the properties that SPYRO synthesizes are sound and provably *best* $\mathcal{L}$-properties.

Astorga et al. [2021] synthesize contracts that are sound with respect to positive examples generated by a test generator. We see two main differences between that work and ours: (*i*) They do not use negative examples, whereas we do. Negative examples are the key to synthesizing best $\mathcal{L}$-properties. (*ii*) Their work does not allow a parametrized DSL and their notion of "tight" is with respect to a syntactic restriction on the logic in which the contract is to be specified.

Synthesis of best $\mathcal{L}$-transformers. The paper that inspired our work synthesizes most-precise abstract transformers in a user-given DSL [Kalita et al. 2022]. We realized that their basic insight—use both positive and negative examples; treat positive examples as hard constraints and negative examples as "maybe" constraints—had broader applicability than just creating abstract transformers.

Our work differs in two key ways. First, because our goal is to obtain a formula rather than a piece of code (their setting), we could take advantage of the structure of formulas—in particular, conjunctions—to decompose the problem into (*i*) an "inner search" to find a best $\mathcal{L}$-property that is an individual conjunct (Alg. 1), and (*ii*) an "outer search" to accumulate best $\mathcal{L}$-properties to form a best $\mathcal{L}$-conjunction (Alg. 2). Second, Alg. 1 exploits monotonicity—i.e., once a sound $\mathcal{L}$-property is found, there must exist a best $\mathcal{L}$-property that implies it (Lemma 3.1). This observation allows us to use a simplified set of primitives: our algorithm uses a SYNTHESIZE primitive, whereas theirs requires a MAXSYNTHESIZE primitive—i.e., one that synthesizes a program that accept all the positive examples and rejects as many negative examples as possible. Our ideas could be back-ported to provide improvements in their setting as well: if the abstract domain supports meet ($\sqcap$), they could run their algorithm multiple times to create a kind of "conjunctive" transformer, which would run multiple, incomparable best $\mathcal{L}$-transformers, and then take the meet of the results.

7 CONCLUSION

This paper presents a formal framework for the problem of synthesizing a best $\mathcal{L}$-conjunction—i.e., a conjunctive specification of a program with respect to a user-defined logic $\mathcal{L}$—and an algorithm for automatically synthesizing a best $\mathcal{L}$-conjunction. The innovations in the algorithm are three-fold: (i) it identifies individual conjuncts that are themselves strongest consequences; (ii) it balances negative examples that *must* be rejected by the $\mathcal{L}$-property being synthesized and ones that *may* be rejected; and (iii) it guarantees progress via monotonic constraint hardening.

Our work opens up many avenues for further study. One is to harness other kinds of synthesis engines to implement SYNTHESIZE, CHECKSOUNDNESS, and CHECKPRECISION. Recent work on Semantics-Guided Synthesis (SEMGUS) [Kim et al. 2021] provides an expressive synthesis framework for expressing complex synthesis problems like the ones discussed in this paper. SEMGUS solvers, such as MESSY [Kim 2022], are able to produce two-sided answers to a problem: either synthesizing a solution, or proving that the problem is unrealizable—i.e., has no solution—exactly what is needed for CHECKPRECISION. On the theoretical side, while we have used first-order logic, it would be interesting to try other logics, such as separation logic [Reynolds 2002] or effectively propositional logic [Itzhaky 2014; Padon 2018]. On the practical side, our work could find applications in invariant generation [Padon et al. 2022] and code deobfuscation [Blazytko et al. 2017].

Acknowledgement

Supported, in part, by a Microsoft Faculty Fellowship; a gift from Rajiv and Ritu Batra; NSF under grants CCF-{1750965,1763871,1918211,2023222,2211968,2212558}; and ONR under grant N00014-17-1-2889. 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 that containing the source code of SPYRO, Dafny proofs, and all necessary dependencies for the experiments [Park et al. 2023b].

REFERENCES

Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. 2019. SyGuS-Comp 2018: Results and Analysis. https://doi.org/10.48550/ARXIV.1904.07146

Angello Astorga, Shambwaditya Saha, Ahmad Dinkins, Felicia Wang, P. Madhusudan, and Tao Xie. 2021. Synthesizing contracts correct modulo a test generator. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485481

Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.* 72, 1-2 (2008), 3–21. https://doi.org/10.1016/j.scico.2007.08.001

Haniel Barbosa, Clark W. 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 (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, Robert J. Simmons, SaiDeep Tetali, and Aditya V. Thakur. 2010. Proofs from Tests. *IEEE Trans. Software Eng.* 36, 4 (2010), 495–508. https://doi.org/10.1109/TSE.2010.49

Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 643–659. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

Patrick Cousot and Radhia Cousot. 1978. Static Determination of Dynamic Properties of Recursive Procedures. In *Formal Descriptions of Programming Concepts*. North-Holland.

Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 84–96. https://doi.org/10.1145/512760.512770

Loris D'Antoni, Marco Gaboardi, Emilio Jesús Gallego Arias, Andreas Haeberlen, and Benjamin C. Pierce. 2013. Sensitivity analysis using type-based constraints. In *Proceedings of the 1st annual workshop on Functional programming concepts in domain-specific languages, FPCDSL@ICFP 2013, Boston, Massachusetts, USA, September 22, 2013*, Richard Lazarus, Assaf J. Kfoury, and Jacob Beal (Eds.). ACM, 43–50. https://doi.org/10.1145/2505351.2505353

Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated error diagnosis using abductive inference. *ACM SIGPLAN Notices* 47, 6 (2012), 181–192.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Trans. Software Eng.* 27, 2 (2001), 99–123. https://doi.org/10.1109/32.908957

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 69–87. https://doi.org/10.1007/978-3-319-08867-9_5

Denis Gopan and Thomas W. Reps. 2007. Low-Level Library Analysis and Summarization. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4590)*, Werner Damm and Holger Hermanns (Eds.). Springer, 68–81. https://doi.org/10.1007/978-3-540-73368-3_10

Kodai Hashimoto and Hiroshi Unno. 2015. Refinement Type Inference via Horn Constraint Optimization. In *SAS*.

Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas W. Reps. 2020. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In *Proceedings of the 41st ACM SIGPLAN International Conference on*

Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1128–1142. https://doi.org/10.1145/3385412.3385979

Shachar Itzhaky. 2014. *Automatic Reasoning for Pointer Programs Using Decidable Logics.* Ph. D. Dissertation. Tel Aviv University.

Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas W. Reps, and Subhajit Roy. 2022. Synthesizing Abstract Transformers. *Proc. ACM Program. Lang.* OOPSLA (2022). https://doi.org/10.1145/3563334

Daniela Kaufmann, Paul Beame, Armin Biere, and Jakob Nordstrom. 2022. Adding Dual Variables to Algebraic Reasoning for Gate-Level Multiplier Verification. In *Proceedings of the 2022 Conference on Design, Automation Test in Europe* (Antwerp, Belgium) *(DATE '22)*. European Design and Automation Association, Leuven, BEL, 1431–1436.

Jinwoo Kim. 2022. Messy-Release. https://github.com/kjw227/Messy-Release.

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas W. Reps. 2021. Semantics-guided synthesis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434311

K. Rustan M. Leino and Valentin Wüstholz. 2014. The Dafny Integrated Development Environment. In *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014 (EPTCS, Vol. 149)*, Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry (Eds.). 3–15. https://doi.org/10.4204/EPTCS.149.2

David Lo, Siau-Cheng Khoo, Jiawei Han, and Chao Liu (Eds.). 2017. *Mining Software Specifications: Methodologies and Applications.* Chapman & Hall.

Benjamin Mariano, Josh Reese, Siyuan Xu, ThanhVu Nguyen, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2019. Program Synthesis with Algebraic Library Specifications. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 132 (Oct. 2019), 25 pages. https://doi.org/10.1145/3360558

Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. 2020. Data-Driven Inference of Representation Invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3385412.3385967

Tom M. Mitchell. 1997. *Machine Learning.* McGraw-Hill.

Or Ozeri, Oded Padon, Noam Rinetzky, and Mooly Sagiv. 2017. Conjunctive Abstract Interpretation Using Paramodulation. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10145)*, Ahmed Bouajjani and David Monniaux (Eds.). Springer, 442–461. https://doi.org/10.1007/978-3-319-52234-0_24

Saswat Padhi, Rahul Sharma, and Todd D. Millstein. 2016. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 42–56. https://doi.org/10.1145/2908080.2908099

Oded Padon. 2018. *Deductive Verification of Distributed Protocols in First-Order Logic.* Ph. D. Dissertation. Tel Aviv University.

Oded Padon, James R. Wilcox, Jason R. Koenig, Kenneth L. McMillan, and Alex Aiken. 2022. Induction duality: primal-dual search for invariants. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498712

Kanghee Park, Loris D'Antoni, and Thomas Reps. 2023a. Synthesizing Specifications. arXiv:2301.11117 [cs.PL]

Kanghee Park, Loris D'Antoni, and Thomas Reps. 2023b. Synthesizing Specifications. https://doi.org/10.5281/zenodo.8327699

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.

Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*. 252–266. https://doi.org/10.1007/978-3-540-24622-0_21

Thomas W. Reps and Aditya V. Thakur. 2016. Automating Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 3–40. https://doi.org/10.1007/978-3-662-49122-5_1

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. https://doi.org/10.1145/1375581.1375602

Micha Sharir and Amir Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall.

Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Algehed. 2017. Quick specifications for the busy programmer. *Journal of Functional Programming* 27 (2017), e18.

Armando Solar-Lezama. 2013. Program sketching. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

Aditya V. Thakur, Matt Elder, and Thomas W. Reps. 2012. Bilateral Algorithms for Symbolic Abstraction. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*. 111–128. https://doi.org/10.1007/978-3-642-33125-1_10

Aditya V. Thakur and Thomas W. Reps. 2012. A Method for Symbolic Computation of Abstract Operations. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 174–192. https://doi.org/10.1007/978-3-642-31424-7_17

Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 269–282.

Yu-Xiang Wang, Jing Lei, and Stephen E. Fienberg. 2016. Learning with Differential Privacy: Stability, Learnability and the Sufficiency and Necessity of ERM Principle. *J. Mach. Learn. Res.* 17, 1 (jan 2016), 6353–6392.

Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2021. Program analysis via efficient symbolic abstraction. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–32. https://doi.org/10.1145/3485495

Zhe Zhou, Robert Dickerson, Benjamin Delaware, and Suresh Jagannathan. 2021. Data-driven abductive inference of library specifications. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485493

He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 707–721. https://doi.org/10.1145/3192366.3192416

Received 2023-04-14; accepted 2023-08-27