Explainable Program Synthesis by Localizing Specifications

AMIRMOHAMMAND NAZARI, University of Southern California, USA
YIFEI HUANG, University of Southern California, USA
ROOPSHA SAMANTA, Purdue University, USA
ARJUN RADHAKRISHNA, Microsoft, USA
MUKUND RAGHOTHAMAN, University of Southern California, USA

The traditional formulation of the program synthesis problem is to find a program that meets a logical correctness specification. When synthesis is successful, there is a guarantee that the implementation satisfies the specification. Unfortunately, synthesis engines are typically monolithic algorithms, and obscure the correspondence between the specification, implementation and user intent. In contrast, humans often include comments in their code to guide future developers towards the purpose and design of different parts of the codebase. In this paper, we introduce *subspecifications* as a mechanism to augment the synthesized implementation with explanatory notes of this form. In this model, the user may ask for explanations of different parts of the implementation; the subspecification generated in response is a logical formula that describes the constraints induced on that subexpression by the global specification and surrounding implementation. We develop our ideas in the context of the syntax-guided synthesis (SyGuS) framework. We develop algorithms to construct and verify subspecifications and investigate their theoretical properties. We perform an experimental evaluation of the subspecification generation procedure, including its effectiveness and running time. Finally, we conduct a user study to determine whether subspecifications are useful: we find that subspecifications greatly aid in understanding the global specification, in identifying alternative implementations, and in debugging faulty implementations.

ACM Reference Format:
Amirmohammand Nazari, Yifei Huang, Roopsha Samanta, Arjun Radhakrishna, and Mukund Raghothaman. 2022. Explainable Program Synthesis by Localizing Specifications. 1, 1 (October 2022), ?? pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Program synthesis technology has made tremendous advances over the past two decades [??????]. It has been applied to diverse domains, including end-user programming [???], data science [?], networking [?], robotics [?] and programmer assistance tools [??].

Despite these developments, one aspect of program synthesis that is being examined only recently concerns questions of trust and interpretability. In particular, most synthesis engines do not explicitly report connections between the specification and the synthesized code. In addition, writing correct specifications is a non-trivial task, and even relatively simple specification mechanisms such as programming-by-example (PBE) are subject to omission, user error and noise [?]. Finally,

Authors' addresses: Amirmohammand Nazari, nazaria@usc.edu, University of Southern California, USA; Yifei Huang, yifeih@usc.edu, University of Southern California, USA; Roopsha Samanta, roopsha@cs.purdue.edu, Purdue University, USA; Arjun Radhakrishna, arradha@microsoft.com, Microsoft, USA; Mukund Raghothaman, raghotha@usc.edu, University of Southern California, USA.

XXXX-XXXX/2022/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

maintaining the synthesized implementation in the face of changing requirements remains an outstanding problem. Some techniques to address these challenges involve guidance from advanced interaction models, specification refinement, and the use of explanatory artifacts [?????]. However, these techniques focus either on disambiguating the specification, such as by augmenting input-output examples with user annotations, or using their guidance to prune the search space and thereby accelerate synthesis. Notably, none of these approaches provide explanations of how the synthesized program satisfies the specification.

In contrast, when human programmers write code, they include comments and other forms of documentation that indicate its design, purpose, and connection to the rest of the codebase. This helps future developers to reason about the software system in question, and enables ongoing maintenance, bug fixes, feature additions, optimization, and porting. Research on modular verification also uses similar approaches, leveraging function summaries and other fine-grained properties instead of directly establishing properties about the program as a whole.

Inspired by these ideas, we introduce the concept of *subspecifications* as a general mechanism to identify the constraints imposed by the global specification on individual parts of the implementation. Consider for example the task of synthesizing a function $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ such that $f(1,2) = 3 \wedge f(2,3) = 5 \wedge f(1,0) = 2$. Say the synthesizer produces the solution:

$$f_1(x,y) = \texttt{if}\ x \geq y\ \texttt{then}\ \underbrace{x+1}_{h_1}\ \texttt{else}\ \underbrace{x+y}_{h_2}. \tag{1}$$

The user might now ask questions about different parts of the program. For example, to understand the subexpression marked h_1 in the then-branch, they might ask what other expressions could be used instead. One can readily observe that alternative implementations exist, such as:

$$f_2(x,y) = \texttt{if}\ x \geq y\ \texttt{then}\ y+2\ \texttt{else}\ x+y, \text{ and}$$
$$f_3(x,y) = \texttt{if}\ x \geq y\ \texttt{then}\ x+y+1\ \texttt{else}\ x+y,$$

among others. Further reflection reveals that any function $f^* : \mathbb{Z} \to \mathbb{Z}$ of the form:

$$f^*(x,y) = \texttt{if}\ x \geq y\ \texttt{then}\ g_1(x,y)\ \texttt{else}\ x+y$$

also satisfies the specification iff the new subexpression $g_1 : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ satisfies the condition $g_1(1,0) = 2$. The condition $g_1(1,0) = 2$ therefore summarizes the constraints on the subexpression labelled h_1 by the global specification and the surrounding implementation. Subspecifications formalize this process of reverse-engineering requirements on different parts of the implementation from the original specification-implementation pair.

Subspecifications (or simply subspecs for short) can be used to connect parts of the implementation back to the original specification, in a manner similar to requirements traceability in software engineering [?]. The user can also use them to refine the specification and gain insight into how the implementation works. For example, they might observe that the then-branch of Equation ?? is under-constrained and this might lead them to provide additional input-output examples to prune the space of feasible implementations. In addition, subspecs could be used to determine connections between different parts of the implementation: for example, one might observe that the subexpressions marked h_1 and h_2—with subspec $g_2(1,2) = 3 \wedge g_2(2,3) = 5$—have independent subspecifications and can therefore be separately manipulated without interfering with global correctness.

Having introduced the concept of subspecifications, we next turn our attention to investigating their theoretical properties. The main algorithmic question is to efficiently obtain a simple subspec from a specification, implementation and subexpression of interest. We begin by describing a construction that allows us to readily obtain a subspec by substituting a probe function into the

location of interest. Unfortunately, subspecifications obtained in this manner can be quite large, and provide limited additional insight to users. Our algorithm then proceeds by deskolemizing this trivial subspecification and simplifying the resulting indicator function.

This two-stage procedure is remarkably effective in experiments: We implemented the algorithm using the state-of-the-art SyGuS solver CVC5 [**?**], and performed an experimental evaluation using benchmarks from the 2017 SyGuS Competition [**?**]. We focus our evaluation on determining whether our implementation, which we call S^3, is effective in producing small subspecifications in reasonable amounts of time. Overall, we discover a 74% reduction in the subspec size for over 74% of the benchmarks. In addition, 77% of the subspecification synthesis tasks can be completed in less than one second, and 83% of the subspecifications can be synthesized in less time than was originally needed to obtain the implementation, potentially enabling its application to interactive program reasoning.

A second algorithmic problem involves determining the correctness of a proposed subspecification. While this problem can be hard in general, indicator functions enable a complete method to determine correctness for the case of point-wise specifications, in which all calls to the target function in the spec are syntactically identical. Next, we investigate the simplifying power of subspecifications in different situations: we prove that point-wise specs always lead to point-wise subspecs, and identify conditions under which PBE synthesis tasks lead to PBE subspecs. Finally, we attempt to formalize the intuition that individually understanding the parts of a program can lead to an understanding of its whole. While this reconstruction theorem admittedly requires some technical assumptions, it naturally leads us to identify connections between different parts of the program and—analogous to the idea of joint probability distributions—motivates the generalized concept of joint subspecifications.

To evaluate the usefulness of subspecifications in assisting users of program synthesis tools, we conducted a small user study with 20 graduate student participants. Across a sequence of four tasks requiring users to assess and manipulate the output of a SyGuS solver, we determined that having access to subspecifications improves the accuracy of responses by 34% and causes a 24% reduction in the time needed to arrive at conclusions. In post-study discussions, participants reported that subspecifications helped in visualizing the output of the synthesized implementation, and indicated a strong preference for having access to subspecifications while answering questions about synthesized implementations.

Contributions. In summary, we make the following contributions in this paper:

(1) We propose the concept of subspecifications as a general framework to facilitate user understanding in program synthesis systems.
(2) We develop algorithms to construct and verify subspecifications and investigate their theoretical properties.
(3) We conduct a user study to determine the value of subspecifications to users of program synthesizers and conclude that subspecifications can help achieve a better understanding of both specifications and implementations.
(4) We present experiments showing that our algorithm is able to efficiently generate small subspecifications for a range of synthesis tasks.

2 FORMALLY DEFINING SUBSPECIFICATIONS

Consider the following SyGuS specification φ which describes an integer-valued function $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$:[1]

$$f(x, y) = f(y, x) \wedge f(x, y) \in \{x - y, y - x\}.$$

The goal is to find a function f which satisfies this specification for all values of the free variables, x and y. Given this specification, a synthesizer such as EUSolver [**?**] may produce the following implementation:

$$f_1(x, y) = \texttt{if}\ \underbrace{x \geq y}_{h_1}\ \texttt{then}\ \underbrace{x - y}_{h_2}\ \texttt{else}\ \underbrace{y - x}_{h_3}. \tag{2}$$

The function f_1 computes the absolute value of the difference between its inputs, i.e., $|x - y|$. An SMT solver can easily verify that the synthesized function f_1 satisfies the specification φ. However, a user may find it hard to trust the synthesis process and use the resulting implementation. To be confident in the implementation, they must first be sure that the specification φ they have written captures their intent correctly. Second, not only must they be convinced that the implementation f_1 satisfies φ, but they must also understand the obscured connection between the implementation and specification in order to maintain and modify it in response to changing requirements in the future.

Let us attempt to understand the reasoning behind choosing $x \geq y$ as the subexpression h_1 in the implementation. We can see that any function f^* of the form:

$$f^*(x, y) = \texttt{if}\ g_1(x, y)\ \texttt{then}\ x - y\ \texttt{else}\ y - x$$

satisfies the specification φ iff g_1 satisfies the property:

$$x \neq y \implies g_1(x, y) \neq g_1(y, x). \tag{3}$$

Therefore, the following alternative implementation also satisfies φ:

$$f_2(x, y) = \texttt{if}\ x < y\ \texttt{then}\ x - y\ \texttt{else}\ y - x. \tag{4}$$

This new implementation f_2 computes $-|x - y|$. This observation surprised one of the authors of this paper, who initially believed that the intent of the specification was to determine the absolute value of the difference, $|x - y|$. This mismatch between the user's intent and the specification becomes more likely as specifications become increasingly complex.

Our thesis is that requirements on subexpressions (such as Equation **??** on $g_1(x, y)$) can help users achieve a better understanding of the specification and the implementation, and can provide them with a mechanism to interrogate the synthesizer. We call these requirements on subexpressions *subspecifications*. We will formally define subspecifications in the rest of this section, and consider several examples in Section **??**.

Synthesis problems. We begin by recalling that a synthesis problem $P = (f, \varphi(f, \boldsymbol{x}))$ consists of: (a) an uninterpreted function f with appropriate type signature, and (b) a quantifier-free formula $\varphi(f, \boldsymbol{x})$ with free variables $\boldsymbol{x}$. The goal of the SyGuS solver is to find a function expression f such that $\varphi(f, \boldsymbol{x})$ holds for all values of the free variables $\boldsymbol{x}$.

Remark 2.1. In this paper, we focus on the syntax-guided synthesis (SyGuS) program synthesis framework where a synthesis problem requires a third parameter, namely a grammar G, which

[1] Problem named `diff.sl` in the CLIA track of the 2018 SyGuS competition.

constrains the syntax of candidate implementations. However, because subspecifications are defined and computed as post-hoc explanations for an already synthesized implementation, we will ignore the grammar specification in the following discussion.

We say that a specification is *point-wise* if all calls to the function f to be synthesized have the same arguments [?]. For example, the specification $f(x,y) = f(y,x) \wedge f(x,y) \in \{x-y, y-x\}$ is not point-wise as f is called both with the arguments (x,y) and (y,x). On the other hand, $f(x,y) \geq x \wedge f(x,y) \geq y$ is a point-wise specification.

Program locations and holes. Given a function expression f, a *program location* h in f is a node in its abstract syntax tree (AST). The *subexpression at* h is the expression corresponding to the subtree rooted at h, which we denote by $f{\downarrow}h$. Given an alternative expression g, we write $f\,[g/h]$ to denote the expression obtained by replacing the subexpression at h with g. We say two holes h_1 and h_2 are *independent* if neither is an ancestor of the other. Given multiple pairwise-independent, holes h_1, ..., h_n and expressions g_1, ..., g_n, we define $f\,[g_1/h_1, \ldots, g_n/h_n]$ to be the expression obtained by simultaneously replacing each expression at h_i with g_i.

Example 2.2. Consider the implementation $f_1(x,y) = \texttt{if}\ \underbrace{x \geq y}_{h_1}\ \texttt{then}\ \underbrace{x-y}_{h_2}\ \texttt{else}\ y-x$ from Equation **??** with the holes h_1 and h_2 given by the highlighted locations. We have that $f_1{\downarrow}h_1 = x \geq y$ and $f_1 \downarrow h_2 = x - y$. If $g_1(x,y) = x < y$ then we have $f_1\,[g_1/h_1] = \texttt{if}\ x < y\ \texttt{then}\ x-y\ \texttt{else}\ y-x$.

Subspecifications. Let $P = (f, \varphi(f, \boldsymbol{x}))$ be a synthesis problem instance, and let f_0 be an implementation which satisfies φ. For a given hole h of f, we say that a formula $\psi(g, \boldsymbol{x})$ is a *subspecification for* h *in* f_0 if every alternative subexpression g_0 satisfies ψ iff the modified implementation $f_0\,[g_0/h]$ satisfies the global specification φ. Formally, we want that for all g_0, $g_0 \models \psi \iff f_0\,[g_0/h] \models \varphi$. Similarly, we can define joint subspecifications $\psi(g_1, \ldots, g_n, \boldsymbol{x})$ for multiple pairwise-independent holes h_1, ..., h_n by requiring for all g_1, ..., g_n, $(g_1, \ldots, g_n) \models \psi \iff f_0\,[\forall i, g_i/h_i] \models \varphi$.

Note that subspecifications are not necessarily unique and multiple formulas may satisfy all the required conditions. We use the notation $\varphi|_h^f$ to denote some arbitrary subspecification for h in f, and $\varphi|_{h_1,\ldots,h_n}^f$ to denote some arbitrary subspecification for multiple holes h_1, ..., h_n in f.

Example 2.3. Continuing from Example **??**, the expression $x \neq y \implies g(x,y) \neq g(y,x)$ from Equation **??** is a valid subspecification of h_1 in f_1. Intuitively, we want that (x,y) and (y,x) to take different branches of the if construct whenever x and y are different.

3 MOTIVATING EXAMPLES

We will now present examples to show how subspecifications can be used to aid in understanding specifications and implementations and for debugging program synthesizers. We hope to show the breadth of potential applications and the value of algorithms that can automatically generate subspecs.

Example 3.1. As a first example, consider the following specification φ_1 from the 2018 SyGuS competition:[2]

$$\begin{array}{lll} & 0 \leq f(x,y) \leq 2 & \\ \wedge & f(x,y) = 0 & \implies x = y \\ \wedge & f(x,y) = 0 \wedge 1 \leq i,j \leq 2 & \implies |x-y| \leq (x-y)(j-i) \vee f(x+i, y+j) = 0 \\ \wedge & f(x,y) \neq 0 & \implies f(x+f(x,y), y+f(x,y)) \neq 0 \wedge |x-y| > 0. \end{array} \tag{5}$$

[2]Problem named `jmbl_fg_VC22_a.sl` in the CLIA track.

Observe that the specification is hard to understand because it spans several lines and has complex Boolean structure. On the other hand, EUSolver produces the following remarkably simple implementation for φ:

$$f(x,y) = \texttt{if}\ x = y\ \texttt{then}\ \underbrace{0}_{h_1}\ \texttt{else}\ \underbrace{1}_{h_2}. \tag{6}$$

Let us manually construct a subspecification for h_1, h_2 in f. We examine all implementations of the form $f^*(x,y) = \texttt{if}\ x = y\ \texttt{then}\ g_1(x,y)\ \texttt{else}\ g_2(x,y)$. From the second and fourth clauses of the global specification, it follows that $f(x,y) = 0$ iff $x = y$. From this, we can see that any function which satisfies the second and fourth clauses will automatically satisfy the third clause as setting $x = y$ in the consequent of the third clause will make it trivially true. The first clause of the specification φ_1 may be equivalently written as $f(x,y) \in \{0,1,2\}$. From our observations, it follows that $f^*(x,y)$ satisfies φ_1 iff $g_1(x,y)$ and $g_2(x,y)$ together satisfy $\varphi_1|^f_{h_1,h_2}$, where:

$$\begin{aligned}\varphi_1|^f_{h_1,h_2} \equiv \quad & x = y \implies g_1(x,y) = 0 \\ \wedge \quad & x \neq y \implies g_2(x,y) \in \{1,2\}.\end{aligned}$$

In other words, this formula is the subspecification of h_1 and h_2 under φ_1.

Observe that the $\varphi_1|^f_{h_1,h_2}$ is significantly smaller than the original specification φ_1. In addition, note that $\varphi_1|^f_{h_1,h_2}$ is a point-wise specification, even though the global specification φ_1 included multiple syntactically unequal calls to f, including $f(x,y)$, $f(x+i, y+j)$, and $f(x+f(x,y), y+f(x,y))$ and was therefore not a point-wise specification. We claim that it is easier to understand $\varphi_1|^f_{h_1,h_2}$ than to understand φ_1 and, by suggesting alternative implementations, provides additional insight into the constraints imposed by φ_1. These empirically verify these claims in Task 1 of the user study in Section **??**, where we observe that users are faster and more accurate in answering questions about this specification-implementation pair when they have access to subspecifications.

Example 3.2 (Traceability). Consider the following PBE specification φ_2, adapted from the problem named `LinExpr_inv1_ex.sl` from the 2017 SyGuS competition.

$$\begin{aligned}&f(11,4) = 1 \wedge f(25,3) = 1 \wedge f(7,21) = 1 \wedge f(2,38) = 1 \wedge \\ &f(26,1) = 3 \wedge f(75,1) = 3 \wedge f(1,38) = 3 \wedge f(1,48) = 3.\end{aligned}$$

Given this specification, EUSolver responds with the following implementation:

$$f(x,y) = \texttt{if}\ x \leq 1\ \texttt{then}\ \underbrace{3x}_{h_1}\ \texttt{else if}\ y \leq 1\ \texttt{then}\ \underbrace{3y}_{h_2}\ \texttt{else}\ \underbrace{1}_{h_3}.$$

Given this implementation, the subspecifications for the program locations marked h_1, h_2, and h_3 are:

$$\begin{aligned}\varphi|^f_{h_1} &\equiv g_1(1,38) = 3 \wedge g_1(1,48) = 3, \\ \varphi|^f_{h_2} &\equiv g_2(26,1) = 3 \wedge g_2(75,1) = 3, \text{ and} \\ \varphi|^f_{h_3} &\equiv g_3(11,4) = 1 \wedge g_3(25,3) = 1 \wedge g_3(7,21) = 1 \wedge g_3(2,38) = 1,\end{aligned}$$

respectively. Computing the subspecifications immediately reveals which examples in the PBE problem are handled by which branches of the `if-then-else` expression. This clearly shows the distribution of the training data between different parts of the implementation, and clarifies the choice of each of the individual subexpressions $f \downarrow h_1$, $f \downarrow h_2$, and $f \downarrow h_3$. For example, focusing only on $f \downarrow h_1$ using $\varphi|^f_{h_1}$ it is easy to see why $3x$ is a valid expression for h_1.

Examining these subspecifications might also indicate to the user which branches and parts of the code have insufficient numbers of examples, and suggest new examples to strengthen the specification. They might also observe that although there are only two groups of examples, of the form $f(\_,\_) = 1$ and $f(\_,\_) = 3$ respectively, there are three branches, and that both branches h_1 and h_2 are responsible for fulfilling examples of the form $f(\_,\_) = 3$. Users can use these observations to confirm that the division of data between different branches is consistent with their intent and knowledge of the problem domain. This reasoning naturally mirrors the way users often conceptualize their code, distinguishing its behavior on general cases from its behavior on exceptional corner cases.

We used a more elaborate version of this specification-implementation pair for in Task 2 of our user study. We broadly observed that participants with subspecifications are able to more readily understand the relevance of individual examples, and that subspecifications aid in better understanding possible user intent.

Example 3.3 (Debugging synthesizers). Finally, consider the following synthesis task φ_3 from PBE SLIA track of the 2017 SyGuS competition:[3]

$$\begin{aligned} &f(\texttt{"Ducati100"}) = \texttt{"Ducati"} \wedge f(\texttt{"Honda125"}) = \texttt{"Honda"} \wedge \\ &f(\texttt{"Ducati125"}) = \texttt{"Ducati"} \wedge f(\texttt{"Honda250"}) = \texttt{"Honda"} \wedge \\ &f(\texttt{"Ducati250"}) = \texttt{"Ducati"} \wedge f(\texttt{"Honda550"}) = \texttt{"Honda"}. \end{aligned} \tag{7}$$

EUSolver solves this problem with the following implementation:

$$f(x) = \underbrace{\texttt{substr}(x, 0, 5)}_{h_1} + \underbrace{\texttt{strat}(\texttt{substr}(x, 5, 4), 0)}_{h_2}. \tag{8}$$

Unfortunately, CVC5 rejects this same implementation as being inconsistent with the examples. In order to diagnose this discrepancy, we used CVC5 to compute the subspecifications of the holes labelled h_1 and h_2 in Equation **??**, which results in $\varphi_3|^f_{h_1} = \texttt{false}$ and the following formula, $\varphi_3|^f_{h_2}$, respectively:

$$\begin{aligned} &f(\texttt{"Ducati100"}) = \texttt{"i"} \wedge f(\texttt{"Honda125"}) = \texttt{""} \wedge \\ &f(\texttt{"Ducati125"}) = \texttt{"i"} \wedge f(\texttt{"Honda250"}) = \texttt{""} \wedge \\ &f(\texttt{"Ducati250"}) = \texttt{"i"} \wedge f(\texttt{"Honda550"}) = \texttt{""}. \end{aligned} \tag{9}$$

This indicates that there is no expression g_1 that can be substituted in $g(x)$+$\texttt{strat}(\texttt{substr}(x, 5, 4), 0)$ to obtain a valid solution, and hence hints that CVC5 and EUSolver understand the expression $\texttt{strat}(\texttt{substr}(x, 5, 4), 0)$ differently. Further investigation reveals that there was a discrepancy between the two SyGuS solvers—CVC5 and EUSolver—in the semantics of the substr function: In general, the function $\texttt{substr}(w, i, j)$ returns the first j characters in w starting from index i, however, when $i + j$ exceeds the length of the string, CVC5 returns the entire suffix starting from position i, whereas EUSolver returns the empty string. Ignoring parts of the program with false subspecifications allowed us to rapidly localize the problem to the second subexpression in the concatenation. Notably, this is the only implementation produced by EUSolver for which we encounter the subspec false, and one of only five specifications overall with inconsistent subspecifications.

In Tasks 3 and 4 of the user study, we presented participants with this faulty implementation and asked them to identify the bug. Participants with access to subspecs had a higher success rate than the control group, although most of them were admittedly confused about the meaning of false subspecifications. We nevertheless believe that experience with logical specifications can eventually make users familiar with unintuitive specifications of this form.

[3]File named `bikes_small.sl` from the PBE strings track.

4 ALGORITHMIC SYNTHESIS OF SUBSPECIFICATIONS

We note that our definition of subspecifications in Section **??** is purely descriptive, and does not explain their construction or even guarantee their existence. We will focus on these algorithmic issues in this section. We first note that a simple but potentially large subspecification may be easily constructed, and our procedures will rely on various algorithmic manipulations of this *trivial subspec*.

4.1 The Trivial Subspecification

As a running example for this section, we consider the following simple specification φ_4 of a function $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$:

$$f(x, y) \geq x \wedge f(x, y) \geq y. \tag{10}$$

Note that this specification is satisfied by the function that returns the larger of its two input arguments:

$$f(x, y) = \texttt{if}\ x \geq y\ \texttt{then}\ \underbrace{x}_{h}\ \texttt{else}\ y. \tag{11}$$

Say the user probes the subexpression x that appears in the `then`-branch of the above function expression and consider all potential alternative implementations of the form:

$$f^*(x, y) = \texttt{if}\ x \geq y\ \texttt{then}\ g(x, y)\ \texttt{else}\ y,$$

where g is a fresh uninterpreted function. Observe that $f^*(x, y)$ satisfies φ_4 iff $g(x, y)$ satisfies the following specification, obtained by substituting f^* into Equation **??**:

$$\begin{array}{l}(\texttt{if}\ x \geq y\ \texttt{then}\ g(x, y)\ \texttt{else}\ y) \geq x \wedge \\ (\texttt{if}\ x \geq y\ \texttt{then}\ g(x, y)\ \texttt{else}\ y) \geq y.\end{array} \tag{12}$$

Equation **??** is a subspecification of h under the global specification φ_4.

Observe that this technique is quite general: in order to obtain the subspecification at a program location, we simply replace the subexpression at that location with a fresh uninterpreted function and substitute this result into the global specification. We may express this construction using the notation of Section **??** as:

$$\text{triv}(\varphi, f, h) = \varphi(f\ [g/h], \boldsymbol{x}). \tag{13}$$

From construction, it follows that:

Lemma 4.1. *For all global specifications $\varphi(f, \boldsymbol{x})$, implementations f and program locations h, if g is a fresh synthesis target,* $\text{triv}(\varphi, f, h)$ *is the subspecification of h under φ.*

We call this formula the *trivial subspec* of h under φ. Note that even though this construction is a valid subspecification, it is often very large—especially when the global specification is long or makes multiple invocations of the synthesis target. In fact, in our experiments in Section **??**, we observe that the trivial subspec is, on average, 5× the combined size of the specification and implementation. In these situations, the trivial subspec constructed in Equation **??** provides limited insight, especially when compared to optimized representations, such as $x \geq y \implies g(x, y) \geq x$.

4.2 From (Sub-)Specifications to Indicator Functions

At its heart, our algorithm constructs subspecifications by using a SyGuS solver to simplify the trivial subspec. The main challenge in this process is the presence of the uninterpreted synthesis function $g(x, y)$, whose second-order quantifier makes it difficult for SyGuS solvers to process. Our

key insight is to replace calls to the synthesis function $g(x, y)$ in specifications such as Equation **??** with a fresh first-order logical variable t, resulting in the Boolean-valued indicator function:

$$\begin{aligned}\text{ind}_{\varphi_4}(x, y, t) = (\texttt{if}\ x \geq y\ \texttt{then}\ t\ \texttt{else}\ y) \geq x \wedge \\ (\texttt{if}\ x \geq y\ \texttt{then}\ t\ \texttt{else}\ y) \geq y.\end{aligned} \tag{14}$$

Notice that the indicator function can be used to test whether a given implementation $g(x, y)$ locally satisfies the specification at $(x = x_0, y = y_0)$ by evaluating $\text{ind}_{\varphi_4}(x_0, y_0, g(x_0, y_0))$. In other words, $g \models \varphi_4$ iff for all values of the inputs x and y, $\text{ind}_{\varphi_4}(x, y, g(x, y))$ evaluates to `true`. Our goal is to obtain a simplified representation of ind_{φ_4} and convert this back into a specification for g.

We begin by formally showing how to formally construct indicator functions. First, fix a map:

$$\begin{aligned}t = \{f(x) \mapsto t_{f(x)}, f(y) \mapsto t_{f(y)}, \ldots, \\ f(0) \mapsto t_{f(0)}, f(1) \mapsto t_{f(1)}, \ldots, \\ f(x + y) \mapsto t_{f(x+y)}, f(f(x)) \mapsto t_{f(f(x))}, \ldots\}\end{aligned} \tag{15}$$

from *all syntactically distinct* calls to the synthesis target $f(\cdots)$ to their corresponding test variables $t_{f(\ldots)}$. Ensure that all test variables $t_\bullet$ are fresh, and do not occur in the specification φ in question.

Now construct the *indicator expression* ind_φ by replacing all calls to the synthesis function $f(\cdots)$ in the specification φ with the corresponding test variable $t_{f(\ldots)}$. For example, the specification $f(x) > x$ would result in the indicator expression $t_{f(x)} > x$, and the specification $f(f(x)) > f(x)$ would result in the indicator expression $t_{f(f(x))} > t_{f(x)}$. Finally, observe that the mapping into test variables $t_\bullet$ defines a bijection, so that it is possible to exactly recover the function specification φ from its indicator representation ind_φ.

Lemma **??** forms the heart of our algorithmic development:

Lemma 4.2. *If two specifications $\varphi(f, \boldsymbol{x})$ and $\psi(f, \boldsymbol{x})$ have equivalent indicator functions, ind_φ and ind_ψ, then for all potential implementations f, $f \models \varphi$ iff $f \models \psi$.*

Proof. Assume otherwise. WLOG, assume that $f \not\models \varphi$ but $f \models \psi$. Therefore, there exists a valuation $\boldsymbol{x} = \boldsymbol{v}$ of the free variables such that $\neg\varphi(f, \boldsymbol{v})$. Let $\boldsymbol{v}_t$ be the instantiation of the test variables $t_\bullet$ according to the values of f at the corresponding input points. Observe that $\text{ind}_\varphi(\boldsymbol{v}, \boldsymbol{v}_t) = \texttt{false} = \text{ind}_\psi(\boldsymbol{v}, \boldsymbol{v}_t)$. It follows that $\neg\psi(f, \boldsymbol{x})$, which contradicts the assumption that $f \models \psi$. □

4.3 Simplifying Specifications

We illustrate the end-to-end subspecification synthesis pipeline in Figure **??**. Starting from the trivial subspecification, we repeatedly apply a SyGuS solver in a bottom-up manner on the indicator representation to obtain an optimized subspecification. We describe this process in Algorithm **??**. As a straightforward consequence of Lemma **??**, we have:

Lemma 4.3. *Let φ be a specification, and let ψ be the specification associated with the simplified form of its indicator function, $\text{ind}_\psi = \textsc{Simplify}(\text{ind}_\varphi, \texttt{true})$. Then, the specifications φ and ψ are equivalent.*

Our implementation includes two notable optimizations over the procedure described in Lemma **??**. First, instead of requiring global equality between the original and simplified indicator functions, ind_φ and ind_ψ, we only require equality over a more restricted space of inputs, thereby permitting more aggressive simplification. Second, in order to reduce load on the SyGuS solver, we perform a preprocessing pass that performs optimizations such as constant folding. We now describe these optimizations in some detail.

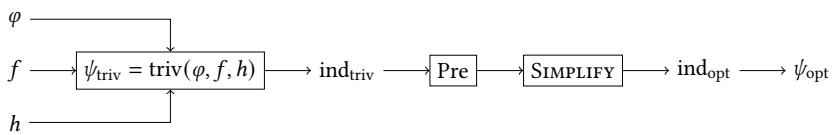


Fig. 1. The subspecification synthesis process using our system S^3. We first construct the trivial subspecification using Equation **??**, and simplify its indicator representation, ind_{triv} using Algorithm **??** to obtain an optimized indicator ind_{opt}. From this optimized indicator, we recover the final output subspecifcation.

Algorithm 1 Simplify(e, π). Recursively simplifies the expression e under the assumptions that its inputs satisfy the condition π.

The following cases arise based on the syntactic form of the expression e:

(1) If $e = c$, for some constant c in the theory: Return c.
(2) If $e = v$, for some formal input variable v: Return v.
(3) If $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$, for some sub-expressions e_1, e_2, e_3, let:

$$e' = \texttt{if}\ e_1'\ \texttt{then}\ e_2'\ \texttt{else}\ e_3', \text{ where}$$
$$e_1' = \textsc{Simplify}(e_1, \pi),$$
$$e_2' = \textsc{Simplify}(e_2, \pi \wedge e_1'), \text{ and}$$
$$e_3' = \textsc{Simplify}(e_3, \pi \wedge \neg e_1').$$

(4) Otherwise, if $e = \text{op}(e_1, e_2, \ldots, e_k)$ for some operator op, let:

$$e' = \text{op}(e_1', e_2', \ldots, e_k'), \text{ where}$$
$$e_i' = \textsc{Simplify}(e_i, \pi), \text{ for each } i.$$

(5) Synthesize a function e'' which is equal to e' on all points which satisfy π:

$$e'' = \textsc{Sygus}(e'' \mid \forall \boldsymbol{x}, \pi(\boldsymbol{x}) \implies e''(\boldsymbol{x}) = e'(\boldsymbol{x})).$$

(6) Return e'' if synthesis was successful and $|e''| < |e'|$. Otherwise, return e'.

Relaxing global indicator equality. Note that even though Lemma **??** holds even for non-pointwise specifications, the requirement that ind_φ and ind_ψ coincide everywhere sometimes limits the effectiveness of simplification. As an example, consider the specification,

$$\varphi \equiv y = z \implies f(x + y) = f(x + z).$$

Note that this specification is satisfied by all functions $f : \mathbb{Z} \to \mathbb{Z}$, and is equivalent to $\psi \equiv \texttt{true}$. However, because their indicator functions:

$$\text{ind}_\varphi(x, y, z, t_{f(x+y)}, t_{f(x+z)}) = (y = z \implies t_{f(x+y)} = t_{f(x+z)}), \text{ and}$$
$$\text{ind}_\psi(x, y, z, t_{f(x+y)}, t_{f(x+z)}) = \texttt{true},$$

are inequivalent, the synthesizer is unable to simplify φ into ψ. On the other hand, notice that their behaviors only diverge on inputs where $y = z$ and $t_{f(x+y)} \neq t_{f(x+z)}$. Since the test variables $t_\bullet$ in the proof of Lemma **??** are instantiated based on the implementation f and the values of x, y, z, these distinguishing inputs would violate the functional constraints on f and are therefore physically unrealizable.

We therefore associate each specification φ with a set of functional constraints FC_φ which is the conjunction of all clauses $\boldsymbol{x} = \boldsymbol{y} \implies t_{f(\boldsymbol{x})} = t_{f(\boldsymbol{y})}$, for each pair $f(\boldsymbol{x})$, $f(\boldsymbol{y})$ of syntactically different function calls in φ. For the example constraint φ considered above, $\text{FC}_\varphi \equiv x + y = x +$

$z \implies t_{f(x+y)} = t_{f(x+z)}$. To compute the optimized subspecification, it now suffices to find a formula ψ such that $\mathrm{FC}_\varphi \implies \mathrm{ind}_\varphi = \mathrm{ind}_\psi$. We therefore pass FC_φ as the seed assumption to the simplification procedure of Algorithm **??**.

Preprocessing passes. Finally, in order to reduce the load on the SyGuS solver in Step **??** of Algorithm **??**, we apply three categories of rewrite rules to the indicator functions before they are submitted to the core synthesis algorithm:

(1) We perform constant folding and eagerly evaluate all subexpressions which do not contain any free variables.
(2) We perform short-circuit evaluation of Boolean connectives and conditional expressions.
(3) We manually cancel identical prefixes and suffixes in string equalities.

These steps are of particular value in PBE problem instances, and in problems with string constraints. These preprocessing passes capture a lot of easy simplifications and significantly reduce the load on the core synthesis phase of the algorithm.

4.4 Verifying Correctness of Subspecifications

A second algorithmic problem with subspecifications involves determining whether a proposed subspecification is indeed a subspecification according to our definition in Section **??**. This is in general a challenging problem as the converse of Lemma **??** does not hold, and indicator functions can no longer be used to prove their correctness. Surprisingly however, the converse holds for the restricted case of pointwise subspecifications, thus permitting algorithms to mechanically check their correctness:

Lemma 4.4. *Consider a specification φ and a conforming implementation f. Let h be a hole in f. Let the trivial subspecification $\psi(h, \boldsymbol{x}) = \mathrm{triv}(\varphi, f, h)$ be pointwise, and consider any other pointwise representation of the same subspec, $\theta(h, \boldsymbol{x})$. Then the corresponding indicator functions, ind_ψ and ind_θ, are equivalent.*

Proof. Assume otherwise. WLOG, assume that in both specifications, the invocations of f in ψ and θ are syntactically equal.

Now, there exists a valuation $\boldsymbol{x} = \boldsymbol{v}$ of the free variables and a valuation $t_\bullet = \boldsymbol{v}_t$ of the test variables such that $\mathrm{ind}_\psi(\boldsymbol{v}, \boldsymbol{v}_t) = \texttt{true}$ and $\mathrm{ind}_\theta(\boldsymbol{v}, \boldsymbol{v}_t) = \texttt{false}$, or vice-versa. In addition, because ψ and θ are equivalent representations of the same subspec $\varphi|_h^f$, there is a function g such that $g \models \psi$ and $g \models \theta$. (Recall that the current subexpression $f \downarrow h$ is itself a natural choice for g.) Now construct the function:

$$g'(x) = \begin{cases} g(x) & \text{if } x \neq v, \text{ and} \\ \boldsymbol{v}_t & \text{otherwise.} \end{cases}$$

In other words, we have surgically constructed a function g' which agrees with g everywhere except at the point $\boldsymbol{v}_t$.

In the first case, where $\mathrm{ind}_\psi(\boldsymbol{v}, \boldsymbol{v}_t) = \texttt{true}$ and $\mathrm{ind}_\theta(\boldsymbol{v}, \boldsymbol{v}_t) = \texttt{false}$, observe that $g' \models \psi$ and $g' \not\models \theta$. This contradicts the assumption that ψ and θ were equivalent. The other case is similar. □

To confirm the correctness of a proposed subspecification, in the case where both the trivial subspec and the candidate subspec are pointwise, it suffices to merely check whether the corresponding indicator functions are globally equal.

5 PROPERTIES OF SUBSPECIFICATIONS

We now discuss some interesting properties about subspecifications in general, and in particular, subspecifications produced by our algorithm from Section **??**.

5.1 Subspecifications for Specialized Classes of Specifications

We discuss the subspecifications that arise for two special classes of specifications, point-wise specifications and programming-by-example (PBE) specifications. These classes of specifications are syntactically simple, easier to comprehend, and are amenable to more efficient synthesis algorithms. Hence, we would like subspecifications arising from these specifications to be of the same class. Below, we show that this is true for point-wise specifications in general, and is true for PBE specifications if the hole is highly constrained.

Subspecifications for point-wise specifications. The theorem below states that for every hole h in an implementation f_0 of a point-wise specification φ, there exists a point-wise subspecification. This follows easily from the definition of trivial subspecification from Section **??**—the trivial subspecifications for point-wise specifications are point-wise. Consequently, we have that the simplified subspecification returned by our algorithm is also point-wise. On the other hand, observe that as Example **??** from Section **??** shows, specifications that are not point-wise might still lead to subspecifications that are point-wise.

Theorem 5.1. *Given a point-wise specification $\varphi(f, \boldsymbol{x})$ and an implementation f_0 that satisfies φ, for every hole h in f, there exists a point-wise specification $\varphi|_h^f$ that is a valid subspecification for h in f.*

Subspecifications for PBE specifications. PBE specifications are simple to reason about and hence, one would wish that subspecifications for PBE are also in the PBE form. Unfortunately, as the example below shows, this is not always true.

Example 5.2. Consider the PBE specification φ given by $f(\texttt{"Alan Turing"}) = \texttt{"Alan"}$, and a corresponding implementation $f(x) = \texttt{substr}(\underbrace{\quad x \quad}_{h}, 0, (\texttt{indexof}(x, \texttt{" "}, 0)))$. The subspecification $\varphi|_h^f$ for h in f is given by $\texttt{substr}(g(\texttt{"Alan Turing"}), 0, 4) = \texttt{"Alan"}$. That is, we can replace h with any function that produces a string that starts with `"Alan"` when the input is `"Alan Turing"`. This subspecification is too loose to be written as a PBE task—it does not constrain the output for the input `"Alan Turing"` to a single value, but any of the infinite set of strings that start with `"Alan"`.

Taking a closer look at the example above, we observe that the problem is caused by the function `substr`. Since it is not a one-to-one function, for a specific output, there are infinite inputs that evaluate to the output, leading to a loose subspecification. On the other hand, in Example **??** in Section **??**, the subspecification for h_2 is indeed a PBE specification. The reason is that when one parameter of the string concatenation is fixed, the resulting function is a one-to-one mapping, thus impose a strict constraint to that part of the code.

This phenomenon where a one-to-one function imposes a strict subspecification also appears in non-PBE settings. Consider the specification $\varphi \equiv f(x, y) = 2x + y$ and the implementation $f(x, y) = x + y + x$. The subspecifications for the holes corresponding to the subexpressions x and y are just $g(x, y) = x$ and $g(x, y) = y$, respectively. This is because when one parameter of the add function is fixed, it becomes a one-to-one mapping.

5.2 Multi-hole subspecifications

Until now, we have only discussed the properties of subspecifications for single holes. Here, we discuss how subspecifications for multiple holes relate to each other. We study two aspects: how are the subspecifications for multiple holes related to the subspecifications for each of the individual holes, and how are the subspecifications for individual holes related to the original specification?

Independent holes. In general, the joint subspecification $\varphi|^f_{h_1,h_2}$ for two holes h_1 and h_2 need not related to the individual subspecifications $\varphi|^f_{h_1}$ and $\varphi|^f_{h_2}$ in a simple way.

Example 5.3. Consider the specification $\varphi \equiv f(x,y) = f(y,x) \wedge f(x,y) \in \{x-y, y-x\}$ and the implementation $f(x,y) = \texttt{if}\ x \geq y\ \texttt{then}\ \underbrace{x-y}_{h_2}\ \texttt{else}\ \underbrace{y-x}_{h_3}$ from Section **??**. The subspecifications $\varphi|^f_{h_2}$ and $\varphi|^f_{h_3}$ are equivalent to $x \geq y \implies g_2(x,y) = x-y$ and $x < y \implies g_3(x,y) = y-x$ respectively. Intuitively, the value of each of the holes h_2 and h_3 is fixed as soon as the subexpression in the opposite branch is decided. However, the joint subspecification is more relaxed—the values of h_2 and h_3 may be interchanged, i.e., we can set h_2 to $y-x$ and h_3 to $x-y$, so the following is also a valid implementation: $f'(x,y) = \texttt{if}\ x \geq y\ \texttt{then}\ y-x\ \texttt{else}\ x-y$. Formally, the joint subspecification can be written as $\varphi|^f_{h_2,h_3} \equiv g_2(x,y) = -g_3(x,y) \wedge (g_2(x,y) = x-y \vee g_2(x,y) = y-x)$.

However, joint subspecifications can be independently computed from individual holes under certain conditions.

Example 5.4. Consider the specification $\varphi \equiv f(x,y) \geq x \wedge f(x,y) \geq y \wedge f(x,y) \in \{x,y\}$ and the corresponding implementation $f(x,y) = \texttt{if}\ x \geq y\ \texttt{then}\ \underbrace{x}_{h_1}\ \texttt{else}\ \underbrace{y}_{h_2}$. Here, the subspecifications for h_2 and h_2 are given by $\varphi|^f_{h_1} \equiv x \geq \implies g_1(x,y) = x$ and $\varphi|^f_{h_2} \equiv x < y \implies g_2(x,y) = y$. Now, joint subspecification is just the conjunction $\varphi|^f_{h_1,h_2} \equiv \varphi|^f_{h_1} \wedge \varphi|^f_{h_2}$. In this case, we can get the joint subspecification by just taking the conjunction because the subexpressions in the holes h_1 and h_2 do not interact with each other in any execution of f.

In the above example, we were able to compute the subspecifications independently because the subexpressions corresponding to the two holes have disjoint path conditions. However, as Example **??** shows, this alone is not sufficient. There, the two holes do not interact with each other in any execution, but they do in the specification as it is not point-wise. Hence, in addition we want that the specification is point-wise. The following theorem formalizes this discussion.

Theorem 5.5. *Let φ be a point-wise specification, f be an implementation for φ, and h_1 and h_2 be two holes in f. If $\mathrm{PC}(h_1) \cap \mathrm{PC}(h_2) = \emptyset$, where $\mathrm{PC}(h)$ is the path condition leading to hole h, then $\varphi|^f_{h_1,h_2} = \varphi|^f_{h_1} \wedge \varphi|^f_{h_2}$ is a valid joint subspecification for h_1 and h_2 in f.*

We postpone the proof to Appendix **??**.

Reconstructing specifications. Given that joint subspecifications for multiple holes capture more information about the specification than subspecifications at individual holes, we might ask whether the joint subspecification captures *all* information about the original subspecification. Unfortunately, the following example shows that this is untrue.

Example 5.6. Consider the specification $\varphi \equiv f(x,y) \geq 0$ and the corresponding implementation $f(x,y) = |\ \underbrace{x}_{h_1} - \underbrace{y}_{h_2}\ |$. Now, the joint subspecification $\varphi|^f_{h_1,h_2}$ is just `true`. From $\varphi|^f_{h_1,h_2}$ and the implementation f alone, it is therefore impossible to reconstruct φ, i.e., there is no procedure to check if an arbitrary implementation f' is correct using just the subspecification.

However, as the theorem below shows, in certain specific circumstances, it is possible to use joint subspecifications in lieu of the original specification.

Theorem 5.7 (Reconstruction). *Let $\varphi(f, \boldsymbol{x})$ be a point-wise specification, $f(\boldsymbol{x}) = \text{op}(\underbrace{e_1}_{h_1}, \underbrace{e_2}_{h_2})$ be an implementation for φ, and $\psi(g_1, g_2, \boldsymbol{x})$ be a joint subspecification for h_1, h_2 in f. Suppose that the operator* op *is surjective. Then, for any f', we can construct g_1' and g_2' such that $f' \models \varphi$ if and only if $(g_1', g_2') \models \psi$.*

The proof of this theorem may also be found in Appendix **??**.

6 USER STUDY

In order to determine whether users can understand the output of program synthesizers, and whether subspecifications help in this process, we conducted a small user study in which we focused on answering the following questions:

RQ1. Do subspecs help in understanding implementation?
RQ2. Do subspecs help in understanding specification?
RQ3. Can subspecs help in debugging faulty implementations?

After approval from the local IRB, we recruited 20 Ph.D. students from the Computer Science, Electrical Engineering, and Industrial Engineering departments of a prominent American university. These participants had a diverse range of research areas, including formal verification and software engineering, cyber-physical systems, IoT, MEMS, optimization, algorithmic privacy, and machine learning. As such, we expect these participants to be potential unexpert users of program synthesis tools.

6.1 Tasks and Study Structure

The study consisted of four tasks inspired by the motivating examples discussed in Section **??**. Before participants attempted these tasks, we presented them with a short introduction to the style of program synthesis used in SyGuS. We showed them examples of specifications that one could present to a synthesizer and examples of implementations obtained in response. To ensure that participants had a minimum level of familiarity with the problem setting, we presented four quiz questions in which we asked them to determine whether an implementation satisfied the constraints imposed by a specification. The text of these questions may be found in Appendix **??**. We only considered responses from participants who correctly answered all quiz questions. 19 of the 20 participants satisfied this requirement and passed the quiz. In addition, one participant withdrew while the study was in progress because of tiredness, leaving 18 participants who provided data for the full study.

In these four tasks, we asked participants to identify which implementations satisfied a given specification, to present alternative implementations, to explain the specification in their own words, and even to debug broken implementations. We present the full text of these tasks in Tables **??**–**??**. We presented Tasks 1, 2, and 4 to the participants in one of two randomly chosen conditions, i.e., with and without access to subspecifications respectively. Each participant attempted at least one task with subspecifications and one task without access to subspecifications. They were able to access the subspecifications on demand through a simple point-and-click web interface as shown in Figure **??**. We designed Task 3 to familiarize participants with `false` subspecifications before encountering them in Task 4. Consequently, all participants had access to subspecifications while attempting Task 3.

We measured the time needed by participants to answer these questions and their accuracy under each of the conditions. We present the time taken to complete the tasks in Figure **??**. After the study was complete, we had a short discussion with each of the participants and obtained their

Table 1. Questions asked in Task 1. We presented the original specification (Equation ??) and original implementation (Equation ??) from Example ??. Only the highlighted subexpressions have changed from the original implementation. We report the number of participants who correctly answered each question in either setting.

	Question	Accuracy With Subspecs	Accuracy W/o Subspecs
	Which of the following implementations also satisfy the original specification?		
Q1.1	$f(x,y) = \texttt{if}\ x = y\ \texttt{then}\ 0\ \texttt{else}\ \underbrace{2}$	9 / 9	8 / 9
Q1.2	$f(x,y) = \texttt{if}\ x = y\ \texttt{then}\ 0\ \texttt{else}\ \underbrace{0}$	9 / 9	6 / 9
Q1.3	$f(x,y) = \texttt{if}\ x = y\ \texttt{then}\ \underbrace{1}\ \texttt{else}\ 1$	9 / 9	7 / 9
Q1.4	$f(x,y) = \texttt{if}\ x = y\ \texttt{then}\ \underbrace{2}\ \texttt{else}\ 1$	9 / 9	7 / 9
Q1.5	Can you provide an alternative implementation?	9 / 9	2 / 9
Q1.6	Can you explain the specification in your own words?	6 / 9	1 / 9

Table 2. Questions asked in Task 2. The PBE specification was similar to that considered in Example ??, but included more examples. The specification and implementation may be found in Appendix ??.

	Question	Accuracy With Subspecs	Accuracy W/o Subspecs
	Which input-output examples are relevant to following subexpressions?		
Q2.1	$f(x,y) = \texttt{if}\ x \leq 1\ \texttt{then}\ \underbrace{3x}\ \texttt{else if}\ y \leq 1\ \texttt{then}\ 3x\ \texttt{else}\ 1$	8.9 / 9	9 / 9
Q2.2	$f(x,y) = \texttt{if}\ x \leq 1\ \texttt{then}\ 3x\ \texttt{else if}\ y \leq 1\ \texttt{then}\ \underbrace{3y}\ \texttt{else}\ 1$	9 / 9	9 / 9
Q2.3	$f(x,y) = \texttt{if}\ x \leq 1\ \texttt{then}\ 3x\ \texttt{else if}\ y \leq 1\ \texttt{then}\ 3y\ \texttt{else}\ \underbrace{1}$	9 / 9	8.9 / 9
Q2.4	What do you think was the user's intent?		
Q2.5	Can you suggest some examples to disambiguate the specification?		

feedback about which aspects of the study they found easy or difficult, and their experience while using subspecification interface.

6.2 ??: Effectiveness in Understanding Implementations

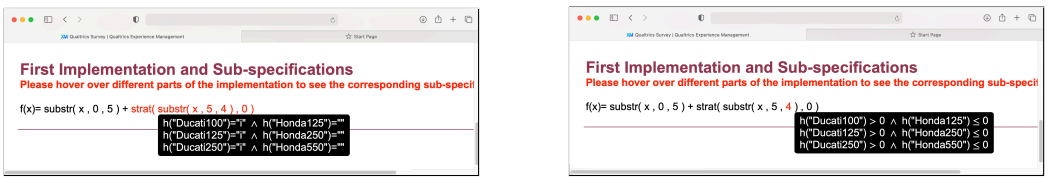
Our first research question involves determining whether participants understood the implementations produced by a program synthesizer. We focus on their responses to Questions 1.1–1.4 of Task 1 and Questions 2.1–2.3 of Task 2. Observe that the questions in Task 1 involve a minor alteration to an existing implementation and asking participants to determine whether the new implementation continues to satisfy the old specification. On the other hand, the questions of Task 2

Table 3. Questions asked in Task 3. The specification required a function such that $\forall x, f(x) \geq 0$, and the proposed implementation was the expression $f(x) = \texttt{if}\ x \geq 0\ \texttt{then}\ x\ \texttt{else}\ -1$.

	Question	Accuracy With Subspecs
Q3.1	There is a bug in the implementation. Where is the bug?	18 / 18
Q3.2	Can you fix the bug?	18 / 18
Q3.3	Consider the following sub-expression in the implementation: $f(x, y) = \texttt{if}\ x \geq 0\ \texttt{then}\ \underbrace{x}\ \texttt{else}\ -1$. Note that the sub-specification is `false` for the sub-expression. Why?	2 / 18

Table 4. Questions asked in Task 4. The specification-implementation pair for this task was the same as in Example **??**.

	Question	Accuracy With Subspecs	Accuracy W/o Subspecs
Q4.1	Which implementation is buggy?	8/9	6 / 9
Q4.2	In which subexpression does the bug occur?	7 / 9	6 / 9
Q4.3	Can you fix the implementation?	5 / 9	3 / 9



(a) (b)

Fig. 2. Interface used by participants to query subspecifications. Upon hovering their mouse pointer over different subexpressions, a tooltip would appear to show the corresponding subspecification.

constitute a form of requirements tracing, and ask the participants to identify relevant parts of the specification for each part of the implementation.

From the accuracy measurements of Table **??**, we observe that subspecifications significantly improve the accuracy of participants: *all* responses from participants with access to subspecifications are correct, while three responses to Question 1.2 from participants without subspecifications are incorrect. A single participant with access to subspecs accidentally missed a response to Q2.1 leading to an imperfect score, and realized this soon after completing the task. Even though subspecifications do not appear to raise overall accuracy for Task 2, they massively reduce the time needed to answer the associated questions: As we can see in Figure **??**, on average, participants with access to subspecifications require only 70% and 48% of the time needed by the control group to complete the respective tasks.

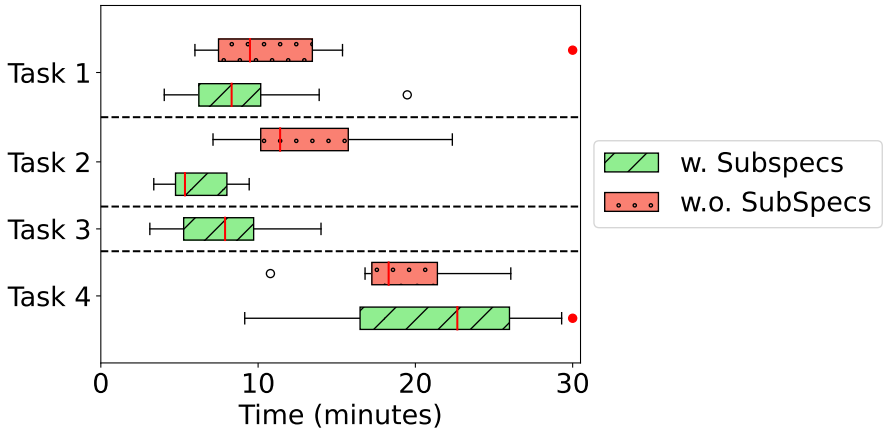


Fig. 3. Distribution of time needed by participants to complete the tasks. In Task 3, all participants had access to subspecifications. One participant required more than 30 minutes to complete Tasks 1 and 4, which we highlight with the red outlier dot at the right end of the figure.

In addition, when reviewing the tasks during the post-study debrief, we discovered that many participants in the control group had imprecise reasons for their responses in Task 1, and had a tendency to guess when they were otherwise unsure of the answer. On the other hand, participants with subspecs were able to immediately complete the task after consulting the corresponding subspecifications. Overall, this indicates that subspecifications do help users in understanding why implementations work.

6.3 ??: Effectiveness in Understanding Specification

Through our second research question, we sought to determine whether users can readily understand specifications—for example, by explaining it in their words—and whether subspecifications can help in this process. In Question 1.5 of Task 1, we asked participants to provide a new implementation for the original specification, and in Question 1.6, we asked them to explain the original specification in English. Along similar lines, Question 2.4 asks participants to guess the intent of the author of the original specification and Question 2.5 asks participants to suggest additional input-output data points to disambiguate this intent.

Observe that all participants with access to subspecifications are able to provide alternative implementations as part of their response to Question 1.5, while only two participants were able to do so in the control group. Four of the responses from the group with access to subspecs were semantically new implementations, with differing behavior when $x \neq y$, while only one of the participants in the control group discovered this approach. The remaining correct responses were merely syntactic variations of the existing implementation.

We manually judged the free-form responses to Question 1.6 and assessed whether they were compatible with the specification. All responses we judged as correct were variations of "*If* $x = y$ *then the function should return* 0 *and otherwise it should return either* 1 *or* 2." On the other hand, many participants who we judged as answering the question incorrectly had difficulty in understanding the required behavior when $x \neq y$: two participants said that the output had to be 1 in this case, two other participants believed that the specification only required a non-zero output when $x \neq y$, and two participants guessed that the spec required a positive value. Three participants in the control group also skipped the question.

While attempting Task 1, 5 participants asked whether subspecs imposed an exact requirement on potential implementations, and were relieved when they realized that they could just consult the subspecs while forming their responses.

When we studied the responses to Question 2.4 of Task 2, we observed that one participant from the control group provided an interpretation that was inconsistent with even the provided input-output examples, and another participant, also from the control group, chose to skip the question entirely. We were able to cluster the remaining responses into two groups: The first group simply provided English readings of the implementation as explanations of user intent. This included 3 responses from the users with subspecs and 4 responses from users without subspecs. On the other hand, the second group of responses attempted to guess the intent from the provided input-output examples, and included responses such as "*The function checks whether either input has value* 1." Note that such responses are not directly available from the implementation and are in fact incompatible with it. 6 of the participants with subspecs provided responses of this kind, while only 3 participants without subspecs provided similar answers. We conclude that subspecs encourage users to more actively interrogate the specification, thereby aiding comprehension.

6.4 ??: Effectiveness in Debugging Faulty Implementations

As part of the third research question, we wanted to confirm whether subspecifications help in debugging faulty implementations, as discussed in Example **??**. In order to familiarize participants with subspecifications for buggy implementations, we first presented them with a relatively simple specification-implementation pair in Task 3:

$$\forall x, f(x) \geq 0, \text{ and}$$

$$f(x) = \texttt{if}\ x \geq 0\ \texttt{then}\ x\ \texttt{else}\ -1.$$

As part of the three questions associated with this task, we asked participants to reflect on the meaning of the `false` subspecification in the `then`-branch. Unfortunately, only two participants gave answers that could be considered correct: one of them deduced that the bug must be elsewhere in the implementation, while the other participant guessed that it was a hint from the synthesizer to "*not worry about this subexpression*." Most other participants reported not knowing the answer.

Despite this, participants with access to subspecifications had a higher accuracy on Task 4, where we reused the specification-implementation pair from Equations **??** and **??** in Example **??**. We asked participants to identify which of two implementations was buggy (the other satisfying implementation may be found in Appendix **??**), to locate the faulty subexpression, and to fix the subexpression in question.

The two participants who successfully completed Task 3 were both in the group who had access to subspecifications in Task 4. They took 16 minutes and 24 minutes respectively to correctly respond to all questions of the last task. In the post-study discussions, participants achieved a better appreciation of what it meant for the subspec to be `false`, and described it as a useful debugging technique. One participant also believed that this represented a powerful alternative way of applying subspecs. One of the participants in the control group who we judged as correctly answering the third question simply copied the reference implementation from before.

Finally, some participants reported being confused by having access to lots of subspecifications. We believe that with greater familiarity with logical specifications as used in SyGuS solvers, users will overcome a number of these difficulties and become more fluent in querying for subspecs on demand.

7 EXPERIMENTAL EVALUATION

We have implemented the subspecification synthesis algorithm of Section **??** in a tool called S^3. It consists of approximately 2,300 lines of Python code, and uses CVC5 to discharge the underlying SyGuS queries. Our evaluation focused on the following research questions:

RQ4. How effective is S^3 in simplifying subspecifications?
RQ5. How long does S^3 take to construct subspecs?
RQ6. How do the preprocessing steps of Section **??** affect simplification effectiveness?

Benchmarks. We performed our evaluation using the benchmarks from the 2017 SyGuS Competition as the specifications. We ran both CVC5 and EUSolver on these specifications and obtained the corresponding implementations. With a timeout of 5 minutes, the solvers are able to successfully discharge 1,381 and 1,359 benchmarks respectively. Subsequently, we narrowed our focus to benchmarks whose implementations had less than 100 holes. This resulted 1,112 specification-implementation pairs from CVC5 and 1,253 specification-implementation pairs from EUSolver. Collectively, they contained 26,437 and 25,659 holes respectively.

We ran S^3 on all these holes with a per-hole timeout of 5 minutes, and a per-implementation deadline of 30 minutes. After this process, we had access to the subspecifications for 24,701 and 24,596 holes respectively.

Experimental setup. We ran our experiments on a workstation machine with an AMD Ryzen 9 5950X CPU and 128 GB of memory running Ubuntu 21.04. We note that the computations are primarily CPU bound rather than memory intensive. Furthermore, our experiments focus primarily on comparative running times rather than absolute values, so largely identical results should be obtained on most contemporary computers.

We devote the rest of this section to examining the research questions listed above.

7.1 ??: Effectiveness in Simplification

To measure the effectiveness of our technique in deriving simple subspecifications, we compared the size of the synthesized subspecification to the size of the trivial subspec. We present these results in Figure **??**, with a separate curves for the implementations obtained from CVC5 and EUSolver respectively. The x-axis indicates the compression ratio, while the y-axis indicates the cumulative number of holes at which S^3 achieve this compression ratio or better.

Although S^3 is marginally more effective at simplifying subspecifications from CVC5 than from EUSolver, it remains uniformly effective, and achieves a 74% reduction in the size of the trivial subspecification for 74% of all holes. This indicates that the system is able to achieve significant simplification. When one focuses on probe points with depth equal to or greater than 10, S^3 achieves an 98.7% reduction in subspecification size. When focusing on benchmarks from the bitvector theory, the system is able to achieve more than 80% reduction for 80% of the probe points. On the other hand, because of fundamental limits, expressions in the SLIA fragment cannot be significantly compressed, and the system has noticeably poorer compression rates. It is only able to achieve 80% compression for 23% of the holes.

We finally note that our system is able to simplify the subspecification to `true` for 464 holes: by indicating unconstrained subexpressions, S^3 is therefore able to reveal potential opportunities for further optimizing the implementation produced by the synthesizer.

7.2 ??: Time Needed to Construct Subspecs

We next measured the time needed by S^3 to construct the optimized subspec. We compared these running times to the time needed by EUSolver to originally solve the synthesis problem instance.

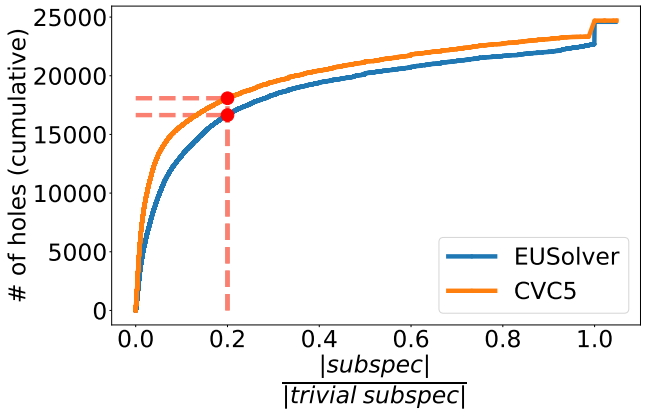


Fig. 4. Effectiveness of S^3 in simplifying subspecifications. We count the number of benchmarks where the algorithm is able to reduce the size of the subspec to the corresponding fraction of the size of the original trivial subspec.

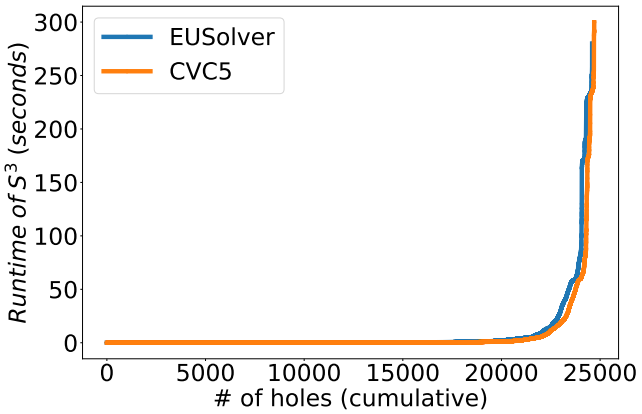


Fig. 5. Cactus plot indicating the time needed by the tool to synthesize subspecifications for implementations originally obtained from CVC5 and EUSolver respectively.

We present the absolute values of running times in Figure **??** and the comparison to the baseline SyGuS solving time in Table **??**.

Observe that subspecs for 77% of the probe points can be constructed in less than 1 second, and as many as 83% of the subspecifications are constructed in less time than was originally needed to construct the implementation by the SyGuS solver. Only 8% of the benchmarks exceed $> 10\times$ of this baseline time, and are mainly from PBE tracks involving strings and bitvectors, and with 1000 examples each. Nevertheless, our systems appears to be sufficiently fast to be used to interactively reason about synthesis problem instances.

Table 5. Comparison of time needed to compute subspecifications to the time needed for originally synthesizing the implementation. Each cell indicates the fraction of holes from each synthesizer for which synthesizing the subspecification took the corresponding amount of time.

Synthesizer	$\leq 1\times$	$\leq 10\times$	$\leq 100\times$	$> 100\times$
EUSolver	82%	92%	98%	2%
CVC5	81%	92%	98%	2%

Table 6. Impact of preprocessing and main simplification loops in reducing the size of the final subspecification.

Synthesizer	**|Preproc| / |Triv|**	**|Final| / |Preproc|**	**|Final| / |Trivial|**
EUSolver	9%	78%	7%
CVC5	5%	77%	4%

7.3 ??: Effect of Preprocessing Passes

Finally, we assessed the impact of preprocessing steps on simplification effectiveness. We describe this data in Table **??**. Observe that the preprocessing passes capture a large fraction of the easy simplification opportunities, and already achieve a 93% reduction in the size of the trivial subspecification. The more expensive recursive simplification process of Algorithm **??** achieves an additional 23% reduction in the size of the simplified subspecifications thus achieving an overall 95% reduction in size of the simplified subspec.

Despite these aggregate statistics, both steps are crucial to the effectiveness of the overall procedure. The most notable of these examples are when the subspecs simplify to `true`, which is typically the result of the main synthesis process and not the preprocessing pass.

8 RELATED WORK

Explainability and interpretability are increasingly important topics in several research areas, including machine learning [**?**], algorithmic transparency [**?**], and reinforcement learning [**?**]. The availability of explicit program representations has made explainability a less pressing issue for program verification and synthesis. Nevertheless, the large body of work on program debugging and comprehension [**??**], program slicing [**??**], and user-guided program synthesis [**?**] underscores the *importance of program explainability*. In this section, we outline notable threads of research that share facets of our work, in particular, research on explainability, local reasoning, and expression simplification.

The closest related work is by Finkbeiner et al.[**?**]. In [**?**], the authors have recently employed a similar idea of decomposing specifications into smaller constituent units and using a divide-and-conquer approach to accelerate reactive synthesis. They divide the original synthesis task into independent subtasks, which can then be solved in parallel. While our use of the term is very close to theirs, there are two notable differences between the papers: first, our concept is slightly more general in that it does not require different parts of the program to be necessarily independent of each other, and second, they use subspecifications primarily to accelerate the synthesis process rather than to facilitate programmer comprehension.

Explainability and Interpretability in AI.. The increased prevalence of black-box or grey-box methods has led to growing interest in explaining and interpreting their functioning and results.

This is especially true in AI [?]. Efforts to better understand the highly expressive and parametric models used for machine learning include model-specific visualization tools [?], model agnostic methods [?], and example-based methods [?]. While there are important differences between program synthesis and machine learning, it *is* possible to adopt the latter's approaches to help explain the program synthesis process, especially for PBE. This is evidenced by recent contributions in visualization for program synthesis [?].

Local reasoning. The concept of local reasoning has been used in multiple areas. For instance, in machine learning, local reasoning is used to explain predictions made on specific inputs (see, for instance, [?] and [?]). In contrast, our local reasoning focuses more on explaining how parts of the model affect the outcome. Local reasoning has, of course, long been used in software verification in the context of modular verification [?]. More recently, local specifications [?] have been adopted in program synthesis by asking a user to provide examples for program snippets; such examples can be viewed as instances of subspecs.

Expression simplification. The core part of our algorithm relies on simplifying Boolean expressions. Expression simplification has been widely studied in compiler optimization, for instance, via constant folding, common subexpression elimination, and partial evaluation [?], with many methods relying on rewriting techniques such as equality saturation [?]. While our method leverages synthesizers to simplify expressions, its performance can potentially be boosted by combining it with such rewriting techniques.

9 CONCLUSION

In this paper, we considered the problem of explaining the output of program synthesizers, and introduced the concept of subspecifications as a mechanism by which users can probe the synthesized program and discover the impact of the global specification on each of its parts. We discussed several examples where subspecifications provided insight into the specification, implementation, and even the semantics of the synthesizer. We presented an algorithm to construct concise subspecifications, and conducted experiments to investigate its effectiveness. In future, we hope to apply the concept more broadly, to debugging specifications, to facilitate user interaction, and in applications beyond program synthesis.

A OMITTED PROOFS

Theorem A.1. *Let φ be a point-wise specification, f be an implementation for φ, and h_1 and h_2 be two holes in f. If $PC(h_1) \cap PC(h_2) = \emptyset$, where $PC(h)$ is the path condition leading up to hole h, then $\varphi|^f_{h_1,h_2} = \varphi|^f_{h_1} \wedge \varphi|^f_{h_2}$ is a valid joint subspecification for h_1 and h_2 in f.*

Proof. We need to show that $\left(g_1 \models \varphi|^f_{h_1} \wedge g_2 \models \varphi|^f_{h_2}\right) \Leftrightarrow f\,[g_1/h_1, g_2/h_2] \models \varphi$.

($\Rightarrow$) Towards a contradiction, assume that $\exists x_0$ such that $\varphi(f\,[g_1/h_1, g_2/h_2], x_0)$ does not hold. Now, at least one of $x_0 \notin \mathrm{PC}(h_1)$ or $x_0 \notin \mathrm{PC}(h_2)$ holds. Without loss of generality, say $x_0 \notin \mathrm{PC}(h_2)$. Now, by the definition of PC, we have that $f\,[g_1/h_1, g_2/h_2](x_0) = f\,[g_1/h_1](x_0)$. Hence, we get that $\varphi(f\,[g_1/h_1], x_0)$ evaluates to false, contradicting the assumption that $g_1 \models \varphi|^f_{h_1}$.

($\Leftarrow$) Towards a contradiction, assume that one of $g_1 \not\models \varphi|^f_{h_1}$ or $g_2 \not\models \varphi|^f_{h_2}$ holds, and without loss of generality, assume that it is the former. Now, there exists an x_0 such that $\varphi(f\,[g_1/h_1, g_2/h_2](x_0), x_0)$ holds and $\varphi(f\,[g_1/h_1](x_0), x_0)$ does not hold. If $x_0 \notin \mathrm{PC}(h_1)$, we have that $f\,[g_1/h_1](x_0) = f(x_0)$. Hence, $\varphi(f\,[g_1/h_1](x_0), x_0)$ is false implies that $f \not\models \varphi$ leading to a contradiction. Therefore, $x_0 \in \mathrm{PC}(h_1)$, which in turn implies that $x_0 \notin \mathrm{PC}(h_2)$. Now, we have that $f\,[g_1/h_1, g_2/h_2](x_0) = f\,[g_1/h_1](x_0)$. This ensures that $\varphi(f\,[g_1/h_1, g_2/h_2](x_0), x_0)$ and $\varphi(f\,[g_1/h_1](x_0), x_0)$ should have the same truth value, which contradicts the assumption. □

Theorem A.2. *Let $\varphi(f, \boldsymbol{x})$ be a point-wise specification, $f(\boldsymbol{x}) = op(\underbrace{e_1}_{h_1}, \underbrace{e_2}_{h_2})$ be an implementation for φ, and $\psi(g_1, g_2, \boldsymbol{x})$ be a subspecification for h_1, h_2 in f. Suppose op is surjective. Then, for any f', we can construct g'_1 and g'_2 such that $f' \models \varphi$ if and only if $(g'_1, g'_2) \models \psi$.*

Proof. By surjectivity of op, for any value v in the range of op, there exists some v_1, v_2 such that $op(v_1, v_2) = v$. Define functions $l(v)$ and $r(v)$ that return some such v_1 and v_2 for every v. Now, set $g'_1 = l \circ f'$ and $g'_2 = r \circ f'$ giving us $op(g'_1(x), g'_2(x)) = f'(x)$. Now, since ψ is a valid subspecification, we get that $op(g'_1, g'_2) \models \varphi \Leftrightarrow (g'_1, g'_2) \models \psi$. Equivalently, $f' \models \varphi$ if and only if $(g'_1, g'_2) \models \psi$. □

B USER STUDY TASKS

B.1 Screening Quiz

(1) Given the following specification:

$$\forall x, y, f(x, y) \geq x \wedge f(x, y) \geq y,$$

and the corresponding implementation,

$$f(x, y) = \texttt{if}\ x \geq y\ \texttt{then}\ x + 2\ \texttt{else}\ y + 3,$$

which of the following alternative implementations also satisfy the specification?

(a) $f'(x, y) = \texttt{if}\ x \geq y\ \texttt{then}\ \underbrace{x - 2}\ \texttt{else}\ y$, and

(b) $f''(x, y) = \texttt{if}\ x \geq y\ \texttt{then}\ \underbrace{x + 1}\ \texttt{else}\ y$

(2) Consider the following implementation:

$$g(a, b) = \texttt{if}\ a \geq b\ \texttt{then}\ a - b\ \texttt{else}\ \underbrace{b - a}_{h},$$

where the subspecification for the highlighted hole is:

$$\forall a, b, a \geq b \implies h(a, b) \geq 0.$$

Which of the following implementations also satisfy the specification?

(a) $g'(a, b) = \texttt{if}\ a \geq b\ \texttt{then}\ a - b\ \texttt{else}\ +8$, and

(b) $g''(a, b) = \texttt{if}\ a \geq b\ \texttt{then}\ a - b\ \texttt{else}\ -11$.

Note that we withheld the specification from users, and that they had to answer this question purely by consulting the provided subspecifications.

B.2 Task 1

The specification and implementation for the first task is identical to that discussed in Example **??**. Here is the specification:

$$\begin{array}{lll} \forall x, y, i, j, & 0 \leq f(x, y) \leq 2 & \\ \wedge & f(x, y) = 0 & \implies x = y \\ \wedge & f(x, y) = 0 \wedge 1 \leq i, j \leq 2 & \implies |x - y| \leq (x - y)(j - i) \vee f(x + i, y + j) = 0 \\ \wedge & f(x, y) \neq 0 & \implies f(x + f(x, y), y + f(x, y)) \neq 0 \wedge |x - y| > 0. \end{array}$$

And here is the implementation:

$$f(x, y) = \texttt{if}\ x = y\ \texttt{then}\ 0\ \texttt{else}\ 1.$$

B.3 Task 2

The specification and implementation for the second task elaborated on that considered in Example **??**. Here is the specification:

$$\begin{array}{lllllllllllllll} f(11, 45) = 1 & \wedge & f(1, 26) = 1 & \wedge & f(39, 29) = 1 & \wedge & f(1, 36) = 1 & \wedge & f(15, 12) = 1 & \wedge \\ f(37, 1) = 1 & \wedge & f(40, 29) = 1 & \wedge & f(1, 33) = 1 & \wedge & f(34, 39) = 1 & \wedge & f(10, 1) = 1 & \wedge \\ f(5, 7) = 1 & \wedge & f(1, 4) = 1 & \wedge & f(7, 21) = 1 & \wedge & f(26, 1) = 1 & \wedge & f(50, 33) = 1 & \wedge \\ f(50, 1) = 3 & \wedge & f(13, 10) = 3 & \wedge & f(1, 39) = 3 & \wedge & f(46, 18) = 3 & \wedge & f(7, 1) = 1, & \end{array}$$

and here is the implementation:

$$f(x, y) = \texttt{if}\ x \leq 1\ \texttt{then}\ 3x\ \texttt{else}\ \texttt{if}\ y \leq 1\ \texttt{then}\ 3y\ \texttt{else}\ 1.$$

B.4 Task 3

In the third task we presented the following specification:

$$\forall x, f(x) \geq 0,$$

and the following (buggy) implementation:

$$f(x) = \texttt{if}\ x \geq 0\ \texttt{then}\ x\ \texttt{else}\ -1.$$

B.5 Task 4

In the final task, we presented the following specification:

$$\begin{aligned} &f(\texttt{"Ducati100"}) = \texttt{"Ducati"} \wedge f(\texttt{"Honda125"}) = \texttt{"Honda"} \wedge \\ &f(\texttt{"Ducati125"}) = \texttt{"Ducati"} \wedge f(\texttt{"Honda250"}) = \texttt{"Honda"} \wedge \\ &f(\texttt{"Ducati250"}) = \texttt{"Ducati"} \wedge f(\texttt{"Honda550"}) = \texttt{"Honda"}. \end{aligned}$$

We then presented two potential implementations:

$$f(x) = \texttt{substr}(x, 0, 5) + \texttt{strat}(\texttt{substr}(x, 5, 4), 0), \text{ and} \tag{16}$$

$$f(x) = \texttt{substr}(x, 0, \texttt{len}(\texttt{substr}(x, 4, 5))) + \texttt{strat}(x, \texttt{len}(\texttt{substr}(x, 4, 5))). \tag{17}$$

The first implementation is buggy because the substring extraction function, $\texttt{substr}(w, i, l)$, returns the string of greatest possible length when fewer than l characters are available starting from index i. We also presented the descriptions of the `substr` and `strat` methods from the SMT-LIB theory reference.