Explainable Program Synthesis by Localizing Specifications

AMIRMOHAMMAD 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 algorithms to construct and verify subspecifications and investigate their theoretical properties. We perform an experimental evaluation of the subspecification generation procedure, and measure 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.

CCS Concepts: • **Software and its engineering → General programming languages**; **Automatic programming**; • **Theory of computation → Semantics and reasoning**.

Additional Key Words and Phrases: Program synthesis, program comprehension, explainability

ACM Reference Format:
Amirmohammad Nazari, Yifei Huang, Roopsha Samanta, Arjun Radhakrishna, and Mukund Raghothaman. 2023. Explainable Program Synthesis by Localizing Specifications. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 298 (October 2023), 25 pages. https://doi.org/10.1145/3622874

1 INTRODUCTION

Program synthesis technology has made tremendous advances over the past two decades [Alur et al. 2013a, 2017; Feng et al. 2018; Jha et al. 2010; Reynolds et al. 2015; Solar-Lezama et al. 2006]. It has been applied to diverse domains, including end-user programming [Gulwani 2011; Le and Gulwani 2014; Singh 2016], data science [Wang et al. 2021a], networking [Shi et al. 2021], robotics [Feniello et al. 2014] and programmer assistance tools [Alur et al. 2013b; Singh et al. 2013].

*Both authors contributed equally to this paper.

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

2475-1421/2023/10-ART298
https://doi.org/10.1145/3622874

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, as has been long recognized by researchers in model checking [Könighofer et al. 2009; Kupferman and Vardi 2003]. Even relatively simple specification mechanisms such as programming-by-example (PBE) are subject to omission, user error and noise [Handa and Rinard 2020]. Finally, 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 [Mayer et al. 2015; Peleg et al. 2018; Tiwari et al. 2020; Zhang et al. 2021, 2020]. 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 [Gotel and Finkelstein 1994]. 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 1 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 subspecs $g_1(1,0) = 2$ and $g_2(1,2) = 3 \wedge g_2(2,3) = 5$ respectively—have independent subspecifications and can therefore be separately manipulated without interfering with global correctness.

Having introduced the concept of subspecifications, we turn our attention to investigating their theoretical properties. First, we present an algorithm to obtain a simple subspec from a specification, implementation and subexpression of interest. This is a two-pass algorithm which initially constructs a "*seed*" subspec and then simplifies this into an optimized representation. In experiments with two synthesis frameworks, SyGuS [Alur et al. 2013a] and DreamCoder [Ellis et al. 2021], our algorithm, which we call S^3, is able to generate small subspecs in reasonable amounts of time: Overall, the size of the resulting subspec is only 74% of the size of the beginning seed subspecifications in over 74% of the SyGuS benchmarks and 59% of the size in 59% of the DreamCoder benchmarks. Furthermore, 78% of the SyGuS and all of the DreamCoder subspecification synthesis tasks can be completed in less than one second, 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, we present an algorithm to verify correctness for the specific case of point-wise specifications, an important category of synthesis problems in which all calls to the target function in the spec are syntactically identical [Alur et al. 2017]. We then investigate the simplifying power of subspecifications in different situations, and identify conditions under which point-wise specs lead to point-wise subspecs and 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 whether subspecifications are helpful to users of program synthesis tools, we conducted a small user study with 20 graduate students. Across six tasks requiring users to assess and manipulate the output of a program synthesizer, we determined that subspecifications lead to a 90% improvement in the accuracy of responses and a 34% reduction in the time needed to arrive at conclusions. In post-study discussions, participants reported that subspecs helped in visualizing the output of the synthesized implementation, and indicated a strong preference for having access to subspecs 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 implement the subspecification generation algorithm for two synthesis frameworks, SyGuS and DreamCoder, and present experiments showing that it is able to efficiently generate small subspecifications for a range of synthesis tasks.

2 FORMALLY DEFINING SUBSPECIFICATIONS

Consider the following 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 [Alur et al. 2017] 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 verify that the synthesized function f_1 satisfies the specification φ. However, the user may find it hard to trust the synthesis process: To be confident in the implementation, they must first be sure that the specification φ they have written accurately captures their intent. 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, $|x - y|$. Mismatches of this kind between specifications and user intentions become more likely as the problem complexity increases.

Our thesis is that requirements on subexpressions (such as Equation 3 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 3.

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 synthesizer 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. Many synthesis frameworks additionally constrain the space of possible implementations, either through grammars [Alur et al. 2013a], components [Jha et al. 2010], or DSLs [Ellis et al. 2021] that may be used to construct well-formed programs. However, because subspecifications are defined and computed as post-hoc explanations for an already synthesized implementation, we will ignore these constraints in the following discussion.

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

We say that a SyGuS specification is *point-wise* if all calls to the function f to be synthesized have syntactically identical arguments [Alur et al. 2017]. 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. The presence of higher-order functions in DreamCoder makes a general definition of a point-wise specification challenging.

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 $h_1, \ldots, h_n$ and expressions $g_1, \ldots, g_n$, 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 2 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_0, 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 $h_1, \ldots, h_n$ by requiring for all $g_1, \ldots, g_n$, $(g_1, \ldots, g_n) \models \psi \iff f_0\ [\forall i, g_i/h_i] \models \varphi$.

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

Example 2.3. Continuing from Example 2.2, the expression $x \neq y \implies g(x,y) \neq g(y,x)$ from Equation 3 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 and debugging the output of program synthesizers. We hope to show the breadth of potential applications and the value of algorithms that can automatically generate subspecs. We adapt our first two examples from Figures 1B and 11B respectively of [Ellis et al. 2021]:

Example 3.1 (Sort). Say the user wishes to synthesize a function f which sorts the list of numbers provided as input. They specify this function using the following input-output example:

$$f(\texttt{[9; 2; 7; 1]}) = \texttt{[1; 2; 7; 9]}. \tag{5}$$

The implementation produced by DreamCoder may be transcribed as follows:

$$\begin{aligned} f(l) &= \texttt{map}\ (\texttt{fun}\, n \to g_{15}(l, 1+n))\ (\texttt{range}(\texttt{len}(l))), \text{ where} \\ g_{15}(l,n) &= g_{13}(\texttt{filter}\ (\texttt{fun}\, z \to n > \texttt{len}\ (\texttt{filter}\ (\texttt{fun}\, u \to z > u)\ l))\ l), \text{ and} \\ g_{13}(l) &= \texttt{hd}\ (\texttt{filter}\ (\texttt{fun}\, y \to \texttt{isnil}\ (\texttt{filter}\ (\texttt{fun}\, z \to z > y)\ l))\ l). \end{aligned}$$

Notice that the implementation is cryptic and the intermediate functions, g_{15} and g_{13} have uninformative names. In fact, in order to supplement their narration, the authors supply these functions with the more informative names n-th smallest element and maximum respectively. As we will observe in Task 5 of our user study, recovering descriptive names from an unannotated codebase is challenging. On the other hand, the global specification, Equation 5, induces the following subspecification for the concept g_{13}:

$$g_{13}(\texttt{[1]}) = 1 \wedge g_{13}(\texttt{[2; 1]}) = 2 \wedge g_{13}(\texttt{[2; 7; 1]}) = 7 \wedge g_{13}(\texttt{[9; 2; 7; 1]}) = 9,$$

which immediately suggests that its purpose is to determine the largest element of the input list.

The reader might wonder whether the user could simply request an explanation from a language model such as ChatGPT [OpenAI 2022] or Codex [Chen et al. 2021]. We note that there are qualitative differences between the two approaches: while LLMs present explanations in natural language, subspecifications are formal objects which potentially lend themselves to further deductive analysis. Furthermore, the lack of guarantees and instability of LLM outputs inhibits their immediate applicability: for example, the online ChatGPT interface to the GPT-4 language model proposed the name `find_smallest_element`, which is the opposite of the purpose revealed by its subspec above.

Example 3.2 (List difference). In our second example, the user seeks a function f which computes the element-wise difference of two lists of numbers supplied as input. They begin with the following example of its behavior:

$$f(\texttt{[10; 11; 8]}, \texttt{[1; 7; 5]}) = \texttt{[9, 4, 3]}. \tag{6}$$

The synthesizer produces the following implementation as a solution:

$$\begin{aligned} f(l_1, l_2) &= g_1(l_1, \texttt{(-)}, l_2), \text{ where} \\ g_1(l_1, \text{op}, l_2) &= \texttt{map}\ (\texttt{fun}\ n \to \text{op}(g_2(n, l_1), g_2(n, l_2)))\ (\texttt{range}(\texttt{len}(l_1))), \text{ and} \\ g_2(n, l) &= \texttt{hd}\ (\texttt{fold}\ (\texttt{fun}\ u, v \to \texttt{tl}(u))\ (\texttt{range}(n))\ l). \end{aligned}$$

As before, the purpose of the intermediate functions g_1 and g_2 is non-obvious. On the other hand, the subspecification for g_2 turns out to be the following:

$$\begin{aligned} g_2(\texttt{0}, \texttt{[10; 11; 8]}) - g_2(\texttt{0}, \texttt{[1; 7; 5]}) &= 9 \wedge \\ g_2(\texttt{1}, \texttt{[10; 11; 8]}) - g_2(\texttt{1}, \texttt{[1; 7; 5]}) &= 4 \wedge \\ g_2(\texttt{2}, \texttt{[10; 11; 8]}) - g_2(\texttt{2}, \texttt{[1; 7; 5]}) &= 3. \end{aligned}$$

This hints that $g_2(n, l)$ computes the n-th element of the list l. Moreover, observe that even though the original specification, Equation 6 was a PBE, the subspecification is not. In particular, the subspecification can be used to identify semantically distinct alternative implementations of g_2, including $g_2'(n, l) = g_2(n, l) + c$, for an arbitrary constant c.

As with the previous example, we once again asked GPT-4 to generate a suitable name for the function g_2. It responded with the suggestion `remove_first_n_chars` which is once again incompatible with its real purpose. Although we are excited by the possibilities of LLMs in program synthesis and associated tasks, their responses remain unreliable in our experience, especially when dealing with unidiomatic code.

Example 3.3. Consider the following specification φ_1 from the 2018 SyGuS competition:[2]

$$\begin{array}{lll} & 0 \le f(x,y) \le 2 & \\ \wedge & f(x,y) = 0 & \implies x = y \\ \wedge & f(x,y) = 0 \wedge 1 \le i,j \le 2 & \implies |x-y| \le (x-y)(j-i) \vee f(x+i,y+j) = 0 \\ \wedge & f(x,y) \neq 0 & \implies f(x+f(x,y), y+f(x,y)) \neq 0 \wedge |x-y| > 0. \end{array} \tag{7}$$

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:

$$f(x,y) = \texttt{if}\ x = y\ \texttt{then}\ \underbrace{0}_{h_1}\ \texttt{else}\ \underbrace{1}_{h_2}. \tag{8}$$

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:

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

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. We empirically verify these claims in Task 1 of the user study in Section 6, 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 3.4 (Traceability). Consider the following PBE specification φ_2, adapted from the problem named `LinExpr_inv1_ex.sl` from the 2017 SyGuS competition.

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

Given this specification, EUSolver responds with the following implementation:

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

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

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

$$\varphi_2|^f_{h_1} \equiv g_1(1,38) = 3 \wedge g_1(1,48) = 3,$$
$$\varphi_2|^f_{h_2} \equiv g_2(26,1) = 3 \wedge g_2(75,1) = 3, \text{ and}$$
$$\varphi_2|^f_{h_3} \equiv g_3(11,4) = 1 \wedge g_3(25,3) = 1 \wedge g_3(7,21) = 1 \wedge g_3(2,38) = 1,$$

respectively. Computing the subspecifications immediately reveals which examples in the PBE problem are handled by which branches of the `if-then-else` expression. This 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 $\varphi_2|^f_{h_1}$ it is easy to see why $3x$ is a valid expression for h_1.

Examining these subspecifications might also indicate to the user which parts of the implementation have insufficient numbers of examples and suggest ways 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 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.

From the examples presented so far, the reader might object that subspecifications merely convey information that developers can already obtain by embedding `print` statements to code. We note however that the main distinction between the two approaches is that `print` statements describe the behavior of the specific implementation under consideration (i.e., the behavior of the code, "*as it is*"), while the subspecification constrains the space of possible alternative implementations. As we argue in the following example, subspecifications describe the code "*as it should be*".

Example 3.5 (Unconstrained subexpressions). We adapt the following PBE specification φ from the problem named `univ_4_short.sl` from the 2017 SyGuS competition:

$$f(\texttt{"NYU"}, \texttt{"New York, New York, USA"}) = \texttt{"New York, NY, USA"} \wedge \\ f(\texttt{"UCLA"}, \texttt{"Los Angeles, CA"}) = \texttt{"Los Angeles, CA, USA"}. \tag{9}$$

Given this specification, CVC5 synthesizes the following implementation:

$$f(w_1, w_2) = \texttt{if endswith}(w_2, \texttt{"USA"})\ \texttt{then} \cdots \texttt{else}\ (\texttt{replace}(w_2, \texttt{"New York"}, \underbrace{\texttt{"PA"}}_{h}) + \texttt{", USA"}).$$

Anecdotally, synthesizers such as EUSolver and CVC5 are prone to producing unusual fragments such as $\texttt{replace}(w_2, \texttt{"New York"}, \texttt{"PA"})$ in the implementation above. Our subspecification synthesis algorithm concludes that the subspecification for location marked h is `true`. In other words, h can be replaced by any other well-formed expression of the same type and is otherwise unconstrained. Furthermore, note that this is true even though the subexpression at h is executed upon evaluating the second input-output example.

Upon further investigation of the code, one notices that the input examples that reach h do not contain `"New York"` as a substring. As a result, the replace statement has no effect on the input

string and changing the replacement string h to any other expression would not alter the global behavior on the input-output examples.

Finally, observe that while subspecifications immediately draw attention to such anomalous subexpressions, traditional debugging techniques such as printing the values of subexpressions do not transparently reveal these phenomena. In any case, as we will remark in Section 6.2, participants in our user study were hesitant to rely on `print` statements and other experimental observations while making judgments about the behavior of code.

Example 3.6 (Debugging synthesizers). We draw our final example, φ_3, from the PBE SLIA track of the 2017 SyGuS competition:[3]

$$\begin{aligned} &f(\texttt{"Ducati100"}) = \texttt{"Ducati"} \wedge f(\texttt{"Honda125"}) = \texttt{"Honda"} \wedge \\ &f(\texttt{"Ducati125"}) = \texttt{"Ducati"} \wedge f(\texttt{"Honda250"}) = \texttt{"Honda"} \wedge \\ &f(\texttt{"Ducati250"}) = \texttt{"Ducati"} \wedge f(\texttt{"Honda550"}) = \texttt{"Honda"}. \end{aligned} \tag{10}$$

EUSolver solves this problem with the following implementation:

$$f(x) = \underbrace{\texttt{substr}(x, 0, 5)}_{h_1} + \underbrace{\texttt{strat}(\texttt{substr}(x, 5, 4), 0)}_{h_2}. \tag{11}$$

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 11, which results in $\varphi_3|^f_{h_1} = \texttt{false}$ and the following formula, $\varphi_3|^f_{h_2}$, respectively:

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

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. This is the one of the few implementations produced by EUSolver for which we encounter the inconsistent subspecification `false`.

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.

4 ALGORITHMIC SYNTHESIS OF SUBSPECIFICATIONS

We note that our definition of subspecifications in Section 2 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 *seed subspec*.

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

4.1 The Seed Subspecification

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

$$f(x, y) \geq x \wedge f(x, y) \geq y. \tag{12}$$

Note that the specification is satisfied by the function that returns the larger of its two inputs:

$$f(x, y) = \texttt{if}\ x \geq y\ \texttt{then}\ \underbrace{x}_{h}\ \texttt{else}\ y. \tag{13}$$

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 12:

$$\begin{array}{l}(\texttt{if}\ x \geq y\ \texttt{then}\ g(x, y)\ \texttt{else}\ y) \geq x \wedge \\ (\texttt{if}\ x \geq y\ \texttt{then}\ g(x, y)\ \texttt{else}\ y) \geq y.\end{array} \tag{14}$$

Equation 14 is a subspecification of h under the global specification φ_4.

Observe that this technique is quite general: We begin by replacing the subexpression at the site of the hole h in the implementation f with a call to a fresh uninterpreted function g. The return type of g would be the same as the type of the subexpression currently residing at h. In addition, the newly inserted call to g would have the same arguments as the formal parameters of f. We can express this construction as:

$$\text{seed}(\varphi, f, h) = \lambda g.\varphi(f\ [g/h], \boldsymbol{x}). \tag{15}$$

Note that $\text{seed}(\varphi, f, h)$ is itself a specification of g and contains the same free variables $\boldsymbol{x}$ as φ. From construction, it follows that:

Lemma 4.1. *For all global specifications $\varphi(f, \boldsymbol{x})$, implementations f_0 and program locations h, $\text{seed}(\varphi, f_0, h)$ is the subspecification of h under φ.*

We call this formula the *seed 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 7, we observe that the seed subspec is, on average, 5× the combined size of the specification and implementation. In these situations, the seed subspec constructed in Equation 15 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 begins with the seed subspecification and attempts to simplify it into a more compact representation. The main challenge in this process is the presence of the uninterpreted synthesis function $g(x, y)$, whose second-order quantifier makes it difficult directly simplify. Our key insight is to replace calls to the synthesis function $g(x, y)$ in specifications such as Equation 14 with a fresh first-order logical variable t, resulting in the Boolean-valued indicator function:

$$\begin{array}{rl}\text{ind}_{\varphi_4}(x, y, t) = & (\texttt{if}\ x \geq y\ \texttt{then}\ t\ \texttt{else}\ y) \geq x \wedge \\ & (\texttt{if}\ x \geq y\ \texttt{then}\ t\ \texttt{else}\ y) \geq y.\end{array} \tag{16}$$

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:

$$\begin{aligned} t = \{ & f(x) \mapsto t_{f(x)}, f(y) \mapsto t_{f(y)}, \dots, \\ & f(0) \mapsto t_{f(0)}, f(1) \mapsto t_{f(1)}, \dots, \\ & f(x+y) \mapsto t_{f(x+y)}, f(f(x)) \mapsto t_{f(f(x))}, \dots \} \end{aligned} \tag{17}$$

from *all syntactically distinct* calls to the synthesis target $f(\cdots)$ to their corresponding test variables $t_{f(\cdots)}$. 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 $t_{f(\cdots)}$. 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 $t_{f(f(x))} > t_{f(x)}$.

Note that we fix the map $t_\bullet$ from syntactically different function calls to their corresponding test variables globally across *all* specifications. Hence, one should conceptualize the indicator function as taking the infinitely long vector of) valuations of all test variables as input, accessing the (necessarily finite) subset of relevant test variables, and discarding the remaining useless inputs. For example, consider the specifications $\varphi(f, x) = f(x) \geq x$ and $\psi(f, x) = f(f(x)) \geq x$. They would result in the indicator functions:

$$\begin{aligned} &\text{ind}_\varphi(x, t_{f(x)}, \dots, t_{f(0)}, t_{f(1)}, \dots, t_{f(f(x))}, \dots) = t_{f(x)} \geq x, \text{ and} \\ &\text{ind}_\psi(x, t_{f(x)}, \dots, t_{f(0)}, t_{f(1)}, \dots, t_{f(f(x))}, \dots) = t_{f(f(x))} \geq x \end{aligned}$$

respectively. Because the mapping into test variables $t_\bullet$ defines a bijection, it is possible to exactly recover the function specification φ from its indicator representation ind_φ.

Lemma 4.2 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 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 1. We start with the seed subspecification and apply a sequence of simplification passes to the corresponding indicator function. We recover the final subspecification from this optimized representation. In the rest of this section, we focus on the simplification process for implementations obtained from SyGuS solvers, which we describe in Algorithm 1. In contrast, the subspecification synthesis procedure for DreamCoder implementations simplifies the indicator function using a set of rewrite rules. We briefly describe this procedure in Appendix A.

It can be proved by induction on e that $\textsc{Simplify}(e, \pi)$ is identically equal to e on all inputs that satisfy the predicate π. As a straightforward consequence of Lemma 4.2, we then 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 4.3. First, instead of requiring global equality between the original and simplified indicator functions,

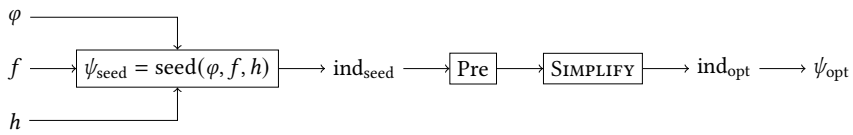


Fig. 1. The subspecification synthesis process using our system S^3. We first construct the seed subspecification using Equation 15, and simplify its indicator representation, ind_{seed} using Algorithm 1 to obtain an optimized indicator ind_{opt}. From this optimized indicator, we recover the final output subspecification.

Algorithm 1 $\textsc{Simplify}(e, \pi)$. 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:

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

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

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

(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'.

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. We describe the preprocessing passes in Appendix A.

Relaxing global indicator equality. Note that even though Lemma 4.2 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:

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

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 4.2 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, $\mathrm{FC}_\varphi \equiv x+y = x+z \implies t_{f(x+y)} = t_{f(x+z)}$. 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 1.

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 2. This is in general a challenging problem as the converse of Lemma 4.2 does not hold, and indicator functions can no longer be used to prove correctness. Surprisingly, 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, a conforming implementation f, and a hole h in f. Let the seed subspecification $\psi(h, \boldsymbol{x}) = \mathrm{seed}(\varphi, f, h)$ 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 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 seed subspec and the candidate subspec are pointwise, it suffices to merely check whether the corresponding indicator functions are globally equal.

The following result is a straightforward consequence of Lemma 4.4 and establishes the optimality of our subspecification synthesis algorithm under mild assumptions:

Corollary 4.5. *If the specification φ is pointwise and assuming the SyGuS solver in Step 5 always returns the syntactically smallest solution, then Algorithm 1 also produces the syntactically smallest pointwise subspecification.*

5 PROPERTIES OF SUBSPECIFICATIONS

We now discuss some interesting properties of subexpressions. While subspecifications are general and can be applied to diverse synthesis tools, in this section, we restrict our discussion to domains that do not contain higher-order functions and forbid recursive function definitions.

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) tasks. These specifications are syntactically simple, easy to comprehend, and are amenable to more efficient synthesis algorithms. We would therefore like subspecifications arising from these specifications to be of the same class. We now 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 seed subspecification from Section 4—the seed 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 3.3 from Section 3 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_0, there exists a point-wise specification $\varphi|_h^{f_0}$ that is a valid subspecification for h in f_0.*

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 $f(x) = \texttt{substr}(\underbrace{x}_{h}, 0, (\texttt{indexof}(x, \texttt{" "}, 0)))$. 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. This leads to a loose subspecification. On the other hand, in Example 3.6 in Section 3, 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 be 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 2. 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_1 and h_2 are given by $\varphi|^f_{h_1} \equiv x \geq y \implies g_1(x,y) = x$ 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 5.3 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 B.

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 main goal of the above result is to formalize the intuition that understanding all the parts of a program can lead to an understanding of its whole. Its proof may also be found in Appendix B.

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 users in understanding implementations?
RQ2. Do subspecs help in understanding specifications?
RQ3. Can subspecs help users in debugging faulty implementations?

After IRB approval, 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 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 six tasks inspired by the motivating examples discussed in Section 3. The study materials may be found in Appendix C. Before participants attempted these tasks, we presented them with a short introduction to the style of program synthesis used in SyGuS and DreamCoder. 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. 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.

These tasks required participants to identify which implementations satisfied a given specification, to present alternative implementations, to explain the specification and implementation in their own words and to debug broken implementations. We presented Tasks 1, 2, 4, 5, and 6 to the participants in one of two randomly chosen conditions, i.e., with and without access to subspecifications respectively. Each participant attempted at least two tasks with subspecifications and two tasks without access to subspecifications. They were able to access the subspecifications on demand through a simple point-and-click web interface such as that shown in Figure 2. We designed Task 3 to familiarize participants with `false` subspecifications before first encountering them in Task 4. Consequently, all participants had access to subspecifications while attempting Task 3.

We present the time needed by participants to answer these questions in Figure 3a and their accuracy under each of the conditions in Figure 3b. After the study was complete, we had a short discussion with each of the participants and obtained their feedback about which aspects of the study they found easy or difficult, and their experience while using subspecification interface.

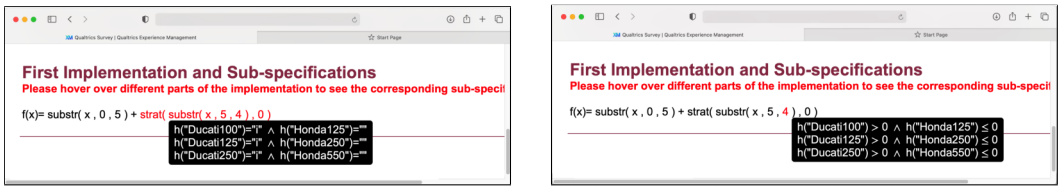


Fig. 2. Interface used by participants to query subspecifications. Upon hovering their mouse pointer over different subexpressions, a tooltip would appear to show the corresponding subspecification.

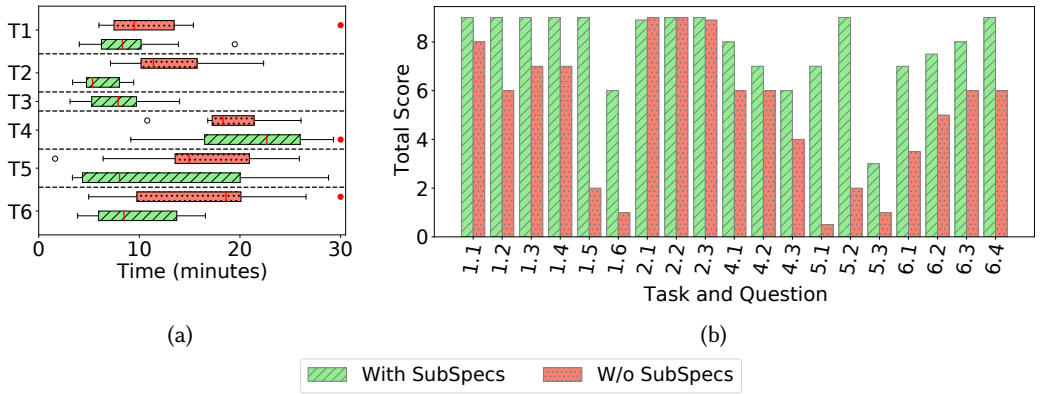


Fig. 3. Time needed by participants to answer questions and their accuracy under each of the conditions. One participant required more than 30 minutes to complete Tasks 1, 4, and 6, which we highlight with the red outlier dot at the right end of Figure 3a. We do not report participant accuracy in Task 3 in Figure 3b because all participants had access to subspecifications. Complete numbers are available in Appendix C.

6.2 RQ1: 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, 2.1–2.3, 5.1–5.3, and 6.1–6.4 of the study materials described in Appendix C. These questions involve assessing the impact of a minor alteration to an existing implementation, some form of requirements tracing, and providing appropriate names to intermediate functions.

From Figure 3b, we observe that subspecifications significantly improve the accuracy of responses: Across Tasks 1 and 2, we recorded only one incorrect response from a participant with access to subspecifications. Notably, this participant realized their mistake soon after completing Task 2. Tasks 5 and 6 were clearly more challenging. Here, subspecifications lead to an $\approx 5.7\times$ improvement in accuracy in Task 5, and measurable accuracy improvements in Task 6.

At the same time, subspecifications reduce the time needed by participants to provide their responses, as we observe in Figure 3a. We see the most dramatic of these improvements in Tasks 2, 5, and 6, where the median response time from participants with subspecifications was only 48%, 46%, and 42% of the time needed by participants from the control group respectively.

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, and had a tendency to guess when they were otherwise unsure of the answer. In particular, for Tasks 5 and 6, participants reported being "*baffled by unnatural implementations*". Of course, one risk in these observations is

that we transcribed DreamCoder output into Python code, which admits more idiomatic implementations for each of the specifications.

We also note that we encouraged participants to use any resources they needed to complete their tasks. In response, only one participant proceeded to test the code in an interpreter. Somewhat amusingly, this participant was also unsuccessful failed to complete the task: We observed that they were unable to determine which part of the code needed instrumentation. They proceeded to exhaustively insert `print` statements into the program, but this led to further confusion when they were unable to interpret the results that they observed. The remaining participants offered varied reasons for not experimenting with the provided implementations: some believed that it was not "*trustworthy*" to understand the logic of implementations, and others found it "*really hard to guess reasonable inputs for the unknown subroutines*". A third class of participants reported not thinking about active experimentation with an interpreter during the study.

As such, participants in the control group responded with humour and mentioned that they wouldn't use the synthesized code, while participants with access to subspecs appeared to better understand the presented code.

6.3 RQ2: Understanding Specifications

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 RQ3: 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 3.6. In order to familiarize participants with subspecifications for buggy implementations, we first presented them with a relatively simple specification-implementation pair in Task 3. The specification was $\forall x, f(x) \geq 0$ and the implementation was $f(x) = \texttt{if}\ x \geq 0\ \texttt{then}\ x\ \texttt{else}\ -1$ in Task 3. 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 10 and 11 in Example 3.6. We asked participants to identify which of two implementations was buggy (the other satisfying implementation may be found in Appendix C.5), 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 these difficulties and become more fluent in querying for subspecs on demand.

7 EXPERIMENTAL EVALUATION

We have implemented, S^3, the subspecification synthesis algorithm of Section 4, in approximately 2,700 lines of Python code. Our implementation is able to generate subspecifications for implementations produced both using SyGuS solvers [Alur et al. 2013a] and using DreamCoder [Ellis et al. 2021]. We use CVC5 to simplify the seed subspec for SyGuS implementations and apply a sequence of rewriting rules to simplify the subspec in the case of DreamCoder. Our evaluation focused on the following research questions:

RQ4. How effective is S^3 in generating simple subspecifications?
RQ5. How long does S^3 take to construct subspecs?
RQ6. How do the preprocessing steps of Section 4 affect simplification effectiveness?

Benchmarks. We ran both CVC5 and EUSolver on the benchmarks from the 2017 SyGuS Competition and obtained the corresponding implementations. With a timeout of 5 minutes, the solvers are able to successfully discharge 1,384 and 1,360 benchmarks respectively. Subsequently, we narrowed our focus to benchmarks whose implementations had less than 100 holes. This resulted in

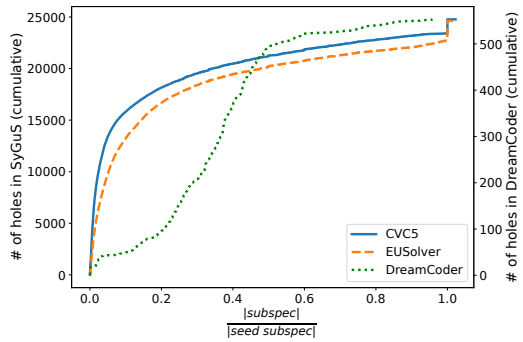


Fig. 4. Effectiveness of S^3 in simplifying subspecifications. We count the number of benchmarks where the algorithm reduces the size of the subspec to the corresponding fraction of the size of the original seed subspec.

1,124 specification-implementation pairs from CVC5 and 1,254 specification-implementation pairs from EUSolver. Collectively, they contained 26,472 and 25,689 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 24,771 and 24,625 holes respectively.

Next, we focused on the reference solutions for the List domain provided as part of the DreamCoder artifact.[4] Of the 550 implementations contained in this file, we excluded 10 implementations because of the tricky semantics of the `mod` operator when applied to negative numbers. We then calculated the subspec for each highlighted library function in these implementations. This resulted in a total of 568 subspecifications.

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.

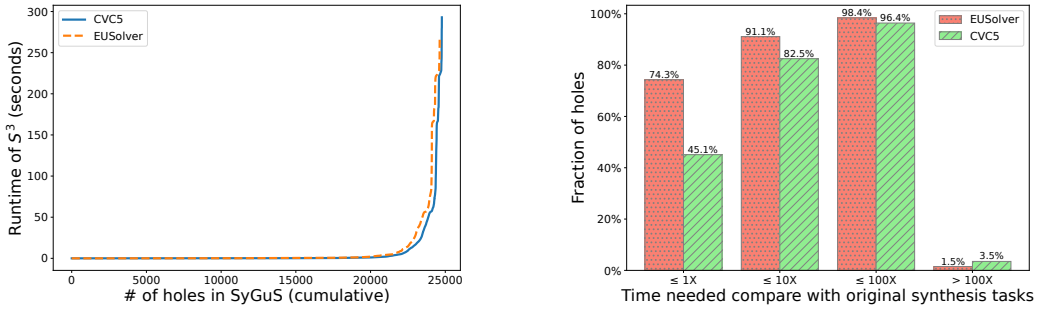
7.1 RQ4: Effectiveness in Simplification

To measure the effectiveness of our technique in deriving simple subspecifications, we compared the size of the synthesized subspecification to the size of the seed subspec. We present these results in Figure 4, with a separate curve for the implementations obtained from CVC5, EUSolver, and DreamCoder 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.

Overall, we observe that S^3 is comparably effective in simplifying subspecs obtained from CVC5 and from EUSolver. For these benchmarks, it achieves a 74% reduction in the size of the seed subspecification for 74% of all holes. On the other hand, it is less effective in achieving high compression rates for subspecs obtained from DreamCoder. In these cases, it only achieves a 59% reduction in the size of the seed subspec for 59% of the holes.

We speculate that the low compression rate is due to higher-order functions. Specifically, when the function at the probe point happens to itself be the parameter of a higher-order operator (such as map or fold), partial evaluation can increase the size of the resulting subspecification. Anecdotally, however, this increase in size does not necessarily cause an increase in perceptual complexity and sometimes even reveals nontrivial properties of the implementation. For example,

[4]File named `jobs/list_hard_test_ellisk_2019-02-15T11.43.28`.



(a) (b)

Fig. 5. (5a) Cactus plot of the time needed by the tool to synthesize subspecs for implementations obtained from the SyGuS solvers. (5b) Comparison of the time needed to produce subspecs to the time originally needed to produce the implementation.

one subspec that resulted from such expansion was $h(h(h(\texttt{[]}, 4), \texttt{0}), 2) = \texttt{[4; 0; 2]}$, suggesting the intermediate function append. Overall, however, the difficulty in obtaining small subspecifications for the DreamCoder implementations highlights the importance of powerful simplification procedures such as those we use in the case of the SyGuS benchmarks.

Finally, we note that our system is able to simplify the subspecification to `true` for 482 holes: by indicating unconstrained subexpressions, S^3 is able to reveal potential opportunities for further optimizing the implementation produced by the synthesizer.

7.2 RQ5: Time Needed to Construct Subspecs

We next measured the time needed by S^3 to construct the optimized subspec. Because of the relatively light-weight rewrite rules used for the DreamCoder implementations, the total time needed to explain all holes is less than 20 seconds. We will therefore only focus on the time needed to explain the SyGuS implementations.

We present a cactus plot of the time needed by S^3 to explain each hole in Figure 5a, and compare these running times to the time needed by CVC5 and EUSolver to originally solve the respective synthesis tasks in Figure 5b. We note that subspecs for 78% of the probe points can be constructed in less than 1 second. Only 13% of the benchmarks exceed 10× of the time it takes for the original synthesis task, and are mainly from PBE tracks involving strings and bit-vectors. Notably, each of these synthesis tasks involve at least 1000 examples. Nevertheless, our system appears to be sufficiently fast to be used to interactively reason about synthesis problem instances.

7.3 RQ6: Effect of Preprocessing Passes

Finally, we assessed the impact of the preprocessing steps in simplifying subspecs for the SyGuS benchmark. We describe this data in Table 1. 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 seed subspec. The more expensive recursive simplification process of Algorithm 1 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, as we also observed in the case of DreamCoder in Figure 4. The most notable of these examples are when the subspecs simplify to `true`, which is typically the result of the main synthesis

Table 1. Impact of preprocessing and main simplification loops in reducing the size of the final subspecification.

Synthesizer	|Preproc| / |Seed|	|Final| / |Preproc|	|Final| / |Seed|
EUSolver	9%	78%	7%
CVC5	5%	77%	4%

process and not the preprocessing pass. Similarly, in the LIA fragment, the main simplification algorithm only contributes a 50% size reduction on average, while the preprocessing pass only results in a 32% drop in subspec size.

8 RELATED WORK

Explainability and interpretability are increasingly important topics in several research areas, including machine learning [Doshi-Velez and Kim 2017], algorithmic transparency [Barocas et al. 2019], and reinforcement learning [Mott et al. 2019]. 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 [Caballero et al. 2017; Zeller 1999], program slicing [Ko and Myers 2004; Weiser 1981], and user-guided program synthesis [Zhang et al. 2021] 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.

Finkbeiner et al. [Finkbeiner et al. 2021] have recently employed a similar idea of decomposing specifications into smaller constituent units and using a divide-and-conquer approach to accelerate reactive synthesis. While our use of the term is very close to theirs, there are two notable differences: 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 [Adadi and Berrada 2018]. Efforts to better understand the highly expressive and parametric models used for machine learning include model-specific visualization tools [Wang et al. 2021b], model agnostic methods [Ribeiro et al. 2016a], and example-based methods [Aamodt and Plaza 2001]. 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 [Zhang et al. 2021].

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, [Ribeiro et al. 2016b] and [Zhou et al. 2016]). In contrast, our approach focuses more on explaining how parts of the model affect the outcome. Local reasoning has, of course, long been used in modular software verification [Chaki et al. 2003]. More recently, local specifications [Ferdowsifard et al. 2020] 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. Our algorithm relies on simplifying Boolean expressions. Expression simplification has been widely studied in compiler optimization via constant folding, common subexpression elimination, and partial evaluation [Muchnick 1998], with many methods relying on

rewriting techniques such as equality saturation [Willsey et al. 2021]. While our method leverages synthesizers, its performance can potentially be boosted using such rewriting techniques.

Connections to top-down program synthesis. As such, subspecifications are also closely related to propagation of constraints during top-down program synthesis, including in deductive program synthesizers such as Leon [Koukoutos et al. 2016] and SuSLik [Polikarpova and Sergey 2019], and top-down enumerative synthesizers such as Myth [Osera and Zdancewic 2015], λ^2 [Feser et al. 2015] and Synquid [Polikarpova et al. 2016]. In this context, subspecs can be thought of as the specification for the part of the program yet to be enumerated.

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. 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.

ARTIFACT AVAILABILITY STATEMENT

The artifact that supports the findings of this paper can be downloaded from Zenodo [Nazari et al. 2023].

ACKNOWLEDGEMENTS

We thank all the participants in our user study and the anonymous reviewers for immeasurably improving this paper. The research described in this paper was supported by the NSF under grants CCF #2146518, #2124431, and #2107261.

REFERENCES

Agnar Aamodt and Enric Plaza. 2001. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications* 7 (2001), 39–59. https://doi.org/10.3233/AIC-1994-7104

Amina Adadi and Mohammed Berrada. 2018. Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI). *IEEE Access* 6 (2018), 52138–52160. https://doi.org/10.1109/ACCESS.2018.2870052

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

Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. 2013b. Automated Grading of DFA Constructions. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI Press, 1976–1982.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 319–336.

Solon Barocas, Moritz Hardt, and Arvind Narayanan. 2019. *Fairness and Machine Learning: Limitations and Opportunities*. fairmlbook.org. http://www.fairmlbook.org.

Rafael Caballero, Adrián Riesco, and Josep Silva. 2017. A Survey of Algorithmic Debugging. *Comput. Surveys* 50 (08 2017), 1–35. https://doi.org/10.1145/3106740

Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. 2003. Modular Verification of Software Components in C. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 385–395.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang,

Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

Finale Doshi-Velez and Been Kim. 2017. Towards A Rigorous Science of Interpretable Machine Learning. https://doi.org/10.48550/ARXIV.1702.08608 arXiv:1702.08608 [stat.ML]

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua Tenenbaum. 2021. DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 835–850. https://doi.org/10.1145/3453483.3454080

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 420–435. https://doi.org/10.1145/3192366.3192382

Ashley Feniello, Hao Dang, and Stan Birchfield. 2014. Program Synthesis by Examples for Object Repositioning Tasks. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 4428–4435. https://doi.org/10.1109/IROS.2014.6943189

Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST)*. Association for Computing Machinery, 614–626. https://doi.org/10.1145/3379337.3415869

John Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, 229–239. https://doi.org/10.1145/2737924.2737977

Bernd Finkbeiner, Gideon Geier, and Noemi Passing. 2021. Specification Decomposition for Reactive Synthesis. In *NASA Formal Methods*. Springer, 113–130.

Orlena Gotel and Anthony Finkelstein. 1994. An Analysis of the Requirements Traceability Problem. In *Proceedings of IEEE International Conference on Requirements Engineering*. 94–101. https://doi.org/10.1109/ICRE.1994.292398

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 317–330. https://doi.org/10.1145/1926385.1926423

Shivam Handa and Martin Rinard. 2020. Inductive Program Synthesis over Noisy Data. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC / FSE)*. ACM, 87–98.

Susmit Jha, Sumit Gulwani, Sanjit Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*. ACM, 215–224. https://doi.org/10.1145/1806799.1806833

Amy J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 151–158. https://doi.org/10.1145/985692.985712

Robert Könighofer, Georg Hofferek, and Roderick Bloem. 2009. Debugging Formal Specifications Using Simple Counterstrategies. In *2009 Formal Methods in Computer-Aided Design*. 152–159. https://doi.org/10.1109/FMCAD.2009.5351127

Manos Koukoutos, Etienne Kneuss, and Viktor Kuncak. 2016. An Update on Deductive Synthesis and Repair in the Leon Tool. In *Proceedings Fifth Workshop on Synthesis (SYNT@CAV, Vol. 229)*. 100–111. https://doi.org/10.4204/EPTCS.229.9

Orna Kupferman and Moshe Vardi. 2003. Vacuity Detection in Temporal Model Checking. *International Journal on Software Tools for Technology Transfer* 4, 2 (Feb. 2003), 224–233. https://doi.org/10.1007/s100090100062

Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 542–553. https://doi.org/10.1145/2594291.2594333

Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, 291–301. https://doi.org/10.1145/2807442.2807459

Alex Mott, Daniel Zoran, Mike Chrzanowski, Daan Wierstra, and Danilo Rezende. 2019. Towards Interpretable Reinforcement Learning Using Attention Augmented Agents. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS)*. Article 1107, 10 pages.

Steven Muchnick. 1998. *Advanced Compiler Design and Implementation.* Morgan Kaufmann.

Amirmohammad Nazari, Yifei Huang, Roopsha Samanta, Arjun Radhakrishna, and Mukund Raghothaman. 2023. *Explainable Program Synthesis by Localizing Specifications.* https://doi.org/10.5281/zenodo.8331495

OpenAI. 2022. ChatGPT. https://openai.com/blog/chatgpt/.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 619–630. https://doi.org/10.1145/2737924.2738007

Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming Not Only by Example. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 1114–1124. https://doi.org/10.1145/3180155.3180189

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 522–538. https://doi.org/10.1145/2908080.2908093

Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 72 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290385

Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification (CAV)*. Springer, 198–216.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016a. Model-Agnostic Interpretability of Machine Learning. (2016). https://doi.org/10.48550/ARXIV.1606.05386

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016b. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 1135–1144. https://doi.org/10.1145/2939672.2939778

Lei Shi, Yahui Li, Boon Thau Loo, and Rajeev Alur. 2021. Network Traffic Classification by Program Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 430–448.

Rishabh Singh. 2016. BlinkFill: Semi-Supervised Programming by Example for Syntactic String Transformations. *Proceedings of the VLDB Endowment* 9, 10 (June 2016), 816–827. https://doi.org/10.14778/2977797.2977807

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 15–26. https://doi.org/10.1145/2491956.2462195

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 404–415. https://doi.org/10.1145/1168857.1168907

Ashish Tiwari, Arjun Radhakrishna, Sumit Gulwani, and Daniel Perelman. 2020. Information-theoretic User Interaction: Significant Inputs for Program Synthesis. *CoRR* abs/2006.12638 (2020). arXiv:2006.12638 https://arxiv.org/abs/2006.12638

Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy Ko. 2021a. Falx: Synthesis-Powered Visualization Authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI)*. ACM, Article 106, 15 pages. https://doi.org/10.1145/3411764.3445249

Zijie Wang, Robert Turko, Omar Shaikh, Haekyu Park, Nilaksh Das, Fred Hohman, Minsuk Kahng, and Duen Horng Polo Chau. 2021b. CNN Explainer: Learning Convolutional Neural Networks with Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (Feb. 2021), 1396–1406. https://doi.org/10.1109/tvcg.2020.3030418

Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*. IEEE Press, 439–449.

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. Springer, 253–267.

Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena Glassman. 2021. Interpretable Program Synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, 16 pages.

Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST)*. 627–648. https://doi.org/10.1145/3379337.3415900

Bolei Zhou, Aditya Khosla, Àgata Lapedriza, Aude Oliva, and Antonio Torralba. 2016. Learning Deep Features for Discriminative Localization. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, 2921–2929. https://doi.org/10.1109/CVPR.2016.319

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