Reconciling Enumerative and Deductive Program Synthesis[*]

Kangjing Huang
Purdue University
West Lafayette, IN, USA
huang989@purdue.edu

Xiaokang Qiu
Purdue University
West Lafayette, IN, USA
xkqiu@purdue.edu

Peiyuan Shen
Purdue University
West Lafayette, IN, USA
shen290@purdue.edu

Yanjun Wang
Purdue University
West Lafayette, IN, USA
wang3204@purdue.edu

Abstract

Syntax-guided synthesis (SyGuS) aims to find a program satisfying semantic specification as well as user-provided structural hypotheses. There are two main synthesis approaches: enumerative synthesis, which repeatedly enumerates possible candidate programs and checks their correctness, and deductive synthesis, which leverages a symbolic procedure to construct implementations from specifications. Neither approach is strictly better than the other: automated deductive synthesis is usually very efficient but only works for special grammars or applications; enumerative synthesis is very generally applicable but limited in scalability.

In this paper, we propose a cooperative synthesis technique for SyGuS problems with the conditional linear integer arithmetic (*CLIA*) background theory, as a novel integration of the two approaches, combining the best of the two worlds. The technique exploits several novel divide-and-conquer strategies to split a large synthesis problem to smaller subproblems. The subproblems are solved separately and their solutions are combined to form a final solution. The technique integrates two synthesis engines: a pure deductive component that can efficiently solve some problems, and a height-based enumeration algorithm that can handle arbitrary grammar. We implemented the cooperative synthesis technique, and evaluated it on a wide range of benchmarks. Experiments showed that our technique can solve many challenging synthesis problems not possible before, and tends to be more scalable than state-of-the-art synthesis algorithms.

[*]The author list has been sorted according to the alphabetical order; this should not be used to determine the extent of authors' contributions.

PLDI '20, June 15–20, 2020, London, UK
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386027

CCS Concepts: • **Software and its engineering → Automatic programming**; *Formal methods*; • **Theory of computation → Automated reasoning**.

Keywords: syntax-guided synthesis, divide-and-conquer, enumerative synthesis, deductive synthesis

ACM Reference Format:
Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling Enumerative and Deductive Program Synthesis. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3386027

1 Introduction

Syntax-guided synthesis (SyGuS) is a common theme underlying many program synthesis systems. The insight behind SyGuS is that to synthesize a large-scale program automatically, the user needs to provide not only a semantic specification but also a syntactic specification, i.e., a grammar of candidate programs as the search space. SyGuS has seen great success in the last decade, including the Sketch [39, 41, 43] synthesizer and the FlashFill feature of Microsoft Excel [20, 21]. The research community has also developed a standard interchange format for SyGuS problems and organized an annual competition, which encourages a plethora of syntax-guided synthesizers [1, 7].

The community usually categorizes synthesis techniques into two classes[1]: *enumerative synthesis* — which systematically enumerates possible implementations of the function to be synthesized and checks if it satisfies the desired specification; and *deductive synthesis* — which tries to reduce the specification to a desired program, purely symbolically by applying a series of deductive rules. Neither strategy clearly outperforms the other.

[1]E.g., see Sec 1.1 of [23], Lectures 2 and 17 of [42], and Table 2 of [19].

Enumerative synthesis traverses the search space following a specific strategy. The simplest strategy begins the search from smaller-sized candidates and moves toward larger-sized candidates. This naïve strategy guarantees to produce the smallest possible program, and is proven efficient for a wide spectrum of syntax-guided synthesis tasks. For example, EUSolver [6] adopts this strategy and has been the winner of the general track in 2016 and 2017 SyGuS competition [4, 5]. Other search strategies may perform better for different classes of problems. Stimulated by earlier success stories and the community's effort of standardization and competition [1, 7], researchers have proposed many novel search strategies, including abstraction-based [13, 15, 24, 25, 31, 48], stochastic enumeration [37, 38], constraint-based [40, 44] and learning-based [8, 29]. Note that the appealing programming-by-example (PBE) and counterexample-guided inductive synthesis (CEGIS) techniques can also be viewed as a class of enumerative synthesis: the synthesizer is given a set of input-output examples and the search will be restricted to the programs whose behavior matches the given examples. As an example, LoopInvGen [30] leverages a learning-based variant of the CEGIS framework and won the invariant track in 2017 and 2018 SyGuS competition [3, 5]. Despite these algorithmic innovation, enumerative search is difficult to scale to large programs, because the search space grows exponentially with the size of the program.

Deductive synthesis is the oldest form of synthesis, dating back to Manna and Waldinger's field-defining paper [28] and even earlier work of Burstall and Darlington [9]. The deduction process essentially accepts a specification S and builds a constructive proof for the theorem "there exists a program satisfying S." Representative examples include Spiral [34], Paraglide [47], Fiat [12] and SusLik [32]. In general, the commonly known challenge for this paradigm of synthesis is the degree of automation, because critical steps of rule applications for synthesizing sophisticated programs, e.g., those involving loops, still rely on some guidance from the user. In recent years, researchers have automated deductive synthesis to symbolic synthesis procedures for several classes of synthesis problems, which run very efficiently. For example, CVC4 [35] embodies a symbolic synthesis algorithm called CEGQI to handle a class of integer arithmetic synthesis problems with the so called single invocation properties, and won the CLIA track of SyGuS competition four years in a row [7]. However, these procedures usually focus on synthesis problems in special domains with fixed grammars, and not applicable to more general synthesis tasks with arbitrary, user-provided grammars.

In recent years, the community has recognized the power of combining enumeration and deduction for synthesis [14–16, 33]. Nonetheless, existing techniques are not directly applicable to SyGuS for arbitrary grammars. In this paper, we focus on the class of SyGuS problems with the *CLIA* background theory but arbitrary grammar, and seek novel and amenable synergies of enumeration and deduction. We present a *cooperative synthesis* technique which switches between the two synthesis strategies to push the scalability. We develop several divide-and-conquer strategies to split a large synthesis problem to smaller subproblems. The subproblems are solved separately and their solutions are combined to form a final solution. The technique integrates two synthesis engines: a pure deductive component for efficiently solving/simplifying the current problem whenever possible, and a height-based enumeration algorithm, as the last resort for handling arbitrary problem instances.

In this paper, we show our technique performs better than existing algorithms, and successfully solves many challenging problems not possible before. We summarize the contributions of this paper as below:

1. a **cooperative synthesis framework** that splits a synthesis problem into subproblems which are solved by deduction or enumeration separately (Section 3);
2. three novel **divide-and-conquer strategies** which allow splitting a wide variety of sophisticated synthesis problems (Section 4);
3. a **height-based enumeration** algorithm that splits the search space based on the height of the tree representation of the program and searches for each height symbolically (Section 5);
4. a set of general **deductive rules** that are powerful enough to solve/simplify many synthesis problems (Section 6);
5. the cooperative synthesis technique has been embodied in a SyGuS **solver** called DryadSynth, which solved more benchmarks than state-of-the-art solvers in every class of benchmarks, and tended to be more scalable for sophisticated benchmarks. 58 out of 715 benchmarks were solved uniquely by DryadSynth (Section 7).

2 Preliminaries

2.1 Syntax-Guided Synthesis

Definition 2.1 (Language)**.** A language $\mathcal{L}$ is a tuple (Σ, τ) where Σ is an alphabet and τ maps every n-ary function name $f \in \Sigma$ to its signature $\tau(f) \in \{Bool, U\}^{n+1}$ (where U represents a universe). An $\mathcal{L}$-expression is an expression over symbols from Σ that conforms to their signatures τ. An $\mathcal{L}$-term is an $\mathcal{L}$-expression of type U. An $\mathcal{L}$-formula is a boolean $\mathcal{L}$-expression.

Definition 2.2 (Background Theory)**.** A decidable background theory $\mathcal{T}$ with respect to a language $\mathcal{L}$ is a set of $\mathcal{L}$-formulae such that there is a decision procedure that takes a quantifier-free Σ-formula $\varphi(\boldsymbol{x})$ as input, and determines if $\mathcal{T} \models \varphi$, generates a counterexample vector $\boldsymbol{C}$ such that $\mathcal{T} \not\models \varphi(\boldsymbol{C})$, if such an $\boldsymbol{C}$ exists.

Example 2.3. Consider a language $(\{0, 1, +, -, \geq, \texttt{ite}\}, \tau)$, where $\tau(0) = \tau(1) = (\mathbb{Z})$ as both 0 and 1 are constants, $\tau(+) = \tau(-) = (U, U, U)$ as they are binary functions over U, $\tau(\geq) = (U, U, \mathit{Bool})$ as $\geq$ is a binary relation. And finally, ite represents the *if-then-else* combination of a formula and two terms; therefore $\tau(\texttt{ite}) = (\mathit{Bool}, U, U, U)$. Then we let *CLIA* denote the standard theory for this language interpreted over $\mathbb{Z}$.

Definition 2.4 (Interpreted Function)**.** An interpreted function for a language $\mathcal{L}$ is a tuple $(f, \Phi(x_1, \dots, x_n))$, where f is the function name, and $\Phi(x_1, \dots, x_n)$ is a well-typed $\mathcal{L}$-expression, i.e., each x_i is of type U or *Bool*, and the whole expression can be typed U or *Bool*.

Example 2.5. Consider a binary function *qm* in the *CLIA* theory that returns the first non-negative argument. We declare this interpreted function as $(qm, \texttt{ite}(x_1 < 0), x_2, x_1)$.

Now we define the expression grammar, which essentially describes syntactic constraints for the expected program using a context-free grammar.

Definition 2.6 (Expression Grammar)**.** An expression grammar $\mathcal{G}$ is a tuple $(\mathcal{T}, \mathcal{R}, \mathcal{N}, S, \mathcal{P})$, where $\mathcal{T}$ is a background theory with alphabet Σ, $\mathcal{R}$ is a set of interpreted functions for $\mathcal{L}$, $\mathcal{N}$ a set of non-terminal symbols (to be typed *Bool* or U, denoted as $\mathcal{N}_b$ and $\mathcal{N}_u$), $S \in \mathcal{N}$ is the start symbol, and $\mathcal{P} \subseteq \mathcal{N} \times \text{Exprs}(\Sigma, \mathcal{R}, \mathcal{N})$ is a set of production rules of form $T \to \epsilon$ or $T \to r(a_1, \dots, a_n)$, where $T \in \mathcal{N}$, $r \in \mathcal{R} \cup \Sigma$, a_i is a free variable or a non-terminal in $\mathcal{N}$. Let $[\![\mathcal{G}]\!]$ denote the set of all expressions generated by $\mathcal{G}$: $\{e \mid S \underset{\mathcal{P}}{\longrightarrow}^* e\}$.

Example 2.7. Consider all possible expressions built using the *qm* function defined in Example 2.5, as well as variables x, y, z, and arbitrary constants. We call these expressions *qm-normal form* (*QNF*). Then formally *QNF* can be defined as an expression grammar

$$\mathcal{G}_{qm} \overset{\text{def}}{=} (\mathit{CLIA}, \{qm\}, \{S\}, S, \{x, y, z\}, \mathcal{P})$$

where $\mathcal{P}$ is the set of production rules presented in Figure 1a.

Example 2.8. We define $\mathcal{G}_{\mathit{CLIA}}$ as a special grammar. It takes *CLIA* as the background theory and allows all standard *CLIA* expressions.

Definition 2.9 (Uninterpreted Function)**.** An n-ary uninterpreted function f is a sequence $\big((x_1, t_1), \dots, (x_n, t_1), rt\big)$ where every pair (x_i, t_i) represents that the name of the i-th argument is x_i with type $t_i \in \{\mathit{Bool}, U\}$, and $rt \in \{\mathit{Bool}, U\}$ is the return type of the function.

Example 2.10. To synthesize a ternary function *max3*, we declare it as an uninterpreted function $((x, U), (y, U), (z, U), U)$.

Now we are ready to define the syntax-guided synthesis (SyGuS) problem we address in this paper.

$$\begin{array}{lcl} S & \to & 0 \mid 1 \mid S + S \mid S - S \\ S & \to & qm(S, S) \end{array}$$

(a) Grammar $\mathcal{G}_{qm}$

$$\begin{array}{lcl} S' & \to & 0 \mid 1 \mid S' + S' \mid S' - S' \\ S' & \to & qm(S', S') \mid aux(S', S') \end{array}$$

(b) Grammar $\mathcal{G}^+_{qm}$

$$\begin{array}{rcl} qm(x_1, x_2) & \overset{\text{def}}{=} & \texttt{ite}(x_1 < 0, x_2, x_1) \\ aux(x_1, x_2) & \overset{\text{def}}{=} & \texttt{ite}(x_1 \geq x_2, x_1, x_2) \end{array}$$

(c) Interpreted functions

Figure 1. Production rules for Examples 2.7 and 3.2.

Definition 2.11 (SyGuS Problem)**.** An instance of the SyGuS problem is given by a tuple $(\mathcal{T}, f, \Phi, \mathcal{G})$ where $\mathcal{T}$ is a background theory with alphabet Σ, f is an n-ary uninterpreted function to be synthesized, Φ is a formula over $\Sigma \cup \{f\}$, and $\mathcal{G} = (\mathcal{T}, \mathcal{R}, \mathcal{N}, \{x_1, \dots, x_n\}, \mathcal{P})$ is an expression grammar. A solution to the SyGuS problem is an expression $E \equiv \lambda x_1, \dots, x_n.e(x_1, \dots, x_n)$ such that: a) $e(x_1, \dots, x_n) \in [\![\mathcal{G}]\!]$; b) $\Phi[E/f]$ is valid, i.e., instantiating f with E makes Φ valid. We use $(\mathcal{T}, f, \Phi, \mathcal{G}) \rightsquigarrow E$ to denote that E is a solution to the SyGuS problem $(\mathcal{T}, f, \Phi, \mathcal{G})$.

Example 2.12. Recall the *max3* function declared in Example 2.10. Now we want to find an implementation of *max3* in $\mathcal{G}_{qm}$ that matches the semantics of the authentic implementation in *CLIA* . This is a SyGuS problem $(\mathit{CLIA}, \mathit{max3}, \Phi, \mathcal{G}_{qm})$ where Φ is the specification for the synthesis task:

$$\mathit{max3}(x, y, z) = \texttt{ite}(x \geq y \wedge x \geq z, x, \texttt{ite}(y \geq z, y, z)) \quad (2.1)$$

One solution to this problem is the following expression

$$\mathit{max3\_sol} \overset{\text{def}}{=} \lambda x, y, z.\big(z + qm(x - z + qm(y - x, 0), 0)\big) \quad (2.2)$$

Remark: To simplify the presentation, we assume there is a single function to be synthesized. However, the SyGuS definition can be easily extended to synthesize multiple functions. In the rest of the paper, we omit the background theory $\mathcal{T}$ when it is *CLIA* from the context. Unless stated otherwise, we also assume the function to be synthesized is always f. Then these components can be omitted from the tuples.

2.2 Counterexample-Guided Inductive Synthesis

The SyGuS problem is typically very challenging. Let $(\mathcal{T}, f, \Phi, \mathcal{G})$ be a SyGuS problem. Note that the specification Φ involves a vector of variables $\boldsymbol{x}$, and the synthesizer needs to find an implementation of f such that $\forall x.\Phi(\boldsymbol{x})$ holds. Checking this quantified formula is already undecidable for most background theories.

A common approach to addressing this problem is the Counterexample Guided Inductive Synthesis (CEGIS) framework [40, 43]. The basic idea is that a set of representative value assignments C is usually sufficient to find a solution that works for all inputs. So the synthesis problem can be reduced to a constraint of the following form:

$$\exists f.\ \bigwedge_{\boldsymbol{c} \in C} \Phi(\boldsymbol{c}) \tag{2.3}$$

The set C is usually initialized to contain a random value. The synthesizer tries to solve (2.3) and find a candidate expression q. Then a verifier can check if the candidate works for all inputs, i.e.,

$$\mathcal{T} \models \Phi[q/f](\boldsymbol{x}) \tag{2.4}$$

Note that this query can be solved by the background decision procedure. If true, then q is a valid solution and the algorithm terminates; otherwise, a counterexample can be found, and the algorithm continues by adding the counterexample to C and the synthesizer tries to solve the inductive constraint again. The loop repeats until it finds a valid solution or hits a timeout.

2.3 Invariant Synthesis

In this paper we also address invariant synthesis, a special class of synthesis problems.

Definition 2.13 (Invariant synthesis problem). An invariant synthesis problem can be represented as $\exists inv \forall \boldsymbol{x} \varphi(inv; \boldsymbol{x})$, where inv is the predicate to be synthesized and $\varphi(inv; \boldsymbol{x})$ is of the form

$$\begin{aligned}\varphi(inv; \boldsymbol{x}) \equiv \Big(pre(\boldsymbol{x}) \to inv(\boldsymbol{x})\Big) \wedge \Big(inv(\boldsymbol{x}) \to inv(\boldsymbol{trans}(\boldsymbol{x}))\Big) \\ \wedge \Big(inv(\boldsymbol{x}) \to post(\boldsymbol{x})\Big)\end{aligned}$$

where $pre(\boldsymbol{x})$ and $post(\boldsymbol{x})$ are *CLIA* formulae , $\boldsymbol{trans}(\boldsymbol{x})$ defines a vector of *CLIA* terms such that $|\boldsymbol{trans}(\boldsymbol{x})| = |\boldsymbol{x}|$.

Intuitively, $(pre(\boldsymbol{x}), \boldsymbol{trans}(\boldsymbol{x}), post(\boldsymbol{x}))$ represents a program with a set of variables $\boldsymbol{x}$. $pre(\boldsymbol{x})$ and $post(\boldsymbol{x})$ are the pre- and post-conditions, respectively. $\boldsymbol{trans}(\boldsymbol{x})$ represents the iterative transition: $\boldsymbol{x} := \boldsymbol{trans}(\boldsymbol{x})$. The loop terminates when $\boldsymbol{trans}(\boldsymbol{x}) = \boldsymbol{x}$. The goal of the synthesis problem is to find a loop invariant guaranteeing the partial correctness of the program with respect to *pre* and *post*.

Example 2.14. Consider a simple program of increasing variable x in a loop by 1 each iteration until it reaches 100:

```
int x = 0; while (x < 100) x = x + 1; assert x == 100;
```

The invariant synthesis problem for this program could be encoded to the following way:

$$\begin{aligned} pre(x) &\equiv (x = 0) \\ \boldsymbol{trans}(x) &\equiv \texttt{ite}(x < 100, x + 1, x) \\ post(x) &\equiv (\neg(x < 100) \Rightarrow (x = 100)) \end{aligned} \tag{2.5}$$

3 A Cooperative Synthesis Framework

In this section, we present a cooperative synthesis framework as a novel synergy of enumerative and deductive synthesis. In a nutshell, this framework encompasses a deductive synthesis engine and an enumerative synthesis engine, and solves synthesis problems by divide-and-conquer: it splits a synthesis problem into subproblems and solves them separately using deduction or enumeration.

3.1 Divide-And-Conquer Splitter

The cooperative synthesis framework features a divide-and-conquer splitter. The common pattern for these strategies is as follows: when the current synthesis problem p cannot be directly solved, the algorithm tries to identify a simpler problem (we call *Type-A Subproblem*) such that a solution to it can help simplify p to an easier-to-solve problem (we call *Type-B Subproblem*). We have identified several strategies that divide the original problem into subproblems A and B in different ways (see more details in Section 4).

Figure 2 illustrates the workflow of cooperative synthesis. Given a synthesis problem p, the deductive synthesis engine attempts to simplify/solve the input synthesis problem p purely deductively. If p is not completely solved, the divide-and-conquer splitter takes over and attempts to split the problem using a strategy. If the problem is divisible, the synthesis switches to the Type-A subproblem of p and starts over from the deductive synthesizer.

Otherwise, as the last resort, the problem is sent to the enumerative synthesizer. Notice that every possible solution has a syntax-tree representation, and the synthesizer just enumerates every height h and searches for syntax trees of fixed-height h, starting from 1, until a solution is found. Notice that this height-based enumeration guarantees to find the smallest possible solution, which is critical for many synthesis tasks that prefer compact solutions.

Whenever p is solved as a Type-A subproblem of another parent problem, the solution is used to generate the corresponding Type-B subproblem, for which the synthesis procedure repeats similarly.

3.2 Subproblem Graph

Notice that a problem can be split in multiple ways, split problem can be further split, and a subproblem can be generated and shared between multiple parent problems. We use a *subproblem graph* to represent the relations between problems.

Definition 3.1 (Subproblem Graph). Given a SyGuS problem S, a subproblem graph with respect to S is a directed acyclic graph (DAG) with a unique source (the node with no incoming edge) such that:

- every node represents a SyGuS problem; in particular, the source node represents S;

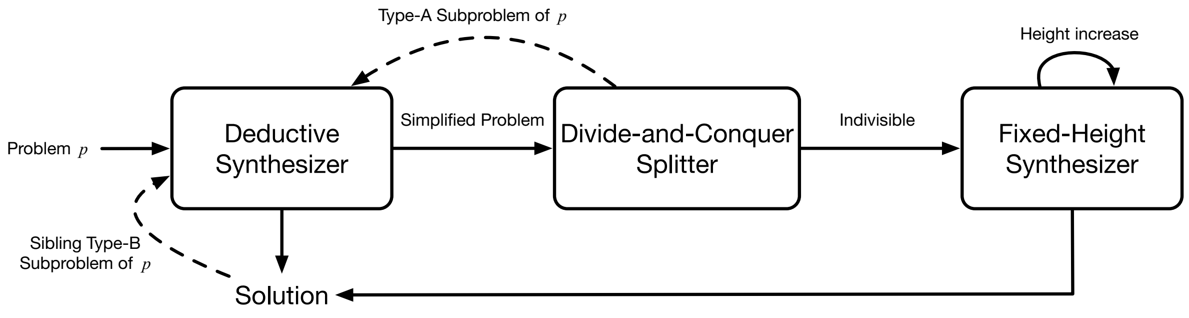

Figure 2. Workflow of cooperative synthesis.

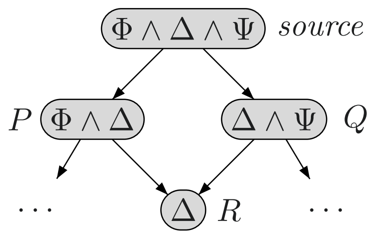

Figure 3. Example of subproblem graph.

- if there is an edge from the node representing P to the node representing Q, then Q is a type-A subproblem of P based on any divide-and-conquer strategy described in Section 6 (subterm-based, fixed-term-based, and weaker-spec-based).

For example, Figure 3 shows the subproblem graph, in which every node is annotated with the specification of the problem it represents. The source node represents the full specification $\Phi \wedge \Delta \wedge \Psi$. According to weaker-spec-based division (see Section 4.3), there are two Type-A subproblems $\Phi \wedge \Delta$ and $\Delta \wedge \Psi$, represented by the two successors P and Q, respectively. Moreover, the two subproblems can be further split to even simpler subproblems, among which R is their common subproblem because the specification Δ is the common conjunct of P and Q.

3.3 Cooperative Synthesis Algorithm

Algorithm 1 presents the overall cooperative synthesis algorithm. The algorithm takes as input a SyGuS problem $(f, \Phi, \mathcal{G})$ and maintains three data structures. PG is a subproblem graph with respect to the synthesis problem, initially built by BUILDGRAPH$(f, \Phi, \mathcal{G})$ (line 2). The procedure just builds a graph with a single source node representing Φ. *DedQueue* is a queue of (sub)problems to be solved by deduction; *EnumQueue* is a priority queue of (sub)problems to be solved by height-based enumeration. A (sub)problem of priority h is to be solved by fixed-height synthesis at height h (cf.

```
   input  : A SyGuS problem $(f, \Phi, \mathcal{G})$
   output : A solution $\lambda \boldsymbol{x}.e(\boldsymbol{x})$, if any; otherwise $\bot$
1  def cooperative-synth($f, \Phi, \mathcal{G}$):
2      $PG \leftarrow$ BUILDGRAPH($f, \Phi, \mathcal{G}$)
3      $DedQueue \leftarrow$ emptyQueue()
4      $EnumQueue \leftarrow$ emptyPriorityQueue()
5      enqueue ($DedQueue$, $PG$.source)
6      repeat
7          if $DedQueue \neq \emptyset$ :
8              $p \leftarrow$ dequeue ($DedQueue$); $h \leftarrow 0$
9              $p$.solution $\leftarrow$ deduct ($p$)
10             if $p$.solution = $\bot$ :
11                 $p$.succ $\leftarrow$ TYPEASUBPROBLEMS ($p$)
12                 foreach $c \in p$.succ :
13                     enqueue ($DedQueue$, $c$)
14         elif $EnumQueue \neq \emptyset$ :
15             $(p, h) \leftarrow$ dequeue ($EnumQueue$)
16             $p$.solution $\leftarrow$ fixed-height ($p$, $h$)
17         if $p$.solution = $\bot$ :
18             enqueue ($EnumQueue$, $p$, $h + 1$)
19         elif $p \neq PG$.source :
20             foreach $t \in p$.pred :
21                 $t \leftarrow$ TYPEBSUBPROBLEM($t$,
                    $p$.solution)
22                 enqueue ($DedQueue$, $t$)
23     until $PG$.source.solution $\neq \bot$ ;
24     return $PG$.source.solution
```

Algorithm 1: Cooperative synthesis framework.

Algorithm 2). Initially, *EnumQueue* is empty and *DedQueue* contains the source node of *PG* only.

The main part of the algorithm is a cooperative loop (lines 6–23) of deductive and enumerative synthesis which ends when a solution to the original problem is found. In each iteration of the loop, the algorithm dequeues one task p from *DedQueue* or *EnumQueue*. As deductive synthesis has higher priority, the task p is always dequeued from *DedQueue* if it is nonempty; otherwise from *EnumQueue*.

If p is from *DedQueue* (lines 7–13), it will be handled by the **deduct** function which conducts pure deductive synthesis (which will be elaborated in Section 6). If p can't be solved, the algorithm will expand *PG* with all possible type-*A* subproblems of p, adding each one as successor of the node representing p. All these newly created problems will be added to *DedQueue* for solving in future iterations. If p is from *EnumQueue* and has priority h (lines 14–16), the algorithm invokes the **fixed-height** function (cf. Algorithm 2 in Section 5) to find a solution of p at height h.

For both cases, if no solution to p is found, the problem will be added back to *EnumQueue* with priority $h + 1$ (lines 17–18). [2] Otherwise, if a solution of p is found, as long as p is not the original problem, the solution will help simplify every parent problem of p to the corresponding type-*B* subproblem. The algorithm does the simplification through the TypeBSubproblem procedure and add the updated problem to *DedQueue* for future iterations (lines 19–22).

Example 3.2. Let us consider Example 2.12 and see how Algorithm 1 solves this problem. Recall that there is no specific rule in our deductive synthesis algorithm for the ad hoc operator *qm*, hence the **deduct** function cannot solve the original problem and adds it to *EnumQueue*. However, the algorithm finds that the reference implementation has a subterm $\texttt{ite}(y \geq z, y, z)$, which allows a subterm-based division (cf. Section 4.1). Again, the **deduct** function cannot solve subproblem A and add it to *EnumQueue*. Then in the next several iterations, the **fixed-height** function takes over and tries to find a height-1 solution of the original problem or the subproblem and fails; when the height is moved up to 2, solution (4.1) is found for subproblem A and simplify the original problem to subproblem B. Finally, **fixed-height** finds a solution subproblem B at height 2 as well and combine the two solutions to form the final solution to the whole problem, as shown in Equation 4.2. The whole procedure takes only 4 seconds to solve this problem. In contrast, this problem can be solved by neither height-based enumeration nor pure deduction alone. State-of-the-art SyGuS solver CVC4 [35] spent 28 minutes to solve it and EUSolver [6] timed out.

4 Divide-And-Conquer Strategies

In this section, we describe the three divide-and-conquer strategies we developed for splitting SyGuS problems. The cooperative synthesis technique can be extended with more splitting strategies in the future. We explain each of them through examples.

4.1 Subterm-Based Division

Let us start from subexpression-based division. Let us continue on Example 2.12. The solution to the SyGuS problem

[2]Note that h is set to 0 for the deduction case and the next search will be at height 1.

$(max3, \Phi, \mathcal{G}_{qm})$ (Expression 2.2) has a large syntax-tree representation (height 6 and size 13) and is difficult to be synthesized. If a synthesizer is stuck with this problem, one may wonder if it is possible to synthesize a simpler, auxiliary function equivalent to a subexpression of the target expression (2.1). For example, can we synthesize an auxiliary function *aux* such that $aux(y, z) = \texttt{ite}(y \geq z, y, z)$? Then the original synthesis problem has been divided into two subproblems:

- *Subproblem A:* synthesize the auxiliary function *aux*;
- *Subproblem B:* once an implementation of *aux* is found, add *aux* to the grammar and synthesize f with the new grammar.

For example, assume the following solution for Subproblem A has been found:

$$aux(x_1, x_2) \overset{\text{def}}{=} x_1 + qm(x_2 - x_1, 0) \tag{4.1}$$

Then we can extend the grammar $\mathcal{G}_{qm}$ with the new operator *aux*. The grammar extension forms Subproblem B and allows us to find the following solution:

$$f(x, y, z) \overset{\text{def}}{=} aux(z, aux(x, y)) \tag{4.2}$$

Note that both solutions (4.1) and (4.2) are small and easier to be synthesized than the original problem, and inlining the implementation of *aux* in (4.1) into (4.2) just yields the expected solution (2.2).

Formally, this divide-and-conquer strategy is formulated as the rule Subterm in Figure 4: when e' is a subexpression of e (denoted as $e' \preccurlyeq e$), we first solve $f(\boldsymbol{y}) = e'$ as subproblem A, then solve $g(\boldsymbol{y}, e') = e$ as subproblem B, which is simpler than the original problem because g is allowed to use an extra argument e'.

4.2 Fixed-Term-Based Division

To understand fixed-term-based division, consider solving Example 6.1 using the CEGIS algorithm. Suppose a candidate solution $max2(x, y)$ is generated, even though it is not the expected solution, the candidate allows us to divide and simplify the synthesis problem. Notice that $max2(x, y)$ is a good implementation and satisfies the specification Φ when the inputs to the program satisfies $\Phi[max2(x, y)/f]$. Therefore, the synthesis problem can be divided into the following subproblems:

- *Subproblem A:* synthesize a function g that satisfies the specification only when the input does not satisfy $\Phi[max2(x, y)/f]$. In other words, the specification for g is $\Phi[max2(x, y)/f] \vee \Phi[g/f]$;
- *Subproblem B:* Combine $max2(x, y)$ and the synthesized function g to form an implementation that satisfies Φ for all inputs, no matter $\Phi[max2(x, y)/f]$ is satisfied or not.

Formally, this divide-and-conquer strategy can be formulated as the rule FixedTerm in Figure 4. Note that we apply this strategy only when $f(\boldsymbol{e}) \sim e \preccurlyeq \Phi$, which means $f(\boldsymbol{e}) \sim e$ occurs in Φ and $\sim$ is an arbitrary connective.

4.3 Weaker-Spec-Based Division

We illustrate how weaker-spec-based division works through the loop invariant synthesis problem defined in Definition 2.13. Recall that the specification consists of three parts:

$$\underbrace{pre(\boldsymbol{x}) \to inv(\boldsymbol{x})}_{\Phi} \wedge \underbrace{inv(\boldsymbol{x}) \to inv(\boldsymbol{trans}(\boldsymbol{x}))}_{\Delta} \wedge \underbrace{inv(\boldsymbol{x}) \to post(\boldsymbol{x})}_{\Psi}$$

When the whole synthesis problem is challenging and it is hard to generate appropriate $inv(\boldsymbol{x})$ to satisfy Φ, Δ and Ψ in tandem, one may divide the problem as follows:

- *Subproblem A:* synthesize an expression $P(\boldsymbol{x})$ that satisfies $\Phi \wedge \Delta$ (resp. $\Delta \wedge \Psi$);
- *Subproblem B:* synthesize an expression $Q(\boldsymbol{x})$ such that $P(\boldsymbol{x}) \wedge Q(\boldsymbol{x})$ (resp. $P(\boldsymbol{x}) \vee Q(\boldsymbol{x})$) satisfies the original specification $\Phi \wedge \Delta \wedge \Psi$.

Notice that both the two subproblems have weaker specifications than the original problem. Subproblem A is obviously easier as $P(\boldsymbol{x})$ only needs to satisfy two of the all three conjuncts. Subproblem B is also easier: imagine there is a solution $Q(\boldsymbol{x})$ to the original problem, it is also a solution to the subproblem B.

In loop invariant synthesis, the solutions P and Q from the subproblems are combined using conjunction or disjunction. However, weaker-spec-based division allows the combination P and Q using arbitrary binary functor $\oplus$. Now we define weaker specification in the most general way:

Definition 4.1 (Weaker Specification)**.** Let $(\mathcal{T}, f, \Phi, \mathcal{G})$ be a SyGuS problem where f's return type is τ, and let $\oplus$ be a binary functor whose two input functions and output function are all of type τ, and $\oplus \overset{\text{def}}{=} \lambda g_1, g_2.\lambda \boldsymbol{y}.E(g_1(\boldsymbol{y}), g_2(\boldsymbol{y}))$ where $E(x, y) \in [\![\mathcal{G}(x, y)]\!]$. Then Ψ is a weaker specification of Φ with respect to $\oplus$, denoted as $\mathcal{T} \models \Psi \preccurlyeq_{\oplus} \Phi$, if the following conditions hold:

1. $\mathcal{T} \models \Phi \to \Psi$;
2. $\mathcal{T} \models \forall g_1, g_2 : \Psi[g_1/f] \wedge \Psi[g_2/f] \to \Psi[g_1 \oplus g_2/f]$;
3. $\mathcal{T} \models \forall g_1, g_2 : (\Psi[g_1/f] \wedge \Psi[g_1 \oplus g_2/f]) \to (\Phi[g_1/f] \to \Phi[g_1 \oplus g_2/f])$.

With this general definition, we formulate the weaker-spec-based division as the rule WeakerSpec in Figure 4. Note that the loop invariant synthesis example we discussed above is just instances of the rule, in which $\oplus$ is instantiated to $\wedge$ or $\vee$.

4.4 Soundness and Completeness

All divide-and-conquer rules in Figure 4 are sound, as readers can verify. Although not all problems are splittable, these rules are complete in the sense that whenever a divide-and-conquer rule is applicable, the problem can be safely divided

```
input  : A SyGuS problem $p = (f, \Phi, \mathcal{G})$ and a
         positive integer $h$
output : A solution $\lambda \boldsymbol{x}.e(\boldsymbol{x})$ such that the syntax-tree
         representation of $e(\boldsymbol{x})$ is a full tree of height
         $h$, if any; otherwise $\bot$
// $E_\Phi$ is the set of counterexamples for
   spec $\Phi$
def fixed-height($p, h$) :
    $f \leftarrow p$.target; $\Phi \leftarrow p$.spec; $\mathcal{G} \leftarrow p$.grammar
    if $h = 1$ :
        $E_\Phi \leftarrow \emptyset$
    $q \leftarrow Init(\mathcal{G}, h)$
    repeat
        $result \leftarrow$ verify($\neg\Phi[q/f]$)
        if $result = unsat$ :
            break
        else:
            $E_\Phi \leftarrow E_\Phi \cup \{result\}$
            $q \leftarrow$ ind-synth($\bigwedge_{\boldsymbol{e} \in E} \Phi[\boldsymbol{e}/\boldsymbol{x}], \mathcal{G}, h$)
    until $q = \bot$;
    return $q$
```

Algorithm 2: Fixed-height synthesis.

into subproblems without missing any possible solution. We formulate the completeness as the following theorem:

Theorem 4.2. *Let Ω be a SyGuS problem, and let Ω_A and Ω_B be a pair of subproblems obtained by dividing Ω using a divide-and-conquer strategy. If Ω has a solution P, then P is also a solution to Ω_A and Ω_B.*

Proof. The completeness can be verified for each rule in Figure 4 separately. In particular, if the division is a Weaker-Spec division with respect to a weaker specification $\Psi \preccurlyeq_{\oplus} \Phi$, then according to Definition 4.1, the first and second conditions guarantee that P is also a solution to Ω_A and Ω_B, respectively. □

5 Fixed-Height Synthesis

Recall that the height-based enumeration algorithm sticks to a fixed size/height limit and searches for a solution within the limit symbolically, and gradually increase the limit when a solution of smaller size can't be found. While the algorithm is straightforward, it has its own merit and the idea does not seem be explored by any existing techniques. On the one hand, it still guarantees to synthesize the smallest satisfying program; on the other hand, it leverages as much power of symbolic solving as possible. In this section, we elaborate how the fixed-height synthesis component is implemented.

5.1 Concrete Height Enumeration

Algorithm 1 solves the fixed-height synthesis problem through a function fixed-height. When the height of the solution's

Subterm
$$\frac{(\mathcal{T}, f, f(\boldsymbol{y}) = e', \mathcal{G}) \rightsquigarrow P \quad (\mathcal{T}, g, g(\boldsymbol{y}, e') = e, \mathcal{G}) \rightsquigarrow Q}{(\mathcal{T}, f, f(\boldsymbol{y}) = e, \mathcal{G}) \rightsquigarrow \lambda \boldsymbol{y}, y'.Q(\boldsymbol{y}, y')[P(\boldsymbol{y})/y']} \quad \text{if } e' \preccurlyeq e$$

FixedTerm
$$\frac{(\mathcal{T}, g, \Phi[e/f(\boldsymbol{e})] \vee \Phi[g/f], \mathcal{G}) \rightsquigarrow P \quad (\mathcal{T}, f, f(\boldsymbol{y}, y') = \mathtt{ite}(\Phi[e/f(\boldsymbol{e})], e, y'), \mathcal{G}) \rightsquigarrow Q}{(\mathcal{T}, f, \Phi, \mathcal{G}) \rightsquigarrow \lambda \boldsymbol{y}.Q(\boldsymbol{y}, P(\boldsymbol{y}))} \quad \text{if } f(\boldsymbol{e}) \sim e \preccurlyeq \Phi \text{ for a connective } \sim$$

WeakerSpec
$$\frac{(\mathcal{T}, f, \Psi, \mathcal{G}) \rightsquigarrow P \quad (\mathcal{T}, g, \Phi[\lambda \boldsymbol{y}.\big(P(\boldsymbol{y}) \oplus g(\boldsymbol{y})\big)/f], \mathcal{G}) \rightsquigarrow Q}{(\mathcal{T}, f, \Phi, \mathcal{G}) \rightsquigarrow P \oplus Q} \quad \text{if } \mathcal{T} \models \Psi \preccurlyeq_{\oplus} \Phi$$

Figure 4. Deductive rules for divide-and-conquer.

syntax tree is fixed to h, the synthesis problem is simplified and usually can be solved *purely symbolically*. In Algorithm 2, we present a simple implementation of **fixed-height** based on the standard CEGIS framework [43]. The algorithm assumes there are function **verify** as the verifier and function **ind-synth** as the inductive synthesizer for fixed-height solutions, and maintains a set of counterexamples E. The algorithm starts with a random candidate solution $Init(\mathcal{G}, h)$ (line 5). In each following iteration, the synthesizer proposes a height-h candidate solution q that satisfies the specification when $\boldsymbol{x}$ is assigned values from E (line 12). Then the verifier checks condition (2.4), i.e., whether the candidate satisfies the specification Φ (line 7). If the result is unsat, then q is the desired implementation and the algorithm terminates; otherwise the verifier reports a counterexample as the witness of the failed verification and add it to E (line 11). Then the CEGIS loop continues with the next iteration repeatedly, until a solution is found.

Parallelization. While the naïve height-based enumeration in Algorithm 1 incrementally searches all possible heights of the decision tree, starting from 1, the enumeration can be naturally parallelized. If there are n cores available on the machine, the parallelized version runs the **fixed-height** algorithm at n different heights on n threads while sharing the set of counterexamples among them with proper synchronization. The algorithm starts with the n smallest heights, $\{1, \ldots, n\}$. It also maintains a variable k as the next height to search, starting from $n + 1$. Whenever a thread concludes that there is no solution at the current height, it starts a new CEGIS loop at height k, and the value of k gets increased. The whole algorithm stops whenever a thread finds a solution.

5.2 Symbolic Inductive Synthesis

In Algorithm 2, **verify** is just the standard background decision procedure, and **ind-synth** is an inductive synthesizer with the assumption that the solution's height is up to h. In our framework, this synthesis task is encoded and symbolically solved by the background decision procedure. We first

$$\begin{array}{rcl} \text{Int Const Vector: } \boldsymbol{c}_i & & \text{Int Const: } d_i \\ \text{Atom Expr: } e & ::= & \boldsymbol{c}_i \cdot x + d_i \\ \text{Atom Cond: } \alpha & ::= & e \geq 0 \\ \text{Expr: } E, E_1, E_2 & ::= & e \mid \mathtt{ite}(\alpha, E_1, E_2) \\ \text{Condition: } \varphi & ::= & \alpha \mid \mathtt{ite}(\alpha, \varphi_1, \varphi_2) \end{array}$$

Figure 5. Decision tree normal form.

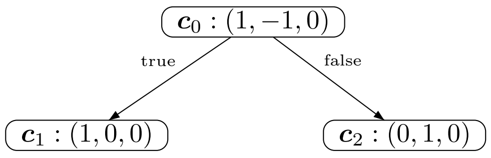
$\boldsymbol{c}_0 : (1, -1, 0)$
true
false
$\boldsymbol{c}_1 : (1, 0, 0)$
$\boldsymbol{c}_2 : (0, 1, 0)$

Figure 6. Representation of the *max*2 function.

illustrate the idea with the assumption that the grammar is $\mathcal{G}_{CLIA}$, then extend the encoding to arbitrary grammar $\mathcal{G}$.

Decision Tree Representation. Let the input SyGuS problem be $(f, \Phi, \mathcal{G}_{CLIA})$, then any implementation of f can be represented in a *decision tree normal form*, as described in Figure 5. It is not hard to see that every *CLIA* expression can be converted to this normal form. The decision tree representation of a *CLIA* expression is a binary tree in which every node with id i contains a vector $\boldsymbol{c}_i$ of integer constants and an extra constant d_i. Then each decision node (non-leaf node) tests whether $\boldsymbol{c}_i \cdot (\boldsymbol{x} \oplus (1)) \geq 0$ and according to the test result proceeds to the "true" child or "false" child. Each leaf node determines the value of the function as $\boldsymbol{c}_i \cdot (\boldsymbol{x} \oplus (1))$. For example, if f is a binary function and the solution is a binary max function of height 2: $f(x_1, x_2) \overset{def}{=} \mathtt{ite}(x_1 \geq x_2, x_1, x_2)$. It can be represented as the tree shown in Figure 6. Notice that the full decision tree of height h consists of $2^h - 1$ nodes, and the node id's can be fixed in the range between 0 and $2^h - 2$. This allows us to reduce **ind-synth** to the problem of searching for the vector $\boldsymbol{c}_i$ for each node i.

Interpret Function. Then for a fixed height h, we can build an interpret_h function that interprets the vectors back to the function of height h. For any function f of height h, its decision tree can be represented as 2^{h-1} vectors $\boldsymbol{c}_0, \ldots, \boldsymbol{c}_{2^h-1}$. Then for any vector of constants $\boldsymbol{d}$, interpret_h essentially interprets the decision tree on $\boldsymbol{d}$ and determines the value of $f(\boldsymbol{d})$. In other words, $\text{interpret}_h(\boldsymbol{c}_0, \ldots, \boldsymbol{c}_{2^h-1}, \boldsymbol{d}) = f(\boldsymbol{d})$. For example, continuing on the example of Figure 6. Assume $\boldsymbol{d} = (1, -2)$, then the value of $f(\boldsymbol{d})$ can be computed as:

$$\begin{aligned}&\text{interpret}_2(\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{d})\\ &\quad = \texttt{ite}\big(\boldsymbol{c}_0 \cdot \boldsymbol{d} \oplus (1) \geq 0,\ \boldsymbol{c}_1 \cdot \boldsymbol{d} \oplus (1),\ \boldsymbol{c}_2 \cdot \boldsymbol{d} \oplus (1)\big)\\ &\quad = \texttt{ite}\big(\boldsymbol{c}_0 \cdot (1, -2, 0) \geq 0,\ \boldsymbol{c}_1 \cdot (1, -2, 0),\ \boldsymbol{c}_2 \cdot (1, -2, 0)\big)\end{aligned}$$

Now to solve **ind-synth(**$\bigwedge_{\boldsymbol{e} \in E} \Phi[\boldsymbol{e}/\boldsymbol{x}], \mathcal{G}, h)$, we can replace every occurrences of f in $\Phi[\boldsymbol{e}/\boldsymbol{x}]$ with a corresponding interpret function. The resulting *CLIA* formula involves variables $\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2$ only and can be solved by a single SMT query.

Extension to General Grammar. We have generalized above encoding to arbitrary grammar $\mathcal{G}$. As an example, consider the $\mathcal{G}_{qm}$ grammar defined in Figure 1a. In the decision-tree representation, non-leaf nodes will represent the qm function invocation, which can be interpreted by the following adapted interpret_2 function:

$$\begin{aligned}&\text{interpret}_2(\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{d})\\ &\quad = qm\big(\text{interpret}_1(\boldsymbol{c}_1, \boldsymbol{d}), \text{interpret}_1(\boldsymbol{c}_2, \boldsymbol{d})\big)\\ &\quad = \texttt{ite}\big(\boldsymbol{c}_1 \cdot (1, -2, 0) < 0,\ \boldsymbol{c}_1 \cdot (1, -2, 0),\ \boldsymbol{c}_2 \cdot (1, -2, 0)\big)\end{aligned}$$

This generalization allows us to solve arbitrary SyGuS problems with the *CLIA* background theory (see the General track benchmarks in Section 7). We leave further generalization to other background theories, e.g., bit vectors, to future work.

6 The Deductive Component

In this section, we introduce the deductive component of the framework (i.e., the deduct function in Algorithm 1). This component integrates a set of deductive rules that can simplify the specification Φ or find a solution directly. The implementation, as shown in Algorithm 3, just repeatedly and exhaustively applies these rules to simplify the specification Φ. If the simplified Φ is already a solution (of the form $f(x_1, \ldots, x_n) = e$), return the solution; otherwise return $\perp$. As deduction can be performed very efficiently, this component serves as the first step for all (sub)problems.

Note that our deductive rules are designed as a component for the cooperative synthesis framework, they are not expected to be complete in any sense. That said, they are already powerful enough to solve many synthesis problems. For instance, the rules in Figures 7 and 8 have already superseded the class of *Single Invocation Problems*, a common class of problems that can be solved using the counterexample-guided quantifier instantiation algorithm [35].

```
input  : A SyGuS problem $p = (f, \Phi, \mathcal{G})$
output : A solution $\lambda \boldsymbol{x}.e(\boldsymbol{x})$, if any; otherwise $\perp$
def deduct($p$):
    $f \leftarrow p$.target; $\Phi \leftarrow p$.spec; $\mathcal{G} \leftarrow p$.grammar
    $\Phi \leftarrow$ Simplify($f, \Phi, \mathcal{G}$); $p$.spec $\leftarrow \Phi$
    if IsSolution($\Phi, \mathcal{G}$) :
        return $\Phi$
    else:
        return $\perp$
```
Algorithm 3: Deductive synthesis.

IntEq		
$f(\boldsymbol{y}) = e \wedge \Psi$	$\Longrightarrow$	$f(\boldsymbol{y}) = e \wedge \Psi[\lambda \boldsymbol{y}.e/f]$
IntNeq		
$f(\boldsymbol{y}) \neq e \vee \Psi$	$\Longrightarrow$	$f(\boldsymbol{y}) \neq e \vee \Psi[\lambda \boldsymbol{y}.e/f]$
BoolPos		
$(f(\boldsymbol{y}) \vee \Phi) \wedge \Psi$	$\Longrightarrow$	$\Psi[\lambda \boldsymbol{y}.((\neg\Phi) \vee f(\boldsymbol{y}))/f]$
		if f does not occur in Φ
BoolNeg		
$(\neg f(\boldsymbol{y}) \vee \Phi) \wedge \Psi$	$\Longrightarrow$	$\Psi[\lambda \boldsymbol{y}.(\Phi \wedge f(\boldsymbol{y}))/f]$
		if f does not occur in Φ
RemoveVar		
Ψ	$\Longrightarrow$	$\Psi[0/y_i]$ if $\mathcal{T} \models \Phi \leftrightarrow \Phi[y_i'/y_i]$
RemoveArg		
$(f, \Phi, \mathcal{G})$	$\Longrightarrow$	$(g, \Phi[g(\boldsymbol{e}, \boldsymbol{e'})/f(\boldsymbol{e}, C, \boldsymbol{e'})], \mathcal{G})$
		if the i-th arg of f is always constant C
Match		
$(f, f(\boldsymbol{y}) = e, \mathcal{G})$	$\Longrightarrow$	$(f, f(\boldsymbol{y}) = e', \mathcal{G})$
		if $e \Longrightarrow^*_{\mathcal{G}} e'$ and $e' \in [\![\mathcal{G}(\boldsymbol{y})]\!]$

Figure 7. Deductive rules for arbitrary grammar.

We next present deductive rules that are general and applicable to arbitrary grammar, followed by special simplification for two special classes of problems. To the best of our knowledge, these rules are not explicitly integrated in any existing deductive synthesizer. Our framework can also integrate more deductive rules in the future.

General Deduction. Figure 7 shows a set of general deductive rules for arbitrary grammar. Assuming f is the function to be synthesized, these rules soundly substitute occurrences of f, arguments or variables with a concrete implementation. Most of the rules are self explanatory. In particular, the last rule Match applies when the specification is a reference implementation $f(\boldsymbol{y}) = e$ but e does not conform to the grammar $\mathcal{G}$. In that case, we can exhaustively match and replace subexpressions of e with interpreted functions in $\mathcal{G}$, and check if the final expression falls in $[\![\mathcal{G}]\!]$. For example, let e be $x + x + x + x$ and let $\mathcal{G}$ be a grammar that contains only one operator $double(x) \overset{\text{def}}{=} x + x$, then e can be rewritten to $double(double(x))$.

GeMax

$f(\boldsymbol{e}) \geq e_1 \wedge f(\boldsymbol{e}) \geq e_2 \implies f(\boldsymbol{e}) \geq \mathtt{ite}(e_1 \geq e_2, e_1, e_2)$

LeMin

$f(\boldsymbol{e}) \leq e_1 \wedge f(\boldsymbol{e}) \leq e_2 \implies f(\boldsymbol{e}) \leq \mathtt{ite}(e_1 \geq e_2, e_2, e_1)$

GeMin

$f(\boldsymbol{e}) \geq e_1 \vee f(\boldsymbol{e}) \geq e_2 \implies f(\boldsymbol{e}) \geq \mathtt{ite}(e_1 \geq e_2, e_2, e_1)$

LeMax

$f(\boldsymbol{e}) \leq e_1 \vee f(\boldsymbol{e}) \leq e_2 \implies f(\boldsymbol{e}) \leq \mathtt{ite}(e_1 \geq e_2, e_1, e_2)$

Eq

$f(\boldsymbol{e}) \geq e_1 \wedge f(\boldsymbol{e}) \leq e_2 \implies f(\boldsymbol{e}) = e_1$ if $\mathcal{T} \models e_1 = e_2$

NotEq

$f(\boldsymbol{e}) \geq e_1 \vee f(\boldsymbol{e}) \leq e_2 \implies f(\boldsymbol{e}) \neq e_1 - 1$ if $\mathcal{T} \models e_1 = e_2 + 2$

CNF

$(\Phi \vee \Psi_1) \wedge (\Phi \vee \Psi_2) \implies \Phi \vee (\Psi_1 \wedge \Psi_2)$ if f does not occur in Ψ_1 or Ψ_2

Figure 8. Deductive rules for $\mathcal{G}_{CLIA}$.

$$\begin{aligned}
&f(x,y,z) \geq x \;\wedge\; f(x,y,z) \geq y \;\wedge\; f(x,y,z) \geq z \;\wedge \\
&\big(f(x,y,z) = x \vee f(x,y,z) = y \vee f(x,y,z) = z\big) \overset{\text{CNF}}{\Longrightarrow} \\
&\quad f(x,y,z) \geq x \;\wedge\; f(x,y,z) \geq y \;\wedge\; f(x,y,z) \geq z \\
&\quad \wedge \big(f(x,y,z) \geq x \vee f(x,y,z) \geq y \vee f(x,y,z) \geq z\big) \\
&\quad \wedge \big(f(x,y,z) \leq x \vee f(x,y,z) \leq y \vee f(x,y,z) \leq z\big) \\
&\quad \wedge \;\ldots \xRightarrow{\text{GeMax,LeMax},\ldots} \\
&f(x,y,z) \geq \mathtt{ite}(\mathtt{ite}(x \geq y, x, y) \geq z, \mathtt{ite}(x \geq y, x, y), z) \\
&\wedge f(x,y,z) \leq \mathtt{ite}(\mathtt{ite}(x \geq y, x, y) \geq z, \mathtt{ite}(x \geq y, x, y), z) \\
&\wedge \;\ldots \xRightarrow{\text{Eq,IntEq}} \\
&\quad f(x,y,z) = \mathtt{ite}(\mathtt{ite}(x \geq y, x, y) \geq z, \mathtt{ite}(x \geq y, x, y), z) \\
&\quad \xRightarrow{\text{Match}} f(x,y,z) = max2(max2(x,y),z)
\end{aligned}$$

Figure 9. Rewriting sequence for Example 6.1.

Merging and Substituting for CLIA.. For $\mathcal{G}_{CLIA}$, a very common grammar for syntax-guided synthesis, we designed a set of ad hoc rules as illustrated in Figure 8. Intuitively, these rules find two occurrences of f and merge them into a single occurrence.

Example 6.1. Let $\mathcal{G}$ be a grammar with only an operator $max2(x,y) \overset{\text{def}}{=} \mathtt{ite}(x \geq y, x, y)$. Our deductive synthesis algorithm can synthesize a ternary maximum function $f(x,y,z)$ using the rewriting sequence shown in Figure 9.

Loop Summary for Invariant Synthesis. We also developed a special class of simplification rules for loop invariant synthesis. The idea is to find a predicate that precisely summarizes the effect of arbitrary k-steps of loop transformation. Formally, if there exists a binary predicate ***fast-trans*** such that

$$\boldsymbol{fast\text{-}trans}(\boldsymbol{x},\boldsymbol{y}) \Leftrightarrow \exists k \geq 0.\boldsymbol{trans}^k(\boldsymbol{x}) = \boldsymbol{y}$$

then the original specification can be reduced to a simpler constraint:

$$\Big(\big(pre(\boldsymbol{x}) \wedge \boldsymbol{fast\text{-}trans}(\boldsymbol{x},\boldsymbol{y})\big) \rightarrow inv(\boldsymbol{y})\Big) \wedge \Big(inv(\boldsymbol{y}) \rightarrow post(\boldsymbol{y})\Big)$$

For example, the loop transformation in Example 2.14 can be summarized as:

$$\boldsymbol{fast\text{-}trans}(x,y) \equiv (x < 100 \wedge x \leq y) \vee x = y$$

We have identified a class of *Acyclic Translational* loop transformations for which such summarization exists and the synthesis problem is decidable. We leave the details in Appendix A of the supplementary material.

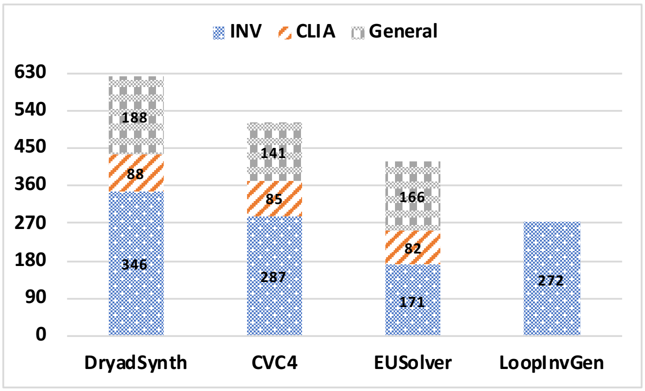

Figure 10. Solved benchmarks (breakdown by tracks).

7 Experimental Evaluation

We have prototyped our cooperative synthesis technique as a system called DryadSynth,[3] which supports the *CLIA* background theory. DryadSynth is written in Java with around 11k LOC, and employs Z3 [11] as the constraint solving engine. This is a relatively small and lightweight implementation in terms of engineering (Comparing to, e.g. 343k+ LOC of the CVC4 code base and 33k+ LOC of the EUSolver code base).

To evaluate our algorithms, we compared DryadSynth with state-of-the-art SyGuS solvers, CVC4, EUSolver and LoopInvGen. [4] They are winning solvers in recent years' SyGuS competition and we used the latest version from their public repositories. CVC4 and EUSolver are two general-purpose solvers that participate in the General, CLIA and INV tracks. LoopInvGen focuses on INV track only.

[3]https://github.com/purdue-cap/DryadSynth

[4]We omitted other solvers as they focus on other background theories and not comparable with ours. For example, while Euphony's AI-guided algorithm [26] is promising, it supports String and BitVector theories only, and its old algorithm was not competitive in the *CLIA* theories.

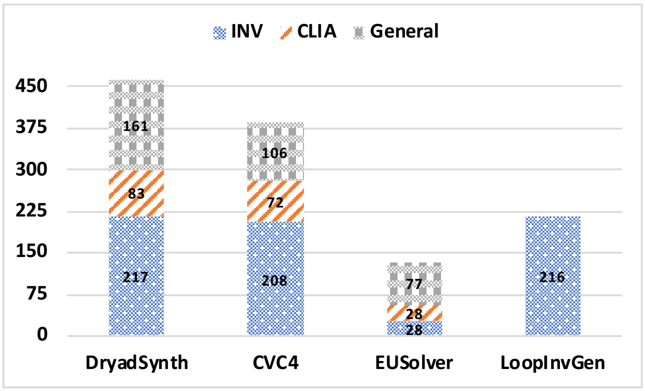

Figure 11. Fastest solved benchmarks (breakdown by tracks).

Experimental Setting. Experiments were conducted on the StarExec platform [45], on which each solver is executed on a 4-core, 2.4GHz CPU and 128GB memory node, with a 30-minute timeout. We adopted 403 INV benchmarks, 88 CLIA benchmarks, and 224 General track benchmarks with the *CLIA* background theory included in the SyGuS competition of 2019. [5] We excluded 372 General benchmarks that are based on the *BitVector* background theory and 426 INV benchmarks that contain let-macros, which DryadSynth does not support at present. It wound up with all of 715 benchmarks.

We summarize the experimental results through a set of figures. Figures 10 and 11 compare the number of benchmarks correctly solved within the 30-minute limit and the number of benchmarks solved the fastest among all solvers, respectively. [6] Figure 12 shows the comparison of the solvers in terms of the number of benchmarks solved and the total amount of time spent. Figure 13 shows the amounts of time spent for every benchmark, sorted in ascending order. All figures break down the comparison by tracks. It is noteworthy that DryadSynth allows us to solve **58** benchmarks which were not solvable by other synthesizers, while LoopInvGen have 9 benchmarks uniquely solved. We list these benchmarks and the time spent by DryadSynth to solve them in Appendix B of the supplementary material.

Observation. Observing the figures, we are encouraged by the following facts: 1) Figures 10 and 11 show that DryadSynth solved and *fastest* solved more benchmarks than all other solvers in all tracks. 2) Figure 12 indicates that DryadSynth solved more CLIA and General benchmarks than all other solvers, with less total time spent. 3) Figure 13 shows that DryadSynth has better scalability than all other synthesizers: although DryadSynth had a constant overhead on easier-to-solve problems, the solving time increases more mildly toward more challenging benchmarks than other synthesizers.

[5] We slightly adapted 21 of General benchmarks to remove the let-macros.
[6] Following the criterion of SyGuS competition, the time amounts are classified into buckets of pseudo-logarithmic scales: [0, 1), [1, 3), [3, 10), . . . , [1000, 1800).

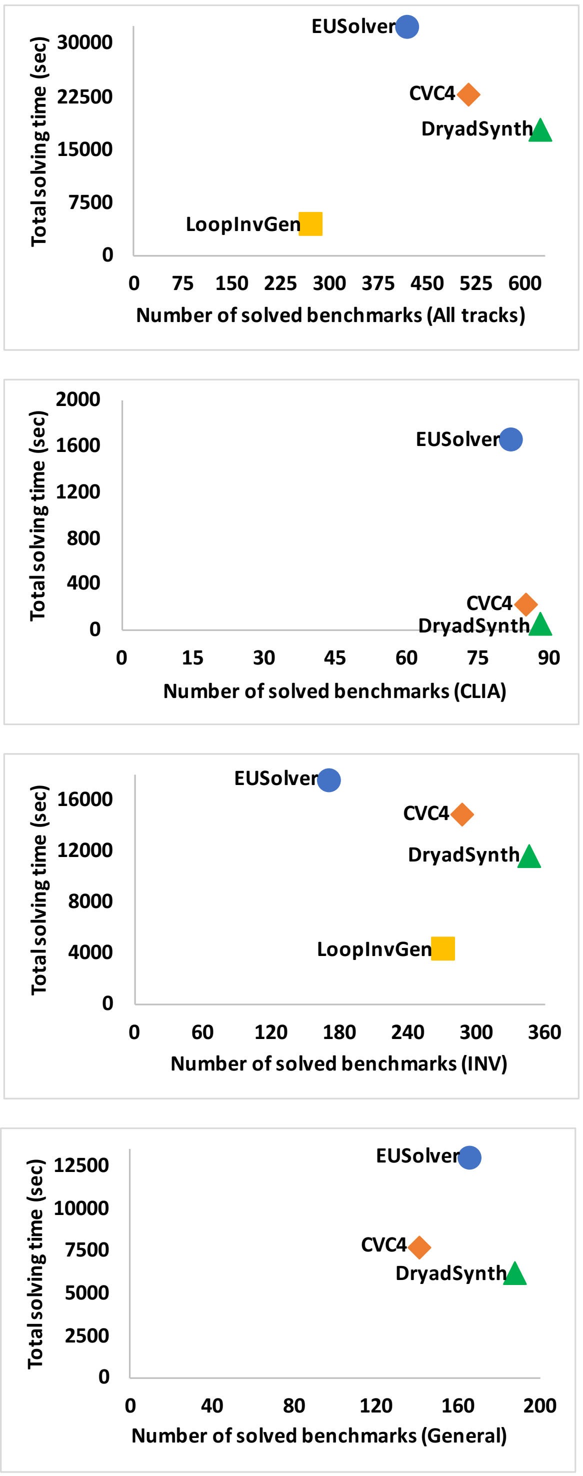

Figure 12. Comparison of solvers on total solved benchmarks and total solving time.

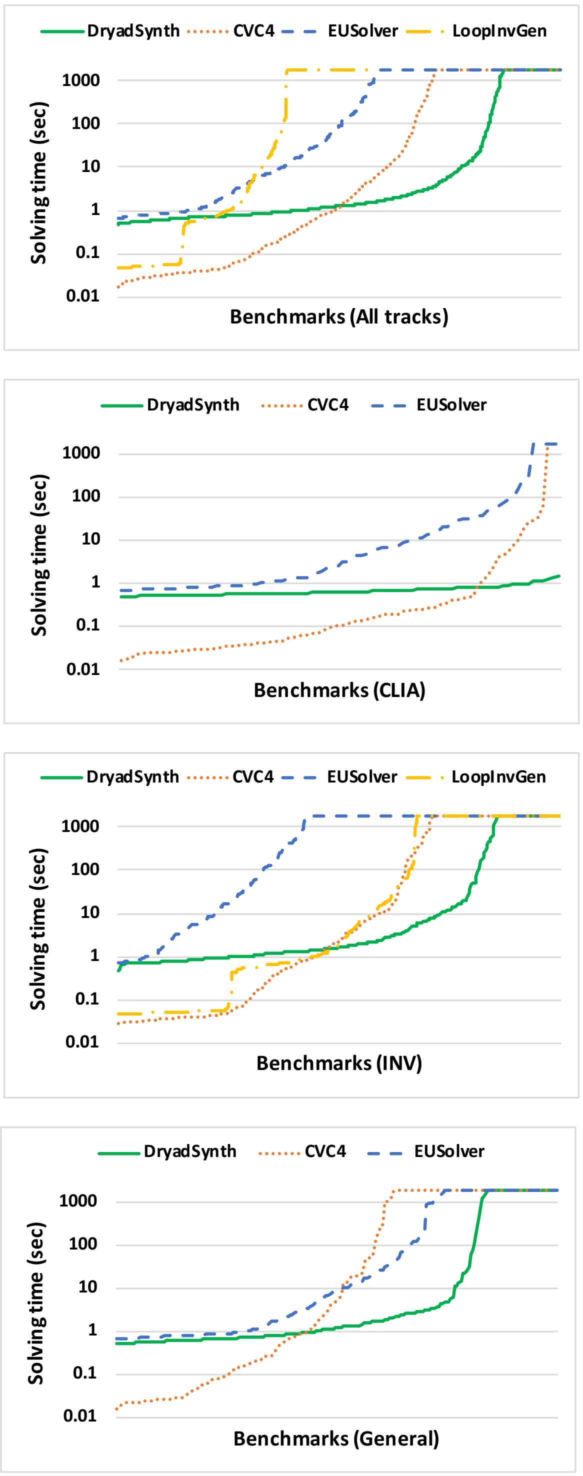

Figure 13. Solving time per benchmark in increasing order.

In summary, our cooperative synthesis technique outperforms state-of-the-art synthesizers in both scalability and diversity of the solved problems, and tends to be a general and efficient synthesis engine for syntax-guided synthesis.

Table 1. Number of smallest solutions and median of solution size (in small text). Best numbers in grey.

Track	DryadSynth	CVC4	EUSolver	LoopInvGen
INV	132 38	118 26	171 3	220 7
CLIA	56 278.5	56 361	67 201.5	-
General	141 19	124 19	166 19	-

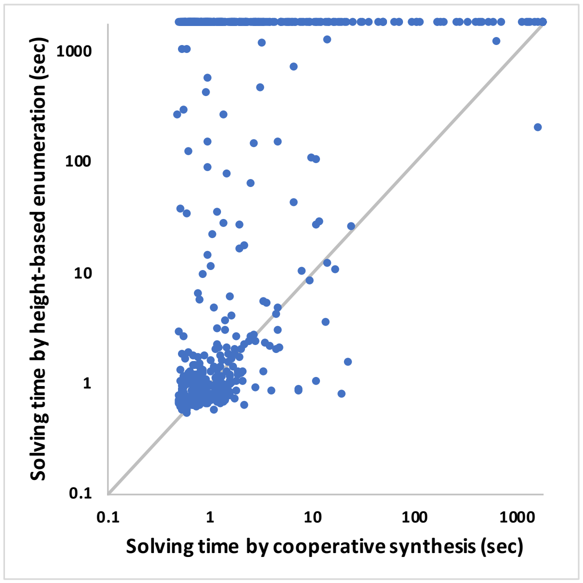

Figure 14. Cooperative synthesis vs. Plain height-based enumeration.

Remark: We also roughly compared the size of solutions as our deductive component does not control the solution size, as shown in Table 1. Based on the number of smallest solutions [7] and the median size of solutions for the commonly solved benchmarks, DryadSynth is slightly better than CVC4 but worse than EUSolver (purely enumerative) and LoopInvGen (for INV track only).

Ablation Studies. To evaluate whether the combination of enumerative and deductive synthesis improves the performance, we also compare DryadSynth, in which the full-fledged cooperative synthesis framework is implemented, with our implementation of the plain height-based enumeration synthesis algorithm (Algorithm 2), the plain deductive synthesis algorithm (Algorithm 3), and our cooperative synthesis framework with the height-based enumeration synthesis algorithm replaced by EUSolver, a representative enumerative synthesizer.

Figure 14 compares the solving time of the full-fledged cooperative synthesis framework and the plain height-based enumeration synthesis algorithm on all benchmarks. As the

[7]Following the criterion of SyGuS competition, the size amounts are classified into buckets of pseudo-logarithmic scales: [1,10), [10,30), [30,100), [100,300), [300,1000), ≥ 1000.

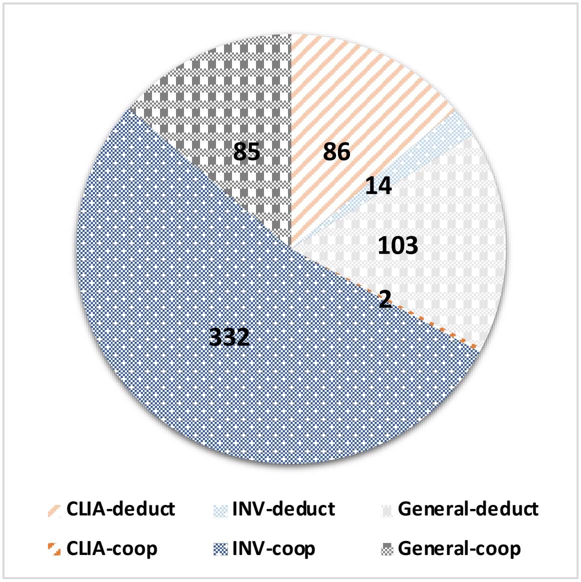

Figure 15. Cooperative synthesis vs. Plain deduction.

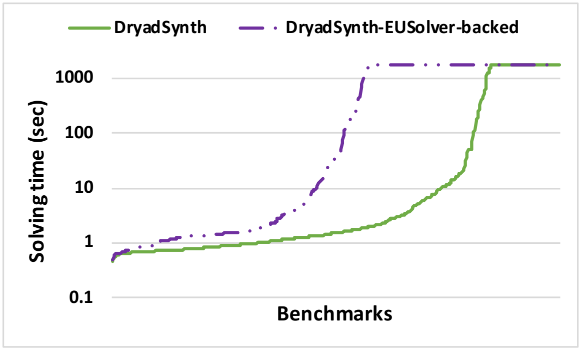

Figure 16. Vanilla DryadSynth vs. EUSolver-backed DryadSynth.

figure illustrates, with the help of divide-and-conquer deduction, our cooperative synthesis clearly outperformed the plain height-based enumeration for the vast majority of all benchmarks. The plain height-based enumeration procedure performed slightly better for several easier-to-solve problems, though, as they are simple enough and divide-and-conquer cannot help much.

Figure 15 shows the number of solved benchmarks by the plain deductive synthesis algorithm (per category) and the number of extra benchmarks solved with the help of the height-based enumeration (per category). Figure 15 shows that among all the benchmarks solved by the cooperative synthesis framework, only 32.6% of them were solved by pure divide-and-conquer deduction. The vast majority of all benchmarks were further solved with the help of height-based enumeration.

Figure 16 compares the amounts of time spent for benchmarks between the vanilla DryadSynth and a version using EUSolver as the enumerative synthesis component. In the EUSolver-backed DryadSynth, every invocation to the fixed-height synthesis algorithm (Algorithm 2) is replaced with a query to EUSolver. As we could not find a proper way to control the search space when invoking EUSolver, the query to EUSolver searches with unbounded height. Therefore, while our height-based enumeration was parallelized, it became ineffectual to parallelize EUSolver. We omitted those benchmarks purely solved by the deductive synthesis algorithm. That wound up with a comparison on 496 benchmarks. The figure indicates that the vanilla DryadSynth consistently performed better and solved 135 more benchmarks than the EUSolver-backed DryadSynth.

In short, our studies show that the cooperation of our enumerative and deductive algorithms improves the performance from the standalone enumeration, and solves more benchmarks than the standalone deduction. Our height-based enumeration algorithm also performs better than an existing enumeration algorithm when serving as the enumerative synthesis component.

8 Related Work

Syntax-Guided Synthesis. As we mentioned in Section 1, the most important dimension along which we characterize existing syntax-guided synthesis approaches is their synthesis strategies. For example, among the winning SyGuS synthesizers we compared with, EUSolver [6] adopts a purely enumerative search strategy, and CVC4 [35] solves synthesis problems purely symbolically through a procedure called counterexample guided quantifier instantiation. The Counterexample-Guided Inductive Synthesis (CEGIS) framework [40] has been a common theme underlying several solvers which differ in how the synthesizer generates candidates from counterexamples: Sketch [39, 41] solves constraints that encode the counterexamples; Alchemist [36] and ICE-DT [18] find likely candidates using learning algorithms.

The decision tree representation for fixed-height synthesis (Sec 5.2) is similar to the representations used in the ICE-DT learning algorithm [18] and the enumerative search algorithm underlying EUSolver [6]. Our decision tree for *CLIA* (e.g., Figure 6) is different: it assumes a fixed height for the decision tree and represents the whole function as a vector of coefficients, and helps us encode the fixed-height synthesis problem to a *CLIA* query.

A lot of general search-based algorithms for syntax-guided synthesis have been developed in the past few years, including learning-based ones [17, 18, 36] and enumerative ones [2, 46]. Caulfield et al. [10] identified several decidable fragments of synthesis problems in the theories of *EUF* and *BitVector*. The SyGuS solver Euphony [26] predicts the *likelihood* of candidate programs, and enumerates them starting from the most likely correct candidates. Their algorithm focuses on theories of *String* and *BitVector*.

Combining Concrete and Symbolic Search. The idea of height-based enumeration is inspired by a recent trend of combining concrete and symbolic search, but differs from existing approaches in the target program and/or the enumeration method.

The work of Gulwani *et al.* [22] is, to the best of our knowledge, the first attempt of combining enumerative and symbolic search. Their system enumerates the number of components and encodes each case as a symbolic constraint. The Adaptive Concretization algorithm [24, 25] developed for Sketch is another instance of enumerative-symbolic combination. As a Sketch-based algorithm, adaptive concretization supports a general class of SyGuS problems. Its algorithm statistically determines a class of *highly influential unknowns* and explicitly enumerates all possible values of these unknowns. Unlike our decision-tree-based enumeration, their enumeration strategy is not supported by integer arithmetic decision procedures and seems not competitive in synthesizing *CLIA* functions [4]. Synquid [31] synthesizes recursive functional programs using an algorithm that enumerates the top part of the program and synthesizes the remaining part of the program through liquid abduction. Hades [48] is a system that synthesizes transformations on hierarchical data trees. A key component of Hades is an algorithm for synthesizing path transformations from examples, which enumerates all possible partitions of the examples, checks the unifiability of each partitioned set using SMT solvers, and combines the unifiers into a decision tree through machine learning.

Our height-based enumeration is different: the shapes (or sketches) of the syntax tree are not explicitly enumerated or learned, but grouped by their heights and then enumerated and solved symbolically. This is a nice combination as it still guarantees to synthesize the smallest satisfying program while leveraging more power of symbolic solving. To the best of our knowledge, this idea is not explored by any existing techniques.

Deductive Synthesis. There has been significant research effort on deductive synthesis. Spiral [34] is an automatic system that synthesizes digital signal processing algorithms and programs. Paraglide [47] derives algorithms for concurrent programs from their sequential implementations using domain specific knowledge to constraint the search space. Fiat [12] also utilizes deductive synthesis to synthesize abstract data types that package methods with private data. As an interactive system, the synthesizer also requires user's guidance to help with the synthesis. Recently, Polikarpova and Sergey [32] present SusLik as a deductive-based synthesizer that generates imperative heap-manipulating programs. The synthesizer takes a pair of pre- and post- conditions written in separation logic as input and derives the programs based on a set of deductive rules with structural constraints of the heap baked in. Above deductive synthesizers are all designed to serve a specific application and seem hard to extend to other synthesis purposes, as domain specific knowledge is critical for these systems. Our approach is more general and not limited to a fixed grammar.

Combining Deduction and Enumeration. We are not the first to combine deduction and enumeration. $\lambda 2$ [16] and FleshMeta [33] have used deduction in novel ways (inverse semantics or refutation) to decompose the synthesis task and guide the program search. They can be perceived as special type-directed search algorithms [42] and comparable to other search strategies. More recently, Morpheus [15] also combines deduction and enumeration to synthesize programs manipulating tabular data. Morpheus uses enumerative search to find possible candidate programs and uses deduction to prune the search space. With similar ideas, Neo [14] synthesizes programs in several domains, including tabular data transformation and list manipulation, by supplying a DSL of the target domain. Our deduction-enumeration combination is different from the techniques above: it repeatedly performs divide-and-conquer and solves subproblems by deduction or enumeration separately.

Li *et al.* [27] present a technique for searching proofs for program correctness. They use abductive inference to decompose the verification task to several lemmas then discharge each lemma separately. Our idea of divide-and-conquer deduction is similar to their lemma abductive inference. The difference from the prior technique is that we focus on syntax-guided program synthesis and design our own deductive rules that are general enough for arbitrary grammars.

9 Conclusion

We introduced cooperative synthesis, a syntax-guided synthesis technique in which enumerative and deductive synthesis strategies are combined for solving SyGuS problems with *CLIA* background. The framework repeatedly splits large synthesis problems to smaller subproblems and have them solved by a deductive synthesis engine or a height-based enumeration algorithm. Then the solutions are combined to form a final solution. We found that, compared to state-of-the-art synthesizers, cooperative synthesis has better scalability and solved many benchmarks not possible before. This synthesis technique may be extended to handle other background theories in the future.

Acknowledgments

We thank our shepherd, Isil Dillig, and the anonymous reviewers for their feedback in improving the paper.

This research was supported in part by the National Science Foundation under Grant No. CCF-1837023. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. 1–8. http://ieeexplore.ieee.org/document/6679385/

[2] Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *CAV (2) (Lecture Notes in Computer Science)*, Vol. 9207. Springer, 163–179.

[3] Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Armando Solar-Lezama. 2018. SyGuS-Comp 2018: Results and Analysis. *https://sygus.org/comp/2018/report.pdf* (2018).

[4] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2016. SyGuS-Comp 2016: Results and Analysis. *https://arxiv.org/abs/1611.07627* (2016). arXiv:1611.07627

[5] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. SyGuS-Comp 2017: Results and Analysis. (11 2017). arXiv:1711.11438 https://arxiv.org/abs/1711.11438

[6] 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*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 319–336.

[7] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-based program synthesis. *Commun. ACM* 61, 12 (Nov 2018), 84–93. https://doi.org/10.1145/3208071

[8] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *CoRR* abs/1611.01989 (2016). arXiv:1611.01989 http://arxiv.org/abs/1611.01989

[9] R. M. Burstall and John Darlington. 1977. A Transformation System for Developing Recursive Programs. *J. ACM* 24, 1 (Jan 1977), 44–67. https://doi.org/10.1145/321992.321996

[10] Benjamin Caulfield, Markus N. Rabe, Sanjit A. Seshia, and Stavros Tripakis. 2015. What's Decidable about Syntax-Guided Synthesis? (10 2015). arXiv:1510.08393 https://arxiv.org/abs/1510.08393

[11] Leonardo de Moura and Nikolaj Bjørner. 2008. *Z3: An Efficient SMT Solver*. Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[12] Benjamin Delaware, Clement Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *POPL'15*. ACM, 689–700.

[13] Dana Drachsler-Cohen, Sharon Shoham, and Eran Yahav. 2017. Synthesis with Abstract Examples. *Lecture Notes in Computer Science* (2017), 254–278. https://doi.org/10.1007/978-3-319-63387-9_13

[14] 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* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

[15] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

[16] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *PLDI'15* (Portland, OR, USA). ACM, 229–239.

[17] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *CAV'14*. 69–87. https://doi.org/10.1007/978-3-319-08867-9_5

[18] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. In *POPL'16* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 499–512. https://doi.org/10.1145/2837614.2837664

[19] Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B. Tenenbaum, and Tim Mattson. 2018. The three pillars of machine programming. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages - MAPL 2018* (2018). https://doi.org/10.1145/3211346.3211355

[20] 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* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[21] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *Commun. ACM* 55, 8 (Aug. 2012), 97–105. https://doi.org/10.1145/2240236.2240260

[22] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

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

[24] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. 2015. Adaptive Concretization for Parallel Program Synthesis. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science)*, Vol. 9207. 377–394.

[25] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. 2017. An empirical study of adaptive concretization for parallel program synthesis. *Formal Methods in System Design* 50, 1 (01 Mar 2017), 75–95. https://doi.org/10.1007/s10703-017-0269-8

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

[27] Boyang Li, Isil Dillig, Thomas Dillig, Ken McMillan, and Mooly Sagiv. 2013. Synthesis of Circular Compositional Program Proofs via Abduction. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 370–384.

[28] Z. Manna and R. Waldinger. 1979. Synthesis: Dreams => Programs. *IEEE Transactions on Software Engineering* 5, 4 (1979), 294–328.

[29] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian Sketch Learning for Program Synthesis. *CoRR* abs/1703.05698 (2017). arXiv:1703.05698 http://arxiv.org/abs/1703.05698

[30] Saswat Padhi and Todd Millstein. 2017. Data-Driven Loop Invariant Inference with Automatic Feature Synthesis. (07 2017). arXiv:1707.02029 https://arxiv.org/abs/1707.02029

[31] 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* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

[32] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the synthesis of heap-manipulating programs. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan 2019), 1–30. https://doi.org/10.1145/3290385

[33] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming,*

Systems, Languages, and Applications (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

[34] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, , F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (Feb 2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306

[35] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9207. Springer, 198–216. https://doi.org/10.1007/978-3-319-21668-3_12

[36] Shambwaditya Saha, Pranav Garg, and P. Madhusudan. 2015. Alchemist: Learning Guarded Affine Functions. In *CAV'15*. 440–446. https://doi.org/10.1007/978-3-319-21690-4_26

[37] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. ACM, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

[38] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic Optimization of Floating-point Programs with Tunable Precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/2594291.2594302

[39] Armando Solar-Lezama. 2008. *Program Synthesis By Sketching*. Ph.D. Dissertation. EECS Dept., UC Berkeley.

[40] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5 (01 Oct 2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

[41] Armando Solar-Lezama. 2016. *The Sketch Programmers Manual*. Version 1.7.2.

[42] Armando Solar-Lezama. 2018. Introduction to Program Synthesis. https://people.csail.mit.edu/asolar/SynthesisCourse/TOC.htm

[43] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *ASPLOS'06* (San Jose, California, USA). ACM, 404–415.

[44] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. In *POPL'10* (Madrid, Spain). ACM, 313–326.

[45] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. 2014. StarExec: A Cross-Community Infrastructure for Logic Solving. In *Automated Reasoning*, Stéphane Demri, Deepak Kapur, and Christoph Weidenbach (Eds.). Springer International Publishing, Cham, 367–373.

[46] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *PLDI*. 287–296.

[47] Martin Vechev and Eran Yahav. 2008. Deriving Linearizable Fine-grained Concurrent Objects. In *PLDI'08*. ACM, 125–135.

[48] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing Transformations on Hierarchically Structured Data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 508–521. https://doi.org/10.1145/2908080.2908088