A Concurrent Approach to String Transformation Synthesis

ANONYMOUS AUTHOR(S)

Program synthesis aims at the automatic generation of programs based on given specifications. Despite significant progress, the inherent complexity of synthesis tasks and the interplay among intention, invention and adaptation limit its scope. A promising yet challenging avenue is the integration of concurrency to enhance synthesis algorithms. While some efforts have applied basic concurrency by parallelizing search spaces, more intricate synthesis scenarios involving interdependent subproblems remain unexplored. In this paper, we focus on string transformation as the target domain and introduce the first concurrent synthesis algorithm that enables asynchronous coordination between deductive and enumerative processes, featuring an asynchronous deducer for dynamic task decomposition, a versatile enumerator for resolving enumeration requests, and an accumulative case splitter for if-then-else condition/branch search and assembling. Our implementation, Synthphonia exhibits substantial performance improvements over state-of-the-art synthesizers, successfully solving 116 challenging string transformation tasks for the first time.

1 Introduction

Program synthesis plays a significant role in computer science, guiding the development of methods for automatically generating programs that fulfill given specifications. This field encompasses various methodologies, with traditional deductive synthesis relying on logical rules [11, 29, 30, 35], generic enumerative synthesis systematically exploring the space of candidate programs [1, 2, 5, 37], emerging learning-based synthesis leveraging the power of neural networks [3, 8, 31], and hybrid approaches merging these techniques [14, 15, 21, 26, 40].

Despite the progress, the scope of what can be synthesized remains constricted due to the inherent algorithmic complexity of the program synthesis task and challenges raised between intention, invention, and adaptation [16]. Amidst these hurdles, the incorporation of concurrency presents a promising avenue. Although concurrency is a well-established principle in computing that accelerates various computational tasks, its application to program synthesis has been limited—not because researchers overlooked its potential, but because synthesis procedures are notoriously difficult to parallelize. Some notable exceptions [14, 22, 23] divide a large search space into smaller ones and solve them in parallel. This form of concurrency is elementary, as each subproblem mirrors the others, adhering to identical specifications without necessitating inter-instance communication.

However, more complex synthesis scenarios, particularly those involving deductive top-down decomposition, present more challenges. In these cases, a major challenge is that, due to the nondeterministic inverse semantics of common operators, there is an explosion of decomposition choices. For example, a string can be decomposed into substrings in *quadratically* many ways for concatenation, and a set of input-output samples can be partitioned into conditional branches in *exponentially* many ways. In these scenarios, simple parallelization would help little and a higher degree of coordination and communication among concurrent components is needed.

The question we pose in this paper is whether concurrency can coordinate the decomposition of subtasks and have them solved appropriately, mitigating the exponential blow-up. This inquiry opens the door to several challenges that must be addressed. Firstly, the plethora of deductive rules presents a maze, as it is unclear when rules should be applied and which rules should be prioritized in the exploration process (also known as search tactics [33]). Secondly, the concurrent subproblems, dynamically generated through deduction, each bear unique specifications and demand resolution via enumeration. This necessitates a significant adaptation of traditional enumerative search techniques to accommodate a dynamic, multi-task environment.

In response to these challenges, this paper introduces the first synthesis algorithm that orchestrates deductive and enumerative synthesis processes concurrently. Our contributions include:

- *Asynchronous Deduction*, a framework that empowers designers to not only delineate the ways a synthesis task can be decomposed but also which subproblems should be solved by enumeration (e.g., a prefix-suffix decomposition should be triggered only when an enumerator finds a solution for the prefix). The deducer and the enumerator coordinate through an asynchronous request-response mechanism.
- *Accumulative Case-Splitting*, a technique which decouples condition search and term search. The two searches now can be done concurrently and the found terms and conditions are sent to a single pool and later assembled to form the final solution.
- *Versatile Enumeration*, a technique that resolves multiple, dynamically generated synthesis requests from external sources (e.g., a deducer). It performs enumeration and request handling simultaneously by harnessing the power of domain-specific *term dispatcher* data structures.
- An implementation of our algorithm, dubbed Synthphonia. Our experimental results showcase that this concurrent approach outperforms leading-edge synthesizers significantly and benefits from multithreading. Notably, Synthphonia solved 116 challenging string transformation tasks for the first time.

While this paper primarily focuses on a specific area, namely string transformation, as shown throughout the paper, the concurrent methodology we propose is new for synthesis and can be adapted to benefit a wide variety of synthesis tasks in other domains in the future.

The remainder of the paper is structured as follows: §2 elucidates the concept of concurrent synthesis and its inherent challenges through a concrete example. §3 delineates the formal framework of our approach. §4 details the asynchronous deduction system. §5 describes our methods for accumulative case-splitting and coordinated enumeration. §6 discusses some notable implementation details of Synthphonia. §7 reports our experimental design and the comprehensive results obtained. §8 compares our method with existing literature, followed by conclusion and future work discussion in §9.

2 Overview

In this section, we illustrate through a simple example the challenges faced by current synthesis methodologies and how our concurrent approach addresses these problems.

Example 2.1 (Address Reordering). Consider a string transformation task that purports to reorder the components of an address. It takes an address as input and produces a reordered address as output. Table 1 shows some sample input addresses from different countries in various formats and their corresponding outputs. Each input address typically includes street number/name, and the names of the city, region, and country. Some addresses also contain a separate room number and/or a postal code. The output rearranges the input to the following order: country, region, city, street number/name, and room number (if any), all delimited by a slash "/" .

One intuitive way to perform the transformation is to distinguish addresses involving room numbers (inputs in the table above the dash line) from others (inputs in the table below the dash line). The former case has 5 components and the latter case has less than 5 components. For each case, one can explicitly split the address into multiple components and re-assemble the components in the desired order. Using common string transformation operators, a solution can be constructed

Table 1. (Example 2.1) Sample input/output for reordering from different countries.

Input / String	Output / String
"456 Oak Lane, Unit 102, London, England, UK"	"UK/England/London/456 Oak Lane/Unit 102"
"101 Pine Avenue, Suite 5, New York, NY 10001, USA"	"USA/NY/New York/101 Pine Avenue/Suite 5"
"202 Birch Road, Apt. 23, Vancouver, BC V6B 1L8, Canada"	"Canada/BC/Vancouver/202 Birch Road/Apt. 23"
......	......
"1234 Elm St., Springfield, CA, USA"	"USA/CA/Springfield/1234 Elm St."
"5678 Maple Avenue, Oakville, ON K0E 0B2, Canada"	"Canada/ON/Oakville/5678 Maple Avenue"
"4321 Cedar Rd., Melbourne, VIC, Australia"	"Australia/VIC/Melbourne/4321 Cedar Rd."
......	......

as below:

$$
\begin{aligned}
&\textbf{if } \mathrm{in}_0.\texttt{split}(\text{``,␣''}).\texttt{length} == 5 \textbf{ then}\\
&\quad \mathrm{in}_0.\texttt{split}(\text{``,␣''})[-1] \mathbin{+\!\!+} \text{``/''} \mathbin{+\!\!+} \mathrm{in}_0.\texttt{split}(\text{``,␣''})[-2].\texttt{split}(\text{``␣''})[0] \mathbin{+\!\!+}\\
&\quad \text{``/''} \mathbin{+\!\!+} \mathrm{in}_0.\texttt{split}(\text{``,␣''})[-3] \mathbin{+\!\!+} \text{``/''} \mathbin{+\!\!+} \mathrm{in}_0.\texttt{split}(\text{``,␣''})[0] \mathbin{+\!\!+} \text{``/''} \mathbin{+\!\!+} \mathrm{in}_0.\texttt{split}(\text{``,␣''})[1]\\
&\textbf{else } \mathrm{in}_0.\texttt{split}(\text{``,␣''})[-1] \mathbin{+\!\!+} \text{``/''} \mathbin{+\!\!+} \mathrm{in}_0.\texttt{split}(\text{``,␣''})[-2].\texttt{split}(\text{``␣''})[0] \mathbin{+\!\!+}\\
&\quad \text{``/''} \mathbin{+\!\!+} \mathrm{in}_0.\texttt{split}(\text{``,␣''})[-3] \mathbin{+\!\!+} \text{``/''} \mathbin{+\!\!+} \mathrm{in}_0.\texttt{split}(\text{``,␣''})[0]
\end{aligned}
\tag{2.1}
$$

Though intuitive, the expression above is not the most compact one. For example, one can build a more succinct but trickier solution:

$$
\begin{aligned}
&\mathrm{in}_0.\texttt{split}(\text{``,␣''})[-1] \mathbin{+\!\!+} \text{``/''} \mathbin{+\!\!+} \mathrm{in}_0.\texttt{split}(\text{``,␣''})[-2].\texttt{split}(\text{``␣''})[0] \mathbin{+\!\!+}\\
&\text{``/''} \mathbin{+\!\!+} \mathrm{in}_0.\texttt{split}(\text{``,␣''})[-3] \mathbin{+\!\!+} (\text{``, /''} \mathbin{+\!\!+} \mathrm{in}_0).\texttt{split}(\text{``,␣''})[-5] \mathbin{+\!\!+}\\
&\text{``/''} \mathbin{+\!\!+} \mathrm{in}_0.\texttt{split}(\text{``,␣''})[-4]
\end{aligned}
\tag{2.2}
$$

2.1 Challenges for Existing Approaches

Despite significant advancements in string transformation synthesis over the past decade following the introduction of FlashFill [18], surprisingly, the straightforward example presented above remains unsolvable by any existing synthesizer to our knowledge, including CVC4 [6], Duet [26], FlashFill++ [9], and Probe [5]. This is due to several critical challenges, which we outline below.

Rich Grammar. Real-world synthesis tasks usually require rich grammars. String transformation, as an example, often requires many non-standard operations beyond the standard theory of Strings [10], such as negative indices, loops, date and time conversions, numerical manipulations, etc. Most of these features cannot be expressed in the standard SyGuS interchange format (SyGuS-IF) which is adopted by solvers such as Duet [26] and Probe [5]. As a concrete example, Duet lacks the capability of defining negative-index operations, which are necessary for both solutions to Example 2.1.

Efficient Concurrency. Decomposing a synthesis task into subtasks and solving them independently is a well-established approach in deductive synthesis. However, a significant challenge arises from the nondeterministic inverse semantics of common operators, leading to a combinatorial explosion of decomposition choices. For instance, a string can be decomposed into substrings in quadratically many ways for concatenation, and a set of input-output examples can be partitioned into conditional branches in exponentially many ways. These complexities are often attributed to the inherent nature of the algorithm. However, the potential benefits of concurrency—specifically, coordinating the decomposition of subtasks and solving them efficiently—are overlooked in existing methods. For example, if a prefix-suffix decomposition is only triggered when an enumerator finds a solution for the prefix, the need to consider quadratically many concatenation options can be effectively eliminated.

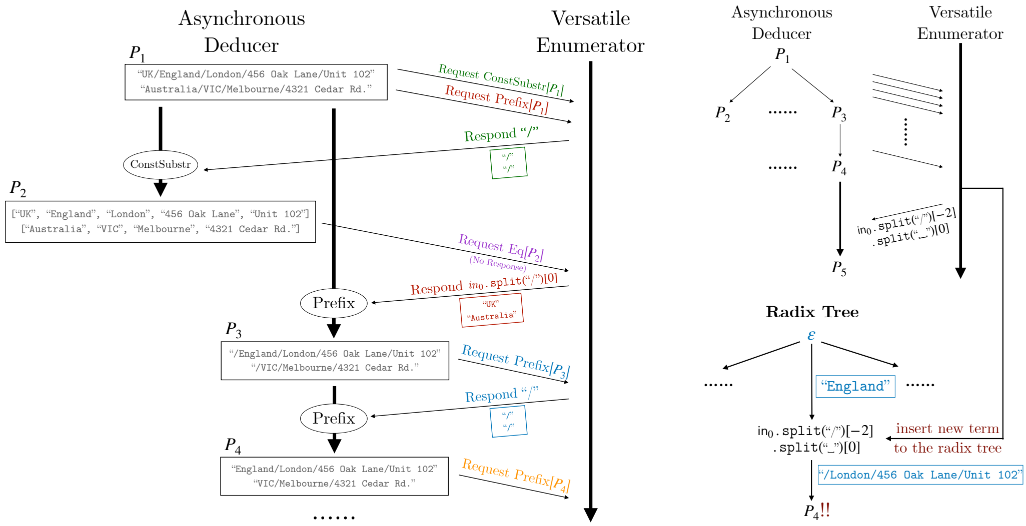
Balanced Scalability and Generality. Another fundamental challenge for program synthesis lies in the tension between scalability and generality. Even a very simple synthesis task corresponds to a gigantic search space, exceeding the capability of generic enumerative or deductive synthesis engines. For example, as we will see soon in the next section, a typical string transformation grammar consists of dozens of operators, and there are astronomically many expressions of similar size to solutions (2.1) and (2.2). Therefore, generic enumerative methods like PROBE [5], though generally applicable, suffer the exponential space explosion and fail to solve Example 2.1. In contrast, FLASHFILL++ [9] as a specialized synthesizer for string transformation, mitigates the problem by employing a hard-coded, regular-expression-based grammar which supports all the non-standard operations mentioned above. However, it enforces a stringent order in which the operations can be applied, which excludes both solutions (2.1) and (2.2).

Customizable Deduction. Decomposing a synthesis task into subtasks by deduction has been a widely accepted approach and has achieved success in numerous domains. Nonetheless, as noted in the introduction, top-down decomposition calls for carefully designed search tactics that have to be provided by domain experts. For example, a reasonable way to deduce Example 2.1 would decompose the problem in a way that the output "`USA/CA/Springfield/1234 Elm St.`" is split into two subproblems with outputs "`USA`" and "`CA/Springfield/1234 Elm St.`", respectively, using a delimiter "`/`". However, there are thousands of different ways to split the output with different delimiters—a generic deductive rule would simply state that "decompose the problem using a delimiter, get a solution for each subproblem, then concatenate these solutions using the delimiter." How can the system prioritize the deduction mentioned above via a more specific rule which specifies that the delimiter must be a simple constant and the first subproblem should be simply solvable by enumeration? All of the existing approaches, including DUET [26] and FLASHFILL++ [9], fail to embed such specific search tactics into their solvers. In particular, the deducing methods in DRYADSYNTH and DUET are restricted to witness functions of operators while FLASHFILL++ allows the DSL designer to create an extended form of witness functions called *cuts*. However, none of these methods allow domain experts to design the prioritization of deduction needed in our example.

Efficient Parallelization. As noted in the introduction, parallelization is widely recognized as a means to speed up computational tasks. Unfortunately, although some synthesizers offer limited parallelism support, they typically either run identical subproblems in parallel [14, 22, 23] or lack specialization for synthesis problem-solving [4]. These approaches fail to yield significant performance improvements for the synthesis problems examined in this paper.

2.2 Our Approach

Driven by the aforementioned challenges, in this paper, we present a concurrent approach designed to fully harness the potential of both deductive and enumerative synthesis techniques. On the one hand, to support rich grammars and customizable deduction, the deducer must support a flexible deduction system which describes not only abundant ways of decomposing synthesis tasks but also what guidance is needed to start a decomposition (e.g., a delimiter is needed for splitting). The guidance per se can be viewed as a simple synthesis problem and solved by the enumerator. On the other hand, to ensure efficient exploration of countless deduction paths, the communication between the deducer and the enumerator must be concurrent—the deducer should try multiple decompositions simultaneously, and the enumerator should be able to provide guidance for multiple decompositions. Below let us see how the deducer and the enumerator in our approach collaborate concurrently to solve Example 2.1 and produce Equation 2.2 as a solution. At the end of the section, we present an overview of our synthesis framework, which involves another accumulative case



(a) Communication between deducer and enumerator.

(b) Request handling.

Fig. 1. Enumeration and Deduction Process for Example 2.1.

splitter component for handling `if-then-else` operators, which are not present in the current simple example.

2.2.1 Asynchronous Deducer. Basically, the deducer splits the synthesis task into subtasks, each aiming at synthesizing a component of the expected output (e.g., country, region, etc.), and concatenates them to form a final solution. How does the deducer know which substrings of the expected output are synthesizable components? It interacts with the enumerator. The detailed communication between the deducer and enumerator used to solve Example 2.1 is shown in Figure 1a. In the figure, box P_1 represents the original synthesis problem and other boxes P_2, P_3, and P_4 each represent distinct subproblems. For simplicity, in each box, we just represent the problem using the expected output for two sample inputs from Table 1, namely "`456 Oak Lane, Unit 102, London, England, UK`" and "`4321 Cedar Rd., Melbourne, VIC, Australia`". The horizontal arrows indicate the message exchanges between the deducer and the enumerator.

At the beginning, based on the rich deductive rules which we will present in §4, the deducer determines that the problem P_1 can be split in two ways: either multiple pieces delimited by a constant, or two pieces—a prefix and a suffix. Therefore, the deducer initiates two requests simultaneously, namely ConstSubstr$[P_1]$ and Prefix$[P_1]$. Intuitively, the former one simply asks for a separator: "Please provide an expression that always evaluates to a substring of the expected output." The latter one asks for a synthesizable prefix: "Please provide an expression that always evaluates to a prefix of the expected output."

On the enumerator side, it maintains a pool of pending requests and solves them simultaneously, as we will discuss shortly in §2.2.2. Whenever a solution for a request is found, the enumerator responds with that expression to the deducer. In the concrete example shown in Figure 1a, the enumerator first responds to request ConstSubstr$[P_1]$ with a simple separator "`/`". Based on this received solution, P_1 can be split into a list of strings like ["`UK`", "`England`", "`London`", "`456 Oak Lane`", "`Unit 102`"]. Thus the deducer creates a corresponding subproblem P_2—once P_2 is solved, P_1 can be assembled by applying `str.join` to the solutions of P_2 with "`/`". Unfortunately, solving

P_2 turns out to be a dead end. Among many attempts, the deducer sends a request Eq[P_2], asking the enumerator to generate an expression to solve P_2. However, it is not an easy task because P_2 has reordered the segments from the original input, and the solution must assemble the segments explicitly using multiple operations. So it takes nearly infinite time for the enumerator to respond to P_2's request.

However, on a different path, the deducer receives the response for the other Prefix[P_1] request: expression in_0`.split("/")[0]` always evaluates to a prefix of P_1. What remains is to synthesize the corresponding suffix, which is denoted as task P_3. Now, similar to the previous case of P_1, to solve P_3, the deducer makes a prefix request Prefix[P_3]. The enumerator, this time, finds the same solution "/" as the delimiter, which yields a new subtask P_4. The concurrent synthesis process continues so forth until the synthesis task is fully solved. Finally, solution (2.2) can be returned using the joint force of deduction and enumeration.

For simplicity, a lot of possible deduction branches for Example 2.1 are omitted in Figure 1a; however, in reality, the number of top-down deductive branches grows exponentially. To deduce synthesis tasks at scale, we allow thousands of requests from the deducer to be handled at the same time using a single enumerator. We call this technique *asynchronous deduction* because numerous deducer requests are handled asynchronously, and deduction can actually be viewed as an asynchronous program which only proceeds once its request gets responded.

2.2.2 Versatile Enumerator. The workhorse for the asynchronous deduction framework is a *versatile enumerator* which solves a large number of synthesis tasks simultaneously. Recall that the enumerator can remember numerous requests from the deducer and respond to them immediately once an expression that satisfies the requests is discovered. The underlying mechanism of the enumerator is depicted in Figure 1b. For each type of request from the deducer, the enumerator maintains a specific data structure to store the relationship between the requests and enumerated expressions.

Here, for "Prefix" requests, the enumerator employs a radix tree (a compact version of a prefix tree) to store all requests from the deducer and all enumerated expressions. Each request or expression is indexed by its output specification or its evaluation, respectively. When the enumerator receives a "Prefix" request from the deducer, it first searches the radix tree and responds with all expressions that already satisfy the constraint. If no expression satisfies the constraint, the enumerator will insert the request into the radix tree. As shown in Figure 1b, request P_4 is added to the radix tree when P_4 requests the enumerator. When the enumerator generates a new expression, it adds the expression to the radix tree and checks if there are any requests that this expression satisfies. It then immediately responds to all such requests with that expression. In Figure 1b, the expression in_0`.split(",␣")[-2].split("␣")[0]` is added to the radix tree, satisfying request P_4. Then the enumerator responds to P_4, causing it to be further reduced into subproblem P_5.

For other types of requests, various data structures are employed to ensure efficiency. We have designed data structures for five different kinds of requests: Eq, ConstSubstr, Prefix, Len and Contains. Because all such data structures are used to efficiently look up the corresponding requests of a given expression, we generalize all these data structures into an abstract data type called *term dispatcher* in our versatile enumerator. With term dispatcher, the enumerator can easily store thousands of requests and respond to them with efficiency.

2.2.3 Overall Architecture. Now, we introduce the overall architecture of our synthesis framework as shown in Figure 2. The framework first relaxes the original input-output example set into different subsets based on a strategy defined by a problem relaxer, and then solves each relaxed subproblem using a worker. The main novelty of our architecture, which we call *accumulative case-splitting* (ACS), is that each worker will simultaneously and independently search: 1) a partial

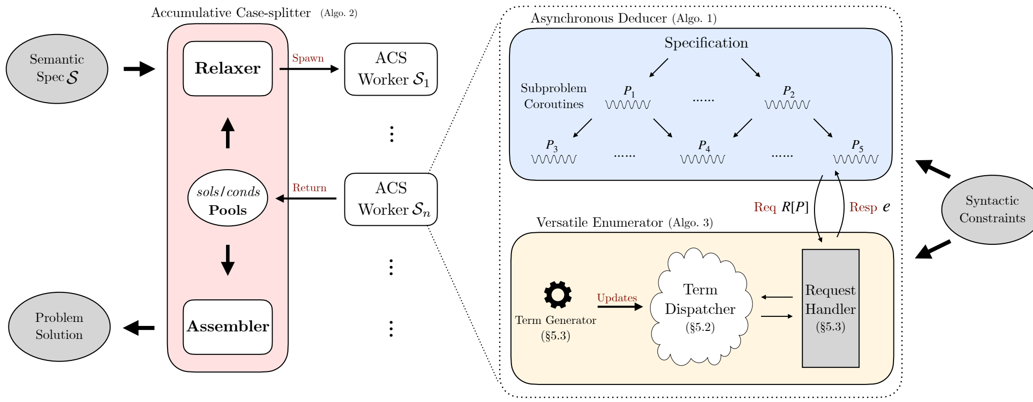


Fig. 2. Overview of Synthphonia.

solution, i.e., a solution for the subset of examples; and 2) one or more conditions that can split the subset. The partial solutions and the conditions found by all workers will be stored in shared pools and then combined into a single solution by a solution assembler. Note that our accumulative case-splitting can be understood as a concurrent version of condition abduction. Traditional condition abduction techniques [2, 25] also separate term search, condition search, and decision tree learning, but interleave these tasks in a fixed order. This rigidity may result in too conservative case-splitting (leading to poor performance) or too aggressive case-splitting (causing overfitting). In contrast, accumulative case-splitting is more flexible and performs these tasks in a completely concurrent and independent manner.

Each ACS worker comprises two components: an asynchronous deducer and a versatile enumerator, as we just introduced above. The deducer initializes the top-down deductive search by recursively splitting the given specification into a range of subproblems and assigns them to different coroutines. The enumerator spawned by the deducer constantly enumerates expressions and maintains a term dispatcher with enumerated expressions and pending requests, and a *request handler* to process requests from the deducer. The subproblem coroutines are solved through interacting with the versatile enumerator. The asynchronous deducer combines all the results from the enumerator to generate a solution for the current worker.

3 Preliminaries

In this section, we provide a formal description of the synthesis problem addressed in this paper.

Definition 3.1 (Background Theory). A background theory is defined as a tuple $\mathcal{T} = (\Sigma, \tau, [\![\cdot]\!])$, where Σ denotes a finite set of symbols, $\tau : \Sigma \to \mathbb{N}$ represents an arity function, and $[\![\cdot]\!]$ is the semantics for the symbols. In particular, a symbol x is considered a constant if $\tau(x) = 0$, or considered an operator if $\tau(x) > 0$. $[\![\cdot]\!]$ will associate each constant with a specific value, and each operator with an operation on the values.

We use *expression grammar* to encompass the syntactical aspect of a synthesis problem.

Definition 3.2 (Expression Grammar). Consider a signature σ. An expression grammar $\mathcal{G}$ with respect to σ can be described as a tuple $(\mathcal{T}, \mathcal{N}, \mathcal{P})$, where $\mathcal{T}$ is the background theory, $\mathcal{N}$ represents a set of non-terminals, and $\mathcal{P}$ comprises a set of production rules. Each production rule is either $N \to f(N_1, \ldots, N_{\tau(f)})$, where $N, N_1, \cdots \in \mathcal{N}$ are non-terminals and $f \in \Sigma$ is a symbol in $\mathcal{T}$, or

$$
\begin{array}{rcl}
S & \rightarrow & S +\!\!+ S \\
 & | & S[I] \\
 & | & \texttt{str.replace}(S, S, S) \\
 & | & \texttt{str.substr}(S, I, I) \\
 & | & \texttt{str.from\_int}(I) \\
 & | & \texttt{str.from\_float}(F) \\
 & | & \texttt{str.uppercase}(S) \\
 & | & \texttt{str.lowercase}(S) \\
 & | & \texttt{str.filter\_char}(S, C) \\
 & | & L[I] \\
 & | & \texttt{list.join}(L, S) \\
 & | & \texttt{month.fmt}[\mathit{Str}](I) \\
 & | & \texttt{weekday.fmt}[\mathit{Str}](I) \\
 & | & \texttt{time.fmt}[\mathit{Str}](T) \\
 & | & \texttt{ITE}(B, S, S) \\
 & | & \ldots\ \text{(Constants)}\ \ldots \\
 & | & \ldots\ \text{(Variables)}\ \ldots
\end{array}
\qquad
\begin{array}{rcl}
I & \rightarrow & \texttt{len}(S) \\
 & | & \texttt{str.count}(S, S) \\
 & | & \texttt{int.+}(I, I) \\
 & | & \texttt{int.-}(I, I) \\
 & | & \texttt{int.from\_str}(S) \\
 & | & \texttt{int.from\_float}(F) \\
 & | & \texttt{date.year}(D) \\
 & | & \texttt{date.month}(D) \\
 & | & \texttt{date.day}(D) \\
 & | & \texttt{date.weekday}(D) \\
 & | & \texttt{str.indexof}(S, S, I) \\
 & | & \texttt{ITE}(B, I, I) \\
 & | & \ldots\ \text{(Constants)}\ \ldots \\
L & \rightarrow & \texttt{str.split}(S, S) \\
 & | & \texttt{list.map}[S \rightarrow S](L) \\
 & | & \texttt{list.filter}[S \rightarrow B](L) \\
C & \rightarrow & \texttt{charset.Ll} \mid \texttt{charset.Lu} \\
 & | & \texttt{charset.L} \mid \texttt{charset.N} \\
 & | & \texttt{charset.LN}
\end{array}
\qquad
\begin{array}{rcl}
B & \rightarrow & \texttt{int.>}(I, I) \\
 & | & \texttt{int.=}(I, I) \\
 & | & \texttt{int.>=}(I, I) \\
 & | & \texttt{str.prefix}(S, S) \\
 & | & \texttt{int.contains}(S, S) \\
 & | & \texttt{int.suffix}(S, S) \\
F & \rightarrow & \texttt{float.from\_str}(S) \\
 & | & \texttt{float.+}(F, F) \\
 & | & \texttt{float.-}(F, F) \\
 & | & \texttt{float.shl10}(F, I) \\
 & | & \texttt{float.floor}(F, F) \\
 & | & \texttt{float.ceil}(F, F) \\
 & | & \texttt{float.round}(F, F) \\
 & | & \ldots\ \text{(Constants)}\ \ldots \\
D & \rightarrow & \texttt{date.parse}(S) \\
T & \rightarrow & \texttt{time.parse}(S) \\
 & | & \texttt{time.floor}(T, T) \\
 & | & \texttt{time.*}(T, I) \\
 & | & 1 \mid 60 \mid 3600
\end{array}
$$

Fig. 3. A grammar for string manipulating programs. (The black part is the core grammar; the green part is an extension used by Duet; the blue part is an extension for loops; the red part is an extension for date/time semantics; brackets $[\cdot]$ are used to indicate arguments that cannot be easily enumerated.)

$N \rightarrow v$, in which v is an input variable whose value changes based on the context. We denote the set of all expressions generated by $\mathcal{G}$ as $[\![\mathcal{G}]\!]$, which is defined as $[\![\mathcal{G}]\!] = \{e \mid N \xrightarrow[\mathcal{P}]{}^{*} e, N \in \mathcal{N}\}$.

We also extend the semantics $[\![\cdot]\!]$ to interpret the input variables. In this paper, we simply denote each input variable as in_0, in_1, ... and assign values to the input variables using an *input vector* $\boldsymbol{i}$, which assigns input variables in_0, in_1, ... to the value $\boldsymbol{i}_0, \boldsymbol{i}_1$, ... The new semantics with input vector $\boldsymbol{i}$ associated is denoted as $[\![\cdot]\!]_{\boldsymbol{i}}$.

Example 3.3. SYNTHPHONIA as a synthesizer specialized for string transformation, uses a grammar for string expressions as shown in Figure 3. This grammar consists of eight non-terminals S, I, L, B, C, F, D, T, corresponding to eight types of expressions *Str*, *Int*, *List*, *Bool*, *CharSet*, *Float*, *Date* and *Time*, respectively. The production rules for each non-terminal are shown in Figure 3. Each non-terminal is associated with a type. Note that this is a very rich grammar which supports not only standard string operations such as `str.concat`, `str.split` or `str.replace`, but also special operations for date and time conversions, as well as numerical operations such as `int.+`, `float.from_str` or `float.ceil`.

In this paper, we solve a class of synthesis problems which describes the syntactical aspect using expression grammars and characterizes the expected behavior of the target expression using examples. We call this class *inductive* SYGUS *problems*, as defined below.

Definition 3.4 (Inductive SYGUS *Problem).* An Inductive SYGUS Problem can be represented as a tuple $P = (\mathcal{G}, \mathcal{S})$, where $\mathcal{G}$ is an expression grammar, and $\mathcal{S}$ is the collection of input-output examples represented as $\boldsymbol{i} \mapsto o$, where $\boldsymbol{i}$ is an input vector and o is the anticipated output. A solution to the SYGUS problem is an expression $e \in [\![\mathcal{G}]\!]$ that satisfies the following condition:

$$\bigwedge_{(\boldsymbol{i} \mapsto o) \in \mathcal{S}} [\![e]\!]_{\boldsymbol{i}} = o$$

In the paper, we use $\mathrm{dom}(\mathcal{S})$, or simply $\mathcal{I}$, to denote set of all the input vectors of $\mathcal{S}$, or domain of $\mathcal{S}$, formally $\mathrm{dom}(\mathcal{S}) = \{\boldsymbol{i} \mid \boldsymbol{i} \mapsto o \in \mathcal{S}\}$. We also use $\mathcal{S}|_I$ to denote the subset of $\mathcal{S}$ which domain is the input vector set I, i.e. $\mathcal{S}|_I = \{\boldsymbol{i} \mapsto o \in \mathcal{S} \mid \boldsymbol{i} \in I\}$.

4 Asynchronous Deduction

In this section, we elaborate on the asynchronous deducer part of our approach. We first introduce a deduction system in which traditional deductive rules are enriched to indicate when and what requests to make to the enumerator. Then we introduce the adaptation needed for conditions, and present the concurrent algorithm that runs the deducer.

4.1 Requests

A salient feature of our deduction system is its asynchronous communication with an enumerator via requests and responses. Intuitively, a *request* denotes a question posed by a deducer to an enumerator at a specific time, asking for a solution to a subproblem.

Definition 4.1 (Request). An enumerator request is of the form $\mathrm{Request}(\mathcal{G}, \mathcal{S}, R)$, where $(\mathcal{G}, \mathcal{S})$ forms the original inductive SyGuS problem to be solved by the deducer, and R is a *subproblem functor* that converts the original, inductive specification $\mathcal{S}$ to the specification for a subproblem denoted as $R(\mathcal{S})$ (see some examples below). A solution (or response) to a request is an expression $e \in [\![\mathcal{G}]\!]$ that satisfies $R(\mathcal{S})$.

Example 4.2 (Subproblem Functors for Strings). In this paper, specialized for the expressive string grammar displayed in Figure 3, we consider five subproblem functors: Eq, ConstSubStr, Prefix, Len and Contains. Each subproblem functor can be characterized as a logical formula regarding the target expression e and the original inductive specification $\mathcal{S}$:

$$\mathrm{Eq}(\mathcal{S}) \coloneqq \bigwedge_{(\boldsymbol{i}\mapsto o)\in\mathcal{S}} [\![e]\!]_{\boldsymbol{i}} = o \qquad \mathrm{Prefix}(\mathcal{S}) \coloneqq \bigwedge_{(\boldsymbol{i}\mapsto o)\in\mathcal{S}} [\![e]\!]_{\boldsymbol{i}}\ \texttt{prefixof}\ o$$

$$\mathrm{ConstSubstr}(\mathcal{S}) \coloneqq \exists c. \bigwedge_{(\boldsymbol{i}\mapsto o)\in\mathcal{S}} [\![e]\!]_{\boldsymbol{i}} = c \wedge c\ \texttt{substrof}\ o \qquad \mathrm{Contains}(\mathcal{S}) \coloneqq \bigwedge_{(\boldsymbol{i}\mapsto o)\in\mathcal{S}} [\![e]\!]_{\boldsymbol{i}}\texttt{.contains}(o)$$

$$\mathrm{Len}(\mathcal{S}) \coloneqq \bigwedge_{(\boldsymbol{i}\mapsto o)\in\mathcal{S}} \texttt{len}([\![e]\!]_{\boldsymbol{i}}) = o$$

In this paper, we simply use $R[\mathcal{G}, \mathcal{S}]$ or $R[P]$ (where $P = (\mathcal{G}, \mathcal{S})$ is an inductive SyGuS problem) to denote $\mathrm{Request}(\mathcal{G}, \mathcal{S}, R)$. For a request r, we use $r.R$, $r.\mathcal{G}$ and $r.\mathcal{S}$ to denote the components R, $\mathcal{G}$, $\mathcal{S}$ associated with r, respectively. We also abuse the notation and use $R[\mathcal{G}, \mathcal{S}]$ to represent the formal specification of the subproblem represented by $R[\mathcal{G}, \mathcal{S}]$. Moreover, we use $e \models_E R[\mathcal{G}, \mathcal{S}]$ to indicate that an expression e is found by an enumerator as the response to request $R[\mathcal{G}, \mathcal{S}]$.

4.2 Asynchronous Deduction Rules

Based on our notion of requests, we can now define the general form of deduction rules used in our synthesis framework, and present the deduction rules used in string transformation synthesis.

Definition 4.3 (Asynchronous Deduction Rule). An *asynchronous deduction rule* is an inference rule in the following form:

$$\frac{(e \models_E R[\mathcal{G}, f(\mathcal{S})]) \rtimes \Big(q(\mathcal{S}, e), \quad e_1 \models \boldsymbol{p}_1(\mathcal{S}, e), \quad \ldots, \quad e_n \models \boldsymbol{p}_n(\mathcal{S}, e)\Big)}{\gamma(e, e_1, \ldots, e_n) \models \mathcal{S}} \quad c(\mathcal{S})$$

where the condition part of the rule is represented by $c(\mathcal{S})$, which is a condition specifying under which condition this rule can be applied. This is primarily used to test if the output example

$$\textsc{Eq}\quad \frac{e \models_E \mathrm{Eq}[\mathcal{G}, \mathcal{S}]}{e \models \mathcal{S}} \qquad \textsc{S-Prefix}\quad \frac{\Big(e \models_E \mathrm{Prefix}[\mathcal{G}, \mathcal{S}]\Big) \rtimes \Big(e_1 \models \texttt{str.substr}(\mathcal{S}, \texttt{str.len}(\llbracket e \rrbracket_{\mathcal{I}}), -1)\Big)}{e +\!\!+\, e_1 \models \mathcal{S}}$$

$$\textsc{S-ConstSubstr}\quad \frac{\Big(e \models_E \mathrm{ConstSubstr}[\mathcal{G}, \mathcal{S}]\Big) \rtimes \Big(e_1 \models \mathcal{S}.\texttt{split\_once}(\llbracket e \rrbracket_{\mathcal{I}})[0], \quad e_2 \models \mathcal{S}.\texttt{split\_once}(\llbracket e \rrbracket_{\mathcal{I}})[1]\Big)}{e_1 +\!\!+\, e +\!\!+\, e_2 \models \mathcal{S}}$$

$$\textsc{L-Map}\quad \frac{\Big(e \models_E \mathrm{Len}[\mathcal{G}, \texttt{len}(\mathcal{S})]\Big) \rtimes \Big(e_f \models \Big\{\llbracket e \rrbracket_{\boldsymbol{i}}[k] \mapsto o[k] \,\Big|\, \boldsymbol{i} \mapsto o \in \mathcal{S}, 0 \le k < \texttt{len}(\mathcal{S}[\boldsymbol{i}])\Big\}\Big)}{\texttt{list.map}[e_f](e) \models \mathcal{S}}$$

$$\textsc{L-Filter}\quad \frac{\begin{gathered}\Big(e \models_E \mathrm{Contains}[\mathcal{G}, \mathcal{S}[0]]\Big) \rtimes \\ \Big(\bigwedge_{\boldsymbol{i} \mapsto o \in \mathcal{S}} o.\texttt{subseqof}(\llbracket e \rrbracket_{\boldsymbol{i}}), \quad e_f \models \Big\{\llbracket e \rrbracket_{\boldsymbol{i}}[k] \mapsto o.\texttt{contains}(\llbracket e \rrbracket_{\boldsymbol{i}}[k]) \,\Big|\, \boldsymbol{i} \mapsto o \in \mathcal{S}, 0 \le k < \texttt{len}(\llbracket e \rrbracket_{\boldsymbol{i}})\Big\}\Big)\end{gathered}}{\texttt{list.filter}[e_f](e) \models \mathcal{S}}$$

Fig. 4. Selected Asynchronous Deduction Rules for String. See Appendix A for full list.

$\mathcal{S}$ matches the type of the rule and the solution generated by this rule can be expressed in the grammar.

The premise part is split by a special $\rtimes$ connective into two parts. The first part represents an asynchronous request to an external enumerator where $R[\mathcal{G}, f(\mathcal{S})]$ is a request and e is a response to the request as defined earlier, where f is a function adapting the original $\mathcal{S}$ for the subproblem (can be simply the identity function). The response to the request (i.e., the solution e) serves as a guard which enables a deduction following the second part of the premise. In the second part of the premise, $q(\mathcal{S}, e)$ is some additional conditions for this deduction rule restricting e (typically ignored for most rules); and $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$ are subproblem functors. Each functor $\boldsymbol{p}_i$ takes the specification $\mathcal{S}$ and an expression e from the enumerator and produces a new input-output example set S_i for the i-th subproblem.

The conclusion part states combining $e, e_1, \ldots, e_n$ can generate a solution of $\mathcal{S}$, where γ is a combinator specified by the rule which is used to generate such a solution.

As such, we can use a tuple $(c, R, f, q, \boldsymbol{p}, \gamma)$ to represent an asynchronous deduction rule, where $\boldsymbol{p}$ is the vector of all subproblem functors $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$.

Figure 4 shows all asynchronous deduction rule designs for synthesizing string transformations using the grammar from Example 3.3. For simplicity, we omit all signature and grammar in the inductive SyGuS problem and all conditions $c(\mathcal{S})$ for all rules. We use prefix S- and L- to denote the type of the input-output examples the rule is applied to. All rules with prefix "S-" must be applied to input-output examples $\mathcal{S}$ of *Str* type, whereas all rules with prefix "L-" must be applied to specifications of *List* type.

For simplicity, we also allow operators to be applied into specification, e.g. $\texttt{len}(\mathcal{S}) = \{\boldsymbol{i} = \texttt{len}(o) \mid \boldsymbol{i} \mapsto o \in \mathcal{S}\}$ is an example-based specification that maps every inputs vector $\boldsymbol{i}$ into the length of output $\texttt{len}(o)$. We also use $\mathcal{I}$ to denote the set of all input vectors in $\mathcal{S}$.

Example 4.4 (Rule S-Prefix). Consider rule S-Prefix in Figure 4. The rule follows the template from Definition 4.3 and can be represented by tuple $(c, R, f, q, \boldsymbol{p}, \gamma)$. For example, S-Prefix can be written as $(c, \text{Prefix}, f, \text{true}, \boldsymbol{p}, +\!\!+)$, where $c(\mathcal{S})$ is trivial and omitted in Figure 4; it just checks that all of $\mathcal{S}$'s outputs have string type; $f(\mathcal{S}) = \mathcal{S}$ simply keeps the original specification $\mathcal{S}$ unchanged; and $\boldsymbol{p}$ just contains a single subproblem functor which generates a new set of input-output examples $S = \texttt{str.substr}(\mathcal{S}, \texttt{str.len}(\llbracket e \rrbracket_{\mathcal{I}}), -1)$ as specification for the subproblem. Intuitively, the rule can be applied if operator $+\!\!+$ is available in $\mathcal{G}$. Upon application, the rule first makes a request to the enumerator asking for an expression that evaluates to a prefix of the expected output. When an expression e is returned from the enumerator, the rule will deduce the original problem to a single subproblem: synthesizing an expression whose outputs can be concatenated to the output of e to form the expected output. Once the subproblem is solved and a solution e_1 is obtained, the concatenation $e +\!\!+ e_1$ forms a solution for $(\mathcal{G}, \mathcal{S})$.

Remark: Our asynchronous deduction rules are different from those used in state-of-the-art deductive systems (e.g., FlashFill++ [9]) in several aspects. On the one hand, our rules are more generalizable, without the need for special customizations like a layer grammar or cuts for restricting the witness function. On the other hand, our rules expect more guidance from the user on how to coordinate between deduction and enumeration for the best performance. For example, rule S-ConstSubstr gives a hint on what the enumerator should solve and how the response determines where the original problem should be split. Other rules for list-related deductions like L-Map and L-Filter also indicate how and in what order these operations' parameters should be synthesized.

4.3 Adaptation for Conditions

Readers may have noticed that Figure 4 misses a key rule for the ITE operator. It is quite natural to extend Figure 4 with a similar S-Ite to let the deducer find a condition that splits the example set $\mathcal{S}$ into two distinct sets, and solves the two subproblems separately. However, as $\mathcal{S}$ can be split arbitrarily, the rule could easily produce exponentially many subproblems to the size of $\mathcal{S}$. How to harness the decomposition and search efficiently? This has been a known open problem for deductive synthesis [33]. Another straightforward alternative is to solve each input-output example independently and then to combine the results using the ITE operator. However, this naïve approach tends to produce large, overfit solutions rather than the optimal solution for the problem. Other methods [2, 25] based on condition abduction have also been proposed. However, as discussed in Section 2.2.3, these methods fail to offer enough flexibility for string transformation synthesis due to their fixed-ordered condition abduction process.

To address this challenge, we introduce the concept of *accumulative case-splitting*. The insight is that condition search should be decoupled from other term search and be agnostic to how the synthesis problem will be decomposed and whether it can be solved. In other words, conditions and terms should be searched independently and then assembled into a solution (see more details in Section 5.1).

In this setting, as we will show shortly in §5, there will be multiple concurrent asynchronous deducers, each working for a distinct spec $\mathcal{S}$. The deducers all contribute to a global pool of conditions and partial solutions for later solution assembling. To this end, we introduce a new subproblem functor (and the corresponding request) called $\text{Cond}(\mathcal{S})$ with the aim of splitting $\mathcal{S}$:

$$\text{Cond}(\mathcal{S}) \coloneqq \Big(\bigvee_{(\boldsymbol{i} \mapsto o) \in \mathcal{S}} \llbracket e \rrbracket_{\boldsymbol{i}} = \textbf{true} \Big) \wedge \Big(\bigvee_{(\boldsymbol{i} \mapsto o) \in \mathcal{S}} \llbracket e \rrbracket_{\boldsymbol{i}} = \textbf{false} \Big) \tag{4.1}$$

Different from other functors/requests presented in Example 4.2, $\text{Cond}(\mathcal{S})$ is not associated will any rule and sent to the enumerator upfront before any deduction. We next illustrate the asynchronous deduction algorithm with accumulative case-splitting.

4.4 The Algorithm

Given a set of rules following the template described above, we now demonstrate how the asynchronous deducer collaborates with an enumerator, as outlined in Algorithm 1. The algorithm takes an inductive SyGuS problem $(\mathcal{G}, \mathcal{S})$ as input and returns both a solution to the problem and a sequence of conditions discovered during the search. These conditions are used in our accumulative case-splitting framework to combine solutions effectively. For clarity, we model this process using Rust-style asynchronous programming primitives such as async-await and generators: a generator continuously yields a stream of conditions and finally returns a solution to the input problem.

The Solve procedure, as the entry of the algorithm, creates the corresponding enumerator *Enum* for the current deducer, which will be used to solve subproblem requests concurrently in a separate coroutine. Then, Solve first sends a $\mathrm{Cond}[\mathcal{G}, \mathcal{S}]$ request to *Enum*, expecting to receive a stream $cond_{\mathrm{STRM}}$ of condition expressions that splits the requested input-output examples. Solve will also start deducing the input SyGuS problem by invoking the Deduce subprocedure, which returns a future sol_{F} of the problem solution. As an ACS worker, Solve will contribute a stream of conditions *cond* as received from *Enum* along the process, until a solution *sol* is generated by the Deduce subprocedure (line 5-8).

The Deduce subprocedure starts by creating a memory location $result_{\mathrm{C}}$ to store the results generated by coroutines created by Deduce. We use the term *one-shot channel* to denote that this location used for inter-coroutine communication can be updated only once. Deduce runs a loop until $result_{\mathrm{C}}$ is set to an expression that will be returned as the solution. The loop considers each rule in the rule set $\mathfrak{R}$, which comprises premise $\mathfrak{p}$, conclusion $\mathfrak{q}$, and condition $\mathfrak{c}$. Note that we use blue color to denote components from the deduction rule and use Fraktur letters to denote variables representing these components. Then the condition $\mathfrak{c}$ for the rule is interpreted under the current grammar $\mathcal{G}$ and specification $\mathcal{S}$. If the condition is evaluated to be true, a coroutine ApplyRule will be created to interpret the rule.

The ApplyRule subprocedure tries to recursively solve all subproblems from the premise $\mathfrak{p}$ and combine the solutions to form a solution for the conclusion $\mathfrak{q}$. There are three possible cases of $\mathfrak{p}$:

- If $\mathfrak{p}$ involves a request of the form $e \models_E R[\mathcal{G}, \mathcal{S}']$, the algorithm generates the request $R[\mathcal{G}, \mathcal{S}']$, sends it to the corresponding enumerator *Enum*, which provides a stream of solutions of this request as response. Whenever a solution is received, the subprocedure applies the solution to the rest of the premise $\mathfrak{p}'$, and continues with ApplyRule in another coroutine (lines 17–20).
- If $\mathfrak{p}$ requires a standard subproblem $\mathcal{S}''$ to be solved by deduction, the subprocedure checks if $\mathcal{S}''$ has the same input vector as the target specification $\mathcal{S}$. If so, the subproblem can be solved by recursively calling Deduce with the same enumerator *Enum*. Otherwise, a new thread must be spawned running the Solve subprocedure. As we restrict every deducer to work in the same threads as the enumerator, Solve must be run on a new thread. Once a solution *sol* to the subproblem is found, it continues by recursively calling ApplyRule for the rest of the premise $\mathfrak{p}'$ (lines 21–26).
- Finally, if $\mathfrak{p}$ is empty, that means all subproblems have been solved and applied to the conclusion $\mathfrak{q}$. The subprocedure can simply take the combined expression $\mathfrak{e}$ and set into $result_{\mathrm{C}}$ (lines 27–29).

For a single Deduce procedure, there are potentially tens or hundreds of concurrent ApplyRule invocations. Once a single instance of ApplyRule sets $result_{\mathrm{C}}$ into some value, Deduce immediately returns, and all running coroutines and pending requests associated with it will be immediately deallocated.

Algorithm 1: Asynchronous Deduction (as an ACS Worker)

Parameters: A set of asynchronous deduction rules $\mathfrak{R}$, an expression grammar $\mathcal{G}$.
Input: A collection of input-output examples $\mathcal{S}$.
Yields: A sequence of conditions *cond* discovered during the enumeration, which splits the input-output examples set $\mathcal{S}$
Output: A solution *sol* to the inductive SyGuS problem $(\mathcal{G}, \mathcal{S})$.

```
async gen Solve(𝒢, 𝒮) :
    Enum ← new Enumerator(𝒢, 𝒮)                       // Create an enumerator coroutine (cf. Algo 3)
    cond_STRM ← Enum.RequestStream(Cond[𝒢, 𝒮])         // Enumeration request (cf. Algo 3).
    sol_F ← Deduce(Enum, 𝒮)
    loop:
        match await ('cond', cond_STRM) ∨ ('sol', sol_F) :
            case 'cond', cond : yield cond                // new condition found
            case 'sol', sol : return sol                  // solution found and return

async fn Deduce(Enum, 𝒮) :
    result_C ← channel.oneshot()                         // Create a oneshot channel to await the result
    for p/q c ∈ ℜ :
        if ⟦c{𝒮 ↦ 𝒮}⟧ :                                   // Apply rule when condition satisfied
            ApplyRule(Enum, p{𝒮 ↦ 𝒮}, q{𝒮 ↦ 𝒮}, result_C)
    return await result_C

async fn ApplyRule(Enum, p, q, result_C) :
    match p :
        case (e ⊨_E R[𝒢, 𝒮′]) ⋊ p′ :
            req_STRM ← Enum.RequestStream(R[𝒢, 𝒮′])       // Send request r to enumerator
            for await e ∈ reqs_STRM :
                ApplyRule(Enum, p′{e ↦ e}, q{e ↦ e}, result_C)
        case e ⊨ 𝒮″, p′ :
            if dom(𝒮″) = dom(𝒮) :
                sol ← await Deduce(Enum, 𝒮″)
            else:
                sol ← await spawn Solve(𝒮″).ret          // Run Solve parallelly, await return value
            ApplyRule(Enum, p′{e ↦ sol}, q{e ↦ sol}, result_C)
        case ε :
            (e ⊨ 𝒮) ← q
            result_C.send(e)
```

5 Enumeration and Case-splitting

In this section, we discuss other enumeration-related components of our concurrent synthesis framework, including the main algorithm which systematically enumerates the relaxed subsets of the original input-output example, the term dispatcher which coordinates the communication between deducer and enumerator, and the underlying enumerator for request handling.

5.1 Accumulative Case-Splitting

We now present the overall synthesis algorithm. In the setting of accumulative case-splitting, the goal of an asynchronous deducer is not to find a full solution satisfying all examples, but conditions and/or partial solutions that can be later assembled to form a decision tree using the `ITE` operator. Therefore, our main synthesis algorithm essentially utilizes multiple worker threads, each solving a different relaxation of the original problem, collects produced conditions and partial solutions, and assembles them to a full solution.

Which relaxations should be enumerated and solved by ACS workers? To strike a balance between failing to find any solution and generating overfit solutions, Our algorithm adopts a size-based enumeration. It begins with the weakest, single-example subproblems, ensuring that an initial, possibly-overfit solution is available. Gradually, it solves stronger, multi-example subproblems, which progressively refine the found solutions and mitigate overfitting. Hence, the likelihood of overfitting can be significantly reduced while maintaining the capability to solve difficult problems.

Algorithm 2 shows our overall synthesis algorithm with Synth as the entrance. The algorithm maintains two global sets of expressions: *conds*, which collects all conditions discovered by the enumerators, and *sols*, which keeps all partial solutions found by worker threads, i.e., expressions that cover at least a subset of the input-output examples. To generate conditions and partial solutions into *conds* and *sols*, the procedure first spawns a set of worker threads. Each worker thread will repeatedly select a subset of $\mathcal{S}$ using GenerateExamples which represents a relaxed, weaker problem, and solve the relaxed problem using Solve as shown in Algorithm 1. GenerateExamples will pick a subset based on a certain strategy, the details of which we leave in Appendix B.1.

The Synth procedure combines terms from *conds* and *sols* to generate the final solution. Once a condition (or, at the end, a partial solution) is available from Solve, it will be immediately added to *conds* (or *sols*). After that, the Synth procedure will try to learn a new decision tree e (shown as the LearnDT call) using *sol* and *conds*, and return e if its size is less than a preset size limit $\theta_{\text{tree-size}}$. The conditions of the decision tree are collected from all the enumerators of the algorithm.

LearnDT learns a decision tree from *sols* and *conds*. The decision tree learning algorithm is based on ID3 [36], but since ID3 does not allow labels to overlap between data points, we slightly updated the information gain defined by ID3 to support the overlap with solutions (that is, an example can be solved by multiple solutions). Specifically, when an example can be solved with multiple solutions, the standard entropy defined by the ID3 algorithm is no longer valid. To address this, each time we compute the entropy, we simply assign each example with multiple solutions to the single solution that covers the most examples. This heuristic technique gives a nearly-minimum entropy of all possible assignments of examples, ensuring the information gain to be nearly-maximal.

5.2 Term Dispatcher

We now elaborate an abstract data type called *term dispatcher* which enables the enumerator to handle a large number of requests simultaneously. Then we present the enumeration algorithm in which a request handler interacts with a term dispatcher. In a nutshell, the term dispatcher $\mathcal{D}$ is an abstract data type that maintains multiple requests and expressions satisfying these requests. We present the definition of term dispatcher as follows:

Definition 5.1. A term dispatcher $\mathcal{D}$ is a structure with the following operations:

- $\mathcal{D}.\text{add-expr}(e)$: Add an expression e to the data structure.
- $\mathcal{D}.\text{add-req}(r)$: Add a request r to the data structure.
- $\mathcal{D}.\text{dispatch}(e)$: Get a set of requests in $\mathcal{D}$ to which e can be a response.
- $\mathcal{D}.\text{select}(r)$: Get a set of expressions in $\mathcal{D}$ that satisfy r.

Algorithm 2: Overall Synthesis Algorithm (Accumulative Case-Splitting)

Parameters : *nthd*, number of worker threads used for the search; and $\theta_{\text{tree-size}}$, size limit of the decision tree.
Input : An inductive SyGuS problem $(\mathcal{G}, \mathcal{S})$.
Output : A solution to $(\mathcal{G}, \mathcal{S})$.

fn Synth$(\mathcal{G}, \mathcal{S})$ **:**
    $conds, sols \leftarrow \varnothing$ // The global sets of conditions and (partial) solutions
    for $i \in [0, nthd)$ **:**
        spawn: // Create worker threads
            loop:
                $R \leftarrow$ GenerateExamples$(\mathcal{S}, sols)$ // Pick a subset of $\mathcal{S}$ (cf. Algo 4 in appx.)
                for await $cond \in$ Solve$(\mathcal{G}, \mathcal{S}\,|_R)$ **:** // Solve the relaxed subproblem (cf. Algo 1)
                    $conds \leftarrow conds \cup \{cond\}$
                finally sol **:** // The return value of Solve (cf. Algo 1 line 8).
                    $sols \leftarrow sols \cup \{sol\}$
    loop:
        wait for *sols* and *conds* to be updated
        $e \leftarrow$ LearnDT$(sols, conds)$ // Learn a decision tree
        if $e \neq \bot$ **and** e.decision-tree-size$() \leq \theta_{\text{tree-size}}$ **:**
            return e

Recall that every request r has a type determined by its subproblem functor $r.R$. For efficiency, for each request type, the term dispatcher should be implemented differently. For Eq, we simply borrow the hash table for checking observational equivalence by allowing it to store requests at the place of the expression if the expression is not available. The operation cost is almost negligible. For ConstSubstr used in S-Join and S-ConstSubstr, we maintain a single interval tree, which is simple and enough for efficient implementation of $\mathcal{D}$.dispatch(e) and $\mathcal{D}$.select(r) when there are not too many expressions that are both constant and a substring of $\mathcal{S}$. For Prefix used in rule S-Prefix, we maintain a radix tree for each input-output example. Note that expressions that evaluate to shorter strings satisfy more requests. In particular, any expression producing empty output for some inputs can trivially satisfy all Prefix requests and can be returned for $\mathcal{D}$.dispatch(e). To avoid this problem, we only traverse the radix tree for which the expression yields the longest output. For Len, we maintain a hash table that uses the length vector as index. For Contains, as the $\mathcal{D}$.select(r) operation is rarely called in practice, we simply keep a hash table that maps a string element to a list of requests. And lastly, for Cond, we maintain a single list $E.C$ to store all conditions discovered by a single enumerator, since the constraint Cond$(\mathcal{S})$ defined in 4.1 doesn't reply on the output of $\mathcal{S}$. We leave more details of the design of each data structures in Table 3 of Appendix B.2, including the data structure we use for implementing every request type, and how $\mathcal{D}$.dispatch(e) and $\mathcal{D}$.select(r) are implemented in each case.

5.3 Enumeration for Request Handling

Algorithm 3 illustrates how the term enumerator operates in response to requests from the deducer. As previously discussed, the enumerator consists of three key components: a term dispatcher, a term generator, and a request handler. To capture their concurrent interaction, we model these components together inside one single actor [17, 19, 20] in Algorithm 3.

Algorithm 3: Enumeration for Request Handling

```
Input    : An inductive SyGuS problem (G, S)
1  actor Enumerator(G, S) :
2      D ← new TermDispatcher(S)                      // Initialize a new Term Dispatcher
3      async gen RequestStream(r) :                   // Generate a stream of response for request r
4          for e ∈ D.select(r) : yield e              // Reply all possible r with expression e
5          r.chan ← channel()                         // Create a channel for FIFO communication
6          D.add-req(r)
7          for await e ∈ r.chan : yield e             // Reply all possible r with expression e
8      async init:
9          loop:
10             e ← NextTerm(G, S)                     // Enumerate the next expression
11             for r ∈ D.dispatch(e) :
12                 await r.chan.send(e)               // Reply all possible r with expression e
13             D.add-expr(e)
```

The enumerator actor created in Algorithm 1 maintains a term dispatcher $\mathcal{D}$ (line 2) to store the relationship between term generation and requests. The enumerator can be requested by the deducer calling RequestStream (line 3-7) to generate a stream of expressions that satisfy the request. It will continually run term generation (line 8-13) once created to answer the requests made by RequestStream.

The RequestStream procedure generates a stream of expressions using the term dispatcher. It first calls $\mathcal{D}$.select to extract expressions that already satisfy the requests. For the undiscovered expressions, it associates each request with a channel, i.e., a message-passing queue for communication, and adds the request into the term dispatcher. During the search, the channel will be asynchronously populated with newly discovered expressions that satisfy the constraints. RequestStream will forward all the expressions given by the channel as the output stream.

Meanwhile, the term generator (line 8-13) repeatedly enumerates new expressions in a bottom-up order (denoted as NextTerm($\mathcal{G}, \mathcal{S}$), with observational equivalence checking over $\mathcal{S}$) and adds them into $\mathcal{D}$. At the time of adding an expression e, the generator also looks up if e satisfies any pending request in $\mathcal{D}$ by calling $\mathcal{D}$.dispatch(e). All such requests will be responded to by adding the newly enumerated e into the associated channel *r.chan*.

6 Implementation

We have refined the synthesis approach detailed in the paper and implemented it in a synthesizer called Synthphonia. The implementation of Synthphonia, written in Rust, comprises approximately 7 KLOC. Below, we describe several significant design choices and optimizations that were employed during the development.

Intra-Thread Coordination of Enumeration and Deduction. Deductive rules, such as S-Prefix, can initiate an extremely aggressive top-down search when a large number of expressions are already stored in the term dispatcher. This extensive top-down search often results in excessive time consumption without yielding significant progress and hinders the enumeration process within the same thread. To achieve a balance between top-down deduction and bottom-up enumeration, we introduce a technique called *delayed deduction*. This technique defers deeper deductive searches to allow more time for enumeration. In our implementation, the deducer is permitted to proceed to the next depth level only after enumerating 100,000 expressions at the current depth. This approach ensures a more efficient allocation of computational resources between deduction and enumeration.

Suppressing Excessive Threads. In Algorithm 1, aside from S-ITE which receives special treatment as described in Section 5.1, L-MAP and L-FILTER can also generate too many subproblems with distinct sets of examples, which lead to too many threads. In practice, these threads mostly search in vain because L-MAP and L-FILTER are not frequently used. To this end, we add additional restriction to Algorithm 1 to suppress the number of threads created by L-MAP and L-FILTER. First, we restrict the depth these rules can be applied with in deduction. In our implementation, we only allow these rules to be applied to subproblems with a depth up to 5. Also, we restrict the execution time of the threads created by L-MAP and L-FILTER to 1 seconds.

Adaptive Size Limit. Because real-world problems vary, it is impossible to find a one-size-fits-all $\theta_{\text{tree-size}}$ for Algorithm 2. Therefore in default setting of SYNTHPHONIA, we allow $\theta_{\text{tree-size}}$ to linearly increase when no new partial solutions are found within a time period. In our setting, $\theta_{\text{tree-size}}$ will increase by 1 (allowing one more `ITE`) each 4 seconds without a new solution found. This adaptive size limit makes SYNTHPHONIA more flexible for solving a wide spectrum of problems with various difficulties.

7 Evaluation

In order to assess the efficiency of our concurrent synthesis approach, we performed comprehensive experiments using SYNTHPHONIA and contrasted its performance against the latest string transformation synthesizers available. All experiments were carried out on a Linux system equipped with two Intel Xeon E5 10-core 2.2GHz CPUs and 128GB of RAM. We use $nthd = 4$ as the default number of ACS workers.

7.1 Experimental Setup

Compared Synthesizers. We compare SYNTHPHONIA with existing, state-of-the-art synthesizers for string transformation: CVC4/CVC5, DUET, PROBE and FLASHFILL++: CVC4 [6] (and its successor CVC5 [4]) is one of the most popular SMT solvers with the capabilities of SYGUS solving. We noticed a significant difference between CVC4 and CVC5 and report the results from both. PROBE [5] is a SYGUS solver that performs a just-in-time bottom-up search with guidance from a probabilistic model. DUET [26] is a tool for solving inductive SYGUS problems. It employs a bidirectional search strategy with a domain specialization technique called top-down propagation which can recursively decompose a given synthesis problem into multiple subproblems. It requires inverse semantics operators that should be designed for each usable operator in the target language. FLASHFILL++ [9] is designed to efficiently synthesize programs using large domain-specific languages (DSLs) containing a large family of operators not expressible in the interchange format SYGUS-IF. It extends DUET's meet-in-the-middle synthesis algorithm with *cuts*, which allows DSL designers to further restrict backward-propagation search space using the domain knowledge from the DSL. We use version `8.25.0` of FLASHFILL++ for our experiments.

CVC4/CVC5, PROBE, and DUET are limited to grammars that can be expressed in the SYGUS-IF. In contrast, FLASHFILL++ has developed a domain-specific grammar for strings that includes numerous operators not supported by SYGUS-IF. Thus, when comparing SYNTHPHONIA to CVC4/CVC5, PROBE, and DUET, we restrict SYNTHPHONIA to utilize only the SYGUS-IF grammar (this version is denoted as SP-G in the following sections). When comparing SYNTHPHONIA to FLASHFILL++, we set the full grammar as presented in Figure 3 as the target grammar which utilizes a broader range of operators such as negative indices, loops, date, time, and float operators.

Benchmarks. We collect our string transformation benchmarks from 3 sources: i) *Duet* Benchmarks, ii) *Prose* benchmarks, and iii) our own *HardBench* benchmarks. These 3 categories of benchmarks entail a collection of 694 benchmarks.

- *Duet.* We grab 205 benchmarks from Duet [26], which consists of 108 from the SyGuS competition and another 97 benchmarks from StackOverflow and ExcelJet. All *Duet* benchmarks provide a grammar in SyGuS interchange format (SyGuS-IF) along with the input-output examples.
- *Prose.* We utilize 354 benchmarks from the Microsoft PROSE team [34]. The challenge of solving *Prose* benchmarks mainly lies in synthesizing transformations involving date, time, and floating-point operations. Because of the limited operator support in the SyGuS-IF, when comparing CVC4/CVC5, Probe, Duet, and Synthphonia (SP-G version), we adapt the target language to the one used in the Duet benchmarks.
- *HardBench.* To further assess a synthesizer's scalability and flexibility, we also crafted 135 challenging benchmarks, involving heavy case-splitting and loops. All these benchmarks are based on real-world scenarios. One of the authors wrote English descriptions of the tasks and produced sample input/output pairs with the aid of ChatGPT. Our Example 2.1 is from this category of benchmarks. We also present an additionally selected benchmark in Appendix C.4.

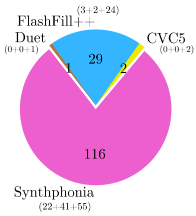
Additional Testing Examples. Every benchmark comes with a set of original examples as an incomplete specification. To ensure that the produced solutions do not overfit to these examples, we also manually crafted two to six additional testing examples for each benchmark. To successfully solve a benchmark, the produced solution must pass all original and additional examples as test cases.

7.2 Comparison to Existing Synthesizers

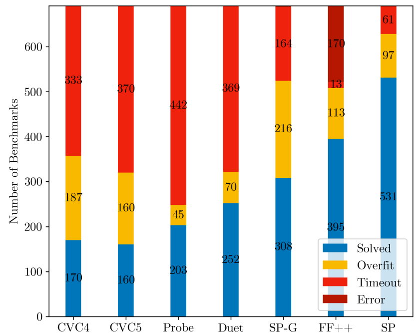
We evaluate Synthphonia on all the benchmarks and compare it with CVC4/CVC5, Duet, Probe and FlashFill++. For each instance, we run the benchmark and measure the running time with a timeout of 1 minute. The experimental result can be Solved, Overfit, Timeout, or Error. Overall, Synthphonia outperforms other solvers in terms of the number of solved problems and execution time.

Synthphonia accurately solved 531 out of 694 benchmarks from three benchmark sets, outperforming all other synthesizers. Synthphonia also generated overfit solutions for 97 benchmarks. Figure 5a shows the number of benchmarks that can be uniquely solved by each solver and for each benchmark (Duet + HardBench + Prose). We do not include Probe in the chart because it did not have any uniquely solved benchmarks in our setting. Among all the solvers listed, Synthphonia stands out by uniquely solving 116 benchmarks. In comparison, FlashFill++ uniquely solved 29 benchmarks; CVC5 and Probe uniquely solved one each. These results showcased the effectiveness of our methodology for string transformation programs.

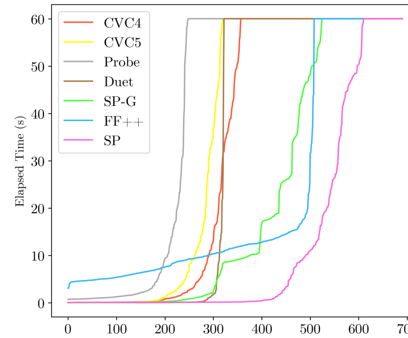
Figure 5b illustrates the number of benchmarks successfully solved by each solver. We denote Synthphonia as SP, FlashFill++ as FF++, and Synthphonia with SyGuS-IF Grammar as SP-G in the figure. According to the figure, Synthphonia solved 531 benchmarks, surpassing all other existing solvers. Even when employing identical grammar, SP-G outperformed Duet by solving 56 more benchmarks. However, because the grammar is not quite optimized on HardBench and Prose benchmark, SP-G exhibited 216 overfits which made SP-G underperform FlashFill++. Figure 5c shows the execution time versus the number of solved benchmarks. Regarding time efficiency, Synthphonia demonstrates faster solution generation for hard problems compared to existing methods.



(a) Number of uniquely solved benchmarks (Duet + HardBench + Prose).

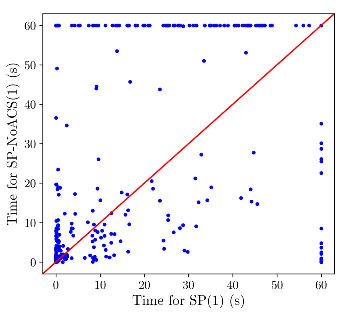


(b) Number of benchmarks solved and overfitted.

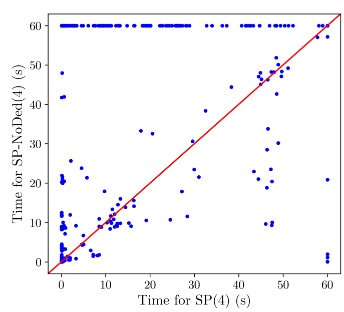


(c) Solution time (Benchmarks in increasing order).

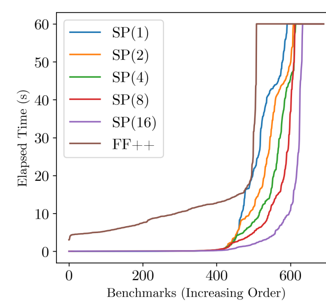
Fig. 5. Experimental results comparing to other solvers.



(a) Accumulative case-splitting.



(b) Asynchronous deduction.



(c) Scalability with multithreading.

Fig. 6. Ablation study.

7.3 Ablation Study

To evaluate the effectiveness of some novel features of our approach, we conducted an ablation study of Synthphonia. This analysis aims to highlight their individual contributions to the overall performance of the solver. To save space, we present only the aggregated results across all benchmark categories; detailed results for each category are available in Appendix C.2. Throughout this subsection, we denote the version of full Synthphonia with n ACS workers as SP(n).

Effectiveness of Accumulative Case-Splitting. We implemented a version of Synthphonia without accumulative case-splitting, which we denoted as SP-NoACS(1). To avoid consuming too many threads in this version, SP-NoACS(1) allows every deduction rule like S-Prefix to be conducted on a subset of the examples S, which enables subproblems with a subset of the examples to be deduced on the same thread. Figure 6a illustrates the performance difference between SP-NoACS(1) and SP(1). Error and Timeout benchmarks are assigned a 1-minute execution time to be included in the charts. As shown in the figure, SP(1) solved more benchmarks compared to SP-NoACS(1). This demonstrates that accumulative case-splitting effectively accelerates the synthesis process and enhances its capability of tackling more challenging benchmarks.

Effectiveness of Asynchronous Deduction. We also tested the performance of Synthphonia without any assistance of asynchronous deduction. We implement a baseline version with only Eq rule in Figure 4, but with accumulative case-splitting enabled. Figure 6b compares the performance between Synthphonia with and without asynchronous deduction. Since our default setting is *nthd* = 4, we denote this version as SP-NoDed(4). According to the figure, SP(4) solves more benchmarks compared to SP-NoDed(4). However, SP(4) spends more time on solving those benchmarks because we increase $\theta_{\text{tree-size}}$ after several seconds without a new partial solution found by any ACS workers, which makes it harder to reach a higher $\theta_{\text{tree-size}}$ for faster workers, since faster workers generate more frequent solutions.

Benefit of Multithreading. We next discuss the performance of our solver when scaled across multiple threads. Figure 6c illustrates the time cost to solve these benchmarks when using different numbers of threads. We also copy the comparable result of FlashFill++ in the figure for reference. According to the figure, about 200 benchmarks can be solved faster using more threads. The version with 16 threads: SP(16) can generate more solutions faster compared to other settings. This showcases Synthphonia's ability to scale across threads using accumulative case-splitting.

7.4 Room for Improvement

While the benchmarks Synthphonia failed to solve vary widely in the target tasks, a few common reasons account for these failures. We now highlight a benchmark that Synthphonia is unable to solve, showcasing certain limitations in our present implementation that future improvements may address. Table 2 presents the input-output examples for the `flight:airport1` benchmark from HardBench, which Synthphonia cannot solve. Each input string of this benchmark contains information about a flight, and the output should consist of the departure and arrival airport codes in lowercase format, extracted from the input string. FlashFill++ successfully generates the following solution:

```
def derived_column(x0):
    index1 =[i for i in range(len(x0)) if x0.startswith("(", i)][1] -1
    return (x0.split(" ")[1] +x0[x0.find(")") +1:index1]).lower()
```

Synthphonia could not solve this benchmark due to the following reasons:

(1) The arrival airport address should be represented by an expression like `str.lowercase(` in_0`.split("->")[1].split("␣")[1])`. However, Synthphonia does not offer enough support the operator "`str.lowercase`". First, it lacks a deductive rule for "`str.lowercase`" (and similarly for "`str.uppercase`" and several date/time operators). On top of that, Synthphonia assigns a low priority to this operator because it is not commonly used, which ultimately causes Synthphonia to fail in generating the solution.

(2) Synthphonia lacks effective heuristics for selecting domain-specific constants (similar to other SyGuS solvers like Duet). The current generic implementation generates an excessive number of irrelevant constants for this problem, such as "`:00 AM`", "`:00 PM`", "`M -> 0`", "`:00 PM -> `", and "`:00 `". This over-generation hampers performance.

8 Related Work

Parallelism for Program Synthesis. Various research has already explored the parallelization of program synthesis algorithms. Morpheus [14] uses multiple threads to search for solutions of different sizes, to maximize the possibility of reaching a large size. Adaptive concretization [22, 23] and Synapse [7] present parallel synthesis algorithms which let each thread search for a non-intersecting portion of the search space independently. However, these techniques only consider parallel instances with an identical specification. Paresy [39] parallelizes an enumeration algorithm for regular expression inference, without consideration of deduction. FlashMeta [33] attempts

Table 2. Input/output examples for benchmark `flight:airport1`.

Input / String	Output/ String
"CZ234; PEK (Beijing, China) -> SYD (Sydney, Australia); 10:00 PM -> 10:00 AM (+1 day)"	"pek -> syd"
"LH789; MUC (Munich, Germany) -> JFK (New York, USA); 01:00 PM -> 05:00 PM"	"muc -> jfk"
"UA789; IAH (Houston, USA) -> ORD (Chicago O'Hare, USA); 08:00 AM -> 11:00 AM"	"iah -> ord"
"UA789; IAH (Houston, USA) -> ORD (Chicago O'Hare, USA); 08:00 AM -> 11:00 AM"	"iah -> ord"
"EK456; JFK (New York, USA) -> DXB (Dubai, UAE); 06:00 PM -> 04:00 PM (+1 day)"	"jfk -> dxb"
"DL567; ATL (Atlanta, USA) -> SFO (San Francisco, USA); 09:00 AM -> 11:00 AM"	"atl -> sfo"
"QF234; SYD (Sydney, Australia) -> LAX (Los Angeles, USA); 09:00 AM -> 06:00 AM"	"syd -> lax"
"LH789; MUC (Munich, Germany) -> JFK (New York, USA); 01:00 PM -> 05:00 PM"	"muc -> jfk"
"AA789; JFK (New York, USA) -> LAX (Los Angeles, USA); 07:00 AM -> 10:00 AM"	"jfk -> lax"
"SQ321; SIN (Singapore) -> JFK (New York, USA); 11:00 PM -> 07:00 AM (+1 day)"	"sin -> jfk"
"DL567; ATL (Atlanta, USA) -> SFO (San Francisco, USA); 09:00 AM -> 11:00 AM"	"atl -> sfo"
"AF567; CDG (Paris Charles de Gaulle, France) -> DXB (Dubai, UAE); 03:00 AM -> 11:00 AM"	"cdg -> dxb"
"CZ345; PEK (Beijing, China) -> LHR (London Heathrow, UK); 11:00 PM -> 05:00 AM (+1 day)"	"pek -> lhr"
.... (67 in total)	....

to parallelize their deduction but faces challenges due to the non-deterministic inverse semantics of common operators, leading to an unnecessary combinatorial explosion in branch possibilities. In contrast, SYNTHPHONIA offers a program synthesis architecture that harnesses concurrency to orchestrate the decomposition, solving, and assembly of subtasks by both deduction and enumeration.

Enumerative Methods for Synthesis. Enumerative program synthesis is widely acknowledged for its efficacy. Here we only highlight those systems supporting string transformation synthesis. Pioneered by EUSOLVER [2], various SYGUS synthesizers navigate expansive search spaces and employ various strategies to efficiently prune those spaces. In bottom-up enumeration, a key technique for pruning is *observational equivalence* (OE) [1, 38], which is also successfully applied in SYNTHPHONIA. CVC4 [6, 32], as a consistent leader in the SYGUS competition, optimizes rewrite rules to enhance equivalence checking during bottom-up approaches.

Recent research has also explored novel enumeration strategies using learning-based methods. For example, PROBE [5] leverages just-in-time learning with probabilistic context-free grammars (PCFG), assigning scores to production rules based on learned contexts. Similarly, EUPHONY [27] incorporates probabilistic higher-order grammars (PHOG) to enrich the search guidance. The concurrent interplay between enumeration and deduction presented in this paper is orthogonal to the choice and enhancement of enumeration strategies.

Combining Deduction and Enumeration. As two major synthesis approaches, deduction and enumeration complement each other, and combining their strengths to achieve the best performance has been a popular direction in recent years. Though to the best of our knowledge, none of those methods considered concurrent coordination between the enumerator and the deducer. Earlier work guides the enumerative search via various kinds of deductions. λ^2 [15] uses deduction to deduce the input-output examples for subproblems and conduct a best-first search on different deductions. Similarly, SMYTH [28] uses live bidirectional evaluation, which propagates examples backward through user-given sketches. Feng et al. [13, 14] employ deduction to effectively limit the search space during enumeration. However, for string transformation, many common operators (`ITE`, concatenation, etc.) have nondeterministic inverse semantics and there are excessive branches to explore. Above techniques help little in these cases.

Recent advancements also proposed "meet-in-the-middle" synthesis, which explores top-down and bottom-up search simultaneously towards the middle, with a mixture of deductive and enumerative methods. DRYADSYNTH [12, 21] explores various ways to combine deductive and enumerative methods, including divide-and-conquer and bottom-up deduction (combining bottom-up enumeration results on-the-fly). Duet [26] combines bottom-up enumeration with top-down propagation, integrating expressions generated from bottom-up processes into a cohesive top-down framework. Simba [40] and FlashFill++ [9] both provide methods to prune the search space during meet-in-the-middle synthesis. Simba utilizes backward abstract interpretation to prune the top-down propagation. FlashFill++ introduces *cuts* in the meet-in-the-middle synthesis system, enabling DSL designers to reduce the witness function based on the DSL.

These existing approaches heavily influenced Synthphonia's cooperation between the deducer and the enumerator. However, in all these methods, the enumeration process is not tailored to react to various specific decomposition needs, and the deducer has to repeatedly sift through a vast pool of enumerated expressions. Compared to these meet-in-the-middle approaches, our concurrent algorithm enables more general and flexible cooperation between the deducer and the enumerator, which effectively accelerates the cooperation and offers more flexibility for deduction.

Synthesis with Conditions. Research on synthesizing conditional expressions encompasses various approaches. Leon [2] introduces an abduction-based reasoning method that guesses conditions based on existing partial solutions. EuSolver [2] formulates the combination of expressions and conditions as a multi-label decision tree problem, using information-gain heuristics to construct compact decision trees. Additionally, PolyGen [24] introduces synthesis through unification (STUN), which unifies synthesized terms after generation, following Occam's learning principles. These methods significantly influenced the development of our accumulative case-splitting technique. As previously stated, accumulative case-splitting offers greater adaptability as it conducts condition search, term search, and the assembly of decision trees entirely concurrently and independently.

9 Conclusion and Future Work

We developed a synthesis algorithm that combines concurrent deductive and enumerative processes, allowing multiple deduction paths to be explored in parallel, guided by enumeration. Our implementation, Synthphonia, designed for string transformation tasks, shows significant performance improvements, successfully solving 116 benchmark tasks for the first time.

While this paper focuses on a special domain of string transformations, some key components of our approach (the framework, the enumerator, and the accumulative case-splitting) are general and have the potential to be applied to many other domains. To migrate Synthphonia to a new domain, two components need to be re-designed carefully: asynchronous deduction rules and the corresponding term dispatcher. Given a new domain, one needs to design a new set of domain-specific, asynchronous deduction rules (similar to those in Figure 4), indicating how to decompose a synthesis problem and which requests to send to the enumerator. Once the deduction rules and requests are determined, on the enumerator side, one has to design corresponding data structures to handle unique requests for the new domain (similar to what we discussed in §5.2).

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 934–950.

[2] 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.

[3] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ByldLrqlx

[4] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng,

Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

[5] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-Time Learning for Bottom-up Enumerative Synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 227 (nov 2020), 29 pages. https://doi.org/10.1145/3428295

[6] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 171–177.

[7] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 775–788. https://doi.org/10.1145/2837614.2837666

[8] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1Xw62kRZ

[9] José Pablo Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proceedings of the ACM on Programming Languages* 7 (2023), 952 – 981.

[10] Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. 2017. What is decidable about string constraints with the ReplaceAll function. *Proc. ACM Program. Lang.* 2, POPL, Article 3 (dec 2017), 29 pages. https://doi.org/10.1145/3158091

[11] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 689–700. https://doi.org/10.1145/2676726.2677006

[12] Yuantian Ding and Xiaokang Qiu. 2024. Enhanced Enumeration Techniques for Syntax-Guided Synthesis of Bit-Vector Manipulations. *Proc. ACM Program. Lang.* 8, POPL, Article 71 (jan 2024), 31 pages. https://doi.org/10.1145/3632913

[13] 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

[14] 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

[15] John K. 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* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

[16] 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

[17] Irene Gloria Greif. 1975. *Semantics of Communicating Parallel Processes.* PhD dissertation. Massachusetts Institute of Technology.

[18] 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

[19] Carl Hewitt. 1976. *Viewing Control Structures as Patterns of Passing Messages.* Technical Report AIM-410. MIT Artificial Intelligence Laboratory. http://dspace.mit.edu/handle/1721.1/6272

[20] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Stanford, USA) *(IJCAI'73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[21] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling Enumerative and Deductive Program Synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1159–1174. https://doi.org/10.1145/3385412.3386027

[22] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. 2015. Adaptive Concretization for Parallel Program Synthesis. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 377–394.

[23] 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 (FMSD)* 50, 1 (March 2017), 75–95.

[24] Ruyi Ji, Jingtao Xia, Yingfei Xiong, and Zhenjiang Hu. 2021. Generalizable Synthesis through Unification. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 167 (oct 2021), 28 pages. https://doi.org/10.1145/3485544

[25] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis Modulo Recursive Functions. In *OOPSLA'13* (Indianapolis, Indiana, USA). ACM, 407–426.

[26] Woosuk Lee. 2021. Combining the Top-down Propagation and Bottom-up Enumeration for Inductive Program Synthesis. *Proc. ACM Program. Lang.* 5, POPL, Article 54 (jan 2021), 28 pages. https://doi.org/10.1145/3434335

[27] 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

[28] Justin Lubin, N. M. Collins, Cyrus Omar, and Ravi Chugh. 2019. Program sketching with live bidirectional evaluation. *Proceedings of the ACM on Programming Languages* 4 (2019), 1 – 29. https://api.semanticscholar.org/CorpusID:213380177

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

[30] Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (1980), 90–121.

[31] Kairo Morton, William Hallahan, Elven Shum, Ruzica Piskac, and Mark Santolucito. 2020. Grammar Filtering for Syntax-Guided Synthesis. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 02 (Apr. 2020), 1611–1618. https://doi.org/10.1609/aaai.v34i02.5522

[32] Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark Barrett, and Cesare Tinelli. 2019. Syntax-Guided Rewrite Rule Enumeration for SMT Solvers. In *Theory and Applications of Satisfiability Testing – SAT 2019*, Mikoláš Janota and Inês Lynce (Eds.). Springer International Publishing, Cham, 279–297.

[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)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

[34] Microsoft PROSE. 2022. PROSE public benchmark suite. Github. https://github.com/microsoft/prose-benchmarks

[35] 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

[36] J. R. Quinlan. 1986. Induction of decision trees. *Machine Learning* 1, 1 (1986), 81–106. https://doi.org/10.1007/BF00116251

[37] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. 2019. cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 74–83.

[38] 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 *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 287–296. https://doi.org/10.1145/2491956.2462174

[39] Mojtaba Valizadeh and Martin Berger. 2023. Search-Based Regular Expression Inference on a GPU. *Proc. ACM Program. Lang.* 7, PLDI, Article 160 (June 2023), 23 pages. https://doi.org/10.1145/3591274

[40] Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. *Proc. ACM Program. Lang.* 7, PLDI, Article 174 (jun 2023), 25 pages. https://doi.org/10.1145/3591288

A Full List of Deduction Rules

Figure 7 presents extra asynchronous deductive rules not shown in Figure 4.

$$\textsc{Eq}\ \frac{e \models_E \mathrm{Eq}[\mathcal{G},\mathcal{S}]}{e \models \mathcal{S}} \qquad \textsc{S-Len L-Len}\ \frac{e \models_E \mathrm{Len}[\mathcal{G},\mathcal{S}]}{\texttt{len}(e) \models \mathcal{S}} \qquad \textsc{S-FromInt}\ \frac{e_i \models I, \quad e \models S}{\texttt{str.from\_int}(e_i) +\!\!+\, e \models \mathcal{S}}\ \texttt{str.from\_int}(I) +\!\!+\, S = \mathcal{S}$$

$$\textsc{S-FromFloat}\ \frac{e_i \models I, \quad e \models S}{\texttt{str.from\_float}(e_i) +\!\!+\, e \models \mathcal{S}}\ \texttt{str.from\_float}(I) +\!\!+\, S = \mathcal{S}$$

$$\textsc{S-ListAt}\ \frac{\left(e \models_E \mathrm{Contains}[\mathcal{G},\mathcal{S}]\right) \rtimes \left(e_i \models [\![e]\!]_{\mathcal{I}}\texttt{.indexof}(\mathcal{S})\right)}{e[e_i] \models \mathcal{S}} \qquad \textsc{S-Join}\ \frac{\left(e \models_E \mathrm{ConstSubstr}[\mathcal{G},\mathcal{S}]\right) \rtimes \left(e_1 \models \mathcal{S}\texttt{.split}([\![e_s]\!]_{\mathcal{I}})\right)}{\texttt{list.join}(e_1, e) \models \mathcal{S}}$$

$$\textsc{S-Ite}\ \frac{\left(e \models_E \mathrm{PartialEq}[\mathcal{G},\mathcal{S}]\right) \rtimes \left(e_1 \models [\![e]\!]_{\mathcal{I}} =_{\mathrm{B}} \mathcal{S}, \quad e_2 \models \left\{\boldsymbol{i} \mapsto o \in \mathcal{S} \,\middle|\, [\![e]\!]_{\boldsymbol{i}} \neq_{\mathrm{B}} o\right\}\right)}{\texttt{ITE}(e_1, e_2, e) \models \mathcal{S}}$$

$$\textsc{S-Prefix}\ \frac{\left(e \models_E \mathrm{Prefix}[\mathcal{G},\mathcal{S}]\right) \rtimes \left(e_1 \models \texttt{str.substr}(\mathcal{S}, \texttt{str.len}([\![e]\!]_{\mathcal{I}}), -1)\right)}{e +\!\!+\, e_1 \models \mathcal{S}}$$

$$\textsc{S-ConstSubstr}\ \frac{\left(e \models_E \mathrm{ConstSubstr}[\mathcal{G},\mathcal{S}]\right) \rtimes \left(e_1 \models \mathcal{S}\texttt{.split\_once}([\![e]\!]_{\mathcal{I}})[0], \quad e_2 \models \mathcal{S}\texttt{.split\_once}([\![e]\!]_{\mathcal{I}})[1]\right)}{e_1 +\!\!+\, e +\!\!+\, e_2 \models \mathcal{S}}$$

$$\textsc{L-Map}\ \frac{\left(e \models_E \mathrm{Len}[\mathcal{G}, \texttt{len}(\mathcal{S})]\right) \rtimes \left(e_f \models \left\{[\![e]\!]_{\boldsymbol{i}}[k] \mapsto o[k] \,\middle|\, \boldsymbol{i} \mapsto o \in \mathcal{S}, 0 \le k < \texttt{len}(\mathcal{S}[\boldsymbol{i}])\right\}\right)}{\texttt{list.map}[e_f](e) \models \mathcal{S}}$$

$$\textsc{L-Filter}\ \frac{\begin{array}{c}\left(e \models_E \mathrm{Contains}[\mathcal{G}, \mathcal{S}[0]]\right) \rtimes \\ \left(\bigwedge_{\boldsymbol{i} \mapsto o \in \mathcal{S}} o\texttt{.subseqof}([\![e]\!]_{\boldsymbol{i}}), \quad e_f \models \left\{[\![e]\!]_{\boldsymbol{i}}[k] \mapsto o\texttt{.contains}([\![e]\!]_{\boldsymbol{i}}[k]) \,\middle|\, \boldsymbol{i} \mapsto o \in \mathcal{S}, 0 \le k < \texttt{len}([\![e]\!]_{\boldsymbol{i}})\right\}\right)\end{array}}{\texttt{list.filter}[e_f](e) \models \mathcal{S}}$$

Fig. 7. Asynchronous Deduction Rules for String.

B Additional Algorithm Details

B.1 Subset Generation Method for Accumulative Case-splitting

In this section, we describe our method to generate subset mentioned in Algorithm 2. Here we present the definition of GenerateExamples in Algorith 4. The subprocedure GenerateExamples generates a minimum subset that is not covered by any existing solutions in *sols*. For example, if the specification $\mathcal{S}$ consists of 6 input-output examples and *sols* has two solutions available: $sols = \{e_1, e_2\}$, and e_1 covers examples 1, 2, 3, 4 and e_2 covers example 3, 4, 5, 6, then example set 1, 5 is a minimum subset that can be generated by GenerateExamples. In contrast, example set 1, 2, 5 is not minimum, and example set 1, 4 is already covered by expression e_1; hence they will not be generated. The subprocedure finds such a subset by performing a size-based enumeration, from size 1 to size $|\mathcal{S}|$. In the i-th iteration, it enumerates all subset of $\mathcal{S}$ with size i in a random order. Whenever a subset R is not covered by any partial solution in *sols*, the subset will be returned for synthesis.

```
fn GenerateExamples(S, sols) :
    for i ∈ [1, |S|] :
        for random R ⊆ dom(S), |R| = i :      // Iterate all i-combinations of dom(S) randomly
            if ⋀_{sol∈sols} ⋁_{(i↦o)∈S|_R} [[sol]]_i ≠ o :   // Ensure not covered by existing partial solutions
                return R
```

Algorithm 4: Subset Generation Method for Accumulative Case-splitting

B.2 Design of each Data Structures in the Term Dispatcher

We list the detail design of each data structures in Table 3.

B.3 Constant Selection

Synthphonia has the capability of inferring suitable string constants from current specification $\mathcal{S}$. We employ a set of heuristic rules to select constants based on their length and frequency within $\mathcal{S}$. For instance, in Example 2.1, the string "␣" is short and frequently appears in the input-output examples. Therefore, we consider "␣" as a suitable constant and incorporate it directly into the enumeration process. This strategic inclusion of suitable constants enhances the synthesizer's ability to effectively synthesize solutions that align closely with the provided examples.

C Additional Experimental Results

C.1 Results for Different Benchmarks

Here we present the specific result for each benchmark category.

Figure 8a illustrates the number of *Duet* benchmarks successfully solved by each solver. Synthphonia solved 188 benchmarks, surpassing all other existing solvers. Additionally, Synthphonia exhibits only 16 overfits, which is lower than all competing solvers. Even when employing identical grammar, Synthphonia-G outperforms Duet by solving 9 more benchmarks. Figure 8b shows the execution time versus the number of solved benchmarks. Regarding time efficiency, both Synthphonia and Synthphonia-G demonstrate faster solution generation for hard problems compared to existing methods.

Figure 9 presents the experimental results on *Prose* benchmarks, demonstrating comparable results with FlashFill++ in terms of both the number of solved benchmarks and the solving time. This showcases Synthphonia's capability of synthesizing programs that involve date, time, and floating-point numbers. In contrast, CVC4/CVC5, Probe, Duet, and Synthphonia-G fall short in achieving comparable performance due to their lack of support for these operators.

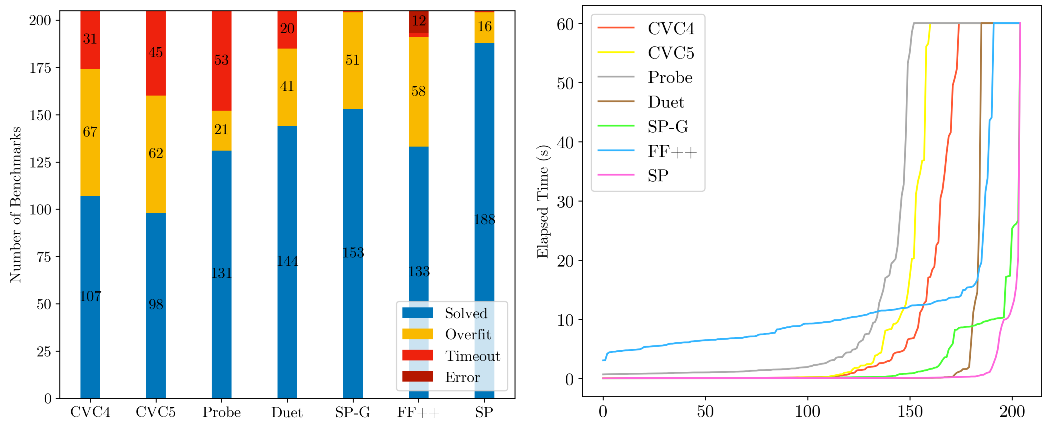
Figure 10 depicts the superior performance of Synthphonia over existing methods on *HardBench* benchmarks. Synthphonia can solve 136 more benchmarks compared to FlashFill++. Furthermore, Synthphonia requires less time to solve these problems compared to FlashFill++. This highlights Synthphonia's capability to tackle difficult problems using accumulative case-splitting and loops.

Table 3. Data structures implementing term dispatcher for different types of requests.

Request Type	Data Structure	$\mathcal{D}$.dispatch(e)	$\mathcal{D}$.select(r)
Eq	A hash table H that maps from a vector of values to expressions/requests. All the expressions and requests stored in H are all indexed by the corresponding value on each input. We also use the same table for checking observational equivalence.	If $H[\llbracket e \rrbracket_{\mathcal{I}}]$ is a request r, returns $\{r\}$, otherwise, return $\varnothing$.	If $H[r.\mathcal{S}]$ is an expression e, returns $\{e\}$, otherwise, return $\varnothing$.
ConstSubstr	Pick a random input-output example $\boldsymbol{i} \mapsto o \in \mathcal{S}$, and maintain an interval tree T that maps any substring of o to expressions and requests. For each enumerated expression e that is a substring of o, we will store an e in the interval tree for each appearance of $\llbracket e \rrbracket_{\boldsymbol{i}}$ in o, indexed by the starting and ending index of that appearance. We will store requests in the interval tree in the same manner.	Look up all the superstrings for $\llbracket e \rrbracket_{\boldsymbol{i}}$ in T and return the set of all the requests that holds on e.	Look up all substrings of $r.\mathcal{S}$ in T and return all the expressions satisfy r.
Prefix	We maintain maintain a radix tree $R_{\boldsymbol{i}}$ for each input example $\boldsymbol{i} \in \mathcal{I}$. For each input example $\boldsymbol{i}$, we store every enumerated expression e into the radix tree $R_{\boldsymbol{i}}$ using $\llbracket e \rrbracket_{\boldsymbol{i}}$ as the prefix. And we store every request r into $R_{\boldsymbol{i}}$ in the same manner.	1) Select $\boldsymbol{i}$ that makes $\llbracket e \rrbracket_{\boldsymbol{i}}$ has longest length. 2) Look up all requests in $R_{\boldsymbol{i}}$ that use $\llbracket e \rrbracket_{\boldsymbol{i}}$ as prefix. 3) Return all requests in 2) that holds on e.	1) Select $\boldsymbol{i}$ that makes $r.\mathcal{S}[\boldsymbol{i}]$ has shortest length. 2) Look up all expressions in $R_{\boldsymbol{i}}$ that is a prefix of $r.\mathcal{S}[\boldsymbol{i}]$. 3) Return all expressions in 2) that satisfy r.
Len	Similar to Eq, the enumerator maintain a hash table H_L that maps from a vector of lengths to the corresponding expressions/requests. Here we allow multiple expressions and requests be associated with the same vector of lengths.	Return all the requests stored in $H_L[\llbracket e \rrbracket_{\mathcal{I}}]$.	Return all the expressions stored in $H[r.\mathcal{S}]$.
Contains	Pick a random input-output example $\boldsymbol{i} \mapsto o \in \mathcal{S}$. A hash table H_C that maps from a string value to a list of requests. H_C stores all the Contains requests r from the deducer and index their value $r_{\boldsymbol{i}}$ with respect to $\boldsymbol{i}$.	If $\llbracket e \rrbracket_{\boldsymbol{i}}$ is a list, for every element $s \in \llbracket e \rrbracket_{\boldsymbol{i}}$, return all the requests in $H_C[s]$ that holds on e, otherwise, return $\varnothing$.	Return $\varnothing$. (For efficiency, we do not keep track of the expression in term dispatcher for Contains requests.)
Cond	A list E_C to store all conditions discovered by a single enumerator and a list R_C to store all the request from the deducers.	If $\llbracket e \rrbracket_{\boldsymbol{i}}$ satisfies the condition defined in 4.1, return R_C.	return E_C.

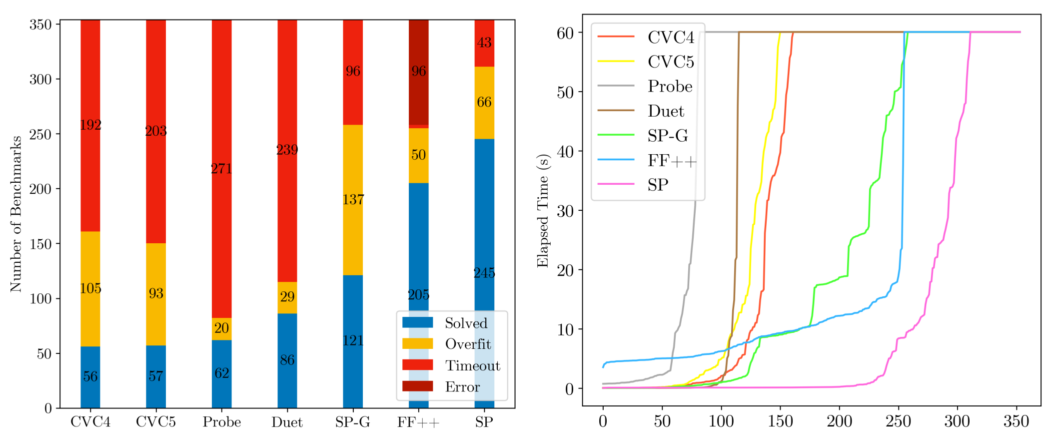
C.2 Ablation Study per Category of Benchmarks

Here we specify the ablation study result in Section 7.3 to different category of benchmarks. Figures 11, 12, 13 show the ablation study results for Duet, HardBench and Prose benchmark respectively.



(a) Number of benchmarks solved and overfit. (b) Solution time (Benchmarks in increasing order).

Fig. 8. Experimental results on *Duet* benchmarks.



(a) Number of Benchmarks solved and overfit. (b) Solution time (Benchmarks in increasing order).

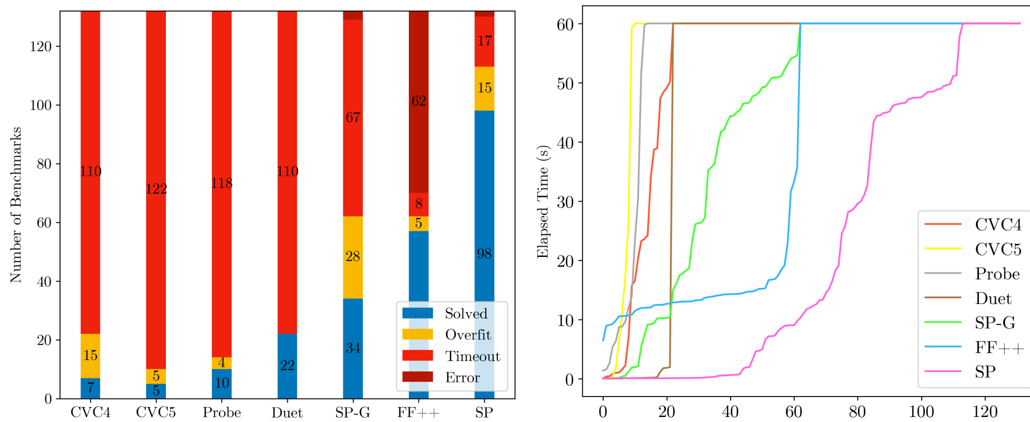
Fig. 9. Experimental results on *Prose* benchmark.

C.3 Additional Ablation Study for Implementation Details

Here we present the ablation study for the optimizations in Sec 6.

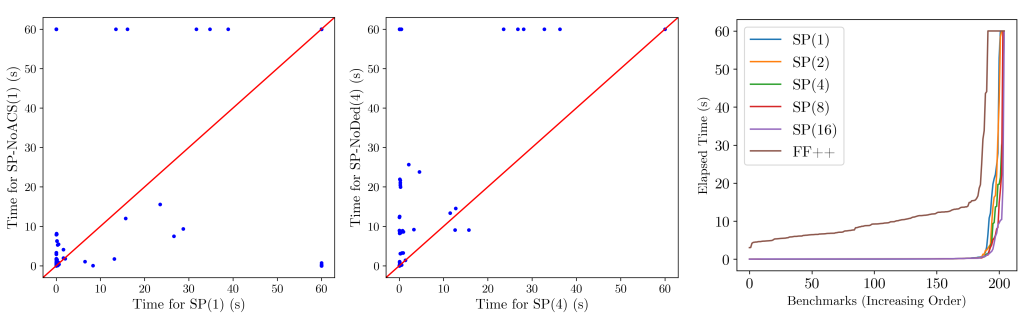
Effectiveness of Delayed Deduction. Fig 14a shows the performance of SYNTHPHONIA with and without delayed deduction (mentioned in Section 6). From the figure, we can see with the help of delayed deduction, more benchmark can be solved by SYNTHPHONIA.

Effectiveness of Adaptive size limit. Fig 14b shows SYNTHPHONIA with different $\theta_{\text{tree-size}}$ increase rate. In the figure, we use SP$(4)/m$s to denote SP(4) with $\theta_{\text{tree-size}}$ increase by 1 for every m seconds without a new partial solution added into *sols*. According to the figure, limiting $\theta_{\text{tree-size}}$ can effectively reduce the number of overfitting benchmarks.



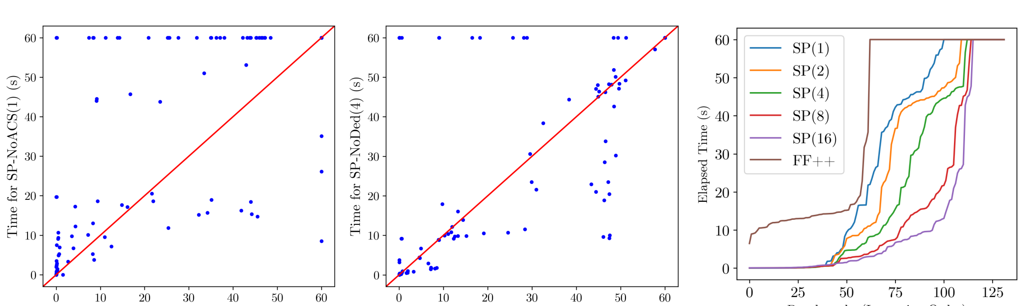
(a) Number of benchmarks solved and overfit. (b) Solution time (Benchmarks in increasing order).

Fig. 10. Experimental results on *HardBench* Benchmarks.



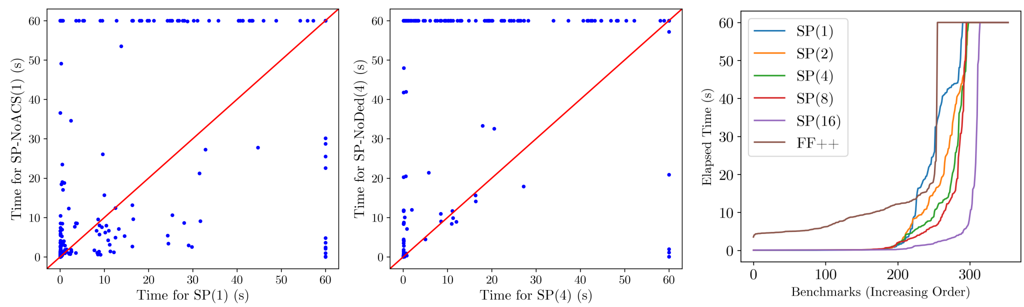
(a) Accumulative case-splitting. (b) Asynchronous deduction. (c) Scalability with multithreading.

Fig. 11. Ablation study for Duet Benchmarks.



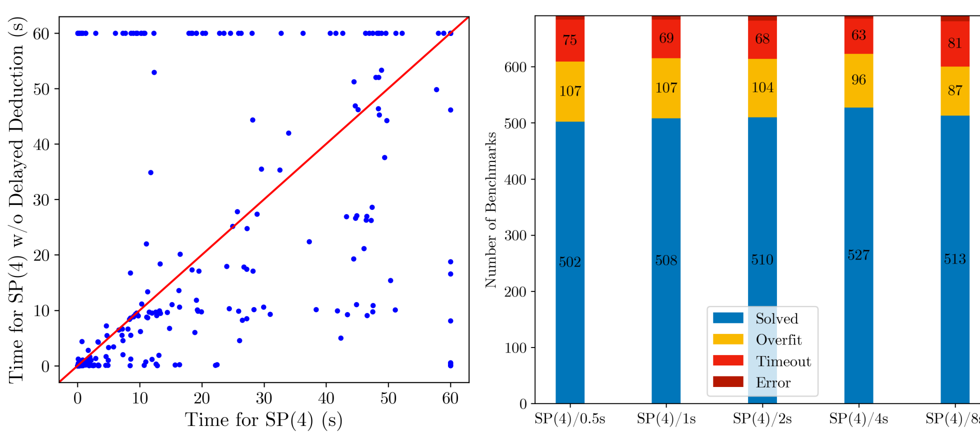
(a) Accumulative case-splitting. (b) Asynchronous deduction. (c) Scalability with multithreading.

Fig. 12. Ablation study for HardBench Benchmarks.



(a) Accumulative case-splitting. (b) Asynchronous deduction. (c) Scalability with multithreading.

Fig. 13. Ablation study for Prose Benchmarks.



(a) Ablation Study for Delayed Deduction. (b) Statistics for Different increase rate.

Fig. 14. Additional Ablation Study for Implementation Details.

C.4 Solution for A Selected Benchmark

In this section we present a concrete benchmark from HardBench: `novel:year1`. This benchmark was created by one of the co-authors. They asked ChatGPT to list some information about famous novels and create a problem to extract the year of that novel from the input. The input/output examples of this benchmark is shown in Table 4.

It takes SYNTHPHONIA 22 seconds to solve this benchmark. It gives the solution:

```
(define-fun f ((arg0 String)) String
    (ite (= (str.count arg0 (str.++ ")" ")")) 0)
        (list.at (str.split (str.++ (list.at (str.split arg0 ")") 0) "(") "(") 1)
        (ite (= (list.len (list.at (str.split arg0 ")") 1)) 0)
            (str.++
                (list.at (str.split arg0 "(") 1) "("
                (list.at (str.split (list.at (str.split arg0 ")") 0) "(") -1) ")")
            (str.++
                (list.at (str.split arg0 (str.++ " " "(")) 1) " "
                (list.at (str.split arg0 " ") 3) " "
                (int.to.str 1) ")"
                (list.at (str.split arg0 ")") 1)
                ")"))))
```

Input / String	Output/ String
“The Scarlet Letter (1850); Historical Fiction, Allegory”	“1850”
“War and Peace (1869); Historical Fiction, Epic”	“1869”
“The Scarlet Letter (1850) by Nathaniel Hawthorne; Historical Fiction, Allegory”	“1850”
“One Hundred Years of Solitude; Magical Realism, Epic”	“ ”
“The Catcher in the Rye; Coming-of-Age, Bildungsroman”	“ ”
“Moby-Dick (1851); Adventure, Symbolic”	“1851”
“Don Quixote (1605 (Part 1), 1615 (Part 2)); Satire, Adventure”	“1605 (Part 1), 1615 (Part 2)”
“The Lord of the Rings; High Fantasy, Adventure”	“ ”
“The Hobbit by J.R.R. Tolkien; Fantasy, Adventure”	“ ”
“The Hobbit by J.R.R. Tolkien; Fantasy, Adventure”	“ ”
“War and Peace; Historical Fiction, Epic”	“ ”
“War and Peace (1869); Historical Fiction, Epic”	“1869”
“Frankenstein (1818); Gothic Horror, Science Fiction”	“1818”
“The Great Gatsby by F. Scott Fitzgerald; Modernist, Tragedy”	“ ”
“War and Peace; Historical Fiction, Epic”	“ ”
“Anna Karenina; Realist Fiction, Tragedy”	“ ”
“To Kill a Mockingbird; Southern Gothic, Bildungsroman”	“ ”
“The Great Gatsby (1925) by F. Scott Fitzgerald; Modernist, Tragedy”	“1925”
.... (67 in total)	....

Table 4. Input/output examples for benchmark `novel:year1`