FlashMeta: A Framework for Inductive Program Synthesis

Oleksandr Polozov
University of Washington, USA
polozov@cs.washington.edu

Sumit Gulwani
Microsoft Research, USA
sumitg@microsoft.com

Abstract

Inductive synthesis, or programming-by-examples (PBE) is gaining prominence with disruptive applications for automating repetitive tasks in end-user programming. However, designing, developing, and maintaining an effective industrial-quality inductive synthesizer is an intellectual and engineering challenge, requiring 1-2 man-years of effort.

Our novel observation is that many PBE algorithms are a natural fall-out of one generic meta-algorithm and the domain-specific properties of the operators in the underlying domain-specific language (DSL). The meta-algorithm propagates example-based constraints on an expression to its subexpressions by leveraging associated *witness functions*, which essentially capture the inverse semantics of the underlying operator. This observation enables a novel program synthesis methodology called *data-driven domain-specific deduction (D^4)*, where domain-specific insight, provided by the DSL designer, is separated from the synthesis algorithm.

Our FlashMeta framework implements this methodology, allowing synthesizer developers to generate an efficient synthesizer from the mere DSL definition (if properties of the DSL operators have been modeled). In our case studies, we found that 10+ existing industrial-quality mass-market applications based on PBE can be cast as instances of D^4. Our evaluation includes reimplementation of some prior works, which in FlashMeta become more efficient, maintainable, and extensible. As a result, FlashMeta-based PBE tools are deployed in several industrial products, including Microsoft PowerShell 3.0 for Windows 10, Azure Operational Management Suite, and Microsoft Cortana digital assistant.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; D.2.13 [*Software Engineering*]: Reusable Software — Domain Engineering; I.2.2 [*Artificial Intelligence*]: Automatic Programming — Program Synthesis

General Terms Algorithms, Languages

Keywords Inductive program synthesis; programming by examples; frameworks; domain-specific languages; deductive inference; search-based synthesis

1. Introduction

Program synthesis is the task of synthesizing a program in an underlying programming language, given the user's intent in the form of some specification [6]. It has applications to a wide variety of domains, including robotics [20], software development [19], and biology [18]. Unfortunately, the true potential of this area still remains elusive: because of the huge and arbitrary nature of the underlying state space of possible programs, program synthesis is a hard combinatorial search problem. Historically, it has been approached in three ways.

Deductive Synthesis The traditional view of program synthesis has been to synthesize code fragments from *declarative* and *complete* logical specifications [25]. A commonly used approach is *deductive synthesis*, which uses a system of axioms and deductive rules to transform the given logical specification into a constructive program. The process of deducing such a program constitutes a proof that the resulting program satisfies the original specification. Deductive synthesis has been successfully used for well-defined application domains such as numeric computations [29, 30].

The advantage of deductive synthesis is its performance – since the technique mostly operates over partial programs that satisfy the current specification, it seldom needs to reject invalid candidate programs and backtrack. Its disadvantage is the amount of manual effort required to axiomatize the application domain into a sound and complete system of deductive rules. Moreover, writing logical specifications is difficult and often daunting for end users and developers.

Syntax-guided Synthesis *Syntax-guided synthesis* (SyGuS) [1] parameterizes generic synthesis algorithms with domain-specific languages (DSLs), and requires a user to provide a specification for the behavior of desired programs in these DSLs. The SyGuS synthesis strategies then apply some kind of search over the DSL to find a satisfying program [13,

31, 35, 37]. Such a problem definition narrows down the space of possible programs to those that are expressible in a given DSL or in a *template* in this DSL, provided by the user. This drastically increases synthesis performance as compared to a complete exploration of entire program space. Besides, general-purpose research in SyGuS synthesis strategies immediately benefits any domain-specific applications that leverage them.

The main drawback of SyGuS is that generic search algorithms cannot leverage any domain-specific insights for improving the performance of synthesis on a given DSL. Moreover, SyGuS restricts the language of specifications to those expressible in some *SMT theory*, which limits the usability of this approach in rich application domains with arbitrary semantics. An example of such a DSL is the language of CSS selectors [4]. Its modeling in SMT requires encoding the semantics of DOM nodes, browser rendering process, and CSS. As a result, the encoding is too challenging for SyGuS solvers even for simplest real-life queries [12].

Domain-specific Inductive Synthesis *Inductive synthesis* is a sub-area of program synthesis where the specification is provided in the form of examples [23]. This area is gaining prominence with its disruptive potential for enabling end users to write small scripts for automating repetitive tasks from examples. Examples are easier to provide than logical specifications in such applications, albeit at the cost of being incomplete. Recent successful mass-market tools based on inductive synthesis include FlashFill feature in Microsoft Excel 2013 [7], FlashExtract feature in PowerShell for Windows 10 [22], and Trifacta's textual data transformation toolkit (`http://www.trifacta.com`). The effectiveness of these tools is based on their algorithms, which are specialized to the underlying domain. However, a domain-specific inductive synthesizer has 3 key limitations:

I. Developing the synthesis algorithm requires deep domain-specific insights.

II. An efficient and correct implementation of such an algorithm can take up to 1-2 man-years of effort. For example, it took the second author 1 year to develop a robust prototype of FlashFill, and it took the Excel product team 6 more months to adapt the prototype into a production-ready feature. Likewise, FlashExtract is a product of more than a year of research & development.

III. The underlying DSL is hard to extend because even small DSL changes might require non-trivial changes to the synthesis algorithm and implementation.

1.1 Data-Driven Domain-Specific Deduction

The synthesis approaches discussed above differ in their strengths and weaknesses. In this work, we propose a program synthesis methodology called *data-driven domain-specific deduction* (D^4), which unifies the strengths of deductive, syntax-guided, and domain-specific inductive approaches in one meta-algorithm:

- D^4 is *domain-specific*: it operates over a DSL, a syntactically restricted program space.
- D^4 is *deductive*: it works by deductively reducing the synthesis problem to smaller synthesis subproblems.
- D^4 is *data-driven*: the deductive rules of D^4 transform the *data space* (I/O examples in the specification). In contrast, deductive rules of classic deductive synthesis transform the *program space* (partial programs and complete logical specifications).

In [6], we broadly categorized the aspects of a program synthesis problem into three main dimensions: *intent specification*, *program space*, and *search strategy*. For D^4, we borrow each dimension of program synthesis from one of the aforementioned approaches that tackles it most efficiently: the intent is specified inductively (via I/O examples), the program space is syntax-guided, and the search strategy is based on deduction.

Program Space Our program space is *domain-specific*: a space of possible programs is restricted to those expressible in a given functional DSL. As in SyGuS, such a restriction drastically increases performance of our search algorithms on complex real-life scenarios. However, unlike SyGuS, our methodology also permits arbitrarily complex functional DSLs with rich domain-specific semantics, and enables scalable program synthesis for such DSLs (less than a second for common real-life tasks). In §6, we show examples of tools built using our technologies that manipulate regular expressions, DOM nodes, and other rich datatypes.

Specification Our specification is *inductive*: it builds on the notion of "input-output examples", one of the most natural specification forms for end users. We generalize it to "properties of the output on specific input states" in order to enable effortless intent specification in complex scenarios.

Input-output examples $\{\sigma_i, o_i\}_{i=1}^m$ are the most natural specification for end users. A program P is considered valid iff it returns o_i when executed on each input state σ_i. However, a similar specification is not as natural in FlashExtract [22]. A program P in FlashExtract takes as input a textual document d, and returns some sequence of selected spans in d. Thus, in a specification above, each o_i would contain an entire sequence of spans in the corresponding input document. Describing the entire output is time-consuming, error-prone, and might in fact constitute the entire one-off task that the user wants to perform in the first place.

Instead, in our specification the user can specify a property of the output (such as some subset of the desired selected spans for FlashExtract). It is a compromise between input-output examples (too restrictive in many applications) and complete logical specifications (inaccessible for end users). In §3.2, we show more complex scenarios and define inductive specifications formally.

Search Strategy Our main contribution in D^4 is a novel *deductive inference* methodology for program synthesis. It is

based on *witness functions*, which propagate example-based specifications on a DSL operator down into specifications on the operator parameters. They essentially capture (a subset of) the inverse semantics of the underlying operator.

In D^4, we combine this deductive inference with *enumerative search*, a complementary state-of-the-art approach to program synthesis [1, 37]. We were inspired by the success of conflict-driven clause learning in SAT solving [39]. It suggests that an efficient algorithm for solving complex logical problems generally cannot be based on either deduction or search alone, but should employ a combination of both. In this work, we present the first generic strategy for domain-specific inductive synthesis that uses such a combination.

Witness Functions In D^4, deductive inference is performed using the standard algorithmic principle of divide-and-conquer. Given a synthesis request for a program of kind $F(N_1, N_2)$ that satisfies a specification φ, D^4 recursively reduces this problem to simpler subproblems of similar kind: a request for a program of kind N_1 that satisfies a certain new specification φ_1, and a request for a program of kind N_2 that satisfies a certain new specification φ_2. The final result is then constructed from the subproblem results.

The key observation that makes D^4 scalable and usable is that *many reduction logics for deducing specifications for simpler sub-problems depends only on the behavior of the involved operator on the concrete values in the original specification*. They do not depend on the DSL structure, program synthesis algorithms and data structures, or axiomatization of the operator semantics. This observation has two important consequences, which alleviate aforementioned limitations I, II, and III of inductive synthesizers:

- If a reduction logic depends only on the operator behavior, this behavior can be written once for any operator, and then reused for its synthesis in any DSL. This enables modularity in domain-specific program synthesis.
- Since such reduction logic depends purely on domain-specific knowledge (i.e. inverse semantics of DSL operators) without details of overall search methodology, it can be written by a domain expert who is not proficient in program synthesis or designing search algorithms.

We call such domain-specific knowledge a *witness function*. The term has first appeared in FlashExtract, where it specifically denoted a function that *witnessed* intermediate inputs for f given I/O examples for the operator $\mathsf{Map}(\lambda\, x \Rightarrow f, L)$. In a general case, given some specification (examples or their properties) for an operator $F(N_1, N_2)$, it "witnesses" a necessary (and often sufficient) specification for N_1 or N_2.

A witness function captures the *inverse semantics* for the corresponding operator. For instance, consider the operator $\mathsf{SubStr}(v, p_1, p_2)$, which selects a substring in v given a pair of positions $\langle p_1, p_2 \rangle$. One of its witness functions, on the other hand, selects a pair of positions $\langle p_1, p_2 \rangle$ given the substring returned by SubStr. In many cases, the operator has no bijective inverse, thus its witness function produces a Boolean statement about the parameters (e.g. a disjunction) instead of concrete values. In §5, we formally define witness functions and the meta-algorithm that leverages them for iterative reduction of the synthesis problem.

D^4 as a Unification of Prior Work In our case studies, we observed that *most* prior work in domain-specific inductive synthesis [2, 3, 5, 7, 8, 16, 21, 22, 28, 32, 33, 38] can be cast as instances of D^4 methodology, in which the original synthesis algorithms naturally arise from the properties of the operators in their DSLs. However, none of the prior works formulated their algorithms as such natural fall-outs; instead their algorithms are manually crafted and defined only for their underlying DSLs. In §6.1, we discuss the benefits new formulations give for implementation of synthesizers.

1.2 The FlashMeta Framework

We implemented D^4 in a declarative framework called FlashMeta to facilitate design, implementation, and maintenance of efficient inductive program synthesizers. FlashMeta requires a synthesizer developer to parameterize the underlying meta-synthesizer algorithm with a DSL, and then automatically generates an inductive synthesizer for this DSL. FlashMeta provides a pre-defined library of generic operators with associated efficient witness functions that can be reused in any conformant DSL. With a pre-defined modeling of properties of various operators, writing a synthesizer becomes an exploration of various design choices in DSL structure. This workflow is similar to *language parsing*, where the burden of producing a domain-specific parser is nowadays carried by parser generators, such as yacc [14].

We released FlashMeta to 9 industrial collaborators who built 5 different inductive synthesizers in FlashMeta for mass-market industrial applications. 3 of these were inspired by existing applications, which provides us a baseline for comparison. The resulting implementations were developed much faster (by a 7x factor on average), partly due to an opportunity to reuse synthesizable portions of underlying DSLs as modules. The applications are also more efficient, maintainable, and extensible than the original ones. In many cases, our collaborators were able to discover optimizations that were not present in the original implementations. We present details of these implementation case studies in §6.1.

We also compared the performance of original systems with their FlashMeta-based counterparts. Notably, even expectations for such a comparison are difficult to formulate. On one hand, one can expect performance to decrease because (a) the new systems have larger feature sets, and (b) the old systems were hand-tuned for a single domain, whereas their reimplementations are instantiations of a general-purpose abstract framework. On the other hand, FlashMeta allows automatic porting of generic algorithmic optimizations to each application developed on top of it. As a result, performance of FlashMeta-based systems varies in a range of $0.5 - 3$x the original ones (with median runtime on real-life scenarios

being < 1 sec). This performance is sufficient for FlashMeta to be successfully used in industry. For example, it is the new reimplementations of FlashFill and FlashExtract that are shipping with Microsoft PowerShell 3.0 for Windows 10. Besides, many new industrial applications have been made possible due to the ease of synthesizer development in FlashMeta.

1.3 Contributions

This paper makes the following contributions:

- We propose a novel strategy for inductive program synthesis using *deductive inference*. It is based on inverse semantics of the DSL operators, expressed in the form of *witness functions*.
- We combine deductive inference and enumerative search in D^4 *methodology*, leveraging complementary strengths of both strategies.
- We extend the *version space algebra* (VSA) data structure, commonly used for memory-efficient representation of the program space in program synthesis. We define three important efficient operations on VSAs: intersection, ranking, and clustering.
- We present the formulation of multiple prior and novel DSLs and their synthesizers as instances of D^4.
- We implement the aforementioned contributions in the FlashMeta framework. It facilitates rapid development of efficient, robust, and maintainable PBE-based applications of industrial quality.
- We discuss performance and development effort for a subset of prior tools that were reimplemented on top of FlashMeta for industrial usage.

2. Illustrative Scenario

In this section, we describe the D^4 approach to each dimension in program synthesis. We first introduce two instances of prior work that can be cast as instances of D^4: FlashFill [7] and FlashExtract [22]. They are used as running examples throughout this paper. We then use this background to illustrate D^4 on real-life motivating examples.

2.1 Background

Figure 1 shows definitions of FlashFill and FlashExtract in the input syntax of our system. This syntax will be explained in detail in §3.1; below, we briefly summarize the structure of both DSLs at a high level.

FlashFill FlashFill is a system for synthesis of string transformations in Microsoft Excel spreadsheets from input-output examples. Each program P in the FlashFill DSL $\mathcal{L}_{\mathsf{FF}}$ takes as input a tuple of *user input strings* $v_1, \ldots, v_k$, and returns an output string.

The FlashFill language $\mathcal{L}_{\mathsf{FF}}$ is structured in three layers. On the topmost layer (omitted in Figure 1), an n-ary Switch operator evaluates n predicates over the input string tuple, and chooses the branch that then produces the output string. Each Switch branch is a *concatenation* of some number of *primitive string expressions*, which produce pieces of the output string. Each such primitive expression can either be a *constant*, or a *substring* of one of the input strings $v_1, \ldots, v_k$. The *position expression* logic for choosing starting and ending positions for the substring operator can either be *absolute* (e.g. "5$^{\text{th}}$ character from the right"), or based on *regular expression matching* (e.g. "the last occurrence of a number").

Example 1 (Adapted from [7, Example 10]). *Consider the problem of formatting phone numbers in a list of attendees:*

Input v_1	*Input v_2*	*Output*
323-708-7700	*Dr. Leslie B. Lamport*	*(323) 708-7700*
(425)-706-7709	*Bill Gates, Sr.*	*(425) 706-7709*
510.220.5586	*George Ciprian Necula*	*(510) 220-5586*

One possible FlashFill program that performs this task is shown below ($\circ$ denotes concatenation):

$$\mathsf{ConstStr}(\text{"("}) \circ \mathsf{Match}(v_1, \text{"\textbackslash d+"}, 1) \circ \mathsf{ConstStr}(\text{")␣"}) \circ \mathsf{Match}(v_1, \text{"\textbackslash d+"}, 2) \circ \mathsf{ConstStr}(\text{"-"}) \circ \mathsf{Match}(v_1, \text{"\textbackslash d+"}, 3)$$

where $\mathsf{Match}(x, r, k)$ *is a* k^{th} *match of a regex* r *in* x. *In* $\mathcal{L}_{\mathsf{FF}}$, $\mathsf{Match}(x, r, k)$ *is an abbreviation for* "`let` $x = v_1$ `in` $\mathsf{SubStr}(x, \langle \mathsf{RegPos}(x, \langle \varepsilon, r\rangle, k), \mathsf{RegPos}(x, \langle r, \varepsilon\rangle, k)\rangle)$". *We also use the notation* $f_1 \circ \ldots \circ f_n$ *for* n*-ary string concatenation:* "$\mathsf{Concat}(f_1,\ \mathsf{Concat}(\ldots, \mathsf{Concat}(f_{n-1},\ f_n)))$".

FlashExtract FlashExtract is a system for synthesis of scripts for extracting data from semi-structured documents. It has been integrated in Microsoft PowerShell 3.0 for release with Windows 10 and in the Azure Operational Management Suite for extracting custom fields from log files. In the original publication, we present three instantiations of FlashExtract; in this work, we focus on the *text* instantiation as a running example, although *Web* and *table* instantiations were also cast in our methodology (see §6.1).

Each program P in the FlashExtract DSL $\mathcal{L}_{\mathsf{FE}}$ takes as input a *textual document* d, and returns a sequence of spans in that document. The sequence is selected based on combinations of `Map` and `Filter` operators, applied to sequences of matches of various regular expressions in d.

Example 2 (Adapted from [22, Examples 1 and 4]). *Consider the textual file shown in Figure 2. One possible* $\mathcal{L}_{\mathsf{FE}}$ *program that extracts a sequence of yellow regions from it is*

$$\mathsf{Map}(\lambda x \Rightarrow \langle \mathsf{AbsPos}(x, 0), \mathsf{AbsPos}(x, -1)\rangle,\ \mathsf{Filter}(\lambda \ell \Rightarrow \mathsf{EndsWith}(\ell, \text{"\textbackslash d+""}),\ \mathsf{SplitLines}(d)))$$

It splits d *into a sequence of lines, filters the lines ending with a number followed by a quote, and on each such line* x *selects a span* $x[0 : -1]$ *(i.e. the entire line* x*).*

2.2 Scenario

We illustrate D^4 on the synthesis problem from Example 1. For simplicity, we only consider the first I/O example given there as our initial specification φ:

Input v_1	**Input v_2**	**Output**
323-708-7700	Dr. Leslie B. Lamport	(323) 708-7700

```
language FlashFill;                                                   (a)

@output string e := f | Concat(f, e);
string f := ConstStr(s)
          | let string x = std.Kth(vs, k) in SubStr(x, pp);
Tuple<int, int> pp := std.Pair(p, p);
int p := AbsPos(x, k) | RegPos(x, rr, k);
Tuple<Regex, Regex> rr := std.Pair(r, r);

@values[FlashFill.Semantics.StaticTokens] Regex r;
@input string[] vs;       string s;       int k;
```

```
string Concat(string f, string e) { return f + e; }                  (b)
string ConstStr(string s) { return s; }
string SubStr(string x, Tuple<int, int> pp) {
    int l = pp.Item1, r = pp.Item2;
    return (l < 0 || r > x.Length) ? null : x.Substring(l, r-l);
}
int AbsPos(string x, int k) { return k < 0 ? x.Length+k+1 : k; }
int? RegPos(string x, Tuple<Regex, Regex> rr, int k) {
    Regex r = new Regex("(?<=" + rr.Item1 + ")" + rr.Item2);
    MatchCollection ms = r.Matches(x);
    int i = k > 0 ? (k - 1) : (k + ms.Count);
    return (i < 0 || i >= ms.Count) ? null : ms[i].Index;
}
static readonly Regex[] StaticTokens = new[] {
  new Regex("\\d+"), new Regex("[a-z]+"), /* more tokens... */ }
```

```
Grammar ff = new Grammar("FlashFill.grammar");                        (c)
ProgramNode p = ff.ParseAST(@"let x = std.Kth(v, 0) in
    SubStr(x, RegPos(x, std.Pair(new Regex(""""),
                             new Regex(@""[A-Z]+"")), -1),
           AbsPos(-1))");
State input = new State(ff.Symbol("vs"),
    new string[] {"Leslie Lamport"});
Assert.Equals(p.Invoke(input), "Lamport");
```

```
language FlashExtract.Text;                                           (d)
using namespace std; using namespace std.list;

StringRegion[] N1 :=
      Map(λ x ⇒ Pair(p, p), Ls)                         // LinesMap
    | Map(λ t ⇒ let string x = GetSuffix(d, t) in
                Pair(t, p), Ps)                          // StartSeqMap
    | Map(λ t ⇒ let string x = GetPrefix(d, t) in
                Pair(p, t), Ps);                         // EndSeqMap
int[] Ps := FilterInt(i0, k, RRs);
int[] RRs := RegexMatches(d, rr);
StringRegion[] Ls := FilterInt(i0, k, FltLs);
StringRegion[] FltLs := Filter(λ ℓ ⇒ B, AllLs);
StringRegion[] AllLs := SplitLines(d);

@extern[FlashFill] int p;          // FV(p) = {x: string}
@extern[text.matches] bool B;      // FV(B) = {ℓ: StringRegion}
@input StringRegion d;       int i0;       int k;
```

Figure 1: **(a)** A DSL of FlashFill substring extraction $\mathcal{L}_{\mathsf{FF}}$. **(b)** Executable semantics of FlashFill operators, defined by the DSL designer in C#, and a set of possible values for the terminal r. **(c)** Example program usage in $\mathcal{L}_{\mathsf{FF}}$. The program selects the last capitalized word (a substring starting from the last capital letter) from the first input string v_1. **(d)** FlashExtract DSL $\mathcal{L}_{\mathsf{FE}}$ for selection of spans in a textual document d. It references position extraction logic p from $\mathcal{L}_{\mathsf{FF}}$ and standard string predicates B.

```
DLZ - Summary Report
"Sample ID:,""5007-01"""
"Sample Date/Time:,""Wednesday, May 30, 2006 00:43:51"""
Intensities
"I/S,""Analyte"",""Mass"",""Conc. Mean"",""Unit"",""Conc. SD"",""RSD"",""Mean"""
"|-,""Be"",9,0.070073,""ug/L"",0.009,12.542,121.334"
"|>,""Sc"",45,,""ug/L"",,,404615.043"
"| ,""Ti"",48,10.653153,""ug/L"",0.847,7.949,181379.200"
"|-,""Se"",82,1.009204,""ug/L"",0.026,2.613,457.487"
"|-,""Sr"",88,20.163079,""ug/L"",2.005,9.943,718014.023"
"|>,""Rh"",103,,""ug/L"",,,438976.176"
DLZ - Summary Report
"Sample ID:,""5007-02"""
"Sample Date/Time:,""Wednesday, May 30, 2006 01:02:38"""
Intensities
"I/S,""Analyte"",""Mass"",""Conc. Mean"",""Unit"",""Conc. SD"",""RSD"",""Mean"""
"| ,""Mn"",55,71.705740,""ug/L"",0.350,0.489,2428667.736"
"| ,""Co"",59,0.131132,""ug/L"",0.004,3.315,3606.816"
"|-,""Ba"",138,129.339264,""ug/L"",3.088,2.387,4648771.382"
"|-,""Hf"",178,,""ug/L"",,,338359.496"
"| ,""Tl"",205,2.876992,""ug/L"",0.730,25.380,129217.588"
"| ,""Pb"",208,3.671043,""ug/L"",0.026,0.702,228830.402"
```

Figure 2: An illustrative scenario for data extraction from semi-structured text using FlashExtract [22, Figure 1].

Below, we denote this spec as $\sigma \rightsquigarrow$ "`(323)␣708-7700`", where σ is the given input state $\{v_1 \mapsto$ "`323-708-7700`", $v_2 \mapsto$ "`Dr.␣Leslie␣B.␣Lamport`"$\}$.

Learning* Concat *Expressions The synthesis starts at the root symbol of $\mathcal{L}_{\mathsf{FF}}$, called e in Figure 1. We are looking for all programs of kind e that satisfy the examples φ (written $e \models \varphi$). According to the $\mathcal{L}_{\mathsf{FF}}$ definition, a program rooted at e can either be of kind f or of kind $\mathsf{Concat}(f, e)$. We reduce the problem $e \models \varphi$ into two independent subproblems $f \models \varphi$ and $\mathsf{Concat}(f, e) \models \varphi$. The union of their results constitutes all possible programs of kind e that satisfy φ.

Consider first the subproblem $\mathsf{Concat}(f, e) \models \varphi$. It involves synthesizing two string programs f and e such that the program f produces some prefix of the output example in φ and the program e produces the remaining suffix. Assume that we know the exact positions in the output string where it is split into a prefix and a suffix being concatenated (i.e. the exact outputs of f and e on the given input state σ). In this case we can reduce our problem straightforwardly to subproblems $f \models \varphi_f$ and $e \models \varphi_e$, where φ_f and φ_e are I/O examples with the prefix and the suffix, respectively. However, there are 13 possible positions that split "`(323) 708-7700`" into two non-empty strings. Thus, we split the synthesis process into 13 branches.

The first witness function $\omega_f(\varphi)$ produces a specification φ_f for the prefix program. On the same input state σ it requires the program's output to equal any of the 13 non-empty prefixes. This is a disjunctive specification:

$$\omega_f(\varphi) = \sigma \rightsquigarrow \text{"(" } \vee \text{ "(3" } \vee \ \ldots \ \vee \text{ "(323)␣708-7700"}$$

Note that independently considering 13 possible suffixes is inefficient and unproductive (f and e are not independent, hence concatenation of "any prefix" with "any suffix" does not necessarily produce the intended result). Instead, on each of 13 branches independently, we fix the assumption about a produced prefix, recursively learn a set of programs for it, and then immediately deduce the corresponding suffix example.

The witness function for the suffix program is thus dependent on the specific value of the prefix. For instance:

$$\omega_e(\varphi \mid f = \text{“(323)␣”}) = (\sigma \rightsquigarrow \text{“708-7700”})$$

In other words, ω_f can be formulated as "*If* $\mathsf{Concat}(f, e) =$ *“(323)␣708-7700”, then f must have produced some prefix of “(323)␣708-7700”*". Likewise, ω_e can be formulated as "*If* $\mathsf{Concat}(f, e) =$ *“(323)␣708-7700” and we know that f produced “(323)␣”, then e must have produced “708-7700”*". Note that both witness functions require only domain-specific knowledge about the behavior of Concat operator and do not specify any details of the synthesis methodology.

Consider now a tree of recursive calls produced by subproblems of kind $f \models \varphi$ and $e \models \varphi$. Learning for the latter subproblem follows the same reduction process over two DSL productions for e, outlined above. Thus, the divide-and-conquer process will generate a subproblem $f \models \sigma \rightsquigarrow s[i : j]$ for each substring $s[i : j]$ of the output $s =$ "(323)␣708-7700". The solution sets of these subproblems can be cached, thereby turning the tree of recursive synthesis calls into a DAG. This DAG has $\mathcal{O}(|s|^2)$ nodes, whereas naïve recursion produces a tree of size $\mathcal{O}(2^{|s|})$. Note that this DAG-based optimization (a key idea in FlashFill [7]) arises automatically within the D^4 methodology.

Learning Primitive String Expressions Consider now each individual subproblem $f \models \varphi_{ij}$ where $\varphi_{ij} = \sigma \rightsquigarrow o[i : j]$. For instance, let $i = 13$, $j = 14$, and $o[i : j] =$ "0", the last character. Following the DSL, we again reduce this problem to two subproblems corresponding to two productions of f: constant expressions and substring expressions.

In the first subproblem, $\mathsf{ConstStr}(s) \models \varphi_{ij}$, we invoke the corresponding witness function for the parameter symbol s. It returns a trivial specification: "*s must have produced “0”, the constant value of $o[13 : 14]$*". This branch of synthesis is now completed, producing a single program ConstStr("0"), which satisfies φ_{ij} by construction.

In the second subproblem, the learning task is "`let` $x = \mathsf{Kth}(vs, k)$ `in` $\mathsf{SubStr}(x, pp) \models \varphi_{ij}$". A `let` expression first calculates the value for a binding, then binds the variable x to it, and evaluates the body. Thus, the only unknown specification to be deduced is the specification for $\mathsf{Kth}(vs, k)$ because the body of the `let` must satisfy the same constraints as the `let` itself (only on an extended input state). At this point, D^4 can proceed either with *deduction* or *search*.

Deductive approach requires a witness function ω_x for the `let` binding. In FlashFill, one possible implementation of ω_x is returning a disjunction of all input strings v_j that contain "0" as a substring. In our example, the new specification φ_x contains only the input v_1. Note that this deduction scheme is only possible because in $\mathcal{L}_{\mathsf{FF}}$, x can be bound only to input string values. If this assumption does not hold, like in the modification below, the logic of ω_x may be more complicated.

```
string f := ConstStr(s) | let string x = y in SubStr(x, pp);
string y := v | ToUpper(v) | ToLower(v);
string v := std.Kth(vs, k);
```

Search-based approach enumerates all possible subexpressions in $\mathcal{L}_{\mathsf{FF}}$ for the binding value, and retains those that make the outer program satisfy the specification φ_{ij} using the corresponding value of x. For example, in the $\mathcal{L}_{\mathsf{FF}}$ modification above, there are 6 possible programs for the x binding: $\mathsf{Kth}(vs, 1)$, $\mathsf{Kth}(vs, 2)$, $\mathsf{ToUpper}(\mathsf{Kth}(vs, 1))$, $\mathsf{ToUpper}(\mathsf{Kth}(vs, 2))$, $\mathsf{ToLower}(\mathsf{Kth}(vs, 1))$, and $\mathsf{ToLower}(\mathsf{Kth}(vs, 2))$. However, on the given input state they produce only 4 distinct outputs, since $\mathsf{ToUpper}(v_1) = \mathsf{ToLower}(v_1) = v_1$. Therefore, we *cluster* all possible subexpressions for the x binding w.r.t. their output on σ, and use these output values to further guide the deductive exploration for $\mathsf{SubStr}(x, pp)$. This enumerative approach is implemented efficiently using dynamic programming [1, 37] and our novel operations on the data structure that succinctly stores the program space of a DSL (see §4). As a result, synthesis splits into 4 branches. The union of their results constitutes all substring programs that satisfy φ_{ij}.

3. Problem Definition

We start with a description of functional *domain-specific languages* supported by FlashMeta in §3.1. We proceed with introducing *inductive specifications* in §3.2 – incomplete specifications of program behavior that describe its properties on a set of representative inputs. Finally, we define the main problem of this paper.

3.1 Domain-Specific Language

A synthesis problem is defined for a given *domain-specific language* (DSL) $\mathcal{L}$. A DSL is specified as a context-free grammar (CFG), where every symbol N is defined through a set of *rules*. Each rule has on its right-hand side an application of an *operator* to some symbols of $\mathcal{L}$, and we denote the set of all possible operators on the right-hand sides of the rules for N as $\mathsf{RHS}(N)$. Every symbol N in this CFG is annotated with a corresponding output type τ, denoted $N : \tau$. We require every operator in $\mathsf{RHS}(N)$ to have the same return type τ, which is the output type of N. A DSL $\mathcal{L}$ also has a designated *input symbol* $\mathsf{input}(\mathcal{L})$, which is a terminal in the CFG of $\mathcal{L}$, and a designated *output symbol* $\mathsf{output}(\mathcal{L})$, which is the topmost nonterminal in the CFG of $\mathcal{L}$.

Every (sub-)program P rooted at a symbol $N : \tau$ in $\mathcal{L}$ maps an *input state* σ to a value of type τ. A state for a program P is a mapping of its free variables $\mathsf{FV}(P)$ to values of their types. By definition, the only free variable of $\mathsf{output}(\mathcal{L})$ is the input symbol $\mathsf{input}(\mathcal{L})$. The other free variables in the DSL are introduced by `let` definitions and λ-functions (described below). A DSL program $P \in \mathcal{L}$ is a function that maps a state $\sigma = \{\mathsf{input}(\mathcal{L}) \mapsto v\}$ to a value of type τ_o, where $\mathsf{output}(\mathcal{L}) : \tau_o$, $\mathsf{input}(\mathcal{L}) : \tau_i$, and $v : \tau_i$.

The language of DSL definitions is given in Figure 3. Apart from user-defined black-box operators, it includes some special constructs: `let` definitions, λ-functions, variable terminals (including an input symbol), external operator

decl ::= *annotation** *type symbol* (**:=** *nonterminal-body*)? **;**
annotation ::= @**input** | @**output** | @**extern**[*namespace*]
| @**values**[*member-name*] | . . .
nonterminal-body ::= *rule* (**|** *rule*)*
rule ::= *symbol*
| (*namespace* **.**)? *operator-name*((*arg* (**,** *arg*)*)?)
| **let** *type symbol* **=** *rule* **in** *rule*
arg ::= *symbol* | **λ***symbol***:** *type* **⇒** *rule*
symbol, operator-name, namespace ::= ⟨id⟩
member-name ::= ⟨member in a target language⟩
type ::= ⟨type in a target language⟩

Figure 3: FlashMeta DSL definition language. A DSL is a set of (typed and annotated) symbol definitions, where each symbol is either a terminal, or a nonterminal defined through a set of rules. Each rule is a conversion of nonterminals, an operator with some arguments (symbols or λ-functions), or a `let` definition. Some auxiliary instructions are omitted for brevity, such as namespace imports or library references.

references (written as *namespace.operator-name*) from the standard library of FlashMeta or from third-party DSLs and modules. These components increase the expressiveness of DSLs, allowing us to capture the semantics of many practical domains in a clean and functional manner.

Example 3. *The original FlashFill DSL definition required two different semantics interpretations: one for the* `SubStr` *operator on a state, and one for the* `AbsPos`/`RegPos` *operators to bind the specific input string that is being processed [see 7, Figure 2]. In Figure 1, to make $\mathcal{L}_{\mathsf{FF}}$ functional, we introduce a* `let` *binding in the* `SubStr` *operator, capturing the specific input string in a variable symbol x. Operators* `AbsPos` *and* `RegPos` *accept x as a parameter.*

Example 4. *Figure 1 also shows an example of DSL modularity (reusing existing DSLs as sublanguages). In it, we define the FlashExtract DSL $\mathcal{L}_{\mathsf{FE}}$ on top of the existing position extraction logic from $\mathcal{L}_{\mathsf{FF}}$ and text predicates in the standard library of FlashMeta. Note how $\mathcal{L}_{\mathsf{FE}}$ binds a variable x before referencing p in $\mathcal{L}_{\mathsf{FF}}$ to make the usage conformant with free variables $\mathsf{FV}(p)$ in $\mathcal{L}_{\mathsf{FF}}$.*

Every (non-external) DSL operator requires an associated executable semantics function, implemented in a *target programming language* (currently C#). We require it to be deterministic and pure, modulo unobservable side effects.

Symbols and rules may be augmented with multiple custom annotations, such as:

- **`@output`** – marks the output symbol of a DSL;
- **`@input`** – marks the input symbol of a DSL;
- **`@values[`***member-name***`]`** – specifies a member (e.g. a field, a static variable, a property) in a target language that stores a set of possible values for the given terminal.
- **`@extern[`***namespace***`]`** – specifies the external namespace (e.g. a third-party DSL) that defines a given symbol;

Notation Throughout the paper, we denote DSL symbols as N, programs as P, concrete values as v, and sets of programs rooted at symbol N as $\widetilde{N}$. When we refer to symbols in a specific DSL (such as $\mathcal{L}_{\mathsf{FF}}$ or $\mathcal{L}_{\mathsf{FE}}$), we usually use their IDs from the DSL definition instead (such as Ps or rr) when the context is clear.

The execution result of a program P on a state σ is written as $[\![P]\!]\sigma$. Updating a state σ with binding $N \mapsto v$ is written as $\sigma[N := v]$. A fixed-length tuple of input states is written as $\vec{\sigma}$. If each element v_i in a tuple $\vec{v}$ has the same type τ, it is denoted as $\vec{v} \colon \vec{\tau}$. We also lift the program execution notation $[\![\cdot]\!]$ onto tuples of states to produce tuples of values: $[\![P]\!]\vec{\sigma} \stackrel{\text{def}}{=} \vec{v}$ where $v_i = [\![P]\!]\sigma_i$. Finally, $\mathcal{L}|_N$ denotes a sublanguage of $\mathcal{L}$ induced by all symbols & operators reachable from N.

3.2 Inductive Specification

Inductive synthesis or *programming-by-examples* traditionally refers to the process of synthesizing a program from a specification that consists of input-output examples. Tools like FlashFill [7], IGOR2 [17], Magic Haskeller [15] fall in this category. Each of them accepts a conjunction of pairs of concrete values for the input state and the corresponding output. We generalize this formulation in two ways: **(a)** by extending the specification to *properties* of program output as opposed to just its *value*, and **(b)** by allowing arbitrary boolean connectives instead of just conjunctions.

Generalization **(a)** is useful when completely describing the output on a given input is too cumbersome for a user. For example, in FlashExtract the user provides instances of strings that should (or should not) *belong* to the output list of selections in a textual document. Describing an entire output list would be too time-consuming and error-prone.

Generalization **(b)** arises as part of the *problem reduction* process that happens internally in the synthesizer algorithms. Specifications on DSL operators get refined into specifications on operator parameters, and the latter specifications are often shaped as arbitrary boolean formulas. For example, in FlashFill, to synthesize a substring program that extracts substrings s from the given input strings v, we synthesize a position extraction program that returns *any* occurrence of s in v (which is a *disjunctive* specification).

Boolean connectives may also appear in the top-level specifications provided by users. For instance, in FlashExtract, a user may provide a *negative example* (an element *not* belonging to the output list).

Definition 1 (Inductive specification)**.** Let $N \colon \tau$ be a symbol in a DSL $\mathcal{L}$. An *inductive specification* ("ispec") φ for a program rooted at N is a quantifier-free first-order predicate of type $\vec{\tau} \to \mathsf{Bool}$ in NNF with n atoms $\langle \sigma_1, \pi_1 \rangle, \ldots, \langle \sigma_{|\vec{\tau}|}, \pi_{|\vec{\tau}|} \rangle$. Each atom is a pair of a concrete input state σ_i over $\mathsf{FV}(N)$ and an *atomic specification* $\pi_i \colon \tau \to \mathsf{Bool}$ constraining the output of a desired program on σ_i.

Definition 2 (Valid program)**.** We say that a program P *satisfies* a specification φ (written $P \models \varphi$) iff the formula

$\varphi[\pi_i(\llbracket P \rrbracket \sigma_i) \ / \ \langle \sigma_i, \pi_i \rangle]$ holds. In other words, φ should hold as a boolean formula over the statements "the output of P on σ_i satisfies the constraint π_i" as atoms. We often denote the output being constrained as $\llbracket \cdot \rrbracket$, assuming implicit P and σ.

Definition 3 (Valid program set)**.** A program set $\widetilde{N}$ is *valid* for ispec φ (written $\widetilde{N} \models \varphi$) iff all programs in $\widetilde{N}$ satisfy φ.

Example 5. *Given a set of I/O examples $\{\sigma_i, o_i\}_{i=1}^m$, a program P is valid iff $\llbracket P \rrbracket \sigma_i = o_i$ for all i. In our formulation, such a specification is represented as a conjunction $\varphi = \bigwedge_{i=1}^m \langle \sigma_i, \pi_i \rangle$. Here each atomic specification π_i on the program output is an **equality predicate**: $\pi_i \stackrel{\text{def}}{=} (\llbracket \cdot \rrbracket = o_i)$.*

Example 6. *In FlashExtract, a program's output is a sequence of spans $\langle l, r \rangle$ in a text document D. A user provides a sequence of **positive** examples U (a subsequence of spans from the desired output), and a set of **negative** examples X (a set of spans that the desired output should not intersect).*

*The ispec here is a conjunction $\varphi = \langle \sigma, \pi^+ \rangle \wedge \langle \sigma, \pi^- \rangle$, where positive atomic specification π^+ is a **subsequence predicate**: $\pi^+ \stackrel{\text{def}}{=} (\llbracket \cdot \rrbracket \sqsubset U)$, and negative atomic specification π^- is a **non-intersection predicate**: $\pi^- \stackrel{\text{def}}{=} (\llbracket \cdot \rrbracket \cap X = \emptyset)$. The input states for both atomic specifications are equal to the same $\sigma = \{d \mapsto D\}$, where symbol $d = \mathsf{input}(\mathcal{L})$.*

Often φ can be decomposed into a conjunction of output properties that must be satisfied on *distinct* input states. In this case we use a simpler notation $\varphi = \{\sigma_i \rightsquigarrow \pi_i\}_{i=1}^m$. A program P satisfies this φ if its output on each input state σ_i satisfies the constraint π_i. For instance, I/O examples are written as $\{\sigma_i \rightsquigarrow o_i\}_{i=1}^m$, and subsequence specifications of FlashExtract are written as $\sigma \rightsquigarrow (\llbracket \cdot \rrbracket \sqsubset U) \wedge (\llbracket \cdot \rrbracket \cap X = \emptyset)$.

Because an ispec is a special case of an arbitrary Boolean predicate over σ and $\llbracket P \rrbracket \sigma$, it can be given to any general-purpose synthesis algorithm that expects a declarative complete specification. However, besides being easier to specify for end users, ispecs also provide two additional benefits that allow us to design more efficient synthesis algorithms:

1. *Representative inputs* $\vec{\sigma}$, embedded in an ispec, prune the search space, allowing us to only consider program behavior on $\vec{\sigma}$ as opposed to analyzing the program behavior over all possible inputs. This property enables enumerative search strategies [37], which cluster semantically equivalent subexpressions w.r.t. $\vec{\sigma}$.[1]
2. An ispec has a restricted shape: it defines a boolean formula over *atomic properties of program outputs on representative inputs*. This benefit is hard to exploit for general-purpose synthesis algorithms, as even a simple ispec turns into a challenging gigantic formula when the semantics of all involved operators and datatypes is expressed in terms of SMT primitives. Our D^4 methodology exploits this shape to make intelligent subgoal choices during deductive program space exploration.

[1] It is noteworthy that many general-purpose synthesis algorithms, given a declarative complete specification, implement their search by *guessing* a representative input σ in a *counterexample-guided synthesis loop*, and then searching in the subspace of programs that give the same output on σ [1].

3.3 Domain-Specific Synthesis

We are now ready to formally define the problem of *domain-specific synthesis*, motivated and tackled in this paper.

Problem 1 (Domain-specific synthesis)**.** *Given a DSL $\mathcal{L}$, and a **learning task** $\langle N, \varphi \rangle$ for a symbol $N \in \mathcal{L}$, the **domain-specific synthesis** problem $\mathsf{Learn}(N, \varphi)$ is to find some set $\widetilde{N}$ of programs in $\mathcal{L}$, valid for φ.*

Some instances of domain-specific synthesis include:

- *Finding **one** program: $|\widetilde{N}| = 1$.*
- *Finding **all** programs (i.e. **complete synthesis**).*
- *Finding **k topmost-ranked** programs according to some domain-specific **ranking function** $h \colon \mathcal{L} \to \mathbb{R}$.*

Problem 1 might be perceived as a special case of syntax-guided synthesis [1]. However, our formulation and approach differ from SyGuS in two important ways.

Deduction-based Approach All known approaches for solving the SyGuS problem employ some kind of search over the program space $\mathcal{L}$. In particular, the participants of the SyGuS-COMP 2014 competition employed: SMT-based search [13], enumerative search [37], sketching [35], and stochastic search [31]. In general, such techniques do not scale to real-life DSLs, described in §1. In D^4, we combine them with domain-specific deductive inference, leveraging both domain knowledge and efficient search techniques.

Set-based Formulation For most applications of the SyGuS problem, it suffices to find any program $P \in \mathcal{L}$ that satisfies the specification φ. However, in end-user PBE applications such as FlashFill and FlashExtract, a typical synthesis problem is satisfied by thousands of ambiguous programs in the DSL. Thus, the main challenge there historically lies in learning a program that is not only consistent with the provided examples φ, but is also most likely to be correct on unseen input data. Problem 1 in PBE is only a single iteration of the *counterexample-guided inductive synthesis loop*, where the overall goal of the loop is to converge to desired end-user program in fewest rounds (end user's specification amendments). To achieve it, inductive synthesizers compute *all* (or *many*) programs in $\mathcal{L}$ that satisfy φ, and then disambiguate among them to satisfy the overall user intent.

One possible approach to disambiguation is ranking. It chooses the right program by maximizing the value of some domain-specific *ranking function* h on P. The function h approximates the likelihood of P to be correct on unseen data. It may be based on prior probability distributions over $\mathcal{L}$ and/or the execution results of P. Designing a good ranking function is a challenging open problem; one promising approach is using machine learning [34], but it requires a sufficient training set of synthesis scenarios.

A second approach to disambiguation is novel user interaction models. For instance, one can paraphrase the ambiguous

program set in English, and display it to the user in such a way that she can agree with a correct portion of a chosen program, but replace an incorrect portion with a better alternative. Another possible model is *active learning*: a system executes several ambiguous programs on the user's input and asks the user to select a correct output among their results (thereby providing an additional example). We have implemented these interaction models in a novel UI on top of FlashMeta and discovered that they significantly increase the correctness of PBE in solving the users' problems [26].

4. Version Space Algebra

A typical practical DSL may have up to 10^{30} programs consistent with a given ispec [33]. Any synthesis strategy for solving Problem 1 requires a data structure for succinctly representing such a huge number of programs in polynomial space. Such a data structure, called *version space algebra* (VSA), was defined by Mitchell [27] in the context of machine learning, and later used for programming by demonstration in SMARTedit [21], FlashFill [7], and other synthesis applications. Its efficiency is based on the fact that typically many of these 10^{30} programs share common subexpressions. In this section, we formalize the generic definition of VSA as an essential primitive for synthesis algorithms, expanding upon specific applications that were explored previously by Mitchell, Lau et al., and us. We define a set of efficient operations over this data structure that are used by several synthesis algorithms, including D^4 (§5).

Intuitively, a VSA can be viewed as a directed graph where each node represents a set of programs. A leaf node in this graph is annotated with a *direct* set of programs, and it explicitly represents this set. There are also two kinds of internal (constructor) nodes. A *union* VSA node represents a set-theoretic union of the program sets represented by its children VSAs. A *join* VSA node with k children VSAs is annotated with a k-ary DSL operator F, and it represents a cross product of all possible applications of F to k parameter programs, independently chosen from the children VSAs. Hereinafter we use the words "program set" and "version space algebra" interchangeably, and use the same notation for both concepts.

Definition 4 (Version space algebra)**.** Let N be a symbol in a DSL $\mathcal{L}$. A *version space algebra* is a representation for a set $\widetilde{N}$ of programs rooted at N. The grammar of VSAs is:

$$\widetilde{N} \;:=\; \{P_1, \ldots, P_k\} \mid \mathsf{U}(\widetilde{N}_1, \ldots, \widetilde{N}_k) \mid F_{\bowtie}(\widetilde{N}_1, \ldots, \widetilde{N}_k)$$

where F is any k-ary operator in $\mathcal{L}$. The semantics of VSA as a set of programs is given as follows:

$$P \in \{P_1, \ldots P_k\} \qquad \text{if } \exists\, j\colon P = P_j$$

$$P \in \mathsf{U}(\widetilde{N}_1, \ldots, \widetilde{N}_k) \qquad \text{if } \exists\, j\colon P \in \widetilde{N}_j$$

$$P \in F_{\bowtie}(\widetilde{N}_1, \ldots, \widetilde{N}_k) \quad \text{if } P = F(P_1, \ldots, P_k) \wedge \forall j\colon P_j \in \widetilde{N}_j$$

Definition 5 (VSA size, width, and volume)**.** Given a VSA $\widetilde{N}$, the *size* of $\widetilde{N}$, denoted $|\widetilde{N}|$, is the number of programs in the set represented by $\widetilde{N}$, the *width* of $\widetilde{N}$, denoted $W(\widetilde{N})$, is the maximum number of children in any constructor node of $\widetilde{N}$, and the *volume* of $\widetilde{N}$, denoted $V(\widetilde{N})$, is the number of nodes in $\widetilde{N}$.

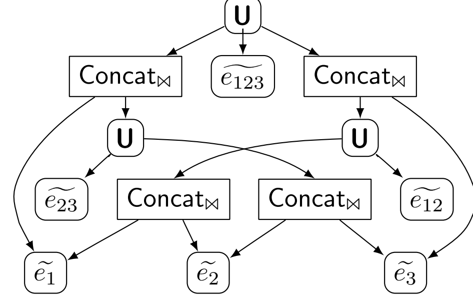

Figure 4: A VSA representing all possible programs output the string "425" in $\mathcal{L}_{\mathsf{FF}}$. Leaf nodes $\widetilde{e_1}, \widetilde{e_2}, \widetilde{e_3}, \widetilde{e_{12}}, \widetilde{e_{23}}$ are VSAs representing all ways to output substrings "4", "2", "5", "42", and "25" respectively (not expanded in the figure).

Note that programs in $\widetilde{N}$ can benefit from two kinds of sharing of common subexpressions. One sharing happens via join VSA nodes, which succinctly represent a cross product of possible subexpression combinations. Another sharing happens by virtue of having multiple incoming edges into a VSA node (i.e. two identical program sets are represented with one VSA instance in memory). Therefore in common (non-degenerate) cases $V(\widetilde{N}) = \mathcal{O}(\log|\widetilde{N}|)$.

Example 7. *Figure 4 shows a VSA of all FlashFill programs that output the string "*`425`*" on a given input state* $\sigma = \{v_1 \mapsto$ "`(425) 706-7709`"$\}$. *The string "*`425`*" can be represented in 4 possible ways as a concatenation of individual substrings. Each substring of "*`425`*" can be produced by a* ConstStr *program or a* SubStr *program.*

We define three operations over VSAs, which are used during the synthesis process: *intersection*, *ranking*, and *clustering*. Intersection of VSAs was first used by Lau et al. [21] and then adapted by us for FlashFill [7]. It has been defined specifically for VSAs built on the SMARTedit and FlashFill DSLs. In this section, we define VSA intersection generically, and also introduce ranking and clustering of VSAs, which are our novel contributions.

4.1 Intersection

Intersection of VSAs enables quick unification of two sets of candidate programs that are consistent with two different specifications. Given a conjunctive ispec $\varphi_1 \wedge \varphi_2$, one possible synthesis approach is to learn a set of programs $\widetilde{N}_1$ consistent with φ_1, another set of programs $\widetilde{N}_2$ consistent with φ_2, and then intersect $\widetilde{N}_1$ with $\widetilde{N}_2$. An efficient algorithm for VSA intersection follows the ideas of automata intersection [11].

Definition 6 (Intersection of VSAs)**.** Given VSAs $\widetilde{N}_1$ and $\widetilde{N}_2$, their *intersection* $\widetilde{N}_1 \cap \widetilde{N}_2$ is a VSA that contains those and only those programs that belong to both $\widetilde{N}_1$ and $\widetilde{N}_2$. Constructively, it is defined as follows (modulo symmetry):

$$[\mathsf{U}(\widetilde{N}_1', ..., \widetilde{N}_k')] \cap \widetilde{N}_2 \stackrel{\text{def}}{=} \mathsf{U}\,(\widetilde{N}_1' \cap \widetilde{N}_2, ..., \widetilde{N}_k' \cap \widetilde{N}_2)$$
$$F_{\bowtie}(\widetilde{N}_1', ..., \widetilde{N}_k') \cap G_{\bowtie}(\widetilde{N}_1'', ..., \widetilde{N}_m'') \stackrel{\text{def}}{=} \emptyset$$
$$F_{\bowtie}(\widetilde{N}_1', ..., \widetilde{N}_k') \cap F_{\bowtie}(\widetilde{N}_1'', ..., \widetilde{N}_k'') \stackrel{\text{def}}{=} F_{\bowtie}(\widetilde{N}_1' \cap \widetilde{N}_1'', ..., \widetilde{N}_k' \cap \widetilde{N}_k'')$$
$$F_{\bowtie}(\widetilde{N}_1', ..., \widetilde{N}_k') \cap \{P_1, ..., P_m\} \stackrel{\text{def}}{=} \{P_j = F(P_1', ..., P_k') \mid P_j' \in \widetilde{N}_j'\}$$
$\widetilde{N}_1 \cap \widetilde{N}_2 \stackrel{\text{def}}{=} \widetilde{N}_1 \cap \widetilde{N}_2$ as direct program sets otherwise

Theorem 1. $V(\widetilde{N}_1 \cap \widetilde{N}_2) = \mathcal{O}(V(\widetilde{N}_1) \cdot V(\widetilde{N}_2))$.

4.2 Ranking

Ranking of VSAs enables us to quickly select the topmost-ranked programs in a set of ambiguous candidates with respect to some domain-specific ranking function.

Definition 7 (Ranking of a VSA)**.** Given a VSA $\widetilde{N}$, a ranking function $h\colon \mathcal{L} \to \mathbb{R}$, and an integer $k \geq 1$, the operation $\mathsf{Top}_h(\widetilde{N}, k)$ returns a (sorted) set of programs that correspond to k highest values of h in $\widetilde{N}$.

$\mathsf{Top}_h(\widetilde{N}, k)$ can be defined constructively, provided the ranking function h is *monotonic* over the program structure (i.e. provided $h(P_1) > h(P_2) \Rightarrow h(F(P_1)) > h(F(P_2))$):

$$\mathsf{Top}_h(\{P_1, \ldots, P_m\}, k) \stackrel{\text{def}}{=} \mathsf{Select}(h, k, \{P_1, \ldots, P_m\})$$
$$\mathsf{Top}_h(\mathsf{U}(\widetilde{N}_1, \ldots, \widetilde{N}_m), k) \stackrel{\text{def}}{=} \mathsf{Select}\big(h, k, \cup_{i=1}^n \mathsf{Top}_h(\widetilde{N}_i, k)\big)$$
$$\mathsf{Top}_h(F_{\bowtie}(\widetilde{N}_1, \ldots, \widetilde{N}_m), k) \stackrel{\text{def}}{=} \mathsf{Select}\big(h, k, \{F(P_1, \ldots, P_m) \mid \forall i\ P_i \in \mathsf{Top}_h(\widetilde{N}_i, k)\}\big)$$

Here Select function implements a *selection algorithm* for top k elements among $\{P_1, \ldots, P_m\}$ according to the ranking function h. It can be efficiently implemented either in $\mathcal{O}(m + k \log k)$ time using Hoare's `quickselect` algorithm [9], or in $\mathcal{O}(m \log k)$ time using a heap.

Theorem 2. *Let $\widetilde{N}$ be a VSA, and let $m = W(\widetilde{N})$. Assume $\mathcal{O}(m \log k)$ implementation of the* Select *function. The time complexity of calculating* $\mathsf{Top}(\widetilde{N}, k)$ *is* $\mathcal{O}(V(\widetilde{N})\, k^m \log k)$.

4.3 Clustering

Clustering of a VSA partitions it into subsets of programs that are semantically indistinguishable w.r.t. the given input state σ, i.e. they give the same output on σ. This operation enables efficient implementation of various computations over a set of candidate programs (e.g., filtering or splitting into independent search subspaces).

Definition 8 (Clustering of a VSA)**.** Given a VSA $\widetilde{N}$ and an input state σ over $\mathsf{FV}(N)$, the *clustering* of $\widetilde{N}$ on σ, denoted $\widetilde{N}|_\sigma$, is a mapping $\{v_1 \mapsto \widetilde{N}_1, \ldots, v_n \mapsto \widetilde{N}_n\}$ such that:

(a) $\widetilde{N}_1 \cup \ldots \cup \widetilde{N}_n = \widetilde{N}$.
(b) $\widetilde{N}_i \cap \widetilde{N}_j = \emptyset$ for all $i \neq j$.
(c) For any $P \in \widetilde{N}_j\colon\ [\![P]\!]\sigma = v_j$.
(d) For any $i \neq j\colon\ v_i \neq v_j$.

In other words, $\widetilde{N}|_\sigma$ is a partitioning of $\widetilde{N}$ into non-intersecting subsets of programs, where each subset contains the programs that give the same output v on the given input state σ. We can also straightforwardly lift the clustering operation onto tuples of input states $\vec{\sigma}$, partitioning $\widetilde{N}$ into subsets of programs that given the same output $\vec{v}$ on $\vec{\sigma}$.

Constructively, $\widetilde{N}|_\sigma$ is defined as follows:

$$\{P_1, ..., P_k\}\,|_\sigma \stackrel{\text{def}}{=} \mathsf{G}\big(\{[\![P_i]\!]\sigma \mapsto \{P_i\} \mid i = 1..k\}\big)$$
$$\mathsf{U}(\widetilde{N}_1, ..., \widetilde{N}_k)|_\sigma \stackrel{\text{def}}{=} \mathsf{G}\big(\textstyle\bigcup_k \widetilde{N}_k|_\sigma\big)$$
$$F_{\bowtie}(\widetilde{N}_1, ..., \widetilde{N}_k)|_\sigma \stackrel{\text{def}}{=} \mathsf{G}\big(\{[\![F(v_1, ..., v_k)]\!]\sigma \mapsto F_{\bowtie}(\widetilde{N}_1', ..., \widetilde{N}_k') \mid \langle v_j, \widetilde{N}_j' \rangle \in \widetilde{N}_j|_{\sigma_j}\}\big)$$

where $\sigma_1, \ldots, \sigma_k$ are input states generated by F for its k parameters during execution on σ, and

$$\mathsf{G}(\{v_1 \mapsto \widetilde{N}_1, \ldots, v_n \mapsto \widetilde{N}_n\}) \stackrel{\text{def}}{=} \big\{v \mapsto \textstyle\bigcup_{v \mapsto \widetilde{N}_j} \widetilde{N}_j \mid \text{all unique } v \text{ among } v_j\big\}$$

is a "group-by" function. It groups a set of bindings by keys, uniting VSAs for equal keys with a U constructor.

The clustering operation allows us to answer many questions about the current set of candidate programs efficiently. For example, it allows us to efficiently filter out the programs in $\widetilde{N}$ that are inconsistent with a given specification:

Definition 9 (Filtered VSA)**.** Given a VSA $\widetilde{N}$ and an ispec φ on N, a *filtered VSA* $\mathsf{Filter}(\widetilde{N}, \varphi)$ is defined as follows: $\mathsf{Filter}(\widetilde{N}, \varphi) \stackrel{\text{def}}{=} \{P \in \widetilde{N} \mid P \models \varphi\}$.

Theorem 3. *If $\vec{\sigma}$ is a state tuple associated with φ, then:*
(1) $\mathsf{Filter}(\widetilde{N}, \varphi) = \mathsf{U}\{\widetilde{N}_j \mid \langle \vec{v}_j \mapsto \widetilde{N}_j \rangle \in \widetilde{N}|_{\vec{\sigma}} \wedge \varphi(\vec{v}_j)\}$.
(2) The construction of $\mathsf{Filter}(\widetilde{N}, \varphi)$ *according to (1) takes* $\mathcal{O}(|\widetilde{N}|_{\vec{\sigma}}|)$ *time after the clustering.*

In §5, we show another application of the clustering operation, where it helps us to partition the search space of candidate programs into subspaces with different behavior on a given input state. We can then parallelize the search process over such independent subspaces of programs.

Estimating the size of $\widetilde{N}|_{\vec{\sigma}}$ and the running time of clustering is difficult, since these values depend not only on $\widetilde{N}$ and σ, but also on the semantics of the operators in $\mathcal{L}$ and the number of collisions in their possible output values. However, in §6.2 we show that these values usually remain small for any practical DSL.

5. Methodology

In this section, we introduce the methodology of *data-driven domain-specific deduction* (D^4), which is a generalization of numerous prior works in inductive program synthesis (some of which are described in §6.1). As we observed, most of prior search algorithms can be decomposed into **(a)** domain-specific insights for refining a specification for the operator into the specifications for its parameters, and **(b)** a common deductive meta-algorithm, which leverages these insights for iterative reduction of the synthesis problem.

```
function GenerateSubstring(σ: Input state, s: String)
    result ← ∅
    for all (i, k) s.t. s is substring of σ(v_i) at position k do
        Y_1 ← GeneratePosition(σ(v_i), k)
        Y_1 ← GeneratePosition(σ(v_i), k + Length(s))
        result ← result ∪ {SubStr(v_i, Y_1, Y_2)}
    return result
function GeneratePosition(s: String, k: int)
    result ← {CPos(k), CPos(−(Length(s) − k))}
    for all r_1 = TokenSeq(T_1, ..., T_n) matching s[k_1 : k − 1] for some k_1 do
        for all r_2 = TokenSeq(T'_1, ..., T'_m) matching s[k : k_2] for some k_2 do
            r_12 ← TokenSeq(T_1, ..., T_n, T'_1, ..., T'_m)
            Let c be s.t. s[k_1 : k_2] is the c^th match for r_12 in s
            Let c' be the total number of matches for r_12 in s
            r̃_1 ← GenerateRegex(r_1, s)
            r̃_2 ← GenerateRegex(r_2, s)
            result ← result ∪ {Pos(r̃_1, r̃_2, {c, −(c' − c + 1)})}
    return result
```

```
function Map.Learn(Examples φ: Dict⟨State, List⟨T⟩⟩)
    Let φ be {σ_1 ↦ Y_1, ..., σ_m ↦ Y_m}
    for j ← 1 ··· do
        Witness subsequence Z_j ← Map.Decompose(σ_j, Y_j)
    φ_1 ← {σ[Z_j[i]/x] ↦ Y_j[i] | i = 0..|Z_j| − 1, j = 1..m}
    Ñ_1 ← F.Learn(φ_1)
    φ_2 ← {σ_j ↦ Z_j | j = 1..m}
    Ñ_2 ← S.Learn(φ_2)
    return Map(Ñ_1, Ñ_2)

function Filter.Learn(Examples φ: Dict⟨State, List⟨T⟩⟩)
    Ñ_1 ← S.Learn(φ)
    φ' ← {σ[Y[i]/x] ↦ true | (σ, Y) ∈ φ, i = 0..|Y| − 1}
    Ñ_2 ← F.Learn(φ')
    return Filter(Ñ_2, Ñ_1)
```

Figure 5: *Left:* FlashFill synthesis algorithm for learning substring expressions [7, Figure 7]; *Right:* FlashExtract synthesis algorithm for learning Map and Filter sequence expressions [22, Figure 6]. Highlighted portions correspond to domain-specific computations, which deduce I/O examples for propagation in the DSL by "inverting" the semantics of the corresponding DSL operator. Non-highlighted portions correspond to the search organization, isomorphic between both FlashFill and FlashExtract.

5.1 Intuition

Figure 5 shows a portion of the synthesis algorithms for FlashFill [7, Figure 7] and FlashExtract [22, Figure 6]. Both algorithms use a divide-and-conquer approach, reducing a synthesis problem for an expression into smaller synthesis problems for its subexpressions. They alternate between 3 phases, implicitly hidden in the "fused" original presentation:

1. Given some examples for a nonterminal N, invoke synthesis on all RHS rules of N, and unite the results.
2. *Deduce* examples that should be propagated to some argument N_j of a current rule $N := F(N_1, \ldots, N_k)$.
3. Invoke learning recursively on the deduced examples, and proceed to phase 2 for the subsequent arguments.

Note that phases 1 and 3 do not exploit the *semantics* of the DSL, they only process its *structure*. In contrast, phase 2 uses domain-specific knowledge to deduce propagated examples for N_j from the examples for N. In Figure 5, we highlight the portions of code that correspond to phase 2.

In FlashExtract, list processing operators $\mathsf{Map}(F, S)$ and $\mathsf{Filter}(F, S)$ are learned generically from a similar *subsequence specification* φ of type $\mathsf{Dict}\langle\mathsf{State}, \mathsf{List}\langle T\rangle\rangle$. Their learning algorithms are also shown in Figure 5. FlashExtract provides generic synthesis for 5 such list-processing operators independently of their DSL instantiations. However, the domain-specific insight required to build a specification for the Map's F parameter depends on the specific DSL instantiation of this Map (such as LinesMap or StartSeqMap in $\mathcal{L}_{\mathsf{FE}}$), and cannot be implemented in a domain-independent manner. This insight was originally called a *witness function* – it *witnessed* sequence elements passed to F. We notice, however, that FlashExtract witness functions are just instances of domain-specific knowledge that appears in phase 2 of the common meta-algorithm for inductive synthesis.

Derivation Consider the operator $\mathsf{LinesMap}(F, L)$ in $\mathcal{L}_{\mathsf{FE}}$. It takes as input a sequence of lines L in the document, and applies a function F on each line in L, which selects some substring within a given line. Thus, its simplest type signature is $((\mathsf{String} \to \mathsf{String}),\ \mathsf{List}\langle\mathsf{String}\rangle) \to \mathsf{List}\langle\mathsf{String}\rangle$.

The simplest form of Problem 1 for LinesMap is *"Learn all programs of kind* $\mathsf{LinesMap}(F, L)$ *that satisfy a given I/O example* $\varphi = \sigma \rightsquigarrow v$*"*. To solve this problem, we need to compute the following set of expressions:

$$\{(F, L) \mid \mathsf{LinesMap}(F, L) \models \varphi\} \tag{1}$$

Applying the principle of divide-and-conquer, in order to solve (1), we can compute an *inverse semantics* of LinesMap:

$$\mathsf{LinesMap}^{-1}(v) \stackrel{\text{def}}{=} \{(f, \ell) \mid \mathsf{LinesMap}(f, \ell) = v\} \tag{2}$$

Then, we can recursively learn all programs F and L that evaluate to f and ℓ respectively on the input state σ.

Computation of $\mathsf{LinesMap}^{-1}(v)$ is challenging. One possible approach could be to enumerate all argument values $f \in \textbf{codom}\ F$, $\ell \in \textbf{codom}\ L$, retaining only those for which $\mathsf{LinesMap}(f, \ell) = v$. However, both **codom** L (all string lists) and **codom** F (all String → String functions) are infinite.

In order to finitize the value space, we apply domain-specific knowledge about the behavior of LinesMap in $\mathcal{L}_{\mathsf{FE}}$. Namely, L must evaluate to a sequence of lines in the input document, and F must evaluate to a function that selects a subregion in a given line. Thus, the "strongly-typed" signature of LinesMap is actually $((\mathsf{Line} \to \mathsf{StringRegion}),\ \mathsf{List}\langle\mathsf{Line}\rangle) \to \mathsf{List}\langle\mathsf{StringRegion}\rangle$. This is a *dependent type*, parameterized by the input state σ, which contains our input document. Therefore the inverse semantics $\mathsf{LinesMap}^{-1}$ is also implicitly parameterized with σ:

$$\mathsf{LinesMap}^{-1}(\sigma \rightsquigarrow v) = \left\{ (f, \ell) \;\middle|\; \begin{array}{l} f \text{ and } \ell \text{ can be obtained from } \sigma \\ \text{and } \mathsf{LinesMap}(f, \ell) = v \end{array} \right\}$$

Our implementation of $\text{LinesMap}^{-1}(\sigma \rightsquigarrow v)$ now enumerates over all line sequences ℓ and substring functions f given an input document. Both codomains are now finite. Note that we had to take into account both input and output in the specification to produce a constructive synthesis procedure.

The synthesis procedure above is finite, but not necessarily efficient. For most values of ℓ, there may be no satisfying program L, and therefore even computing matching values of f for it is redundant. Let $v = [r_1, r_2]$. In this case ℓ must be the list of two document lines $[l_1, l_2]$ containing r_1 and r_2, respectively; any other value for ℓ cannot yield v as an output of $\text{LinesMap}(f, \ell)$ regardless of the value of f. In general, there are operators $F(X, Y)$ (e.g. substring extraction), for which computing all viable arguments (x, y) is much slower than computing matching ys only for realizable values of X.

This observation leads to another key idea in D^4: *decomposition* of inverse semantics. Namely, for our LinesMap example, instead of computing $\text{LinesMap}^{-1}(\sigma \rightsquigarrow v)$ explicitly, we ask two simpler questions separately:

1. If $[\![\text{LinesMap}(F, L)]\!]\sigma = v$, what could be the possible output of L?
2. If $[\![\text{LinesMap}(F, L)]\!]\sigma = v$ and we know that $[\![L]\!]\sigma = \ell$, what could be the possible output of F?

The answers to these questions define, respectively, *partial* and *conditional* inverse semantics of LinesMap with respect to its individual parameters:

$$\text{LinesMap}_L^{-1}(v) \supset \{\ell \mid \exists f \colon \text{LinesMap}(f, \ell) = v\} \quad (3)$$

$$\text{LinesMap}_F^{-1}(v \mid L = \ell) = \{f \mid \text{LinesMap}(f, \ell) = v\} \quad (4)$$

The key insight of this technique is that, by the principle of *skolemization* [10], inverse semantics $\text{LinesMap}^{-1}(v)$ can be expressed as a cross-product computation of parameter inverses $\text{LinesMap}_L^{-1}(v)$ and $\text{LinesMap}_F^{-1}(v \mid L = \ell)$ for all possible $\ell \in \text{LinesMap}_L^{-1}(v)$:

$$\begin{aligned} &\text{LinesMap}^{-1}(v) = \{(f, \ell) \mid \text{LinesMap}(f, \ell) = v\} \quad (5) \\ &= \left\{(f, \ell) \mid \ell \in \text{LinesMap}^{-1}(v), f \in \text{LinesMap}_F^{-1}(v \mid L = \ell)\right\} \end{aligned}$$

Such a decomposition naturally leads to an efficient synthesis procedure for $\text{Learn}(\text{LinesMap}(F, L),\ \sigma \rightsquigarrow v)$, which is based on a divide-and-conquer algorithmic paradigm:

1. Enumerate $\ell \in \text{LinesMap}_L^{-1}(v)$.
2. For each ℓ recursively find programs $\widetilde{L}_\ell$ s.t. $[\![L]\!]\sigma = \ell$.
3. Enumerate $f \in \text{LinesMap}_F^{-1}(v \mid L = \ell)$ for all those ℓ for which the program set $\widetilde{L}_\ell$ is non-empty.
4. For each f recursively find programs $\widetilde{F}_{\ell,f}$ s.t. $[\![F]\!]\sigma = f$.
5. For any combination of parameter programs $L \in \widetilde{L}_\ell$, $F \in \widetilde{F}_{\ell,f}$ we now have $\text{LinesMap}(F, L) \models \varphi$ by construction.

In practice, Problem 1 for LinesMap is rarely solved for example-based specifications: as discussed in §3.2, providing the entire output v of regions selected by LinesMap is impractical for end users. Thus, instead of concrete outputs the procedure above should manipulate *inductive specifications with properties of concrete outputs* (e.g. a prefix of the output list v instead of entire v). A corresponding generalization of "inverse semantics of LinesMap" is a function that deduces a specification for L given a specification φ on $\text{LinesMap}(F,\ L)$ (or for F, under the assumption of fixed L). We call such a generalization a *witness function*.

In our LinesMap example, a user might provide us a *prefix specification* $\varphi = [r_1, r_2, \ldots]$ for the output list of regions. A witness function for L in this case follows the same observation above: the output list of L should begin with two lines containing r_1 and r_2. This is also a prefix specification. A witness function for F (conditional on ℓ, the output value of L) is universal for all Maps: it requires the output of F to be a function that maps the ith element of ℓ into the ith element of $[r_1, r_2]$. Such modularity of synthesis (a common witness function for any domain-specific Map operator) is another advantage of decomposing inverse semantics into partial and conditional witness functions.

5.2 Witness Functions

Definition 10 (Witness function)**.** Let $F(N_1, \ldots, N_k)$ be a operator in a DSL $\mathcal{L}$. A *witness function* of F for N_j is a function $\omega_j(\varphi)$ that deduces a *necessary* specification φ_j on N_j given a specification φ on $F(N_1, \ldots, N_k)$. Formally, $\omega_j(\varphi) = \varphi_j$ iff $F(N_1, \ldots, N_k) \models \varphi \implies N_j \models \varphi_j$.[2]

Definition 11 (Precise witness function)**.** A witness function ω_j of $F(N_1, \ldots, N_k)$ for N_j is *precise* if its deduced specification is *necessary and sufficient*. Formally, $\omega_j(\varphi) = \varphi_j$ is precise iff $N_j \models \varphi_j \iff \exists\, N_1, \ldots, N_{j-1}, N_{j+1}, \ldots, N_k \colon F(N_1, \ldots, N_k) \models \varphi$.

Definition 12 (Conditional witness function)**.** A *(precise) conditional witness function* of $F(N_1, \ldots, N_k)$ for N_j is a function $\omega_j(\varphi \mid N_{k_1} = v_1, \ldots, N_{k_s} = v_s)$ that deduces a necessary (and sufficient) specification φ_j on N_j given a specification φ on $F(N_1, \ldots, N_k)$ under the assumption that some other *prerequisite* parameters $N_{k_1}, \ldots, N_{k_s}$ of F have fixed values $v_1, \ldots, v_k$. Formally, $\omega_j(\varphi \mid N_t = v_t) = \varphi_j$ iff $F(N_1, \ldots, N_k)[v_t/N_t] \models \varphi \implies N_j \models \varphi_j$.

Example 8. *Figure 6a shows witness functions for all FlashFill substring extraction operators from Figure 1:*

- *A* $\text{ConstStr}(s)$ *expression returns w iff s is equal to w.*
- *An expression "let $x = \text{Kth}(vs, k)$ in b" returns w iff x is bound to an element of vs that has w as a substring.*
- *A* $\text{SubStr}(x, pp)$ *expression returns w (given that x returns v) iff pp returns a position span of any occurrence of w in v as a substring.*
- *An* $\text{AbsPos}(x, k)$ *expression returns c (given that x returns v) iff k is equal to either c or $c - |v| - 1$ (since k may represent a left or right offset depending on its sign).*
- *An expression $P = \text{RegPos}(x, rr, k)$ returns c (given that x returns v) iff rr is equal to any pair of regular expressions that matches the boundaries of position c in*

[2] All free variables are universally quantified unless otherwise specified.

the string v. Additionally, given that rr is equal to $\langle r_1, r_2\rangle$, P returns c iff k is equal to a index of c (from the left or right) among all matches of $\langle r_1, r_2\rangle$ in v.

$\mathcal{L}_{\mathsf{FF}}$ makes use of several external operators from the FlashMeta library. Figure 6c shows their witness functions:

- *A $\mathsf{Kth}(x, k)$ expression returns w (given that x returns $\vec{v}$) iff k is an index of some occurrence of w in $\vec{v}$.*
- *If $\mathsf{Pair}(p_1, p_2)$ returns $\langle v_1, v_2\rangle$, then p_j returns v_j.*

Most witness functions are domain-specific w.r.t. the operator that they characterize. However, once formulated in a module for a domain such as substring extraction, they can be reused by any DSL. In our example, witness functions for Pair and Kth operators in Figure 6c do not depend on the domain of their parameters, and are therefore formulated generically, for any DSL. Witness functions in Figure 6a hold only for their respective operators, but they do not depend on the rest of the DSL in which these operators are used, provided the operator semantics is conformant with its (strongly-typed) signature.[3] This property allows us to define witness functions as generally as possible in order to reuse the corresponding operators in any conformant DSL.

Example 9. *Figure 6b shows witness functions for selected domain-specific portions of $\mathcal{L}_{\mathsf{FE}}$. Since most of the $\mathcal{L}_{\mathsf{FE}}$ operators are generic (e.g.* Map, Filter, FilterInt*), their witness functions are domain-independent, and hold for any DSL that conforms with the operator's type signature. These properties are listed in Figure 6c. However, there is no domain-independent witness function ω_L for the $\mathsf{Map}(F, L)$ operator. Hence, it is provided specifically for every strongly-typed* Map *instantiation in a DSL. For instance, the witness function for the aforementioned* LinesMap *operator $\omega_F(\sigma \rightsquigarrow [r_1, r_2, \ldots] \sqsubset [\![\cdot]\!])$ simply maps each output region r_1, r_2, $\ldots$ into its containing line.*

5.3 Meta-algorithm

A set of witness functions for all the parameters of an operator allows us to reduce the inductive synthesis problem $\langle N, \varphi\rangle$ to the synthesis subproblems for its parameters. We introduce a simple non-conditional case first, and then proceed to complete presentation of the meta-algorithm.

Theorem 4. *Let $N := F(N_1, \ldots, N_k)$ be a rule in a DSL $\mathcal{L}$, and φ be an ispec on N. Assume that F has k non-conditional witness functions $\omega_j(\varphi) = \varphi_j$, and $\widetilde{N}_j \models \varphi_j$ for all $j = 1..k$ respectively. Then:*

1. *$\mathsf{Filter}(F_{\bowtie}(\widetilde{N}_1, \ldots, \widetilde{N}_k), \varphi) \models \varphi$.*
2. *If all ω_j are precise, then $F_{\bowtie}(\widetilde{N}_1, \ldots, \widetilde{N}_k) \models \varphi$.*

[3] FlashMeta currently cannot *statically verify* that the strong type requirements hold in a given DSL, therefore a DSL designer has to ensure she reuses an operator in her DSL only if its type signature is satisfied. However, we *manually proved* that all witness functions from our case studies in §6.1 hold, which follows from the basic properties of their underlying domains.

Theorem 4 gives a straightforward recipe for synthesis of operators with independent parameters, such as $\mathsf{Pair}(p_1, p_2)$. However, in most real-life cases operator parameters are dependent on each other. Consider an example of $\mathsf{Concat}(f, e)$ from FlashFill, and an ispec $\sigma \rightsquigarrow s$. It is possible to design individual witness functions ω_f and ω_e that return a disjunction φ_f of prefixes of s and a disjunction φ_e of suffixes of s, respectively. Both of these witness functions individually are precise (i.e. sound and complete); however, there is no straightforward way to combine recursive synthesis results $\widetilde{f} \models \varphi_f$ and $\widetilde{e} \models \varphi_e$ into a valid program set for φ.

In order to enable divide-and-conquer approach for dependent operator parameters, we apply *skolemization* [10]. Instead of deducing specifications φ_f and φ_e that independently entail φ, we deduce only one independent specification (say, φ_f), and then *fix the value of f*. For each fixed value of f a *conditional witness function* $\omega_e(\varphi \mid f = v)$ deduces a specification $\varphi_{e,v}$ that is a necessary and sufficient characterization for φ. Namely, $\varphi_{e,v}$ in our example is $\sigma \rightsquigarrow s[|v|..]$ if v is a prefix of s, or $\bot$ otherwise.

Skolemization splits synthesis into multiple independent branches, one per distinct value of f. These values are efficiently determined by VSA clustering: $\widetilde{f}|_\sigma = \{v_1 \mapsto \widetilde{f_1}, \ldots, v_k \mapsto \widetilde{f_k}\}$. Within each branch, the program sets $\widetilde{f_j}$ and the corresponding $\widetilde{e}_{v_j} \models \varphi_{e,v_j}$ are independent, hence $\mathsf{Concat}_{\bowtie}(\widetilde{f_j}, \widetilde{e}_{v_j}) \models \varphi$ by Theorem 4. The union of k branch results constitutes all Concat programs that satisfy φ.

Definition 13. Let $N := F(N_1, \ldots, N_k)$ be a rule in a DSL $\mathcal{L}$ with k associated (possibly conditional) witness functions $\omega_1, \ldots, \omega_k$. A *dependency graph* of witness functions of F is a directed graph $G(F) = \langle V, E\rangle$ where $V = \{N_1, \ldots, N_k\}$, and $\langle N_i, N_j\rangle \in E$ iff N_i is a prerequisite for N_j.

A dependency graph can be thought of as a union of all possible Bayesian networks over parameters of F. It is not a single Bayesian network because $G(F)$ may contain cycles: it is often possible to independently express N_i in terms of N_j as a witness function $w_i(\varphi \mid N_j = v)$ and N_j in terms of N_i as a different witness function $w_j(\varphi \mid N_i = v)$.

Theorem 5. *Let $N := F(N_1, \ldots, N_k)$ be a rule in a DSL $\mathcal{L}$, and φ be an ispec on N. If there exists an acyclic spanning subgraph of $G(F)$ that includes each node with all its prerequisite edges, then there exists a polynomial procedure that constructs a valid program set $\widetilde{N} \models \varphi$ from the valid parameter program sets $\widetilde{N}_j \models \varphi_j$ for some choice of parameter specifications φ_j.*

Proof. We define the learning procedure for F in Figure 7 algorithmically. It recursively explores the dependency graph $G(F)$ in a topological order, maintaining a *prerequisite path* Q – a set of parameters N_j that have already been skolemized together with their fixed bindings $\vec{v}_j$ and valid program sets $\widetilde{N}_j$. In the prerequisite path, we maintain the invariant: *for each program set $\widetilde{N}_j$ in the path, all programs in it produce*

(a)

$\mathsf{ConstStr}(s):\ \omega_s(\sigma \rightsquigarrow w) = \sigma \rightsquigarrow w$

$\textbf{let}\ x = \mathsf{Kth}(vs, k)\ \textbf{in}\ b:\ \omega_x(\sigma \rightsquigarrow w) = \sigma \rightsquigarrow \bigvee_{w \prec v_j} v_j$

$\mathsf{SubStr}(x, pp):\ \omega_{pp}(\sigma \rightsquigarrow w \mid x = v) = \sigma \rightsquigarrow \bigvee_{w \text{ occurs in } v \text{ at position } l} \langle l, l + |w| \rangle$

$\mathsf{AbsPos}(x, k):\ \omega_k(\sigma \rightsquigarrow c \mid x = v) = \sigma \rightsquigarrow c \vee (c - |v| - 1)$

$\mathsf{RegPos}(x, rr, k):\ \omega_k\ (\sigma \rightsquigarrow c \mid x = v, rr = \langle r_1, r_2 \rangle) = \sigma \rightsquigarrow j \vee j - |\vec{c}| - 1$ where $\vec{c} = \mathsf{matches}(v, r_1, r_2),\ j = \mathsf{indexof}(c, \vec{c})$

$\mathsf{RegPos}(x, rr, k):\ \omega_{rr}(\sigma \rightsquigarrow c)\sigma \rightsquigarrow \bigvee_{\langle r_1, r_2 \rangle \in R(c,v)} \langle r_1, r_2 \rangle$ where $R(c, v) = \{\langle r_1, r_2 \rangle \mid r_1$ matches left of $c \wedge r_2$ matches right of $c\}$

(b)

$\mathsf{StartSeqMap}(F, L):\ \omega_L\ (\sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \ell) = \sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \left[p_1 \mid \langle p_1, p_2 \rangle \in \ell\right]$

$\mathsf{EndSeqMap}(F, L):\ \omega_L\ (\sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \ell) = \sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \left[p_2 \mid \langle p_1, p_2 \rangle \in \ell\right]$

$\mathsf{LinesMap}(F, L):\ \omega_L\ (\sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \ell) = \sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \left[\mathsf{lineof}(s, \sigma[d]) \mid s \in \ell\right]$

$\mathsf{RegexMatches}(d, rr):\ \omega_{rr}\ (\sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \ell \mid d = v) = \sigma \rightsquigarrow \bigvee_{\langle r_1, r_2 \rangle \in \bigcap_{p \in \ell} R(p, v)} \langle r_1, r_2 \rangle$

(c)

$\mathsf{Kth}(x, k):\ \omega_x(\sigma \rightsquigarrow v) = \sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq [v]$

$\mathsf{Kth}(x, k):\ \omega_k(\sigma \rightsquigarrow w \mid x = \vec{v}) = \sigma \rightsquigarrow \bigvee_{v_j = w} j$

$\mathsf{Pair}(p_1, p_2):\ \omega_j(\sigma \rightsquigarrow \langle v_1, v_2 \rangle) = \sigma \rightsquigarrow v_j$

$\mathsf{Map}(F, L):\ \omega_f(\sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \ell \mid L = v) = \sigma \rightsquigarrow \bigwedge_{i=1}^{|\ell|} [\![\cdot]\!](v_i) = \ell_i$

$\lambda x \Rightarrow b:\ \omega_b(\sigma \rightsquigarrow [\![\cdot]\!](v) = y) = \sigma[x := v] \rightsquigarrow y$

$\mathsf{Filter}(p, L):\ \omega_L(\sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \ell) = \sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \ell$

$\mathsf{Filter}(p, L):\ \omega_p(\sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \ell \mid L = v) = \sigma \rightsquigarrow \bigwedge_{i=1}^{|v|} [\![\cdot]\!](\ell_i) = \text{“}v_i \in \ell\text{”}$

$\mathsf{FilterInt}(i_0, k, L):\ \omega_L(\sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \ell) = \sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \ell$

$\mathsf{FilterInt}(i_0, k, L):\ \omega_{i_0}(\sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \ell \mid L = v) = \sigma \rightsquigarrow \mathsf{indexof}(\ell_1, v)$

$\mathsf{FilterInt}(i_0, k, L):\ \omega_k(\sigma \rightsquigarrow [\![\cdot]\!] \sqsupseteq \ell \mid L = v) =$
$\sigma \rightsquigarrow \mathsf{divisors}\big(\mathsf{gcd}\{\mathsf{indexof}(\ell_{i+1}, v) - \mathsf{indexof}(\ell_i, v)\}_{i=1..|\ell|-1}\big)$

Figure 6: Witness functions for **(a)** FlashFill substring extraction DSL operators in Figure 1; **(b)** FlashExtract DSL operators; **(c)** selected generic operators from the pre-defined library of FlashMeta. Here $s \prec w$ denotes that the string s is a substring of w, $\ell_1 \sqsupseteq \ell_2$ denotes that the list ℓ_2 is a prefix of a list ℓ_1, and $\ell_1 \sqsupset \ell_2$ denotes that the list ℓ_2 is a subsequence of a list ℓ_1. $\mathsf{Matches}(w, r_1, r_2)$ denotes the list of positions c in w such that r_1 matches on the left of c, and r_2 matches on the right of c.

the same values $\vec{v}_j$ on the provided input states $\vec{\sigma}$. This allows each conditional witness function ω_i to deduce an ispec φ_i for the current parameter N_i, given the values of the bindings $\vec{v}_{k_1}, \ldots, \vec{v}_{k_s}$ for the prerequisites $N_{k_1}, \ldots, N_{k_s}$ of N_i.

The program sets in each path are valid for the subproblems deduced by applying witness functions. If all the witness functions in $G(F)$ are precise, then any combination of programs $P_1, \ldots, P_k$ from these program sets yields a valid program $F(P_1, \ldots, P_k)$ for φ. If some witness functions are imprecise, then a filtered join of parameter program sets for each path is valid for N. Thus, the procedure in Figure 7 computes a valid program set $\widetilde{N} \models \varphi$. □

Theorems 4 and 5 give a *constructive* definition of the refinement procedure that splits the search space for N into smaller parameter search spaces for $N_1, \ldots, N_k$. If the corresponding witness functions are precise, then *every* combination of valid parameter programs from these subspaces yields a valid program for the original synthesis problem. Alternatively, if some of the accessible witness functions are imprecise, we use them to *narrow down* the parameter search space, and filter the constructed program set for validity. The Filter operation (§4) filters out inconsistent programs from $\widetilde{N}$ in time proportional to $|\widetilde{N}|_{\vec{\sigma}}|$.

Handling Boolean Connectives Witness functions for DSL operators (such as the ones in Figure 6) are typically defined on *atomic* specification constraints (such as equality or subsequence specifications). To complete the definition of the D^4 methodology, Figure 8a gives inference rules for handling of boolean connectives in an ispec φ. Since an ispec is defined as a NNF, we give the rules for handling conjunctions and disjunctions of ispecs, and positive/negative literals. These rules directly map to corresponding VSA operations:

Theorem 6.
1. *$\widetilde{N}_1 \models \varphi_1$ and $\widetilde{N}_2 \models \varphi_2 \iff \widetilde{N}_1 \mathbf{U} \widetilde{N}_2 \models \varphi_1 \vee \varphi_2$.*
2. *$\widetilde{N}_1 \models \varphi_1$ and $\widetilde{N}_2 \models \varphi_2 \iff \widetilde{N}_1 \cap \widetilde{N}_2 \models \varphi_1 \wedge \varphi_2$.*
3. *$\widetilde{N} \models \varphi_1 \iff \mathsf{Filter}(\widetilde{N}, \varphi_2) \models \varphi_1 \wedge \varphi_2$.*

Handling negative literals is more difficult. They can only be efficiently resolved in two cases: (a) if a witness function supports the negated ispec directly, or (b) if the negative literal occurs in a conjunction with a positive literal, in which case we use the latter to generate a base set of candidate programs, which is then filtered to also satisfy the former. If neither (a) nor (b) holds, the set of possible programs satisfying a negative literal is bounded only by the DSL.

Our pre-defined generic witness functions in Figure 6c, together with witness functions for common syntactic features of FlashMeta DSLs (`let` rules, variables, and literals) constitute the FlashMeta standard library.

Search Tactics Theorems 5 and 6 and Fig. 8 entail a non-deterministic choice among numerous possible ways to explore the program space deductively. For instance, one can have many different witness functions for the same operator F in $G(F)$, and they may deduce subproblems of different complexity. A specific exploration choice in the program space constitutes a *search tactic* over a DSL. We have identified several effective generic search tactics, with different advantages and disadvantages; however, a comprehensive study on their selection is left for future work.

Consider a conjunctive problem $\mathsf{Learn}(N, \varphi_1 \wedge \varphi_2)$. One possible way to solve it is given by Theorem 6: handle two conjuncts independently, producing VSAs $\widetilde{N}_1$ and $\widetilde{N}_2$, and intersect them. This approach has a drawback: the complexity of VSA intersection is quadratic. Even if φ_1 and φ_2 are inconsistent (i.e. $\widetilde{N}_1 \cap \widetilde{N}_2 = \emptyset$), each conjunct individually

(a)

$$\frac{N := F_1(N_1, \ldots, N_k) \mid F_2(M_1, \ldots, M_n) \quad \textsc{LearnRule}(G(F_1), \varphi) = \widetilde{N}_1 \quad \textsc{LearnRule}(G(F_2), \varphi) = \widetilde{N}_2}{\mathsf{Learn}(N, \varphi) = \widetilde{N}_1 \mathbf{U} \widetilde{N}_2} \quad \frac{\forall j = 1..2\colon \mathsf{Learn}(N, \varphi_j) = \widetilde{N}_j}{\mathsf{Learn}(N, \varphi_1 \wedge \varphi_2) = \widetilde{N}_1 \cap \widetilde{N}_2} \quad \frac{\forall j = 1..2\colon \mathsf{Learn}(N, \varphi_j) = \widetilde{N}_j}{\mathsf{Learn}(N, \varphi_1 \vee \varphi_2) = \widetilde{N}_1 \mathbf{U} \widetilde{N}_2}$$

$$\frac{\mathsf{Learn}(N, \varphi_1) = \widetilde{N} \quad \varphi_2 = \neg\langle\sigma, \pi\rangle}{\mathsf{Learn}(N, \varphi_1 \wedge \varphi_2) = \mathsf{Filter}(\widetilde{N}, \varphi_2)} \quad \frac{N := F(N_1, \ldots, N_k) \quad \text{All witness functions in } G(F) \text{ accept } \varphi}{\mathsf{Learn}(N, \varphi) = \textsc{LearnRule}(G(F), \varphi)} \quad \frac{}{\mathsf{Learn}(N, \varphi) = \mathsf{Filter}(\mathcal{L}|_N, \varphi)} \quad \frac{}{\mathsf{Learn}(N, \top) = \mathcal{L}|_N}$$

(b)

$$\frac{N := \text{let } x = e_1 \text{ in } e_2}{\omega_{e_2}(\varphi \mid e_1 = v) = \sigma[x := v] \rightsquigarrow \varphi} \quad \frac{N \text{ is a variable}}{\mathsf{Learn}(N, \sigma \rightsquigarrow \pi) = \{N\} \text{ if } \pi(\sigma[N]) \text{ else } \emptyset} \quad \frac{N \text{ is a literal}}{\mathsf{Learn}(N, \sigma \rightsquigarrow v) = \{v\}}$$

Figure 8: **(a)** Constructive inference rules for processing of boolean connectives in the inductive specifications φ; **(b)** Witness functions and inference rules for common syntactic features for FlashMeta DSLs: `let` definitions, variables, and literals.

```
    Input G(F): dependency graph of witness functions for the rule F
    Input φ: specification for the rule F
    function LearnRule(G(F), φ)
1:      Permutation π ← TopologicalSort(G(F))
2:      Ñ ← ⋃{Ñ' | Ñ' ∈ LearnPaths(G(F), φ, π, 1, ∅)}
3:      if all witness functions in G(F) are precise then
4:          return Ñ
5:      else
6:          return Filter(Ñ, φ)

    Input π: permutation of the parameters of F
    Input i: index of a current deduced parameter in π
    Input Q: a mapping of prerequisite values v⃗_k and corresponding
             learnt program sets Ñ_k on the current path
    function LearnPaths(G(F), φ, π, i, Q)
7:      if i > k then
8:          Let Ñ_1, ..., Ñ_k be learnt program sets for N_1, ..., N_k in Q
9:          return {F⋈(Ñ_1, ..., Ñ_k)}
10:     s ← π_i  // Current iteration deduces the rule parameter N_s
11:     Let ω_s(φ | N_{k_1} = v⃗_1, ..., N_{k_m} = v⃗_s) be the w.f. for N_s
    // Extract the prerequisite values for N_s from the mapping Q
12:     {v⃗_{k_1} ↦ Ñ_{k_1}, ..., v⃗_{k_m} ↦ Ñ_{k_m}} ← Q[k_1, ..., k_m]
    // Deduce the ispec for N_s given φ and the prerequisites
13:     φ_s ← ω_s(φ | N_{k_1} = v⃗_{k_1}, ..., N_{k_m} = v⃗_{k_m})
14:     if ω_s = ⊥ then return ∅
    // Recursively learn a valid program set Ñ_s ⊨ φ_s
15:     Ñ_s ← Learn(N_s, φ_s)
    // If no other parameters depend on N_s, proceed without clustering
16:     if N_s is a leaf in G(F) then
17:         Q' ← Q[s := ⊤ ↦ Ñ_s]
18:         return LearnPaths(G(F), φ, π, i + 1, Q')
    // Otherwise cluster Ñ_s on σ⃗ and unite the results across branches
19:     else
20:         σ⃗ ← the input states associated with φ
21:         for all (v⃗'_j ↦ Ñ'_{s,j}) ∈ Ñ_s|_σ⃗ do
22:             Q' ← Q[s := v⃗'_j ↦ Ñ'_{s,j}]
23:             yield return all LearnPaths(G(F), φ, π, i + 1, Q')
```

Figure 7: A learning procedure for the DSL rule $N := F(N_1, \ldots, N_k)$ that uses k conditional witness functions for $N_1, \ldots, N_k$, expressed as a dependency graph $G(F)$.

may be satisfiable. In this case the unsatisfiability of the original problem is determined only after $T(\mathsf{Learn}(N, \varphi_1)) + T(\mathsf{Learn}(N, \varphi_2)) + \mathcal{O}(V(\widetilde{N}_1) \cdot V(\widetilde{N}_2))$ time.

An alternative search tactic for conjunctive ispecs arises when φ_1 and φ_2 constrain *different* input states σ_1 and σ_2, respectively. In this case each conjunct represents an independent "world", and witness functions can deduce subproblems in each "world" independently and concurrently. FlashMeta applies witness functions to each conjunct in the specification in parallel, conjuncts the resulting parameter ispecs, and makes a single recursive learning call. Such "parallel processing" of conjuncts in the ispec continues up to the terminal level, where the deduced sets of concrete values for each terminal are intersected across all input states.[4]

The main benefit of this approach is that unsatisfiable branches are eliminated much sooner. For instance, if among m I/O examples one example is inconsistent with the rest, a parallel approach approach discovers it as soon as the relevant DSL level is reached, whereas an intersection-based approach has to first construct m VSAs (of which one is empty) and intersect them. Its main disadvantage is that in presence of disjunction the number of branches grows exponentially in a number of input states in the specification.

Optimizations FlashMeta performs many practical optimizations in the algorithm in Figure 7. We parallelize the loop in Line 21, since it explores non-intersecting portions of the program space. For ranked inductive synthesis, we only calculate top k programs for leaf nodes of $G(F)$, provided the ranking function is monotonic. We also cache synthesis results for every distinct learning subproblem $\langle N, \varphi\rangle$, which makes D^4 an instance of *dynamic programming*. This optimization is crucial for efficient synthesis of many common DSL operators, as we explain in more details in §6.1.

For bottom portions of the DSL (when $\mathcal{L}|_N$ is small) we switch to enumerative search [37], which in such conditions is more efficient than deduction, provided no constants need to be synthesized. In principle, every subproblem $\mathsf{Learn}(N, \varphi)$ in FlashMeta can be solved by any sound strategy, not necessarily D^4 or enumerative search. Possible alternatives include constraint solving or stochastic techniques [1].

6. Evaluation

Our evaluation of FlashMeta aims to answer two classes of questions: its *applicability* (§6.1) and its *performance* (§6.2).

[4] The "parallel" approach can also be thought of as a deduction over a new isomorphic DSL, in which operators (and witness functions) are straightforwardly lifted to accept *tuples* of values instead of single values.

Table 1: Case studies of FlashMeta: prior works in inductive program synthesis. "D^4" means "Is it an instance of the D^4 methodology?", "Imp" means "Has it been (re)implemented?", π is a top-level specification kind, φ' lists notable intermediate specification kinds (for D^4 only).

Project	Domain	D^4	Imp	π	φ'
Gulwani [7]	String transformation	✓	✓	=	=
Le and Gulwani [22]	Text extraction	✓	✓	$\sqsupset$	$\sqsupset$
Kini and Gulwani [16]	Text normalization	✓	✓	=	soft
Barowy et al. [3]	Table normalization	✓	✓	=	=
Singh and Gulwani [33]	Semantic text editing	✓	✗	=	=
Harris and Gulwani [8]	Table transformation	✓	✗	=	=
Singh and Gulwani [32]	Number transformation	✓	✗	=	=
Andersen et al. [2]	Algebra education	✓	✗	trace	=
Lau et al. [21]	Editor scripting	✓	✗	trace	=
Feser et al. [5]	ADT transformation	✓	✗	=	=
Osera and Zdancewic [28]	ADT transformation	✓	✗	=	=
Yessenov et al. [38]	Editor scripting	✓	✗	=	=
Udupa et al. [37]	Concurrent protocols	✗	✗	trace	N/A
Katayama [15]	Haskell programs	✗	✗	=	N/A
Lu and Bodik [24]	Relational queries	✗	✗	=	N/A
WebExtract	Web data extraction	✓	✓	$\sqsupset$	$\sqsupset$

Table 2: Development data on (re)implemented projects.

Project	LOC		Development time	
	Original	FlashMeta	Original	FlashMeta
Gulwani [7]	12K	3K	9 months	1 month
Le and Gulwani [22]	7K	4K	8 months	1 month
Kini and Gulwani [16]	17K	2K	7 months	2 months
Barowy et al. [3]	5K	2K	8 months	1 month
WebExtract	—	2.5K	—	1.5 months

6.1 Case Studies

Table 1 summarizes our case studies: the prior works in inductive synthesis over numerous different applications that we studied for evaluation of FlashMeta. Of the 15 inductive synthesis tools we studied, 12 can be cast as a special case of D^4 methodology, which we verified by manually formulating corresponding inductive properties for their algorithms. In the other 3 tools, the application domain is inductive synthesis, and our problem definition covers their application, but the original technique is not an instance of D^4: namely, enumerative search [15, 37] or constraint solving [24].

Our industrial collaborators reimplemented 4 existing systems and created a new system WebExtract for synthesis of CSS selectors for extraction of webpage data by example. We present data on these development efforts in Table 2.

Q1: How motivated is our generalization of inductive specification? Input-output examples is the most popular specification kind, observed in 12/15 projects. However, 3 projects require *program traces* as their top-level specification, and 2 projects (1 prior) require *subsequences of program output*. Boolean connectives such as $\vee$ and $\neg$ are omnipresent in subproblems across all 12 D^4 projects.

FlashNormalize [16] is a noteworthy case. Its witness functions introduce *soft specifications*. Such specifications inspire an extension of Problem 1, where it suffices to satisfy a *maximal* set of conjuncts in a CNF specification. In [16], such soft specifications are used to learn conditionals $\mathsf{ITE}(B, T, E)$: first a collection of program sets that cover different maximals of examples is learned as candidates for T, and then one of these maximals is selected as a soft specification for B. After a subset of examples is covered by the learned $\widetilde{B}$ and $\widetilde{T}$, the remaining examples are passed down recursively to E (as a regular, "hard" specification). We are aware of at least two more applications of soft specifications: (a) composition of Filter operators, where each program may satisfy only a subset of negative examples (assuming other Filters handle the rest), and (b) composition of Merge operators (for similar reasons). Complete exploration of soft specifications is left for future work.

Q2: How applicable is our generic operator library? Most common operators across our case studies are string processing functions, due to the most popular domain being data manipulation (11/16 projects). Almost all projects include some version of learning conditional operators (equivalent to that of FlashFill). List processing operators (e.g. Map, Filter) appear in 9/16 projects, often without explicit realization by the original authors (for example, the awkwardly defined Loop operator in FlashFill is actually a combination of Concatenate and Map). Feser et al. [5] define an extensive library of synthesis strategies for list-processing operators in the λ^2 project. These synthesis strategies are isomorphic to FlashExtract witness functions; both approaches can be cast as instances of D^4 (see §8 for detailed comparison).

Q3: How usable is FlashMeta? Table 2 presents some development stats on the projects that were reimplemented. In all cases, FlashMeta was shorter, cleaner, more stable and extensible. The reason is that with FlashMeta, our collaborators did not concern themselves with tricky details of synthesis algorithms, since they were implemented once and for all, as in §5.3. Instead, they focused only on domain-specific witness functions, for which design, implementation, and maintenance are much easier. Notably, in case of FlashRelate [3] reimplementation and WebExtract, our collaborators did not have any experience in program synthesis.

The development time in Table 2 includes the time required for an implementation to mature (i.e. cover the required use cases), which required multiple experiments with DSLs. With FlashMeta, various improvements over DSLs were possible on a daily basis. FlashMeta also allowed our collaborators to discover optimizations not present in the original implementations. We share some anecdotes of FlashMeta simplifying synthesizer development below.

Scenario 1. One of the main algorithmic insights of FlashFill is synthesis of $\mathsf{Concat}(e_1, \ldots, e_k)$ expressions using *DAG program sharing*. A DAG over the positions in the output string s is maintained, each edge $s[i : j]$ annotated with a set of programs that output this substring on a given

state σ. Most of the formalism in the paper and code in their implementation is spent on describing and performing operations on such a DAG. In FlashMeta, the same grammar symbol is instead defined through a recursive binary operator: $f := e \mid \mathsf{Concat}(e, f)$. The witness function for e in Concat constructs φ' as a disjunction of all prefixes of the output string in φ. The property for f is conditional on e and simply selects the suffix of the output string after the given prefix $[\![e]\!]\sigma$. Since FlashMeta caches the results of learning calls $\langle f, \varphi\rangle$ for same φs, the tree of recursive $\mathsf{Learn}(f, \varphi)$ calls becomes a DAG. This is *the same DAG* as in FlashFill – but in FlashMeta, it arises implicitly and at no cost. Moreover, it becomes obvious now that DAG sharing happens for any foldable operator, e.g. ITE, $\wedge$, $\vee$, sequential statements.

Scenario 2. During reimplementation of FlashFill, a new operator was added to its substring extraction logic: *relative positioning*, which defines the right boundary of a substring depending on the value of its left boundary. For example, it enables extracting substrings as in "ten characters after the first digit". This extension simply involved adding three `let` rules in the DSL, which (a) define the left boundary position using existing operators; (b) cut the suffix starting from that position; (c) define the right boundary in the suffix. This extension was inspired by some practically useful tasks that FlashFill fails to handle. While such an extension in the original FlashFill implementation would consume a couple of weeks, in FlashMeta it took only a few minutes.

6.2 Experiments

Domain-specific Tools To investigate performance of synthesizers generated by FlashMeta, we chose FlashExtract as a representative implementation, since its test cases cover huge instances of real-life data. We used the entire original FlashExtract benchmark set of 139 tests. Figure 9 shows performance & maximum VSA volume of the new system.

The overall performance is comparable to that of the original system, even though the implementations differ drastically. The original implementation's runtime varies from 0.1 to 4 sec, with a median of 0.3 sec. The new implementation (despite being more expressive and built on a general-purpose framework) has a runtime of $0.5-3$x the original implementation, with a median of 0.6 sec. This performance is sufficient for the FlashMeta-based implementation to be successfully used in industry instead of the original one.

There is no good theoretical bound on the time of VSA clustering (the most time-consuming operation during D^4 deduction). However, it is evident that the output VSA volume is proportional to the clustering time. Thus, to evaluate it, we measured the VSA volume on our real-life benchmark suite. As Figure 9 shows, even for large inputs it never exceeds 8000 nodes, thus explaining efficient runtime.

Generic Tools Comparing FlashMeta to general-purpose meta-synthesizers proved challenging. Most related projects are participants of the SyGuS-COMP 2014 competition [1]. However, their synthesis procedures are currently defined only for DSLs based on SMT theories. In contrast, our case studies explore domains with rich semantics, such as regular expressions or webpage DOM. Even most recent extensions of the Z3 SMT solver to strings [40] lack expressiveness to model such DSLs. We believe that lowering the FlashExtract DSL, for instance, to theory of arrays would deem an unfair comparison because of huge size of the resulting encoding. Therefore we leave comparison of complementary meta-synthesis approaches to future work. To breach this comparison gap, we intend to organize an inductive synthesis competition on a rich industrial benchmark suite.

7. Discussion

7.1 Strengths and Limitations

D^4 methodology works best under the following conditions:

Decidability A majority of the DSL should be characterized by witness functions, capturing a subset of inverse semantics of the DSL operators.

Counterexample An example of an operator that cannot be characterized by any witness function is an integral multivariate polynomial $\mathsf{Poly}(a_0, \ldots, a_k, X_1, \ldots, X_n)$. Here $a_0, \ldots, a_k$ are integer polynomial coefficients, which are input variables in the DSL, and $X_1, \ldots, X_n$ are integer nonterminals in the DSL. Given a specification $\varphi = (a_0, \ldots, a_k) \rightsquigarrow y$ stating that a specific Poly executed with coefficients $a_0, \ldots, a_k$ evaluated to y on *some* $X_1, \ldots, X_n$, a witness function ω_j has to find a set of possible values for X_j. This requires finding roots of a multivariate integral polynomial, which is undecidable.

Deduction Witness functions should not introduce many disjunctions. Each disjunct (assuming it can be materialized by at least one program) starts a new deduction branch. In certain domains this problem can only be efficiently solved with a corresponding SMT solver.

Counterexample Consider the bitwise operator $\mathsf{BitOr}\colon (\mathsf{Bit32}, \mathsf{Bit32}) \to \mathsf{Bit32}$. Given a specification $\sigma \rightsquigarrow b$ where $b\colon \mathsf{Bit32}$, witness functions for BitOr have to construct each possible pair of bitvectors $\langle b_1, b_2\rangle$ such that $\mathsf{BitOr}(b_1, b_2) = b$. If $b = 2^{32} - 1$, there exist 3^{32} such pairs. A deduction over 3^{32} branches is infeasible.

Performance Witness functions should be efficient, preferably polynomial in low degrees over the specification size.

Counterexample Consider the multiplication operator $\mathsf{Mul}\colon (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int}$. Given a specification $\sigma \rightsquigarrow n$ with a multiplication result, a witness function for Mul has to factor n. This problem is decidable, and the number of possible results is at most $\mathcal{O}(\log n)$, but the factoring itself is infeasible for large n.

All counterexamples above feature real-life operators, which commonly arise in embedded systems, control theory,

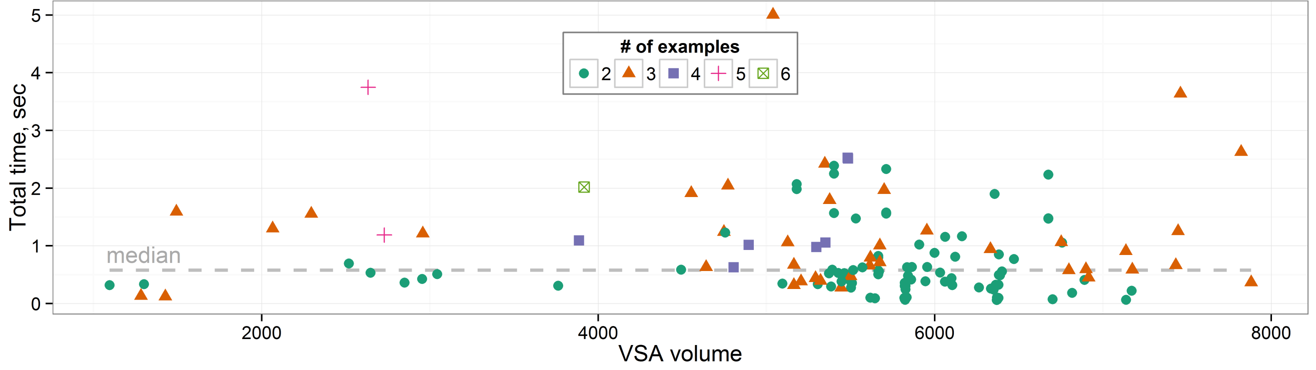

Figure 9: Performance and maximum VSA volume during 139 learning sessions of the FlashExtract reimplementation.

and other domains. The best known synthesis strategies for them are based on specialized SMT solvers [1]. On the other hand, to our knowledge D^4 is the *only* synthesis strategy when the following (also real-life) conditions hold:

- The DSL contains arbitrary executable operators that manipulate domain-specific objects with rich semantics.
- The programs may contain domain-specific constants.
- The specifications are inherently ambiguous, and resolving user's intent requires learning a set of valid programs to enable ranking or additional user interaction.
- The engineering & maintenance cost of a PBE-based tool is limited by industrial budget and available developers.

7.2 Remarks

VSA as a Language In formal logic, a *language* is defined as a set of programs. We commonly represent languages as CFGs, which are based on two syntactic features: *union of rules* (|) and *sharing of nonterminals*. A VSA (§4) is also a representation for a set of programs; it is an AST-based representation that leverages two syntactic features: *union* ($\bigcup$) and *join of shared VSAs* ($F_{\bowtie}$). These representations are isomorphic; in fact, a VSA over a DSL $\mathcal{L}$ is essentially a CFG for some DSL $\mathcal{L}'$ built from $\mathcal{L}$. For instance, if $N = \mathtt{output}(\mathcal{L})$, then the VSA $\widetilde{N} = \mathcal{L}|_N$ is isomorphic to the CFG of $\mathcal{L}$. VSAs produced by D^4 are *granulations* of the CFG of $\mathcal{L}$ (i.e. subsets of $\mathcal{L}$ with some nonterminals split into multiple independent symbols).

8. Related Work

The dream of program synthesis in a *general-purpose language* remains far from reality; the underlying state space is just too huge to synthesize any useful programs. Creative solutions have thus emerged including three main lines of work: *deductive*, *syntax-guided*, and *domain-specific inductive* synthesis. In §1, we outlined the main techniques, strengths, and weaknesses of all three approaches. Our D^4 methodology combines the best features of all approaches: it takes as input an inductive specification, its program space is restricted by a DSL, and its algorithm is a combination of deductive inference with enumerative search. Such a combination makes inductive synthesizers scale to real-world practical domains.

Recently there has been considerable effort in standardizing the program synthesis space and developing generic synthesis strategies and frameworks. Three main initiatives in this space are SKETCH [35], ROSETTE [36], and SyGuS [1].

SKETCH *and* ROSETTE SKETCH, developed by Solar-Lezama [35], is a pioneering work in the space of program synthesis frameworks. It takes as input a partial program, i.e. a program template with holes in place of desired subexpressions. SKETCH translates this template to SAT encoding, and applies counterexample-guided inductive synthesis to fill in the holes with expressions that satisfy the specification.

ROSETTE by Torlak and Bodik [36] is a DSL-parameterized framework, which supports a rich suite of capabilities including verification, synthesis, and repair. Unlike SKETCH, its input language is a limited subset of Racket programming language, which ROSETTE translates to SMT constraints via symbolic execution.

Both SKETCH and ROSETTE allow seamless translation of their input languages (custom C-like in SKETCH or limited Racket in ROSETTE) to SAT/SMT encoding at runtime. It reduces the level of synthesis awareness required from the developer (Torlak and Bodik call such a methodology *solver-aided programming*). However, our experiments show that constraint-based techniques scale poorly to real-world industrial application domains, which do not have a direct SMT theory [2, 32]. To enable program synthesis in such cases, our D^4 methodology separates domain-specific insights into *witness functions*, and uses a common deductive meta-algorithm in all application domains. The resulting workflow is as transparent to the developer as solver-aided programming, but it does not require domain axiomatization.

Syntax-guided Synthesis SyGuS [1] is a recent initiative that aims to (a) standardize the input specification language of program synthesis problems; (b) develop general-purpose synthesis algorithms for these problems, and (c) establish an annual competition of such algorithms on a standardized benchmark suite. Currently, their input language is also based on SMT theories, which makes it inapplicable for complex

industrial domains (see §1). The main synthesis algorithms in SyGuS are enumerative search [37], constraint-based search [13], and stochastic search [31]. Based on the SyGuS-COMP results, they handle different application domains best, although enumerative search is generally effective on medium-sized problems in most settings. We have integrated enumerative search in our D^4 methodology, which makes D^4 the first domain-specific deduction&search-based technique in program synthesis. We also plan to organize an inductive synthesis competition on a suite of industrial benchmarks.

Deduction for ADT Transformations Concurrently with this work, Feser et al. [5] and Osera and Zdancewic [28] developed two approaches to ADT transformations from I/O examples based on deductive reasoning over the program structure. These techniques operate over languages with ADT-processing operators and limited forms of recursion. Both works include deductive synthesis strategies for supported operators, such as Map and Filter. However, they are limited to special cases presented in the respective works:

- Osera and Zdancewic cannot proceed with synthesis if a deduced recursive subproblem is absent in the list of I/O examples (so called "trace completeness property"). For instance, recursive synthesis for the function $\mathsf{Length}\colon \mathsf{List}\langle T\rangle \to \mathsf{Int}$ requires an I/O example per each suffix of the longest input list in the specification.
- Feser et al. cannot derive a general-purpose deduction scheme for $\mathsf{Map}(F, L)$ since they do not leverage domain-specific knowledge about the behavior of L (recall from §5 that witness functions for L exist only for strongly-typed Maps). In fact, their deduction rules are formulated only for Maps whose input is also an input to the overall program. As a result, they are unable to learn programs like $\lambda x \Rightarrow \mathsf{Map}(F, \mathsf{Filter}(B, x))$. Deduction in such cases requires understanding of the domain-specific nature of L, which D^4 captures via strong types and witness functions.

In this work, we present the first comprehensive theory of deduction&search-based inductive synthesis for arbitrary application domains. It presents a practically useful problem definition, in which the synthesis specification describes *properties* of the output on some concrete inputs (as opposed to only output *values*). I/O examples are too limited in many real-life settings, whereas complete logical specifications are difficult to satisfy efficiently. Our intent specification is a trade-off between these two extremes.

Furthermore, we parameterize FlashMeta by a ranking function h, which is a quantitative constraint over the program space. By using VSAs, D^4 supports learning k topmost-rated programs in the DSL w.r.t. h. This enables learning from a small number of examples and rich user interaction models for disambiguation [26].

9. Conclusion and Future Work

The notoriously complex problem of program synthesis has usually been approached in two ways, which fall on two sides of the abstraction spectrum. On one hand, syntax-guided meta-synthesizers restrict the underlying program space to DSLs, but they struggle with semantics of rich practical domains and their algorithms are based on expensive search over DSL. On another hand, domain-specific inductive synthesizers allow incomplete example-based specifications over arbitrary domains, but are incredibly difficult to develop. Our D^4 methodology bridges the gap between these approaches by allowing the DSL designer to separate their domain-specific insights into witness functions of DSL operators, and exploit them in our common deductive meta-synthesis algorithm. We implemented it in the FlashMeta framework, whose effectiveness is evident from its immediate adoption for developing mass-market applications of industrial quality.

The inductive synthesis problem opens a new subfield in the program synthesis domain. To compare many possible solutions to it, we intend to release our framework and organize an inductive synthesis competition on a diverse real-life benchmark set. We foresee it facilitating novel cross-disciplinary research on program synthesis, such as: (a) UIs & environments for debugging/guiding inductive synthesis sessions; (b) Integrating probabilistic techniques with logical reasoning for handling noisy input data in specifications and building a statistical program ranking framework.

Acknowledgments

We thank Microsoft Corporation for sponsoring this work and supporting the first author's 18-months position at Microsoft Research Redmond. We also thank Alex Clemmer, Maxim Grechkin, Dileep Kini, Vu Le, Mikaël Mayer, Mohammad Raza, Rishabh Singh, Gustavo Soares, and Varadarajan Thiruvillamalai for their invaluable help in discussing the FlashMeta initiative and for (re)implementation of multiple PBE-based applications on top of FlashMeta.

References

[1] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17. IEEE, 2013.

[2] E. Andersen, S. Gulwani, and Z. Popović. Programming by demonstration framework applied to procedural math problems. Technical Report MSR-TR-2014-61, Microsoft Research, 2014.

[3] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. FlashRelate: Extracting relational data from semi-structured spreadsheets using examples. In *PLDI*, 2015.

[4] T. Celik, E. J. Etemad, D. Glazman, I. Hickson, P. Linss, and J. Williams. Selectors level 3. W3C recommendation. *World Wide Web Consortium*, page 76, 2011.

[5] J. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.

[6] S. Gulwani. Dimensions in program synthesis. In *PPDP*, pages 13–24. ACM, 2010.

[7] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[8] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

[9] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7): 321–322, July 1961. ISSN 0001-0782. .

[10] W. Hodges. *A shorter model theory*. Cambridge university press, 1997.

[11] J. E. Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979.

[12] T. Hottelier and R. Bodik. Synthesis of layout engines from relational constraints. Technical Report UCB/EECS-2014-181, University of California at Berkeley, 2014.

[13] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, volume 1, pages 215–224. IEEE, 2010.

[14] S. C. Johnson. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[15] S. Katayama. Systematic search for lambda expressions. *Trends in functional programming*, 6:111–126, 2005.

[16] D. Kini and S. Gulwani. FlashNormalize: Programming by examples for text normalization. IJCAI, 2015.

[17] E. Kitzelmann. A combined analytical and search-based approach for the inductive synthesis of functional programs. *KI-Künstliche Intelligenz*, 25(2):179–182, 2011.

[18] A. S. Koksal, Y. Pu, S. Srivastava, R. Bodik, J. Fisher, and N. Piterman. Synthesis of biological models from mutation experiments. In *POPL*, 2013.

[19] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Software synthesis procedures. *CACM*, 55(2):103–111, 2012.

[20] Y. Kuniyoshi, M. Inaba, and H. Inoue. Learning by watching: Extracting reusable task knowledge from visual observation of human performance. *IEEE Transactions on Robotics and Automation*, 10(6):799–822, 1994.

[21] T. A. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534, 2000.

[22] V. Le and S. Gulwani. FlashExtract: A framework for data extraction by examples. In *PLDI*, page 55. ACM, 2014.

[23] H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[24] E. Lu and R. Bodik. Quicksilver: Automatic synthesis of relational queries. Master's thesis, EECS Department, University of California, Berkeley, May 2013.

[25] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *TOPLAS*, 2(1):90–121, 1980.

[26] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. Zorn, and S. Gulwani. User interaction models for disambiguation in programming by example. UIST, 2015.

[27] T. M. Mitchell. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.

[28] P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

[29] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. PLDI, 2015.

[30] M. Püschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačic, Y. Voronenko, et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[31] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 305–316. ACM, 2013.

[32] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification*, pages 634–651. Springer, 2012.

[33] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *VLDB*, 5(8):740–751, 2012.

[34] R. Singh and S. Gulwani. Predicting a correct program in programming by example. CAV, 2015.

[35] A. Solar-Lezama. *Program synthesis by sketching*. ProQuest, 2008.

[36] E. Torlak and R. Bodik. Growing solver-aided languages with Rosette. In *Onward!*, pages 135–152. ACM, 2013.

[37] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. Transit: specifying protocols with concolic snippets. In *PLDI*, pages 287–296. ACM, 2013.

[38] K. Yessenov, S. Tulsiani, A. Menon, R. C. Miller, S. Gulwani, B. Lampson, and A. Kalai. A colorful approach to text processing by example. In *UIST*, pages 495–504. ACM, 2013.

[39] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, pages 279–285. IEEE Press, 2001.

[40] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *FSE*, pages 114–124. ACM, 2013.

A. Proof of Theorem 2

Theorem 2. *Let $\widetilde{N}$ be a VSA, and let $m = W(\widetilde{N})$. Assume $\mathcal{O}(m \log k)$ implementation of the* Select *function. The time complexity of calculating* $\mathsf{Top}(\widetilde{N}, k)$ *is* $\mathcal{O}(V(\widetilde{N})\, k^m \log k)$.

Proof. Let d be the depth (number of levels) of $\widetilde{N}$. Note that $V(\widetilde{N}) = \mathcal{O}(m^d)$. Let $T(n)$ denote the time complexity of calculating $\mathsf{Top}_h(\widetilde{N}_{(n)}, k)$, where $\widetilde{N}_{(n)}$ is the nth level of $\widetilde{N}$. For a leaf node we have $T(1) = \mathcal{O}(m \log k)$. For a union node we have $T(n) = \mathcal{O}(m \cdot T(n-1) + k\, m \log k)$, where the first term represents calculation of Top_h over the children, and the second term represents the selection algorithm. For a join node we similarly have $T(n) = \mathcal{O}(m \cdot T(n-1) + k^m \log k)$. Since $T(n)$ grows faster if $\widetilde{N}_{(n)}$ is a join rather than a union, we can ignore the non-dominant union case. Solving the recurrence for $T(n)$, we get:

$$\begin{aligned} T(d) &= \mathcal{O}\left(m^d \log k + (m-1)^{-1} k^m (m^{d-1} - 1) \log k\right) \\ &= \mathcal{O}(m^d k^m \log k) = \mathcal{O}(V(\widetilde{N}) \cdot k^m \log k) \end{aligned}$$

□