A Framework for Mass-Market Inductive Program Synthesis

Oleksandr Polozov

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2017

Reading Committee:

Sumit Gulwani, Chair

Rastislav Bodik

Emina Torlak

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

A Framework for Mass-Market Inductive Program Synthesis

Oleksandr Polozov

Co-Chairs of the Supervisory Committee:
Affiliate Professor Sumit Gulwani
Paul G. Allen School of Computer Science & Engineering

Professor Zoran Popović
Paul G. Allen School of Computer Science & Engineering

Programming by examples (PBE), or inductive program synthesis, is a problem of finding a program in the underlying domain-specific language (DSL) that is consistent with the given input-output examples or constraints. In the last decade, it has gained a lot of prominence thanks to the mass-market deployments of several PBE-based technologies for data wrangling – the widespread problem of transforming raw datasets into a structured form, more amenable to analysis. However, deployment of a mass-market application of program synthesis is challenging. First, an efficient implementation requires non-trivial engineering insight, often overlooked in a research prototype. This insight takes the form of domain-specific knowledge and heuristics, which complicate the implementation, extensibility, and maintenance of the underlying synthesis algorithm. Second, application development should be fast and agile, tailoring to versatile market requirements. Third, the underlying synthesis algorithm should be accessible to the engineers responsible for product maintenance.

In this work, I show how to generalize the ideas of 10+ previous specialized inductive synthesizers into a single framework, which facilitates automatic generation of a domain-specific synthesizer from the mere definition of the corresponding DSL and its properties. PROSE (PROgram Synthesis using Examples) is the first program synthesis framework that

explicitly separates domain-agnostic search algorithms from domain-specific expert insight, making the resulting synthesizer both fast and accessible. The underlying synthesis algorithm pioneers the use of deductive reasoning for designer-defined domain-specific operators and languages, which enables mean synthesis times of 1-3 sec on real-life datasets.

A dedicated team at Microsoft has built and deployed 10+ technologies on top of the PROSE framework. Using them as case studies, I examine the user interaction challenges that arise after a mass-market deployment of a PBE-powered application. I show how expressing program synthesis as an interactive problem facilitates user intent disambiguation, incremental learning from additional examples, and increases the users' confidence in the system.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am incredibly grateful to Sumit Gulwani, who first introduced me to the world of research, mentored and advised me, believed in my ideas, supported me, and filled our collaboration with indescribable energy, support, and drive. Sumit taught me to value simple problem solutions that leave an impact on millions of people, to persevere in the face of failure or rejection, and to recognize good research skills. Whether in work or in personal life, I knew I could always rely on Sumit's kind words and advice, and for that I remain eternally grateful and indebted.

I have also been blessed to be advised by Zoran Popović. His vision and multi-disciplinary ambitions have always pushed me beyond my comfort zone, helped to keep a bigger picture in mind, and motivated to explore further. Working with Zoran has been a pleasure, and I will always remember his support, patience, and advice.

I would like to thank all my co-authors, collaborators, colleagues, and friends who have helped me with these and other projects over the course of my graduate education: Erik Andersen, Rastislav Bodik, Marc Brockschmidt, Eric Butler, Sarah Chasins, Loris D'Antoni, Kevin Ellis, Michael Ernst, Elena Glassman, Maxim Grechkin, Björn Hartmann, Pushmeet Kohli, Ranvijay Kumar, Vu Le, Mark Marron, Mikaël Mayer, Saswat Padhi, Daniel Perelman, Mark Plesko, Mohammad Raza, Eleanor O'Rourke, Danny Simmons, Rishabh Singh, Adam M. Smith, Gustavo A. Soares, Emina Torlak, Abhishek Udupa, Luke Zettlemoyer, and Ben Zorn.

Last but not least, I want to thank my parents and Sasha for their unending love and support, for words of pride and encouragement, and for tolerating my stressful schedule and mood swings. Without you, none of this would ever happen.

Chapter 1

INTRODUCTION

Program synthesis is the task of finding a program in the underlying language that satisfies a given user intent, expressed in the form of some specification [13]. The origin of the idea of program synthesis dates back to the early days of AI [12, 58], with the original aim of automatizing the programming experience, given a formal specification of the desired software. In addition, has applications to a wide variety of domains, including robotics [30], software development [29], and biology [28]. However, the true potential of this area still remains elusive due to its inherent complexity: because of the huge and arbitrary nature of the underlying state space of possible programs, program synthesis is an incredibly hard combinatorial search problem.

In the last decade, synthesis-based technologies have resurged in popularity, thanks to the recent advancements in efficient search algorithms, constraint solving, and stochastic techniques like deep learning. The key insight to scaling program synthesis is *syntactic guidance*, first formalized universally as a *syntax-guided synthesis* (SyGuS) problem [1]. Instead of a general-purpose programming language, SyGuS-based techniques perform their search in a *domain-specific language* (DSL). A DSL is a syntactic restriction on the search space, which narrows it down to only those programs that are practical in a certain application domain. The SyGuS initiative consists in development of general-purpose synthesis algorithms that take as input a DSL and a formal specification φ, and find a program in the DSL that satisfies φ. However, generality of such algorithms implies that (a) their specification format has to be limited to a common set of formally defined *theories* (currently based on the SMT-LIB format [5]), and (b) they cannot leverage any domain-specific algorithmic

insights for improving synthesis performance on a given DSL. As a result, at the moment performance of all SyGuS algorithms is inapplicable in real-life industrial settings even for simplest tasks [2, 20, 52].

This state of affairs in mass-market industrial deployment of program synthesis has changed in 2011 thanks to FlashFill [14], a technology for automatic synthesis of string transformations, available in Microsoft Excel 2013. FlashFill was widely successful and appraised, thanks to a combination of important features:

- It aids with the widespread problem of *data wrangling* — transforming, cleaning, and reshaping a dataset from a semi-structured form into a structured one, amenable to automated processing [24].
- It is driven by input-output examples of the desired transformation's behavior, which makes it easily accessible to end users.
- It is able to synthesize the desired transformation from as few as 1-3 examples of intent, and in a fraction of a second.

The success of FlashFill has prompted a series of successor technologies based on the same methodology – *programming by examples* (PBE). These include FlashExtract [33] – a technology for automatic extraction of data from semi-structured log files and webpages, FlashRelate [4] – a technology for automatic reshaping of spreadsheets into a relational format, and others. Many of these technologies were also deployed in mass-market products, including Microsoft Windows 10, Cortana, Exchange, PowerShell, and Azure Log Analytics.

Unfortunately, the engineering effort associated with implementation and deployment of a domain-specific PBE-based technology is enormous. Domain-specific synthesis applications implement program synthesis algorithms that are designed specifically for a given DSL, and are tightly dependent on its structure. This approach drastically improves synthesis performance, but also limits industrial applicability of the resulting codebase. Specifically, the underlying DSL is hard to extend because even small DSL changes might require non-trivial changes to the synthesis algorithm and implementation. Both the original implementation of a PBE-based

synthesizer and the subsequent software evolution may take up to 1-2 man-years of effort and require PhD-level expertise in program synthesis.

In this work, I show that most domain-specific PBE-based synthesis applications produced in the last decade [2, 4, 9, 14, 16, 26, 33, 40, 52, 53, 60, 61] can be cast as instance of one common methodology. We decomposed all of them into:

1. A set of domain-specific insights that describe properties of the application domain (i.e. the DSL).
2. A general-purpose meta-algorithm that is parameterized with a DSL and the aforementioned insights.

The algorithm can be written independently once and for all domains. The insights are domain-specific, but independent of the chosen synthesis strategies, thus can be defined by domain experts and software engineers.

We have implemented this common algorithm as the PROSE framework ("PROgram Synthesis using Examples") [43].[1] Over the last three years, PBE technologies developed on top of PROSE have been deployed in 10+ Microsoft applications and external projects. In this work, I am using them as case studies, exploring the consequences of making example-driven program synthesis an industrial commodity. As it turns out, deploying PBE-based technologies in the mass markets leads to a new set of HCI and AI challenges, which do not arise on a smaller scale. These include:

- Disambiguating user intent from 1-3 input-output examples.
- Incorporating iterative refinements of the same intent in a single synthesis session.
- Intelligently and proactively leading the user toward better examples.
- Transparently communicating the synthesizer's current understanding of the task, as well as all the information it has used to reach this understanding.
- Efficiently analyzing, storing, and pruning sets of up to 10^{30} candidate programs.

[1] Available at https://microsoft.github.io/prose.

- Ensuring that the synthesizer's implementation *and* the implementation of its produced artifacts (learned programs) is efficient and maintainable.
- Adapting and improving all the aforementioned capabilities over time, learning on the completed tasks.

I study these and related questions in the second half of this work, presenting our solutions to the challenges of mass-market industrial deployment of program synthesis.

Thesis Statement: *Interactive example-driven program synthesis for any data-centric domain can be expressed as a common domain-agnostic top-down search algorithm parameterized by concise domain-specific deduction procedures. Such decomposition makes program synthesizers scale to real-life application domains, enables domain-agnostic intent disambiguation techniques, incremental synthesis sessions, and mass-market deployment of the derived technologies.*

Outline Chapter 3 begins with the relevant background for presenting the rest of the work. I introduce the prior work in domain-specific inductive program synthesis (FlashFill, FlashExtract, and their successors) in more detail, and use them as running examples thereafter throughout the work. I also introduce formally the notions of a domain-specific language, inductive specification, and several variations of the domain-specific program synthesis problem, motivated and tackled in this work.

Chapter 4 presents version space algebras, an important data structure for storing, pruning, and reasoning about huge program spaces that arise in the process of solving a program synthesis problem. It has been first used in the context of program synthesis by Lau et al. [32]; in this work, I present the first comprehensive formalism that explores all the aspects of VSAs in the context of PBE, as well as various extensions to them that make VSAs suitable for ranked and incremental flavors of the synthesis problem.

Chapter 5 introduces the PROSE framework. It presents and formalizes the underlying algorithmic concepts that makes PROSE efficiently scale to real-life application domains: domain-specific witness functions and domain-agnostic deductive search. I give a comprehen-

sive overview of expressing prior work as instances on top of the PROSE framework, design trade-offs, and key benefits of such transition. I then present our evaluation of the PROSE framework and numerous PBE technologies developed on top of it in terms of synthesizer performance, disambiguation effectiveness, and software engineering effort. Finally, I discuss some limitations and extensions of the technique.

Chapter 6 studies the consequences of mass-market deployment of PROSE-based PBE applications. It introduces *interactive program synthesis*, a novel view on the program synthesis problem that arise in real-life interactions of end users with a synthesis-powered application. This view embraces program synthesis as an iterative process, in which the user is constantly refining his/her expression of the intent based on the behavior of the current program candidate on the held out data. I show how automatically and proactively reusing the learned insights from the previous iterations of synthesis in the same session aids the debugging and improves the user's confidence in the system. Specifically, I show how the speed of every synthesis iteration can be improved by performing program synthesis incrementally; how to lead the user toward better examples by automatically constructing clarifying questions that optimize the disambiguating power of the next refining example, and how incorporating these user interaction models in the synthesizer interface improves the quality of the synthesized programs and the users' confidence in them.

Chapter 2

RELATED WORK

The scope of this work encompasses three large areas: program synthesis, data wrangling, and active learning. All three communities have developed numerous alternative solutions to the problems related to the ones I discuss in this discuss. This chapter presents an overview of related work in both areas, focusing on significant, influential, highly relevant papers, as well as comparing and contrasting PROSE with the existing efforts.

2.1 Program Synthesis

In Chapter 1, we outlined the main techniques, strengths, and weaknesses of two large families of recent program synthesis works, *syntax-guided* and *domain-specific inductive synthesis*. The PROSE framework is our contribution to unifying the strengths of both approaches into a single system. It follows the recent line of effort in standardizing the program synthesis space and developing generic synthesis strategies and frameworks. Apart from PROSE, three main initiatives in this space are SKETCH [55], ROSETTE [56], and SyGuS [1].

2.1.1 SKETCH and ROSETTE

SKETCH, developed by Solar-Lezama [55], 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 [56] 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 this 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, 52]. To enable program synthesis in such cases, PROSE 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.

2.1.2 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 SyGuS-COMP 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 Chapter 1). The main synthesis algorithms in SyGuS are enumerative search [57], constraint-based search [22], and stochastic search [50]. 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.

2.1.3 Deductive Synthesis

The *deductive* approach to program synthesis [36] uses a formal system of deduction rules to build a constructive definition of a program in a way that proves the program's validity. This approach is efficient because at every point it works upon a partially correct program

and never needs to eliminate erroneous candidates (like enumerative search). Its drawback is substantial manual effort to axiomatize the application domain into sound and complete deduction rules.

The deductive search of PROSE is an extension of the described ideas, where we reduce this effort by using local deductive rules in the form of domain-specific *witness functions*. They characterize the behavior of a *single operator* on a *single input*, whereas deduction rules historically describe the behavior of *partial programs* on *all inputs*. This allows synthesis designers to easily provide domain-specific insight for program synthesis in complex DSLs.

2.1.4 Type-Based Synthesis

Type-based synthesis is another recent area in program synthesis research [10, 42]. It defines a synthesis problem as a problem of *finding a type inhabitant*, and uses the theory of refinement types to resolve it. A particularly attractive feature of type-based synthesis is that, similarly to PROSE, it uses deductive reasoning, building the desired program *top-down* from the specified properties. The spec, expressed as a refinement on the program's output value type, can range from input-output examples [10] to fully formalised properties [42], as long as these refinements allow logical reasoning. The prior work relies on *liquid type inference* [48] or conventional inference rules [40] to perform such reasoning.

Program synthesis with refinement types is very effective in domains that enable logical reasoning on simple type properties, such as data structures, functional programs, and security verification. On these domains it significantly outperforms alternative solutions (constraint solving and enumeration), due to the same feature that makes deduction efficient: it always operates on a partially correct program, filling in its holes by deducing necessary properties (in this case, refinements) of these holes and applying the same inference recursively. Similarly to constraint solves, applying this approach to data wrangling domains would require significant domain axiomatization and development of a domain-specific reasoning engine, which is highly non-trivial for industrial applications.

2.1.5 Oracle-Guided Inductive Synthesis

Jha and Seshia recently developed a novel formalism for inductive synthesis called *oracle-guided inductive synthesis* (OGIS) [21]. It unifies several commonly used approaches such as counterexample-guided inductive synthesis (CEGIS) [55] and distinguishing inputs [22]. In OGIS, an inductive learning engine is parameterized with a concept class (the set of possible programs), and it learns a concept from the class that satisfies a given partial specification by issuing queries to a given oracle. The oracle has access to the complete specification, and is parameterized with the types of queries it is able to answer. Queries and responses range over a finite set of types, including membership queries, witnesses, counterexamples, verification queries, and distinguishing inputs. They also present a theoretical analysis for the CEGIS learner variants, establishing relations between concept classes recognizable by learners under various constraints.

The problem of interactive program synthesis, presented in this work, can be mapped to the OGIS formalism (with the end user playing the role of an oracle). Hence, any theoretical results established by Jha and Seshia for CEGIS automatically hold for the settings of interactive program synthesis where we only issue counterexample queries.

In addition, inspired by our study of mass-market deployment of PBE-based systems, we present further formalism for the user's interaction with the synthesis system. While the "learner" component in the OGIS formalism is limited to a pre-defined class of queries, our formalism adds a separate modal "hypothesizer" component. Its job is to analyze the current set of candidate programs and to ask the questions that best resolve the ambiguity between programs in the set. The hypothesizer is domain-specific, not learner-specific, and therefore can be refactored out of the learner and reused with different synthesis strategies.

2.2 Data Wrangling

Data wrangling is a process of cleaning and transforming raw semi-structured data, converting it into a form amenable to analysis. Typically data is locked up in text logs, Web pages,

manually compiled spreadsheets, PDF reports, unstandardized CSV files, or even images (screenshots). By various estimates, data scientists spend over 80% of their time preparing data for analysis and only 20% gaining business insights from it [24].

Data wrangling is not unique to statisticians. Here are some unrelated applications that we have personally observed over the last several years:

- IT admins have to analyze server logs (usually plain text or JSON) to find a cause of failure. That requires extracting a sequence of entries from the log that match complex syntactic patterns.
- Security/forensics specialists similarly analyze server logs, looking for malicious events. However, patterns of malicious events are *semantic*, not *syntactic*, and typically cannot be described by a regular expression.
- Database integrators for large-scale applications (e.g. Bing, Google, Facebook graphs) use Web mining to extract facts and entities to compile their knowledge graphs. It requires analysis of raw HTML/DOM, extracting data based on learned or manually constructed patterns, and cross-validating extracted facts with existing databases or other webpages.

Various forms of data cleaning, parsing, and text manipulation have been addressed by the field of data mining. Most system for discovering cleaning transformations on the data perform some form of *active learning* by combining intelligent search with human assistance. Two prominent examples of this idea are Wrangler [23] (later Trifacta[1]) and DataXFormer [39]. Both of them apply an algorithm similar to enumerative search in program synthesis to discover transformations. However, they significantly differ in their (a) search space, and (b) kind of specification from a human.

Wrangler employs a method coined *predictive interaction*, where at each step in the session the system makes a conjecture about an atomic transformation step that the user might want to apply on the dataset next. To enable such interaction, their search space is a large loosely-

[1] https://trifacta.com

connected DSL of standard atomic transformations. In contrast, in PBE-based approaches the program being synthesized is an end-to-end sequence of multiple transformation steps. This makes the system much more accessible to end users (who do not necessarily understand an exact sequence of steps that should be applied), but also increases the level of ambiguity, since the number of programs consistent with a given set of input-output examples grows exponentially with program size.

DataXFormer builds transformations that combine together multiple data sources, such as web table or knowledge bases. It does not have a pre-defined DSL of transformations, inferring them automatically. In order to enable automatic inference of transformations (which may require domain-specific knowledge or table lookup), DataXFormer involves a human at each step of the search process, presenting partial results and asking for clarifications, additional transformation examples, and data filtering. The key component of this system is *table discovery*, which determines which data sources from a giant data lake may be relevant to the given example-based query. Since DataXFormer is interactive at every step, it can employ an EM-based ranking algorithm for resolving this ambiguity [39]. The confidence scores associated with each table are seeded with the number of examples it covers, and then iteratively recomputed as DataXFormer changes its beliefs about relevance of each table and each data point to the answer.

2.3 Active Learning

In machine learning, *active learning* is a subfield of semi-supervised learning where the learning algorithm is permitted to issue queries to an oracle (usually a human) [51]. As applied to, e.g., classification problems, a typical query asks for a classification label on a new data point. The goal is to achieve maximum classification accuracy with a minimum number of queries.

Research in active learning has focused on two important problems: (a) when to issue a query, and (b) how to pick a data point for it. For instance, in *uncertainty sampling* [34] an active learner queries the user on the input about which it is least certain of the labeling. The

particular uncertainty measure varies by model. A probabilistic model for learning a binary classifier could query about the input whose posterior probability is closest to 0.5 [34]. Other probabilistic classifiers use margin sampling [49] or (like one of our suggested disambiguation scores) the more general entropy approach.

This work borrows the ideas of active learning, extends them, and applies them in the domain of program synthesis. In our setting, the issued queries do not necessarily ask for the exact output of the desired program on a given input (an equivalent of "label" in ML), but may also ask for weaker output properties (e.g. verify a candidate element of the output sequence). In all cases, though, our queries are *actionable*: they are convertible to constraints, which automatically trigger a new iteration of synthesis. We also develop a novel approach for picking an input for the query based on its *disambiguation score* w.r.t. the current set of candidate programs.

Chapter 3

BACKGROUND

This chapter presents the preliminaries that are used throughout the thesis. First, I introduce FlashFill [14] and FlashExtract [33], two instances of prior synthesis work that are generalized by the PROSE framework. Then, I use them to describe the notation for PROSE DSLs and illustrate it on real-life motivating examples.

3.1 Programming by Examples

As discussed in Chapter 2, the field of domain-specific programming by examples (PBE) started with FlashFill [14] and its successors: FlashExtract [33], FlashRelate [4], and others. In this work, I use FlashFill and FlashExtract as the running examples of PBE DSLs generalized by the PROSE framework. For future reference, Figure 3.1 shows definitions of FlashFill and FlashExtract in the input syntax of PROSE.

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 $\mathscr{L}_{\mathsf{FF}}$ takes as input a tuple of *user input strings* $v_1, \ldots, v_k$, and returns an output string.

The FlashFill language $\mathscr{L}_{\mathsf{FF}}$ is structured in three layers. On the topmost layer, an n-ary ITE ("if-then-else") operator evaluates n predicates over the input string tuple, and chooses the branch that then produces the output string. Each ITE branch is a *concatenation* of some number of *atomic string expressions*, which produce pieces of the output string. Each such atomic 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 [14, Example 10])**.** Consider the problem of formatting phone numbers in a list of conference 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:

```
ConstStr('(') ∘ Match(v1, '\d+', 1) ∘ ConstStr(')␣') ∘
    Match(v1, '\d+', 2) ∘ ConstStr('-') ∘ Match(v1, '\d+', 3)
```

where `Match(`w`,` r`,` k`)` in $\mathcal{L}_{\mathsf{FF}}$ is a k^{th} match of a regex r in w — an abbreviation for "`let` $x = w$ `in Substring(`x`,` ⟨`RegexPos(`x`,` ⟨ε, r⟩`,` k`),` `RegexPos(`x`,` ⟨r, ε⟩`,` k`)`⟩`)`". Here we also use the notation ⟨x, y⟩ for "`std.Pair(`x`,` y`)`" and $f_1 \circ \ldots \circ f_n$ for n-ary string concatenation: "`Concat(`f_1`, Concat(..., 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, Le and Gulwani 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 reimplemented on top of the PROSE framework (see Section 5.5.1).

Each program in the FlashExtract DSL $\mathcal{L}_{\mathsf{FE}}$ takes as input a *textual document doc*, 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 regexes in *doc*.

(a)

```
language FlashFill;
// Nonterminals
@start bool expr[string[] inputs] :=
    let string ℓ = std.Kth(inputs, k) in std.ITE(match[ℓ], transform, expr[inputs]);
string transform := atom | Concat(atom, transform);
string atom := ConstStr(s) | let string x = std.Kth(inputs, k) in Substring(x, pp[x]);
Tuple<int, int> pp[string x] := std.Pair(pos[x], pos[x]);
int pos[string x] := AbsPos(x, k) | RegexPos(x, std.Pair(r, r), k);
bool match[string ℓ] := Contains(ℓ, r) | StartsWith(ℓ, r) | EndsWith(ℓ, r);
// Terminals
string s;      int k;      Regex r;
```

(b)

```
string Concat(string atom, string transform) => atom + transform;
string ConstStr(string s) => s;
string Substring(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) => k < 0 ? x.Length + k + 1 : k;
int? RegexPos(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;
}
[Values("r")] static readonly Regex[] StaticTokens =
    { new Regex("\\d+"), new Regex("[a-z]+"), /* more tokens... */ }
```

(c)

```
language FlashExtract.Text;
using grammar FF = FlashFill.Language;
// Nonterminals
@start StringRegion[] seq[StringRegion doc] :=
      @id['LinesMap'] std.Map(λ x ⇒ std.Pair(pos[x], pos[x]), lines)
    | @id['StartSeqMap'] std.Map(λ p ⇒ std.Pair(p, pos[GetSuffix(doc, p)]), positions)
    | @id['EndSeqMap'] std.Map(λ p ⇒ std.Pair(pos[GetPrefix(doc, p)], p), positions);
int[] positions := std.FilterInt(init, step, regexPositions);
int[] regexPositions := RegexMatches(doc, std.Pair(r, r));
int pos[string x] := FF.pos[x]; // External nonterminal
StringRegion[] lines := std.FilterInt(init, step, filterLines);
StringRegion[] filterLines := std.Filter(λ ℓ ⇒ FF.match[ℓ], docLines); // External nonterminal
StringRegion[] docLines := SplitLines(doc);
// Terminals
int init;      int step;      Regex r;
```

Figure 3.1: **(a)** A DSL of FlashFill substring extraction $\mathscr{L}_{\mathsf{FF}}$. **(b)** Executable semantics of FlashFill operators, defined by the DSL designer in C#, and a set of possible values for the terminal r. **(c)** FlashExtract DSL $\mathscr{L}_{\mathsf{FE}}$ for selection of spans in a textual document doc. It references position extraction logic pos and Boolean predicates $match$ from $\mathscr{L}_{\mathsf{FF}}$.

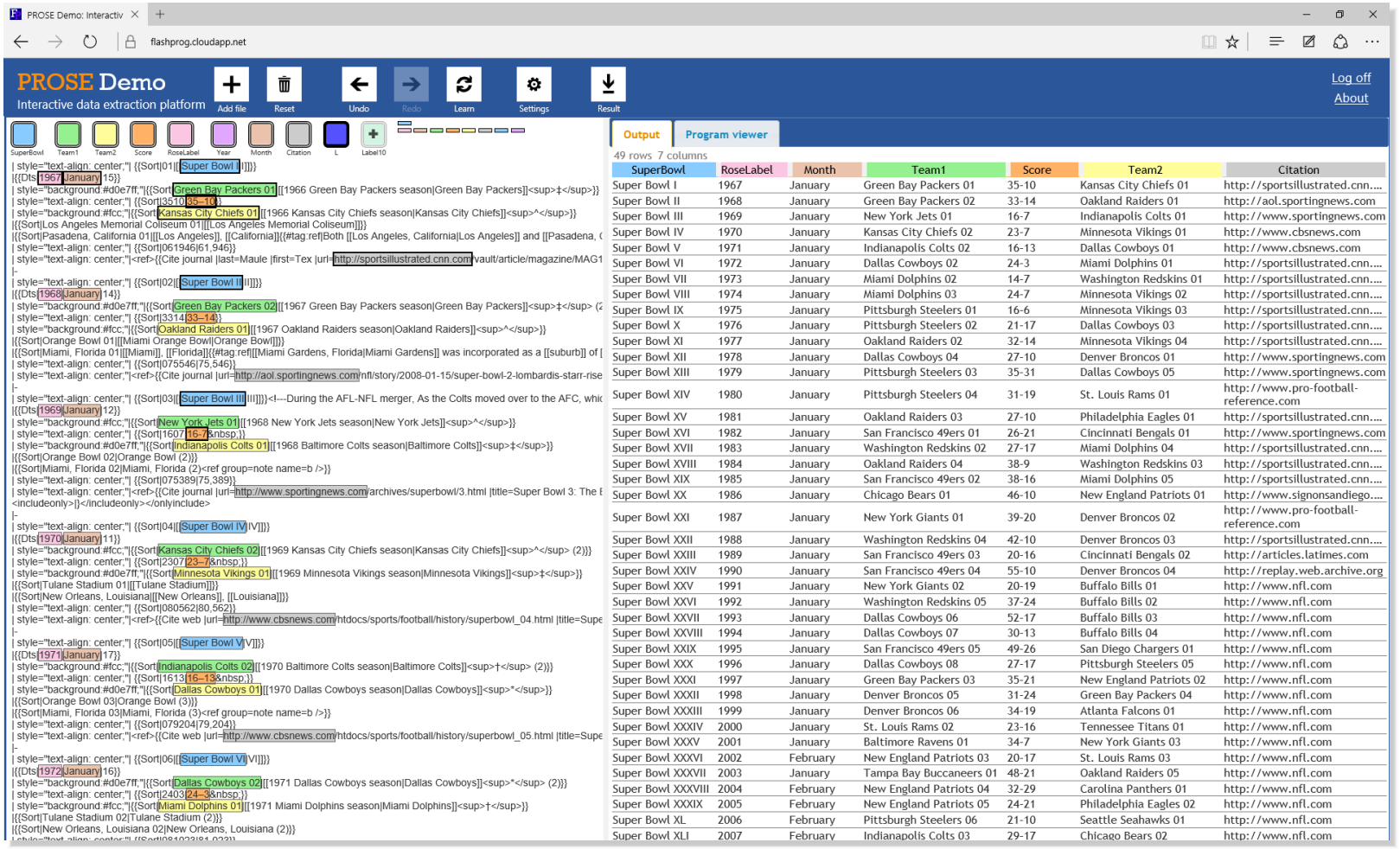


Figure 3.2: An illustrative scenario for data extraction from semi-structured text using FlashExtract.

Example 2. Consider the textual file shown in Figure 3.2. One possible $\mathscr{L}_{\mathsf{FE}}$ program that extracts a sequence of yellow regions (the scores) from it is

```
LinesMap(λx ⇒ ⟨RegexPos(x, ⟨'\|', ε⟩, -1), RegexPos(x, ⟨'\d+', ε⟩, -1)⟩,
    FilterInt(1, 3, Filter(λℓ ⇒
        StartsWith(ℓ, '\|␣style="text-align:␣center;"\|␣{␣{␣Sort\|' ∘ '[\w\d]+'),
        SplitLines(doc))))
```

It splits *doc* into a sequence of lines, selecting only the lines that start with a certain constant string followed by an alphanumeric token (i.e., a sequence of alphanumeric characters). Then, from every third such line starting from the second one, it extracts the string between the last occurrence of "|" and the last occurrence of a number.

3.2 Domain-Specific Language

A synthesis problem is defined for a given *domain-specific language* (DSL) $\mathscr{L}$. The language of DSL definitions is given in Figure 3.3. A DSL is specified as a context-free grammar (CFG), with each nonterminal symbol N defined through a set of *rules*. Each rule has on its right-hand side an application of an *operator* to some symbols of $\mathscr{L}$, and we denote the set of all possible operators on the right-hand sides of the rules for N as $\mathsf{RHS}(N)$. All symbols and operators are *typed*. Every symbol N in the CFG is annotated with a corresponding output type τ, denoted $N : \tau$. If $N := F(N_1, \ldots, N_k)$ is a grammar rule and $N : \tau$, then the output type of F must be τ. A DSL has a designated *start symbol* $\mathsf{start}(\mathscr{L})$, which is a root nonterminal in the CFG of $\mathscr{L}$.

Every (sub-)program P rooted at a symbol $N : \tau$ in $\mathscr{L}$ maps an *input state*[1] σ to a value of type τ. The execution result of a program P on a state σ is written as $[\![P]\!]\sigma$. A state is a mapping of free variables $\mathsf{FV}(P)$ to their bound values. Variables in a DSL are either explicitly declared as *nonterminal parameters*, or introduced by `let` definitions and λ-functions. The start symbol has a single parameter variable — an *input symbol* $\mathsf{input}(\mathscr{L})$ of the DSL.

[1]DSLs in PBE typically do not involve mutation, so an input σ is technically an *environment*, not a *state*. We keep the term "state" for historical reasons.

document ::= *reference* using* languge-name decl*$^{+}$

reference ::= **#reference** *dll-path* **;**

using ::= **using** *namespace* **;**
| **using semantics** *type* **;**
| **using learners** *type* **;**
| **using language** *namespace* = *member-name* **;**

language-name ::= **language** *namespace* **;**

decl ::= *annotation* type symbol* (**[** *params* **]**)$_{\text{opt}}$ (**:=** *nonterminal-body*)$_{\text{opt}}$ **;**

annotation ::= **@start** | **@id[***symbol-name***]** | **@values[***member-name***]** | ...

params ::= *param* (**,** *param*)*

param ::= *type symbol*

nonterminal-body ::= *rule* (**|** *rule*)*

rule ::= (*namespace* **.**)$_{\text{opt}}$ *symbol*
| (*namespace* **.**)$_{\text{opt}}$ *operator-name***(**(*arg* (**,** *arg*)*)$_{\text{opt}}$ **)**
| **let** *type symbol* **=** *rule* **in** *rule*

arg ::= *symbol* | $\boldsymbol{\lambda}$ *symbol* **⇒** *rule*

symbol, operator-name, namespace ::= ⟨id⟩

dll-path, symbol-name ::= **'**⟨string⟩**'**

member-name ::= ⟨member in a target language⟩

type ::= ⟨type in a target language⟩

Figure 3.3: PROSE DSL definition language. A DSL is a set of (typed and annotated) symbol definitions, where each symbol is either a terminal, or a nonterminal (possibly with parameters) 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. A DSL may reference types and members from external libraries, as well as other DSLs (treated as namespace imports).

Every operator F in a DSL $\mathscr{L}$ has some executable semantics. Many operators are generic, included in the standard library (`std`), and typically reused across different DSLs (e.g. `Filter` and `Map` list combinators). Others are domain-specific, and defined only for a given DSL. Operators are assumed to be deterministic and pure, modulo unobservable side effects.

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

- `@start` – marks the start symbol of a DSL;
- `@id['`*symbol-name*`']` – gives a designated name to the following rule, which may be used to reference it from the accompanying source code;
- `@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.

3.3 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 [14], IGOR2 [27], Magic Haskeller [25] 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 instance, in FlashFill, to synthesize a substring program that extracts a substring s from a given input string v, we synthesize a position extraction program that returns *any*

occurrence of s in v (which is a *disjunctive* specification). In addition, 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 $N : \tau$ be a symbol in a DSL $\mathcal{L}$. An **inductive specification** ("spec") 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 $\langle \sigma_1, \psi_1 \rangle, \ldots, \langle \sigma_{|\vec{\tau}|}, \psi_{|\vec{\tau}|} \rangle$. Each atom is a pair of a concrete input state σ_i over $\mathsf{FV}(N)$ and a **constraint** $\psi_i : \tau \to \mathsf{Bool}$ constraining the output of a desired program on the input σ_i.*

Definition 2 (Valid program)**.** *We say that a program P **satisfies** a spec φ (written $P \models \varphi$) iff the formula $\varphi\,[\langle \sigma_i, \psi_i \rangle := \psi_i\,([\![P]\!]\sigma_i)]$ holds. In other words, φ should hold as a boolean formula over the statements "the output of P on σ_i satisfies the constraint ψ_i" as atoms.*

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

Example 3. Given a set of I/O examples $\{\sigma_i, o_i\}_{i=1}^m$, a program P is valid iff $[\![P]\!]\sigma_i = o_i$ for all i. In our formulation, such a spec is represented as a conjunction $\varphi = \bigwedge_{i=1}^m \langle \sigma_i, \psi_i \rangle$. Here each constraint ψ_i on the program output is an *equality predicate*: $\psi_i(v) := [v = o_i]$.

Example 4. 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 Y (a subsequence of spans from the desired output), and a set of *negative* examples N (a set of spans that the desired output should not intersect).

The spec here is a conjunction $\varphi = \langle \sigma, \psi^+ \rangle \wedge \langle \sigma, \psi^- \rangle$, where positive constraint ψ^+ is a *subsequence predicate*: $\psi^+(v) := [v \sqsubseteq Y]$, and negative constraint ψ^- is a *non-intersection predicate*: $\psi^-(v) := [v \cap N = \emptyset]$. The associated input states for both constraints are equal to the same $\sigma = \{doc \mapsto D\}$, where symbol $doc = \mathsf{input}(\mathcal{L})$.

Often a spec 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 $\varphi = \{\sigma_i \rightsquigarrow \psi_i\}_{i=1}^m$. A

program P satisfies this spec if its output on each input state σ_i satisfies the constraint ψ_i. For instance, I/O examples are written as $\{\sigma_i \rightsquigarrow o_i\}_{i=1}^{m}$, and subsequence specs of FlashExtract are written as $\sigma \rightsquigarrow (\mathbf{?} \sqsubset Y) \wedge (\mathbf{?} \cap N = \emptyset)$, where **?** denotes the desired output of P.

3.4 Domain-Specific Program Synthesis

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

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

Some instances of domain-specific synthesis include:

- Finding **one** program: $|\mathcal{S}| = 1$.
- **Complete synthesis:** finding *all* programs.
- **Ranked synthesis:** finding k *topmost-ranked* programs according to some domain-specific *ranking function* $h : \mathscr{L} \to \mathbb{R}$.
- **Incremental synthesis:** finding one, all, or some programs assuming that the given spec φ *refines* some previously accumulated spec φ_0, and the previous learned result for φ_0 is $\mathsf{Learn}(N, \varphi_0) = \mathcal{S}_0$.

We discuss all these specific problems and their implementation in the PROSE framework in the following sections.

By definition, any algorithm for solving Problem 1 must be *sound* — every program in its returned program set $\mathcal{S}$ must satisfy φ. A synthesis algorithm is said to be *complete* if it learns *all possible* programs in $\mathscr{L}$ that satisfy φ. Since $\mathscr{L}$ is usually infinite because of unrestricted constants and recursion, completeness is usually defined w.r.t. some finitization of $\mathscr{L}$ (possibly dependent on a given synthesis problem).

Chapter 4

VERSION SPACE ALGEBRA

A typical practical DSL may have up to 10^{30} programs consistent with a given spec [53]. Solving a synthesis problem over such a DSL 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 [38] in the context of machine learning, and later used for programming by demonstration in SMARTedit [32], FlashFill [14], 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, I 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., Gulwani, and others. I define a set of efficient operations over this data structure that are used by several synthesis algorithms, including the deductive algorithm of PROSE.

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 I use the words "program set" and "version space algebra" interchangeably, and use the same notation for both concepts.

Definition 4 (Version space algebra)**.** *Let $\mathscr{L}$ be a DSL. A **version space algebra** is a representation for a set $\mathcal{S}$ of programs rooted at the same symbol of $\mathscr{L}$, with the following grammar:*

$$\mathcal{S} \;:=\; \{P_1, \ldots, P_k\} \;\mid\; \cup(\mathcal{S}_1, \ldots, \mathcal{S}_k) \;\mid\; F_{\bowtie}(\mathcal{S}_1, \ldots, \mathcal{S}_k)$$

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

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

$$P \in \bigcup(\mathcal{S}_1, \ldots, \mathcal{S}_k) \qquad \text{if } \exists j\colon P \in \mathcal{S}_j$$

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

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

Note that programs in a VSA 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 $V(\mathcal{S}) = \mathcal{O}(\log|\mathcal{S}|)$.

Example 5. Figure 4.1 shows a VSA of all FlashFill programs that output the string "425" on a given input state $\sigma = \{v_1 \mapsto$ "(425) 555-7709"$\}$. The string "425" can be represented in four possible ways as a concatenation of individual substrings. Each substring of "425" can be produced by a ConstStr program or a SubStr program.

I 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. [32] and then adapted by us for FlashFill [14]. It has been defined specifically for VSAs built on the SMARTedit and FlashFill DSLs. In this section, I define VSA intersection generically, and also introduce ranking and clustering of VSAs, which are novel contributions of this thesis.

4.1 Intersection

Intersection of VSAs enables quick unification of two sets of candidate programs that are consistent with two different specs. Given a conjunctive spec $\varphi_1 \wedge \varphi_2$, one possible synthesis

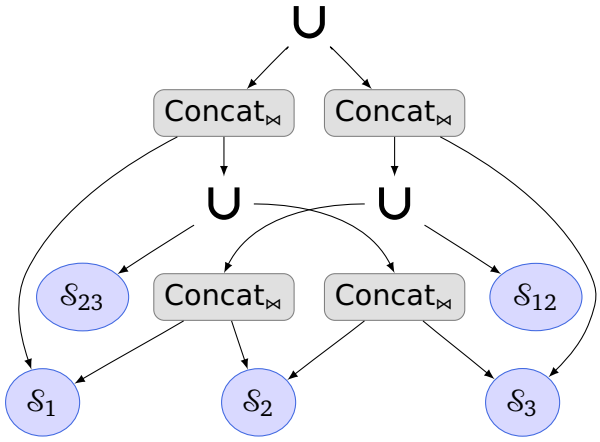


Figure 4.1: A VSA representing all possible programs output the string "425" in $\mathscr{L}_{\mathsf{FF}}$. Leaf nodes $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$, $\mathcal{S}_{12}$, $\mathcal{S}_{23}$ are VSAs representing all ways to output substrings "4", "2", "5", "42", and "25" respectively (not expanded in the figure).

approach is to learn a set of programs $\mathcal{S}_1$ consistent with φ_1, another set of programs $\mathcal{S}_2$ consistent with φ_2, and then intersect $\mathcal{S}_1$ with $\mathcal{S}_2$. An efficient algorithm for VSA intersection follows the ideas of automata intersection [19].

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

$$
\begin{aligned}
\left[\mathsf{U}(\mathcal{S}'_1, \ldots, \mathcal{S}'_k)\right] \cap \mathcal{S}_2 &\stackrel{\text{def}}{=} \mathsf{U}\left(\mathcal{S}'_1 \cap \mathcal{S}_2, \ldots, \mathcal{S}'_k \cap \mathcal{S}_2\right) \\
F_{\bowtie}(\mathcal{S}'_1, \ldots, \mathcal{S}'_k) \cap G_{\bowtie}(\mathcal{S}''_1, \ldots, \mathcal{S}''_m) &\stackrel{\text{def}}{=} \emptyset \\
F_{\bowtie}(\mathcal{S}'_1, \ldots, \mathcal{S}'_k) \cap F_{\bowtie}(\mathcal{S}''_1, \ldots, \mathcal{S}''_k) &\stackrel{\text{def}}{=} F_{\bowtie}(\mathcal{S}'_1 \cap \mathcal{S}''_1, \ldots, \mathcal{S}'_k \cap \mathcal{S}''_k) \\
F_{\bowtie}(\mathcal{S}'_1, \ldots, \mathcal{S}'_k) \cap \{P_1, \ldots, P_m\} &\stackrel{\text{def}}{=} \left\{P_j = F(P'_1, \ldots, P'_k) \mid P'_j \in \mathcal{S}'_j\right\} \\
\mathcal{S}_1 \cap \mathcal{S}_2 &\stackrel{\text{def}}{=} \mathcal{S}_1 \cap \mathcal{S}_2 \ \textit{as direct program sets otherwise}
\end{aligned}
$$

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

Proof. Follows by induction over the structure of $\mathcal{S}_1$ and $\mathcal{S}_2$ in Definition 6. The only non-trivial case is union intersection $[\mathsf{U}(\mathcal{S}'_1, \ldots, \mathcal{S}'_k)] \cap [\mathsf{U}(\mathcal{S}''_1, \ldots, \mathcal{S}''_m)]$, which introduces $k \cdot m$ constructors per level in $\mathcal{S}_1 \cap \mathcal{S}_2$. □

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

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

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

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 [17], or in $\mathcal{O}(m \log k)$ time using a heap.

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

Proof. Let d be the depth (number of levels) of $\mathcal{S}$. Note that $V(\mathcal{S}) = \mathcal{O}(m^d)$. Let $T(n)$ denote the time complexity of calculating $\mathsf{Top}_h(\mathcal{S}_{(n)}, k)$, where $\mathcal{S}_{(n)}$ is the n^{th} level of $\mathcal{S}$. 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 $\mathcal{S}_{(n)}$ is a join rather than a union, we can ignore the non-dominant union case. Solving the recurrence for $T(n)$, we get:

$$T(d) = \mathcal{O}\left(m^d \log k + (m-1)^{-1} k^m (m^{d-1} - 1) \log k\right) = \mathcal{O}(m^d k^m \log k) = \mathcal{O}(V(\mathcal{S}) \cdot k^m \log k) \quad \square$$

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 $\mathcal{S}$ and an input state σ over $\mathsf{FV}(N)$, the **clustering** of $\mathcal{S}$ on σ, denoted $\mathcal{S}/_\sigma$, is a mapping $\{v_1 \mapsto \mathcal{S}_1, \ldots, v_n \mapsto \mathcal{S}_n\}$ such that:*

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

In other words, $\mathcal{S}/_\sigma$ is a partitioning of $\mathcal{S}$ 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 $\mathcal{S}$ into subsets of programs that given the same output $\vec{v}$ on $\vec{\sigma}$.

Constructively, $\mathcal{S}/_\sigma$ is defined as follows:

$$\{P_1, \ldots, P_k\}/_\sigma \stackrel{\text{def}}{=} \mathsf{G}\left(\{[\![P_i]\!]\sigma \mapsto \{P_i\} \;\middle|\; i = 1..k\}\right)$$

$$\mathsf{U}(\mathcal{S}_1, \ldots, \mathcal{S}_k)/_\sigma \stackrel{\text{def}}{=} \mathsf{G}\left(\bigcup\nolimits_k \mathcal{S}_k/_\sigma\right)$$

$$F_\bowtie(\mathcal{S}_1, \ldots, \mathcal{S}_k)/_\sigma \stackrel{\text{def}}{=} \mathsf{G}\left(\left\{[\![F(v_1, \ldots, v_k)]\!]\sigma \mapsto F_\bowtie(\mathcal{S}'_1, \ldots, \mathcal{S}'_k) \;\middle|\; \langle v_j, \mathcal{S}'_j\rangle \in \mathcal{S}_j/_{\sigma_j}\right\}\right)$$

where $\sigma_1, \ldots, \sigma_k$ are input states passed by F into its arguments during execution on σ, and

$$\mathsf{G}(\{v_1 \mapsto \mathcal{S}_1, \ldots, v_n \mapsto \mathcal{S}_n\}) \stackrel{\text{def}}{=} \left\{v \mapsto \bigcup_{v \mapsto \mathcal{S}_j} \mathcal{S}_j \;\middle|\; \text{all unique } v \text{ among } v_j\right\}$$

is a "group-by" function. It groups a given set of bindings by keys, uniting VSAs that correspond to the same key with a $\cup$ 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 $\mathcal{S}$ that are inconsistent with a given spec:

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

Theorem 3. *If $\vec{\sigma}$ is a state tuple associated with φ, then:*

(1) $\mathsf{Filter}(\mathcal{S}, \varphi) = \bigcup\left\{\mathcal{S}_j \;\middle|\; \langle \vec{v}_j \mapsto \mathcal{S}_j\rangle \in \mathcal{S}/_{\vec{\sigma}} \wedge \varphi(\vec{v}_j)\right\}$.

(2) The construction of $\mathsf{Filter}(\mathcal{S}, \varphi)$ *according to (1) takes* $\mathcal{O}\left(\left|\mathcal{S}/_{\vec{\sigma}}\right|\right)$ *time after the clustering.*

Proof. Follows by construction from Definition 9. □

Estimating the size of $\mathcal{S}/_{\vec{\sigma}}$ and the running time of clustering is difficult, since these values depend not only on $\mathcal{S}$ 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 Section 5.5.2 I show that these values usually remain small for any practical DSL.

4.4 VSA as a Language

Another view on a version space algebra is that it is a different representation of a *language*, isomorphic to a context-free grammar (CFG). 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 is also a representation for a set of programs; it is an AST-based representation that leverages two syntactic features: *union* ($\cup$) and *join of shared VSAs* ($F_{\bowtie}$). These representations are isomorphic; in fact, a VSA built from a DSL $\mathcal{L}$ is essentially a CFG for some DSL $\mathcal{L}' \subseteq \mathcal{L}$. For instance, Figure 4.2 shows an example of a small DSL (an excerpt from FlashFill) and a VSA that represents a finite sub-language of this DSL.

```
int position[string s] :=
    AbsPos(s, k)
  | RegexPos(s, std.Pair(r, r), k);
int k;
Regex r;
```

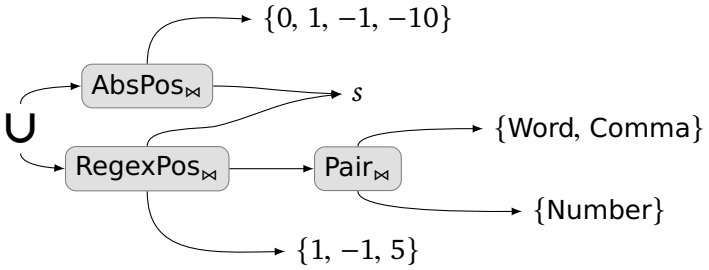


Figure 4.2: A VSA can be seen as an isomorphic representation for a CFG of some sub-language of the original DSL. *Left:* An excerpt from the FlashFill DSL definition. *Right:* A sample VSA that represents a sub-language of the DSL on the top.

This observation is important because it allows us (with careful implementation) to treat VSAs and CFGs interchangeably. Such treatment is the main driver behind seamless *incremental program synthesis* in the PROSE framework, which I describe in Section 6.4.

Chapter 5

THE PROSE FRAMEWORK

In this section, I describe the methodology of program synthesis the PROSE framework: *deductive synthesis driven by domain-specific witness functions*. I first introduce the notion of witness functions informally in Section 5.1, deriving it from various instances of prior work in PBE (such as FlashFill and FlashExtract). Then, in Section 5.3, I give a formal definition of witness functions, as well as a comprehensive review of their usage in PROSE-based PBE technologies. Section 5.4 presents the main synthesis algorithm of PROSE, *deductive search*.[1] In Section 5.5, I show its evaluation on 12+ real-life case studies – PBE technologies developed on top of the framework. Finally, Section 5.6 discusses various limitations and extensions to the algorithm.

5.1 Intuition

The main observation behind PROSE is that most of prior synthesis algorithms in PBE can be decomposed into **(a)** domain-specific insights for refining a spec for the operator into the specs for its parameters, and **(b)** a common deductive search algorithm, which leverages these insights for iterative reduction of the synthesis problem.

For instance, Figure 5.1 shows a portion of the synthesis algorithms for FlashFill [14, Figure 7] and FlashExtract [33, 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 three phases, implicitly hidden in the "fused" original presentation:

[1] Also known in the literature as *divide-and-conquer search*, *top-down search*, and *backpropagation-based search* (not to be confused with the backpropagation algorithm for training neural networks [7], although this name was inspired by similarities between the two techniques).

(a)

function GENERATESUBSTRING(σ: Input state, s: String)
  $result \leftarrow \emptyset$
  for all (i, k) s.t. s is substring of $\sigma(v_i)$ at position k **do**
    $Y_1 \leftarrow$ GENERATEPOSITION($\sigma(v_i)$, k)
    $Y_1 \leftarrow$ GENERATEPOSITION($\sigma(v_i)$, $k + \mathsf{Length}(s)$)
    $result \leftarrow result \cup \{\mathsf{SubStr}(v_i, Y_1, Y_2)\}$
  return *result*
function GENERATEPOSITION(s: String, k: int)
  $result \leftarrow \{\mathsf{CPos}(k), \mathsf{CPos}(-(\mathsf{Length}(s) - k))\}$
  for all $r_1 = \mathsf{TokenSeq}(T_1, \ldots, T_n)$ matching $s[k_1 : k-1]$ for some k_1 **do**
    for all $r_2 = \mathsf{TokenSeq}(T'_1, \ldots, T'_m)$ matching $s[k : k_2]$ for some k_2 **do**
      $r_{12} \leftarrow \mathsf{TokenSeq}(T_1, \ldots, T_n, T'_1, \ldots, T'_m)$
      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
      $\tilde{r_1} \leftarrow$ GENERATEREGEX(r_1, s)
      $\tilde{r_2} \leftarrow$ GENERATEREGEX(r_2, s)
      $result \leftarrow result \cup \{\mathsf{Pos}(\tilde{r_1}, \tilde{r_2}, \{c, -(c' - c + 1)\})\}$
  return *result*

(b)

function MAP.LEARN(Examples φ: Dict⟨State, List⟨T⟩⟩)
  Let φ be $\{\sigma_1 \mapsto Y_1, \ldots, \sigma_m \mapsto Y_m\}$
  for $j \leftarrow 1 \ldots m$ **do**
    Witness subsequence $Z_j \leftarrow \mathsf{Map.Decompose}(\sigma_j, Y_j)$
  $\varphi_1 \leftarrow \{\sigma[Z_j[i]/x] \mapsto Y_j[i] \mid i = 0..|Z_j| - 1,\ j = 1..m\}$
  $\mathcal{S}_1 \leftarrow F.\mathsf{Learn}(\varphi_1)$
  $\varphi_2 \leftarrow \{\sigma_j \mapsto Z_j \mid j = 1..m\}$
  $\mathcal{S}_2 \leftarrow S.\mathsf{Learn}(\varphi_2)$
  return $\mathsf{Map}(\mathcal{S}_1, \mathcal{S}_2)$

function FILTER.LEARN(Examples φ: Dict⟨State, List⟨T⟩⟩)
  $\mathcal{S}_1 \leftarrow S.\mathsf{Learn}(\varphi)$
  $\varphi' \leftarrow \{\sigma[Y[i]/x] \mapsto \mathsf{true} \mid (\sigma, Y) \in \varphi,\ i = 0..|Y| - 1\}$
  $\mathcal{S}_2 \leftarrow F.\mathsf{Learn}(\varphi')$
  return $\mathsf{Filter}(\mathcal{S}_2, \mathcal{S}_1)$

Figure 5.1: **(a)** FlashFill synthesis algorithm for learning substring expressions [14, Figure 7]; **(b)** FlashExtract synthesis algorithm for learning Map and Filter sequence expressions [33, 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 FlashFill and FlashExtract.

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 5.1, we highlight parts of the code that implement phase 2.

It is important that phases 2 and 3 interleave as nested loops for each argument N_j of the currently synthesized rule, because the examples deduced for an argument N_j may depend on the possible value of the previously processed argument N_{j-1}. For instance, the example positions k, deduced for the nonterminal p in GenerateSubstring, depend on the currently selected possible input string v_i.

In FlashExtract, list processing operators $\mathsf{Map}(F, S)$ and $\mathsf{Filter}(F, S)$ are learned generically from a similar *subsequence spec* φ of type $\{\sigma \rightsquigarrow \mathbf{?} \sqsupseteq \mathsf{List}\langle T\rangle\}$. Their learning algorithms are also shown in Figure 5.1. Similarly, they are organized in the "divide-and-conquer" manner, wherein the given spec is first transformed into the corresponding parameter specs, and then the synthesis on them is invoked recursively. FlashExtract provides generic synthesis for five such list-processing operators independently of their DSL instantiations. However, the domain-specific insight required to build a spec for the Map's F parameter depends on the specific DSL instantiation of this Map (such as LinesMap or StartSeqMap in $\mathscr{L}_{\mathsf{FE}}$), 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.

5.2 *Derivation*

Consider the operator $\mathsf{LinesMap}(F, L)$ in $\mathscr{L}_{\mathsf{FE}}$ (see Figure 3.1). It takes as input a (sub-)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, the simplest type signature of LinesMap 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* $\varphi = \sigma \rightsquigarrow v$*"*. Here v is an output value, i.e. a list of extracted strings. To solve this problem, we need to compute the following set of expressions:

$$\{(F, L) \mid \mathsf{LinesMap}(F, L) \models \varphi\} \tag{5.1}$$

Applying the principle of divide-and-conquer, in order to solve (5.1), we can compute an *inverse semantics* of LinesMap:

$$\mathsf{LinesMap}^{-1}(v) \stackrel{\text{def}}{=} \{(f, \ell) \mid \mathsf{LinesMap}(f, \ell) = v\} \tag{5.2}$$

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

Finitization Computation of $\mathsf{LinesMap}^{-1}(v)$ is challenging. One possible approach could be to enumerate all argument values $f \in \mathbf{codom}\, F$, $\ell \in \mathbf{codom}\, L$, retaining only those for which $\mathsf{LinesMap}(f, \ell) = v$. However, both $\mathbf{codom}\, L$ (all string lists) and $\mathbf{codom}\, F$ (all $\mathsf{String} \to \mathsf{String}$ functions) are infinite.[2]

In order to finitize the value space, we apply domain-specific knowledge about the behavior of LinesMap in $\mathscr{L}_{\mathsf{FE}}$. Namely, we use that 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 $((\mathsf{Line} \to \mathsf{StringRegion}),\ \mathsf{List}\langle\mathsf{Line}\rangle) \to \mathsf{List}\langle\mathsf{StringRegion}\rangle$ where StringRegion is a domain-specific type for "a region within a given

[2]Here $\mathbf{codom}\, F$ denotes the *codomain* of an operator F, i.e. the set of its possible outputs.

document". This is a *dependent type*, parameterized by the state σ that contains our input document. Thus, the inverse semantics $\mathsf{LinesMap}^{-1}$ is also implicitly parameterized with σ:

$$\mathsf{LinesMap}^{-1}(\sigma \rightsquigarrow v) = \{(f, \ell) \mid f \text{ and } \ell \text{ can be obtained from } \sigma \ \wedge \ \mathsf{LinesMap}(f, \ell) = v\}$$

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

Decomposition The synthesis procedure above is finite, but not necessarily efficient. For most values of ℓ, there may be no satisfying program L in the DSL to produce it, and therefore even computing matching values of f for it is redundant. For instance, 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 $\mathsf{LinesMap}(f, \ell)$ regardless of f. In general, *for many domain-specific operators $F(X, Y)$ computing all viable arguments (x, y) is much slower than computing matching ys only for the realizable values of X*.

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

1. If $[\![\mathsf{LinesMap}(F, L)]\!]\sigma = v$, what could be the possible output of L?
2. If $[\![\mathsf{LinesMap}(F, L)]\!]\sigma = v$ 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:

$$\mathsf{LinesMap}_L^{-1}(v) \ \supset \ \{\ell \mid \exists f : \mathsf{LinesMap}(f, \ell) = v\} \tag{5.3}$$

$$\mathsf{LinesMap}_F^{-1}(v \mid L = \ell) \ = \ \{f \mid \mathsf{LinesMap}(f, \ell) = v\} \tag{5.4}$$

The key insight of this technique is that, by the principle of *skolemization* [18], inverse semantics $\mathsf{LinesMap}^{-1}(v)$ can be expressed as a cross-product computation of parameter inverses $\mathsf{LinesMap}_L^{-1}(v)$ and $\mathsf{LinesMap}_F^{-1}(v \mid L = \ell)$ for all possible $\ell \in \mathsf{LinesMap}_L^{-1}(v)$:

$$\begin{aligned}\mathsf{LinesMap}^{-1}(v) &= \{(f, \ell) \mid \mathsf{LinesMap}(f, \ell) = v\} \\ &= \left\{(f, \ell) \mid \ell \in \mathsf{LinesMap}^{-1}(v), f \in \mathsf{LinesMap}_F^{-1}(v \;\middle|\; L = \ell)\right\}\end{aligned} \tag{5.5}$$

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

1. Enumerate $\ell \in \mathsf{LinesMap}_L^{-1}(v)$.
2. For each ℓ recursively find a set of programs $\mathcal{S}_\ell$ rooted at L such that $[\![L]\!]\sigma = \ell$.
3. Enumerate $f \in \mathsf{LinesMap}_F^{-1}(v \mid L = \ell)$ for all those ℓ for which the program set $\mathcal{S}_\ell$ is non-empty.
4. For each f recursively find a set of programs $\mathcal{S}_{\ell,f}$ rooted at F such that $[\![F]\!]\sigma = f$.
5. Now, for any such combination of parameter programs $L \in \mathcal{S}_\ell$, $F \in \mathcal{S}_{\ell,f}$ we guarantee $\mathsf{LinesMap}(F, L) \models \varphi$ by construction.

Generalization In practice, Problem 1 for LinesMap is rarely solved for example-based specs: as discussed in Section 3.3, 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 *specs 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 spec for L given a spec φ on $\mathsf{LinesMap}(F,\ L)$ (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 a *prefix spec* $\varphi = \{\sigma \rightsquigarrow [r_1, r_2, \ldots]\}$ for the output list of regions. The witness functions for such a spec arise naturally:

- 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 i^{th} element of ℓ into the i^{th} 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.

5.3 Witness Functions

As described above, a *witness function* is a generalization of inverse semantics for an operator. In other words, it is a *problem reduction logic*, which deduces a necessary (or sufficient) spec on the operator's parameters given a desired spec on the operator's output. In this section, I define witness functions formally and give various examples of their usages in the existing PBE technologies.

Definition 10 (Witness function)**.** *Let $F(N_1, \ldots, N_k)$ be an operator in a DSL $\mathscr{L}$. A **witness function** of F for N_j is a function $\omega_j(\varphi)$ that deduces a **necessary** spec φ_j on N_j given a spec φ on $F(N_1, \ldots, N_k)$. Formally, $\omega_j(\varphi) = \varphi_j$ iff the following implication holds:*[3]

$$F(N_1, \ldots, N_k) \models \varphi \qquad \Longrightarrow \qquad N_j \models \varphi_j.$$

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 spec is **necessary and sufficient**. Formally, $\omega_j(\varphi) = \varphi_j$ is precise iff*

$$N_j \models \varphi_j \qquad \Longleftrightarrow \qquad \exists\, N_1, \ldots, N_{j-1}, N_{j+1}, \ldots, N_k \colon\ F(N_1, \ldots, N_k) \models \varphi.$$

Definition 12 (Conditional witness function)**.** *A **(precise) conditional witness function** of $F(N_1, \ldots, N_k)$ for N_j is a function $\omega_j\left(\varphi \mid N_{k_1} = v_1, \ldots, N_{k_s} = v_s\right)$ that deduces a necessary (and sufficient) spec φ_j on N_j given a spec φ on $F(N_1, \ldots, N_k)$ under the assumption that a subset of other parameters $N_{k_1}, \ldots, N_{k_s}$ of F (called **prerequisites**) have fixed values $v_1, \ldots, v_k$. Formally, $\omega_j(\varphi \mid N_t = v_t) = \varphi_j$ iff the following implication holds:*

$$F(N_1, \ldots, N_{t-1}, v_t, N_{t+1}, \ldots, N_k) \models \varphi \qquad \Longrightarrow \qquad N_j \models \varphi_j.$$

Operator	**Input spec**	**Parameter**	**Prerequisites**	**Output spec**
`Concat(`$atom$`,` $transform$`)`	$\sigma \rightsquigarrow w$	$atom$		$\sigma \rightsquigarrow \bigvee_{j=1}^{\lvert w\rvert-1} w[0..j]$
		$transform$	$atom = v$	$\sigma \rightsquigarrow w[\lvert v\rvert..]$
`ConstStr(`s`)`	$\sigma \rightsquigarrow w$	s		$\sigma \rightsquigarrow w$
`let` x `=` `std.Kth(`$\underbrace{inputs, k}_{binding}$`)` `in Substring(`$x$`,` pp`)`	$\sigma \rightsquigarrow w$	$binding$		$\sigma \rightsquigarrow \bigvee_{j:\ w \text{ occurs in } v_j} v_j$
		pp	$x = v$	$\sigma \rightsquigarrow \bigvee_{\substack{w \text{ occurs in } v \\ \text{at position } l}} \langle l, l + \lvert w\rvert\rangle$
`AbsPos(`x`,` k`)`	$\sigma \rightsquigarrow c$	k	$x = v$	$\sigma \rightsquigarrow c \vee c - \lvert v\rvert - 1$
`RegexPos(`x`,` `std.Pair(`$\underbrace{r, r}_{rr}$`)`, k`)`	$\sigma \rightsquigarrow c$	rr	$x = v$	$\sigma \rightsquigarrow \bigvee_{\substack{\langle r_1, r_2\rangle:\ r_1 \text{ matches left of } c \\ \wedge\ r_2 \text{ matches right of } c}} \langle r_1, r_2\rangle$
		k	$x = v$, $rr = \langle r_1, r_2\rangle$	$\sigma \rightsquigarrow j \vee j - \lvert\vec{c}\rvert - 1$ where $\vec{c}$ are the matches of $\langle r_1, r_2\rangle$ in v, j is an index of c in $\vec{c}$

Table 5.1: Witness functions for various FlashFill operators.

Operator	Input spec	Parameter	Prerequisites	Output spec
`Kth(`xs`,` k`)`	$\sigma \rightsquigarrow w$	xs		$\sigma \rightsquigarrow [\ldots, w, \ldots]$
		k	$xs = \vec{v}$	$\sigma \rightsquigarrow \bigvee_{v_j = w} j$
`Pair(`p_1`,` p_2`)`	$\sigma \rightsquigarrow \langle v_1, v_2 \rangle$	p_j		$\sigma \rightsquigarrow v_j$
`Map(`F`,` L`)`	$\sigma \rightsquigarrow ? \sqsupseteq \vec{\ell}$ as a prefix	F	$L = \vec{v}$	$\sigma \rightsquigarrow f$ s.t. $\bigwedge_{i=1}^{\lvert\vec{\ell}\rvert} f(v_i) = \ell_i$
$\lambda\, x \Rightarrow b$	$\sigma \rightsquigarrow f$ s.t. $f(v) = y$	b		$\sigma[x := v] \rightsquigarrow y$
`Filter(`P`,` L`)`	$\sigma \rightsquigarrow ? \sqsupseteq \vec{\ell}$	L		$\sigma \rightsquigarrow ? \sqsupseteq \vec{\ell}$
		P	$L = \vec{v}$	$\sigma \rightsquigarrow f$ s.t. $\bigwedge_{i=1}^{\lvert\vec{v}\rvert} f(\ell_i) = \left[v_i \in \vec{\ell}\right]$
`FilterInt(`i`,` k`,` L`)`	$\sigma \rightsquigarrow ? \sqsupseteq \vec{\ell}$	L		$\sigma \rightsquigarrow ? \sqsupseteq \vec{\ell}$
		i	$L = \vec{v}$	$\sigma \rightsquigarrow$ index of ℓ_1 in $\vec{v}$
		k	$L = \vec{v}$	$\sigma \rightsquigarrow \bigvee_{d \mid g} d$ where $g = \mathsf{gcd}(\Delta_1, \ldots, \Delta_{\lvert\vec{\ell}\rvert - 1})$, $\Delta_i = p_{i+1} - p_i$, p_j is an index of ℓ_j in $\vec{v}$

Table 5.2: Witness functions for various operators from the standard library of PROSE (`std`).

Example 6. Table 5.1 shows the witness functions for all $\mathscr{L}_{\mathsf{FF}}$ operators from Figure 3.1:

- A `Concat(`$atom$`,` $transform$`)` expression returns w iff $atom$ returns some prefix of w. In addition, assuming that $atom$ returns v, $transform$ mush return the remaining suffix of w after the end of v.
- A `ConstStr(`s`)` expression returns w iff s is equal to w.
- An expression "`let` x `=` `std.Kth(`$inputs$`,` k`)` `in` ..." returns w iff x is bound to an element of $inputs$ that has w as a substring.
- A `Substring(`x`,` pp`)` expression returns w (assuming that x returns v) iff pp returns a position span of any occurrence of w in v as a substring.
- An `AbsPos(`x`,` k`)` expression returns c (assuming 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 `RegexPos(`x`,` rr`,` k`)` returns c (assuming that x returns v) iff rr is equal to any pair of regular expressions that matches the boundaries of position c in the string v. In addition, assuming 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.

Example 7. Table 5.2 shows the witness functions for various operators from the standard library of PROSE that are used in $\mathscr{L}_{\mathsf{FF}}$ and $\mathscr{L}_{\mathsf{FE}}$:

- A `Kth(`xs`,` k`)` expression returns w (given that xs returns $\vec{v}$) iff k is an index of some occurrence of w in $\vec{v}$.
- If `Pair(`p_1`,` p_2`)` returns $\langle v_1, v_2 \rangle$, then p_j returns v_j. Note that this witness function is imprecise, since it restricts only a single parameter (p_1 or p_2).
- A `Map(`F`,` L`)` expression returns a list that starts with a sublist $\vec{\ell}$ as a prefix (given that L returns $\vec{v}$) iff F is any λ-function that maps the first $|\vec{\ell}|$ elements of $\vec{v}$ to the corresponding elements of $\vec{\ell}$. The fact that L returns $\vec{v}$ is usually established thanks to a corresponding domain-specific witness function for L (defined by a DSL designer for a particular instantiation of `Map`).

---

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

Operator	Output spec
`LinesMap`	$\sigma \rightsquigarrow ? \sqsupseteq \left[\ell_1, \ldots, \ell_{\lvert\vec{v}\rvert}\right]$ where ℓ_i is a line of the input document $\sigma[doc]$ that contains the region v_i
`StartSeqMap`	$\sigma \rightsquigarrow ? \sqsupseteq \left[p_1 \mid \langle p_1, p_2\rangle \in \vec{v}\right]$
`EndSeqMap`	$\sigma \rightsquigarrow ? \sqsupseteq \left[p_2 \mid \langle p_1, p_2\rangle \in \vec{v}\right]$

Table 5.3: Witness functions $\omega_L(\varphi) \mapsto \varphi'$ for the list parameter L of various FlashExtract instantiations of the `Map` $(F,\ L)$ operator, where φ is a *prefix spec* $\sigma \rightsquigarrow ? \sqsupseteq \vec{v}$.

- A `Filter(`P`,` L`)` expression returns a list that contains $\vec{\ell}$ as a sublist iff the result of L also contains $\vec{\ell}$ as a sublist. In addition, assuming that L returns a list $\vec{v}$, P must be any λ-function that maps all the elements of $\vec{v}$ that are also present in $\vec{\ell}$ to `true`, and the rest of the elements to `false`.
- A `FilterInt(`i`,` k`,` L`)` expression returns a list that contains $\vec{\ell}$ as a sublist iff the result of L also contains $\vec{\ell}$ as a sublist. In addition, assuming that L returns a list $\vec{v}$, the initial value expression i must evaluate to the index of ℓ_1 in $\vec{v}$, and the step expression k must evaluate to some divisor of the GCD of the gaps between the consecutive occurrences of the elements of $\vec{\ell}$ in $\vec{v}$.

Example 8. As mentioned above, a witness function for the parameter L of `Map(`F`,` L`)` must be defined specifically for every instantiation of `Map` in a given DSL. For instance, the FlashExtract DSL $\mathcal{L}_{\mathsf{FE}}$ contains three `Map` instantiations: `LinesMap`, `StartSeqMap`, and `EndSeqMap` (as defined in Figure 3.1). Their respective witness functions ω_L for the same *prefix spec* $\sigma \rightsquigarrow ? \sqsupseteq \vec{v}$ are shown in Table 5.3.

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 most operators in Table 5.2 (namely,

all except Map) do not depend on the domain of their parameters, and are therefore formulated generically, for any DSL. Witness functions in Table 5.1 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. This property allows us to define witness functions as generally as possible in order to reuse the corresponding operators in any conformant DSL.

5.4 Deductive Search

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 entire algorithm.

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

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

Proof.

1. By definition of $\mathsf{Filter}(\mathcal{S}, \varphi)$.
2. All ω_j deduce specs for N_j given only the outer spec φ, therefore they are independent from each other. Also, all ω_j are precise, therefore each $\mathcal{S}_j$ individually is necessary and sufficient to satisfy φ. □

Theorem 4 gives a straightforward recipe for synthesis of operators with independent parameters, such as $\texttt{Pair}(p_1, p_2)$. However, in most real-life cases operator parameters are dependent on each other. Consider an operator $\texttt{Concat}(atom, transform)$ from FlashFill, and a spec $\sigma \rightsquigarrow s$. It is possible to design individual witness functions ω_a and ω_t that return a disjunction φ_a of prefixes of s and a disjunction φ_t 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 $\mathcal{S}_a \models \varphi_a$ and $\mathcal{S}_t \models \varphi_t$ into a valid program set for φ.

In order to enable deductive search for dependent operator parameters, we apply *skolemization* [18]. Instead of deducing specs φ_a and φ_t that independently entail φ, we deduce only one independent spec (say, φ_a), and then *fix the value of "atom"*. For each fixed value of $atom$ a *conditional witness function* $\omega_t(\varphi \mid atom = v)$ deduces a spec $\varphi_{t,v}$ that is a necessary and sufficient characterization for φ. Namely, $\varphi_{t,v}$ in our example is $\sigma \rightsquigarrow s[|v|..]$ (i.e. the remaining suffix) if v is a prefix of s, or $\bot$ otherwise.

Skolemization splits the deduction into multiple independent branches, one per each value of $atom$. These values are determined by VSA clustering: $\mathcal{S}_a/_\sigma = \{v_1 \mapsto \mathcal{S}_a^1, \ldots, v_k \mapsto \mathcal{S}_a^k\}$. Within each branch, the program sets $\mathcal{S}_a^j$ and the corresponding $\mathcal{S}_t^j \models \varphi_{t,v_j}$ are independent, hence $\mathsf{Concat}_\bowtie(\mathcal{S}_a^j, \mathcal{S}_t^j) \models \varphi$ by Theorem 4. The union of k branch results constitutes a comprehensive set of 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 $w_j(\varphi \mid N_i = v)$. One simple example of such phenomenon is `Concat(`$atom$`,` $transform$`)`: we showed above how to decompose its inverse semantics into a witness function for prefixes and a conditional witness function for the suffix, but a symmetrical decomposition into a witness function for suffixes and a conditional witness function for prefixes is also possible.

Theorem 5. *Let $N := F(N_1, \ldots, N_k)$ be a rule in a DSL $\mathcal{L}$, and φ be a spec on N. If there exists an acyclic spanning subgraph of $G(F)$ that includes each node with all its prerequisite edges, then*

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 $\psi \leftarrow \mathsf{TopologicalSort}(G(F))$
2: $\mathcal{S} \leftarrow \bigcup \{\mathcal{S}' \mid \mathcal{S}' \in \textsc{LearnPaths}(G(F), \varphi, \psi, 1, \varnothing)\}$
3: **if** all witness functions in $G(F)$ are precise **then**
4: **return** $\mathcal{S}$
5: **else**
6: **return** $\mathsf{Filter}(\mathcal{S}, \varphi)$

Input ψ: permutation of the parameters of F
Input i: index of a current deduced parameter in ψ
Input Q: a mapping of prerequisite values $\vec{v}_k$ and corresponding learnt program sets $\mathcal{S}_k$ on the current path
function LearnPaths($G(F)$, φ, ψ, i, Q)
7: **if** $i > k$ **then**
8: Let $\mathcal{S}_1, \ldots, \mathcal{S}_k$ be learnt program sets for $N_1, \ldots, N_k$ in Q
9: **return** $\{F_{\bowtie}(\mathcal{S}_1, \ldots, \mathcal{S}_k)\}$
10: $p \leftarrow \psi_i$ *// Current iteration deduces the rule parameter N_p*
11: Let $\omega_p(\varphi \mid N_{k_1} = \vec{v}_1, \ldots, N_{k_m} = \vec{v}_m)$ be the witness function for N_p
// Extract the prerequisite values for N_p from the mapping Q
12: $\{\vec{v}_{k_1} \mapsto \mathcal{S}_{k_1}, \ldots, \vec{v}_{k_m} \mapsto \mathcal{S}_{k_m}\} \leftarrow Q[k_1, \ldots, k_m]$
// Deduce the spec for N_p given φ and the prerequisites
13: $\varphi_p \leftarrow \omega_p(\varphi \mid N_{k_1} = \vec{v}_{k_1}, \ldots, N_{k_m} = \vec{v}_{k_m})$
14: **if** $\omega_p = \bot$ **then return** $\varnothing$
// Recursively learn a valid program set $\mathcal{S}_p \models \varphi_p$
15: $\mathcal{S}_p \leftarrow \mathsf{Learn}(N_p, \varphi_p)$
// If no other parameters depend on N_p, proceed without clustering
16: **if** N_p is a leaf in $G(F)$ **then**
17: $Q' \leftarrow Q[p := \top \mapsto \mathcal{S}_p]$
18: **return** $\textsc{LearnPaths}(G(F), \varphi, \psi, i+1, Q')$
// Otherwise cluster $\mathcal{S}_p$ on $\vec{\sigma}$ and unite the results across branches
19: **else**
20: $\vec{\sigma} \leftarrow$ the input states associated with φ
21: **for all** $(\vec{v}'_j \mapsto \mathcal{S}'_{s,j}) \in \mathcal{S}_p/_{\vec{\sigma}}$ **do**
22: $Q' \leftarrow Q\left[s := \vec{v}'_j \mapsto \mathcal{S}'_{s,j}\right]$
23: **yield return all** $\textsc{LearnPaths}(G(F), \varphi, \psi, i+1, Q')$

Figure 5.2: 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 $G(F)$.

there exists a polynomial procedure that constructs a valid program set $\mathcal{S} \models \varphi$ from the valid parameter program sets $\mathcal{S}_j \models \varphi_j$ for some choice of parameter specifications φ_j.

Proof. We define the learning procedure for F in Figure 5.2 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 $\mathcal{S}_j$. In the prerequisite path, we maintain the invariant: *for each program set $\mathcal{S}_j$ in the path, all programs in it produce the same values $\vec{v}_j$ on the provided input states $\vec{\sigma}$*. This allows each conditional witness function ω_i to deduce a spec φ_i for the current parameter N_i assuming the bound values $\vec{v}_{k_1}, \ldots, \vec{v}_{k_s}$ 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 5.2 computes a valid program set $\mathcal{S} \models \varphi$. □

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 (defined in Chapter 4) filters out inconsistent programs from $\mathcal{S}$ in time proportional to $\mathcal{S}/_{\sigma}$.

Example 9. Figure 5.3 shows an illustrative diagram for the outermost layer of the learning process for the *transform* symbol from $\mathscr{L}_{\mathsf{FF}}$ and a given spec φ = "`(202) 555-0126`" $\rightsquigarrow$ "`202`". First, the framework splits the search process into two branches: one for `Concat(`*atom*`,` *transform*`)` and one for *atom*, according to the definition of *transform* in $\mathscr{L}_{\mathsf{FF}}$. The figure shows the outermost layer of the first branch.

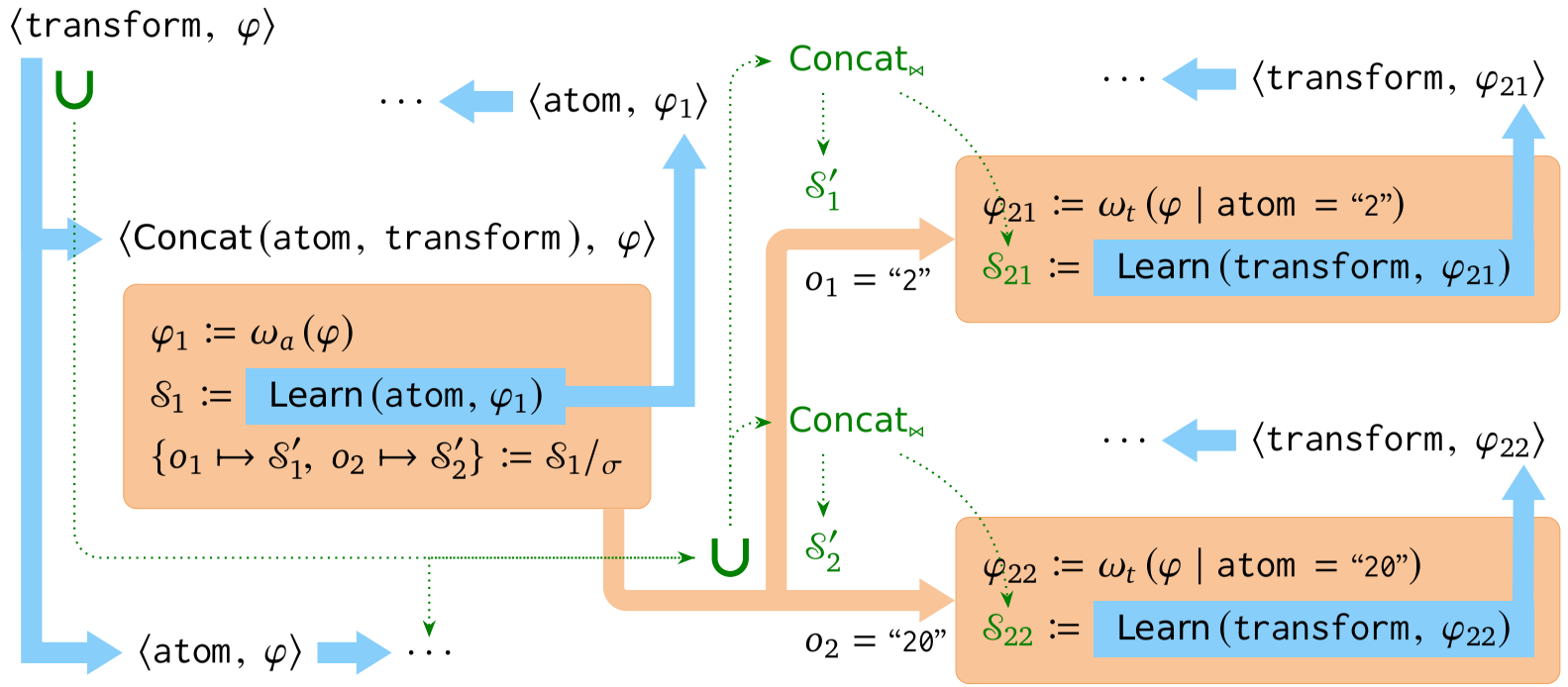


Figure 5.3: An illustrative diagram showing the outermost layer of deductive search for a given problem Learn(transform, "(202) 555-0126" ⇝ "202"). Solid blue arrows show the recursive calls of the search process (the search subtrees below the outermost layers are not expanded and shown as "..."). Rounded orange blocks and arrows shows the nested iterations of the LearnPaths procedure from Figure 5.2. Dotted green arrows show the VSA structure that is returned to the caller.

The LearnPaths procedure first invokes the witness function ω_a for the first parameter *atom* of the `Concat` rule. It returns a necessary spec $\varphi_1 = \sigma \rightsquigarrow$ "2" $\vee$ "20" for the *atom* symbol (where σ is the shared input state from φ). The PROSE framework then recursively resolves this spec into a VSA $\mathcal{S}_1$ of all *atom* programs that satisfy φ_1.

Next, the LearnPaths procedure clusters $\mathcal{S}_1$ and splits its further execution into two branches, one per each cluster of programs in $\mathcal{S}_1$ that give the same output on the given input state σ. There are two clusters: programs $\mathcal{S}'_1$ that return o_1 = "2" and programs $\mathcal{S}'_2$ that return o_2 = "20". For each of them the PROSE framework independently invokes a nested call to LearnPaths with the corresponding output binding for *atom* (i.e. o_1 or o_2 respectively) recorded in the prerequisite path Q. Within each nested invocation, LearnPaths constructs a necessary and sufficient spec on the second parameter *transform* of `Concat` by invoking its conditional witness function ω_t. It returns a spec φ_{21}: $\sigma \rightsquigarrow$ "02" for the branch with o_1 = "2" and a spec φ_{22}: $\sigma \rightsquigarrow$ "2" for the branch with o_2 = "20", respectively. The framework then recursively resolves them into the corresponding program sets $\mathcal{S}_{21}$ and $\mathcal{S}_{22}$.

Since both witness functions ω_a and ω_t are precise, the final result returned from LearnPaths is the VSA $\bigcup \left\{\mathsf{Concat}_{\bowtie}\left(\mathcal{S}'_1,\ \mathcal{S}_{21}\right),\ \mathsf{Concat}_{\bowtie}\left(\mathcal{S}'_2,\ \mathcal{S}_{22}\right)\right\}$.

Handling Boolean Connectives Witness functions for DSL operators (such as the ones in Table 5.1) are typically defined on the *atomic* constraints (such as equality or subsequence predicates). To complete the definition of deductive search, Figure 5.4a gives inference rules for handling of boolean connectives in a spec φ. Since a spec is defined as a NNF, we give the rules for handling conjunctions and disjunctions of specs, and positive/negative literals. These rules directly map to corresponding VSA operations:

Theorem 6.

(1) $\mathcal{S}_1 \models \varphi_1$ *and* $\mathcal{S}_2 \models \varphi_2 \iff \mathcal{S}_1 \cup \mathcal{S}_2 \models \varphi_1 \vee \varphi_2$.

(2) $\mathcal{S}_1 \models \varphi_1$ *and* $\mathcal{S}_2 \models \varphi_2 \iff \mathcal{S}_1 \cap \mathcal{S}_2 \models \varphi_1 \wedge \varphi_2$.

(3) $\mathcal{S} \models \varphi_1 \iff \mathsf{Filter}(\mathcal{S}, \varphi_2) \models \varphi_1 \wedge \varphi_2$.

(a)

$$\frac{N := F_1(N_1, \ldots, N_k) \mid F_2(M_1, \ldots, M_n) \quad \textsc{LearnRule}(G(F_1), \varphi) = \mathcal{S}_1 \quad \textsc{LearnRule}(G(F_2), \varphi) = \mathcal{S}_2}{\mathsf{Learn}(N, \varphi) = \mathcal{S}_1 \cup \mathcal{S}_2} \qquad \frac{}{\mathsf{Learn}(N, \top) = \mathscr{L}|_N}$$

$$\frac{\forall j = 1..2 \colon \mathsf{Learn}(N, \varphi_j) = \mathcal{S}_j}{\mathsf{Learn}(N, \varphi_1 \wedge \varphi_2) = \mathcal{S}_1 \cap \mathcal{S}_2} \qquad \frac{\forall j = 1..2 \colon \mathsf{Learn}(N, \varphi_j) = \mathcal{S}_j}{\mathsf{Learn}(N, \varphi_1 \vee \varphi_2) = \mathcal{S}_1 \cup \mathcal{S}_2} \qquad \frac{}{\mathsf{Learn}(N, \varphi) = \mathsf{Filter}(\mathscr{L}|_N, \varphi)}$$

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

(b)

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

$$\frac{N \text{ is a variable}}{\mathsf{Learn}(N, \sigma \rightsquigarrow \psi) = \{N\} \text{ if } \psi(\sigma[N]) \text{ else } \emptyset}$$

Figure 5.4: **(a)** Constructive inference rules for processing of boolean connectives in the inductive specifications φ; **(b)** Witness functions and inference rules for common syntactic features of PROSE DSLs: `let` definitions, variables, and literals.

Handling negative literals is more difficult. They can only be efficiently resolved in two cases: (a) if a witness function supports the negated spec 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 programs satisfying a negative literal is bounded only by the DSL.

Search Tactics Theorems 5 and 6 and Fig. 5.4 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 $\mathcal{S}_1$ and $\mathcal{S}_2$, 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. $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$), each conjunct individually may be satisfiable. In this case the unsatisfiability of the original problem is determined only after $T(\mathsf{Learn}(N, \varphi_1)) + T(\mathsf{Learn}(N, \varphi_2)) + \mathcal{O}(V(\mathcal{S}_1) \cdot V(\mathcal{S}_2))$ time.

An alternative search tactic for conjunctive specs 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. PROSE applies witness functions to each conjunct in the specification in parallel, conjuncts the resulting parameter specs, and makes a single recursive learning call. Such "parallel processing" of conjuncts in the spec 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 PROSE performs many practical optimizations in the algorithm in Figure 5.2. We parallelize the loop in Line 21, since it explores non-intersecting portions of the program space. For ranked synthesis, we only calculate top k programs for leaf nodes of $G(F)$, provided

[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.

the ranking function is monotonic. We also cache synthesis results for every distinct learning subproblem $\langle N, \varphi \rangle$, which makes deductive search an instance of *dynamic programming*. This optimization is crucial for efficient synthesis of many common DSL operators, as we explain in more details in Section 5.5.1.

For bottom portions of the DSL we switch to enumerative search [57], 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 PROSE can be solved by any sound strategy, not necessarily deduction or enumerative search. Possible alternatives include constraint solving or stochastic techniques [1].

5.5 Evaluation

Our evaluation of PROSE aims to answer two classes of questions: its *applicability* and its *performance*. Applicability questions concern (a) our generalization of prior work in PBE in terms of inductive specifications and witness functions; (b) generality of our library of witness functions; (c) engineering usability of PROSE. Performance questions concern the running time of synthesizers generated by PROSE, and the comparison of PROSE to general-purpose non-inductive synthesizers, such as SyGuS [1].

5.5.1 Case Studies

Table 5.4 summarizes our case studies: the prior works in inductive synthesis over numerous different applications that we studied for evaluation of PROSE. Of the 15 inductive synthesis tools we studied, 12 can be cast as a special case of the deductive search algorithm methodology, which we verified by manually formulating corresponding witness functions 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 deductive search: namely, it is enumerative search [25, 57] or constraint solving [35].

Project	Domain	Ded.	Impl.	ψ	φ'
Gulwani [14]	String transformation	✓	✓	=	=
Le and Gulwani [33]	Text extraction			⊐	⊐
Kini and Gulwani [26]	Text normalization			=	soft
Barowy et al. [4]	Table normalization			=	=
Singh and Gulwani [52]	Number transformation			=	=
Singh and Gulwani [53]	Semantic text editing	✓	✗	=	=
Harris and Gulwani [16]	Table transformation			=	=
Andersen et al. [2]	Algebra education			trace	=
Lau et al. [32]	Editor scripting			trace	=
Feser et al. [9]	ADT transformation			=	=
Osera and Zdancewic [40]	ADT transformation			=	=
Yessenov et al. [61]	Editor scripting			=	=
Udupa et al. [57]	Concurrent protocols	✗	✗	trace	N/A
Katayama [25]	Haskell programs			=	N/A
Lu and Bodik [35]	Relational queries			=	N/A
Raza and Gulwani [46]	Splitting of text into columns	✓	✓	=	=
Rolim et al. [47]	Software refactoring			=	=
Gorinova et al. [11]	Reshaping of healthcare data			=	≈
Transformation.JSON	Transformation of JSON trees			=	⊐
Extraction.JSON	Extraction of data from JSON files			⊐	⊐
Matching.Text	Data profiling/clustering			—	=
Extraction.Web	Web data extraction			⊐	⊐

Table 5.4: Case studies of PROSE: prior works in inductive program synthesis. “Ded.” means “Is it an instance of the deductive methodology?”, “Impl.” means “Have we (re-)implemented it on top of PROSE?”, ψ is a top-level constraint kind, φ' lists notable intermediate constraint kinds (for the deductive techniques only). The bottommost section shows the new projects implemented on top of PROSE since its creation.

Project	LOC		Development time	
	Original	PROSE	Original	PROSE
Gulwani [14]	12K	3K	9 months	1 month
Le and Gulwani [33]	7K	4K	8 months	1 month
Kini and Gulwani [26]	17K	2K	7 months	2 months
Barowy et al. [4]	5K	2K	8 months	1 month
Singh and Gulwani [52]	—	1K	—	2 months
Raza and Gulwani [46]	—	10K	—	2 months
Rolim et al. [47]		6K		3 months
Transformation.JSON		2K		1 month
Extraction.JSON		3K		1 month
Matching.Text		2K		3 months
Extraction.Web		2.5K		1.5 months

Table 5.5: Development data on the (re-)implemented projects. The cells marked with "—" either do not have an original implementation or we could not obtain historical data on them.

Our industrial collaborators reimplemented 5 existing systems and created 7 new ones since PROSE was created in 2015. We present data on these development efforts in Table 5.5.

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 projects implemented using deductive search.

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 (11/22 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 9/22 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. [9] 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 deductive search (see Chapter 2 for detailed comparison).

Q3: How usable is PROSE? Table 5.5 presents some development stats on the projects that were reimplemented. In all cases, PROSE-based implementations were shorter, cleaner, more stable and extensible. The reason is that with PROSE, our collaborators did not concern themselves with tricky details of synthesis algorithms, since they were implemented once and for all, as in Section 5.4. Instead, they focused only on domain-specific witness functions, for which design, implementation, and maintenance are much easier. Notably, in case of the FlashRelate [4] reimplementation and `Extraction.Web`, our collaborators did not have any experience in program synthesis.

The development time in Table 5.5 includes the time required for an implementation to mature (i.e. cover the required use cases), which required multiple experiments with DSLs. With PROSE, various improvements over DSLs were possible on a daily basis. PROSE also allowed our collaborators to discover optimizations not present in the original implementations. We share some anecdotes of PROSE 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

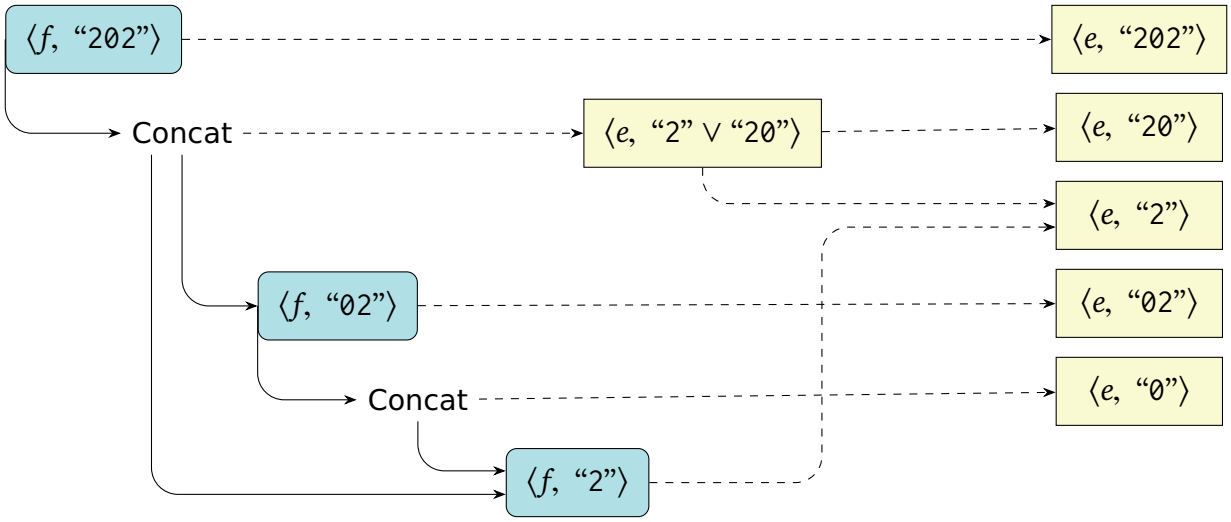


Figure 5.5: A DAG of recursive calls that arises during the deductive search process for a recursive binary operator $f := e \mid \mathsf{Concat}(e,\ f)$. As described in Scenario 1, it is isomorphic to an explicitly maintained DAG of substring programs in the original FlashFill implementation [14].

spent on describing and performing operations on such a DAG. In PROSE, 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 PROSE 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, as shown in Figure 5.5. This is *the same DAG* as in FlashFill – but with PROSE, 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. While such an extension in the original FlashFill implementation would consume a couple of weeks, in PROSE it took only a few minutes.

Scenario 3. A CSS selector is a function $\mathsf{Document} \rightarrow \mathsf{Set}\langle\mathsf{DOMNode}\rangle$. It is a path specification for a DOM node where each element in the path is a predicate on the corresponding ancestor (i.e. the ancestor's tag or its class), and each edge in the path descends to all children of the preceding element that satisfy a certain property [6].

A spec synthesis of CSS selectors is a subset of selected DOM nodes. Using an enumerative search for this problem induces an exponential blowup: it starts with an input state (an HTML document) and iteratively constructs all possible CSS selectors. Since they may select arbitrary subsets of the DOM tree, the resulting search is infeasible.

In contrast, a deductive approach starts with an *output* (a set of nodes), and deduces examples for the intermediate subexpressions (prefixes of the desired CSS selector). This process follows the DOM tree *upwards*, instead of *downwards*, and therefore is by construction finite. Moreover, the number of deduction steps is bounded by the tree depth.

5.5.2 Experiments

Performance & Number of examples Figure 5.6 shows performance and the number of examples of our FlashFill and FlashExtract reimplementations on top of the PROSE framework. The overall performance is comparable to that of the original system, even though the implementations differ drastically. For example, the runtime of the original implementation of FlashExtract varies from 0.1 to 4 sec, with a median of 0.3 sec [33]. The new implementation (despite being more expressive and built on a general 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 PROSE-based implementation to be successfully used in industry instead of the original one.

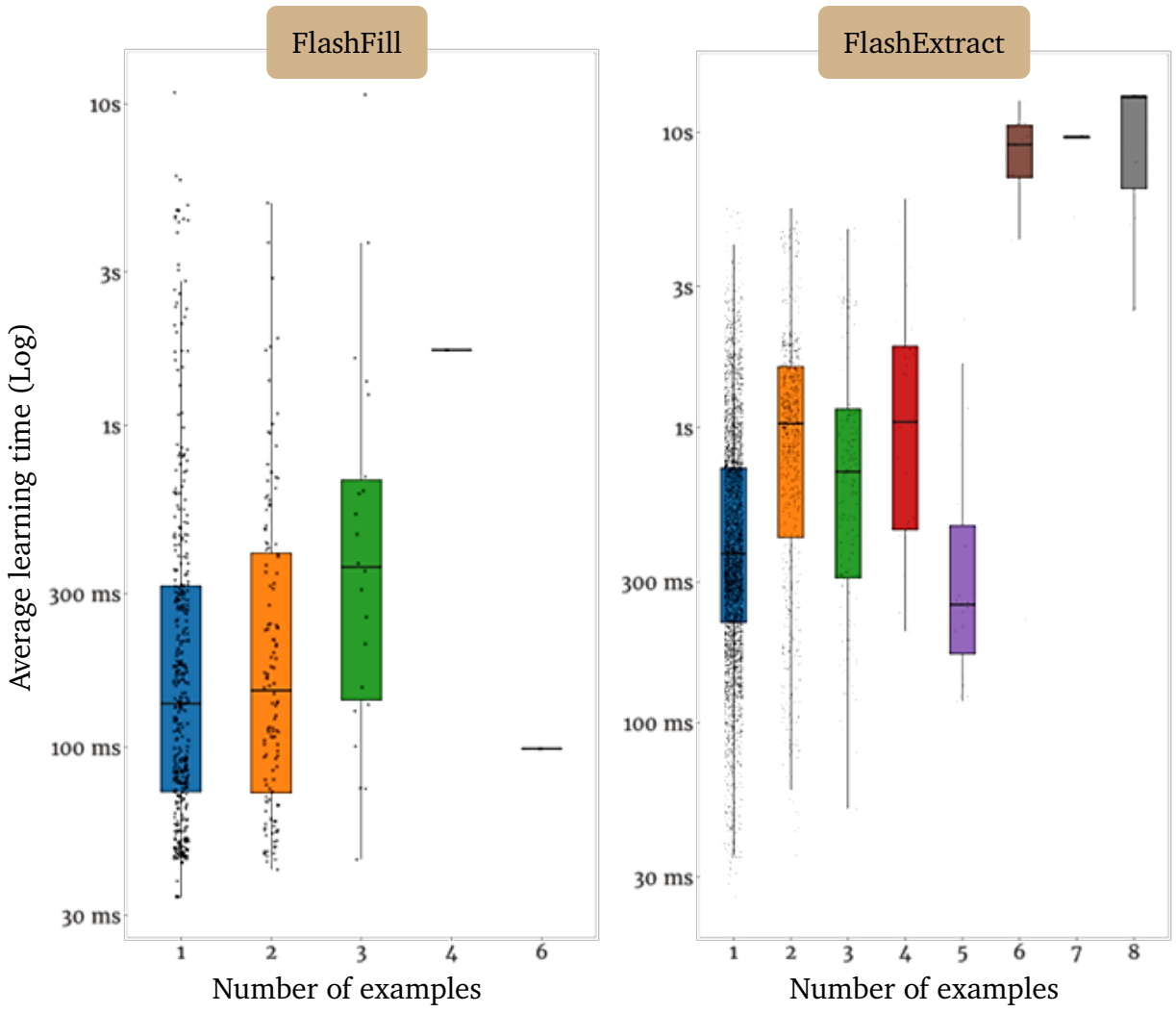


Figure 5.6: Performance evaluation of the reimplementations of FlashFill [14] and FlashExtract [33] on top of the PROSE framework. Each dot shows average learning time per iteration on a given scenario. The scenarios are clustered by the number of examples (iterations) required to complete the task. *Left:* 531 FlashFill scenarios. *Right:* 6464 FlashExtract scenarios.

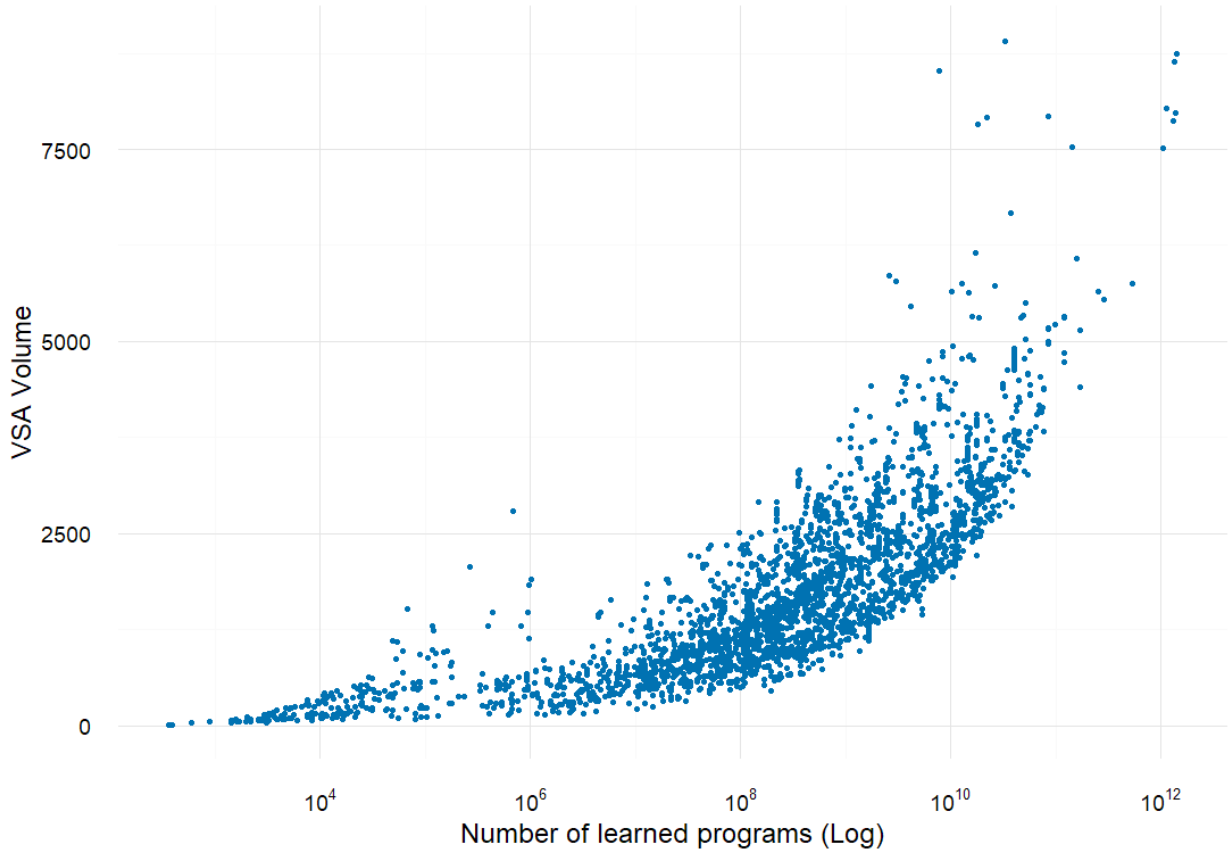


Figure 5.7: The relationship between VSA volume and VSA size (i.e. number of programs) for the complete VSAs learned to solve 6464 real-life FlashExtract scenarios.

VSA Volume There is no good theoretical bound on the time of VSA clustering (the most time-consuming operation in the deductive search). 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 5.7 shows, even for large inputs it never exceeds 8000 nodes (thus explaining efficient runtime), whereas VSA size (i.e. number of learned programs) may approach 10^{13}.

5.6 Strengths and Limitations

The methodology of deductive search that lies at the core of PROSE 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.

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.

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.

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

- The programs may contain domain-specific constants.
- The DSL contains arbitrary executable operators that manipulate domain-specific objects with rich semantics.
- 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 and maintenance cost of a PBE-based tool is limited by industrial budget and available developers.

Chapter 6

INTERACTIVE PROGRAM SYNTHESIS

The key challenge of PBE and, more generally, programming with under-specifications, is *intent ambiguity*. Examples are an inherently ambiguous form of specification. A typical industrial DSL usually may contain up to 10^{20} programs that are consistent with a given input-output example [52]. The underlying PBE system might end up synthesizing an unintended program that is consistent with the examples provided by the user but does not generate the intended results on some other inputs that the user cares about. In 2008, Lau presented a critical discussion of PBE systems noting that adoption of PBE systems is not yet widespread, and proposing that this is mainly due to lack of usability and confidence in such systems [31]. We have observed the issues of confidence, transparency, and debuggability to arise in most interactions with PBE-based synthesis systems.

In the past, intent ambiguity in PBE has been primarily handled by imposing a sophisticated ranking on the DSL [54]. While ranking goes a long way in avoiding undesirable interpretations of the user's intent, it is not a complete solution. For example, FlashFill is designed to cater to users that care not about the program but about its behavior on the small number of input rows in the spreadsheet. Such users can simply eye-ball the outputs of the synthesized program and provide another example if they are incorrect. However, this becomes much more cumbersome (or impossible) with a larger spreadsheet.[1] Moreover, inspecting the synthesized program directly also does not establish enough confidence in it even if the user knows programming.

[1] John Walkenbach, famous for his Excel textbooks, labeled FlashFill as a "controversial" feature. He wrote: *"It's a great concept, but it can also lead to lots of bad data. I think many users will look at a few "flash filled" cells, and just assume that it worked. But my preliminary tests leads me to this conclusion: Be very careful."* [59]

Two main reasons for this are (i) program readability,[2] and (ii) the users' uncertainty in the desired intent due to hypothetical unseen corner cases in the data.

Due to ambiguity of intent in PBE, the standard user interaction model in this setting is for the user to provide constraints *iteratively* until the user is satisfied with the synthesized program or its behavior on the known inputs. However, most work in this area, including FlashFill and FlashExtract, has not been formally modeled as an iterative process. In this work, I describe our interactive formulation of program synthesis that leverages the inherent iterative nature of synthesis from under-specifications, as well as multiple novel techniques for intent disambiguation and user interaction models that leverage the innate strengths of this interactive formulation.

In Section 6.1, I formally introduce the problem of *interactive program synthesis*, wherein an application-specific *synthesis session state* is kept after each iteration. This state is used to improve the transparency, performance, and debuggability of the synthesis process via various means, including proactive feedback to the user, directing the user toward better examples, and speeding up the next synthesis iterations incrementally.

In Section 6.2, I describe our exploration and evaluation of user interaction models that aim to improve the transparency of the iterative synthesis process. Our proposed two novel interaction models alleviate above-mentioned transparency concerns by exposing more information to the user in a form that can be easily understood and acted upon. These models help resolve ambiguity in the example-based specification, thereby increasing user's trust in the results produced by the PBE engine. The first model, *Program Navigation*, allows the user to navigate between all programs synthesized by the underlying PBE engine (as opposed to displaying only the top-ranked program) and to pick one that is intended. We leverage the implicit sharing present in a VSA (Chapter 4) to create a navigational interface that allows the user to select from different ranked choices for various parts of the top-ranked program. Furthermore, these programs are paraphrased in English for easy readability. The second

[2] Stephen Owen, a certified MVP ("Most Valued Professional") in Microsoft technologies, said the following of a program synthesized by FlashExtract: *"If you can understand this, you're a better person than I am."* [41]

model, *Conversational Clarification*, is based on active learning. In it, the system asks questions to the user to resolve ambiguities in the user's specification with respect to the available test data. These questions are generated after the PBE engine has synthesized multiple programs that are consistent with the user-provided examples. The system executes these multiple programs on the test data to identify any discrepancies in the execution and uses that as the basis for asking questions to the user. The user responses are used to refine the initial example-based specification and the process of program synthesis is repeated.

Section 6.3 builds on the ideas of Conversational Clarification, generalizing it into a universal formal technique that is applicable to an arbitrary interactive synthesis process – *feedback-based synthesis*. It builds on our interactive problem definition from Section 6.1. I present a method for leading the user toward the examples with maximum disambiguation potential, and discuss our usage of feedback-based synthesis in PROSE-based applications.

Finally, in Section 6.4, I show how the same problem formulation can be used to improve the performance of the synthesis process. The standard PBE model requires the user to refine her intent in iterative rounds by providing additional constraints on the current candidate program. The standard approach has been to re-run the synthesizer afresh with the conjunction of the original constraints and the new constraints. In this work, I introduce an alternative technique: leveraging the *interpretation of previously learned VSA as a sub-language (Section 4.4)* to carry out the new round of synthesis on a smaller program space.

6.1 Problem Definition

In this section, I extend the conventional program synthesis problem definition (Problem 1) to incorporate learner-user interaction. It allows us to model the inherent interactive workflow in a first-class manner in the program synthesis formalism and associated techniques.

Problem 2 (Interactive Program Synthesis)**.** Let $\mathscr{L}$ be a DSL, and N be a symbol in $\mathscr{L}$. Let $\mathscr{A}$ be an *inductive synthesis algorithm* for $\mathscr{L}$, which solves problems of type $\mathsf{Learn}(N, \varphi)$ where φ is an *inductive spec* on a program rooted at N. The specs φ are chosen from a fixed class of

supported spec types Φ. The result of $\mathsf{Learn}(N, \varphi)$ is some set $\mathcal{S}$ of programs rooted at N that are consistent with φ.

Let φ^* be a spec on the output symbol of $\mathcal{L}$, called a *task spec*. A **φ^*-driven interactive program synthesis process** is a finite series of 4-tuples $\langle N_0, \varphi_0, \mathcal{S}_0, \Sigma_0\rangle, \ldots, \langle N_m, \varphi_m, \mathcal{S}_m, \Sigma_m\rangle$, where

- Each N_i is a nonterminal in $\mathcal{L}$,
- Each φ_i is a spec on N_i,
- Each $\mathcal{S}_i$ is some set of programs rooted at N_i s.t. $\mathcal{S}_i \models \varphi_i$,
- Each Σ_i is an **interaction state**, explained below,

which satisfies the following axioms for any program $P \in \mathcal{L}$:

A. $(P \models \varphi^*) \Rightarrow (P \models \varphi_i)$ for any $0 \leq i \leq m$;

B. $(P \models \varphi_j) \Rightarrow (P \models \varphi_i)$ for any $0 \leq i < j \leq m$ s.t. $N_i = N_j$.

We say that the process is **converging** iff the top-ranked program of the last program set in the process satisfies the task spec: $P^* = \mathsf{Top}_h(\mathcal{S}_m, 1) \models \varphi^*$, and the process is **failing** iff the last program set is empty: $\mathcal{S}_m = \emptyset$.

An **interactive synthesis algorithm** $\widehat{\mathcal{A}}$ is a procedure (parameterized by $\mathcal{L}$, $\mathcal{A}$, and h) that solves the following problem:

$$\mathsf{LearnIter}: \begin{cases} \langle N_0, \varphi_0, \bot\rangle \mapsto \langle \mathcal{S}_0, \Sigma_0\rangle \\ \langle N_i, \varphi_i, \Sigma_{i-1}\rangle \mapsto \langle \mathcal{S}_i, \Sigma_i\rangle, \quad i > 0 \end{cases}$$

In other words, at each iteration i the algorithm receives the i^{th} learning task $\langle N_i, \varphi_i\rangle$ and its own interaction state Σ_{i-1} from the previous iteration. The type and content of Σ_i is unspecified and can be implemented by $\widehat{\mathcal{A}}$ arbitrarily.

Definition 14. *An interactive synthesis algorithm $\widehat{\mathcal{A}}$ is **complete** iff for any task spec φ^*:*

- *If $\exists P \in \mathcal{L}$ such that $P \models \varphi^*$ then $\widehat{\mathcal{A}}$ eventually converges for any φ^*-driven interactive synthesis process.*
- *Otherwise, $\widehat{\mathcal{A}}$ eventually fails for any φ^*-driven interactive synthesis process.*

The notion of an interactive synthesis process formally models a typical learner-user interaction where φ^* describes the desired program. The task spec φ^* is not known to the synthesizer – it is an "ideal spec" that would unambiguously specify the user intent. The general nature of definitions in Problem 2 allows many different implementations for $\widehat{\mathcal{A}}$. In addition to completeness, different implementations (and choices for the state Σ) strive to satisfy different *performance objectives*, such as:

- Number of interaction rounds (e.g. examples) m,
- The total amount of information communicated by the user,
- Cumulative execution time of all $m + 1$ learning calls.

In the rest of this chapter, I present several specific instantiations of interactive synthesis algorithms that optimize these objectives.

6.2 User Interaction Models

This section formally introduces *Program Navigation* and *Conversational Clarification* – our proposed user interaction models to improve the transparency of intent disambiguation process. We have built a prototype UI called PROSE Playground,[3] which incorporates both interaction models, and used it to conduct a user study. In the study, we asked participants to extract structured data from semi-structured text files using Playground. We observe that participants perform more correct extraction when they make use of the new interaction models.

To our surprise, participants preferred Conversational Clarification over Program Navigation slightly more even though past case studies suggested that users wanted to look at the synthesized programs. We believe this is explained by the fact that because Conversational Clarification is a *proactive* interface that asks clarifying questions, whereas Program Navigation is a *reactive* interface that expects an explicit correction of a mistake.

[3]Available at https://prose-playground.cloudapp.net.

6.2.1 User Interface

Figure 6.1 shows a Playground window after providing several examples (on the left), and after invoking the learning process (on the right). The Playground window consists of 3 sections: Top Toolbar (1), Input Text View (2), and PBE Interaction View (3).

The Input Text View is the main area. It gives users the ability to provide examples by highlighting desired sections of the document, producing a set of nested colored blocks. Additionally, users may omit the structure boundary and only provide examples for the fields as shown in Figure 6.1. After an automated learning phase, the output of the highest ranked program is displayed in the Output pane. Each new row in the output is also matched to the corresponding region of the original document that is highlighted with dimmer colors. The user can also provide *negative examples* by clicking on previously marked regions to communicate to the PBE system that the region should not be selected as part of the output.

The PBE Interaction View is a tabbed pane giving users an opportunity to interact with the PBE system in three different ways: (i) exploring the produced output, (ii) exploring the learned program set paraphrased into English in the program viewer (Program Navigation), and (iii) engaging in an active learning session through the "Disambiguation" feature (Conversational Clarification).[4]

The Output pane displays the current state of data extraction result either as a relational table or as a tree. To facilitate exploration of the data, the Input Text View is scrolled to the source position of each cell when the user hovers over it. The user can also mark incorrect table rows as negative examples.

The Program viewer pane (Figure 6.2) lets users explore the learned programs. We concisely describe regexes that are used to match strings in the input text. For instance, "`Words/dots/hyphens ∘ WhiteSpace`" represents `[-.\pLu\pLl]+o\pZs+` (viewable in code

[4] Note that throughout this chapter, I refer to the "disambiguation" as an overall problem of selecting the program that realizes user's intent in PBE. However, in our UI we use the word "Disambiguation" as a header of a pane with one iteration of the Conversational Clarification process. We found that it describes Conversational Clarification most lucidly to the users.

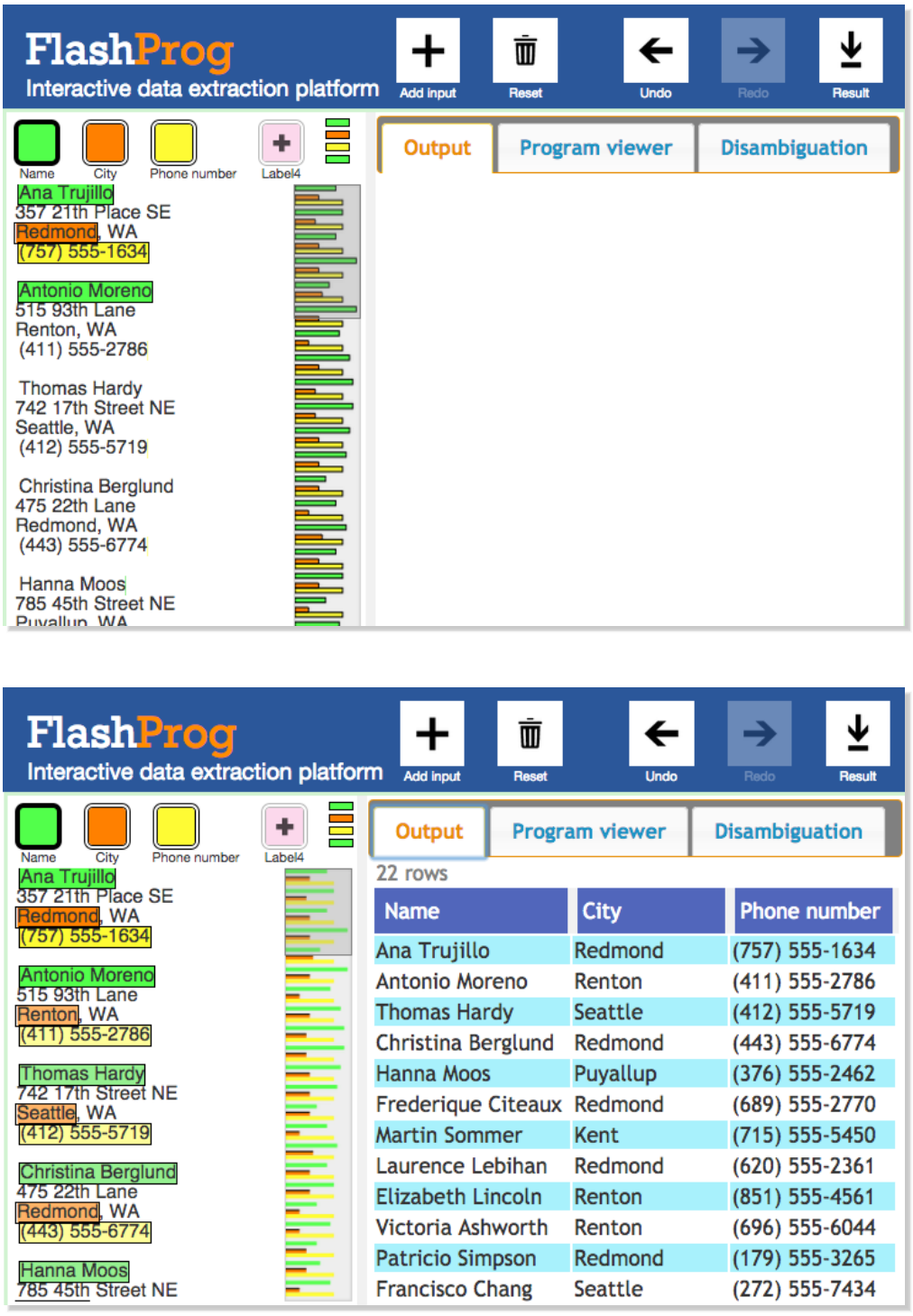


Figure 6.1: Playground UI with PBE Interaction View in the “Output” mode, before and after the learning process. 1 – Top Toolbar, 2 – Input Text View, 3 – PBE Interaction View.

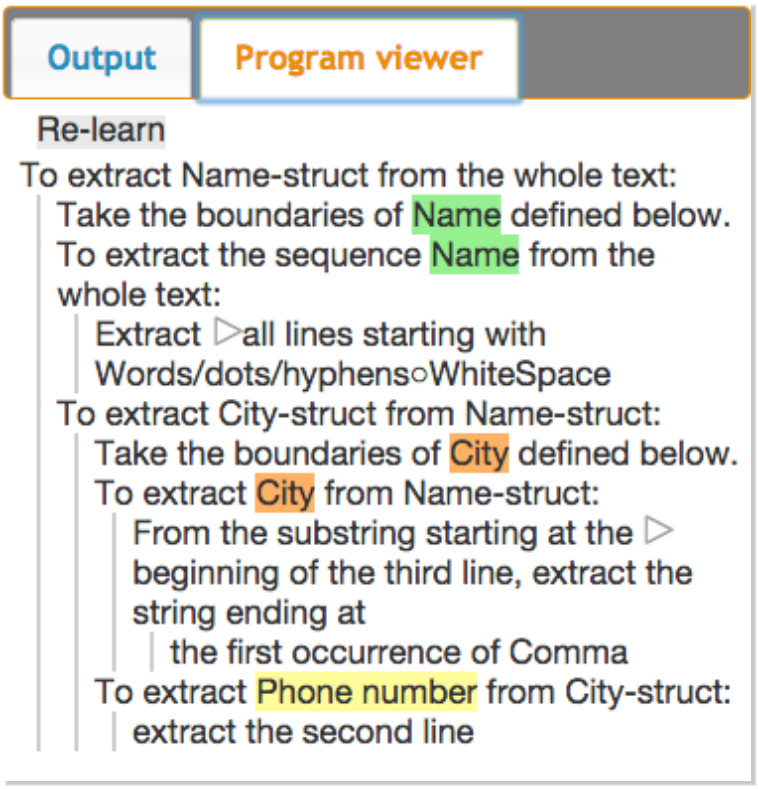


Figure 6.2: Program Viewer tab of the PROSE Playground. It shows the extraction programs that were learned in the session in Figure 6.1. They are paraphrased in English and indented.

mode). To facilitate understanding of these regexes, when the user hovers over part of a regex, our UI highlights matches of that part in the text. In Figure 6.2, `Name-Struct` refers to the region between two consecutive names; `City-Struct` refers to the region between `City` and the end of the enclosing `Name-Struct` region. Learned programs reference these regions to extract data. For instance, `Phone` is learnt relatively to enclosing `City-struct` region: "`second line`" refers to the line in the `City` region. In addition, clicking on the triangular marker opens a list of alternative suggestions for each subexpression. We show number of highlights that will be added (or changed/removed) by the alternative program as a +number (or a -number). If the program is incorrect, the user can replace some expressions with alternatives from the suggested list (Figure 6.3).

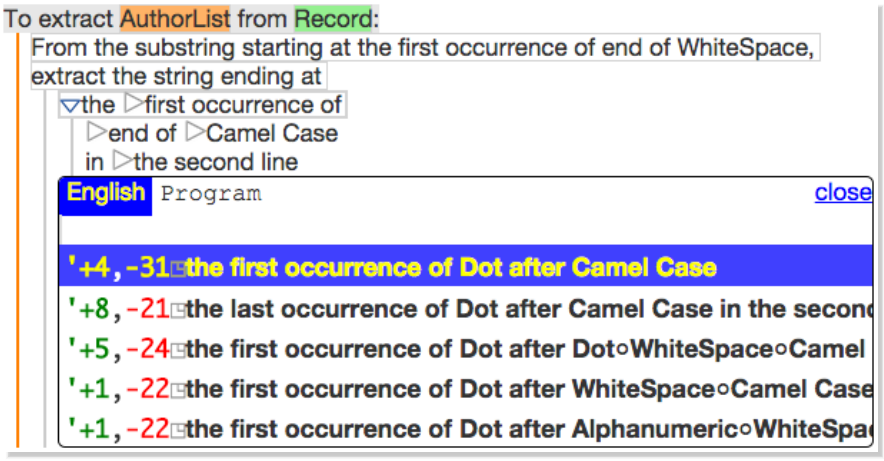


Figure 6.3: Program Viewer tab & alternative subexpressions.

The Disambiguation pane (Figure 6.4) presents the Conversational Clarification interaction model. The PBE engine often learns multiple programs that are consistent with the examples but produce different outputs on the rest of the document. In such cases, this difference is highlighted and user is presented with an option to choose between the two behaviors. Choosing one of the options is always equivalent to providing one more example (either positive or negative), thereby invoking the learning again on the extended specification.

6.2.2 Program Navigation

The two key challenges in Program Navigation are: paraphrasing of the DSL programs in English, and providing alternative suggestions for program expressions.

Templating language To enable paraphrasing, we implemented a high-level templating language, which maps *partial programs* into *partial English phrases*. Lau stated [31]:

> *"Users complained about arcane instructions such as "set the* `CharWeight` *to* `1`*" (make the text bold).* [. . .] *SMARTedit users also thought a higher-level description*

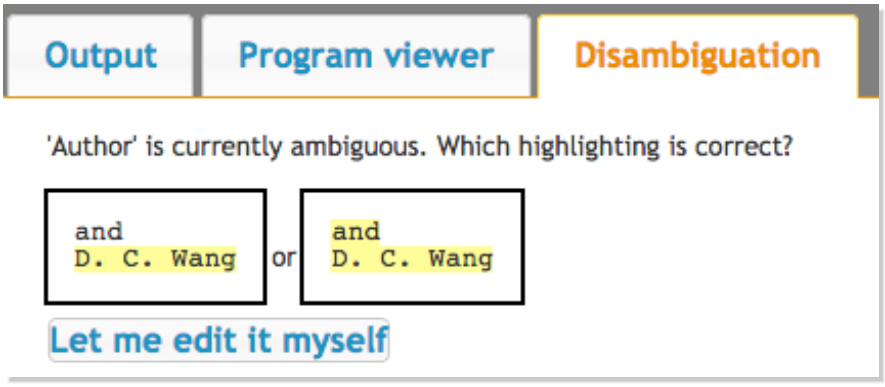


Figure 6.4: Conversational Clarification being used to disambiguate different programs that extract individual authors.

> *such as "delete all hyperlinks" would be more understandable than a series of lower level editing commands."*

Our template-based strategy for paraphrasing avoids arcane instructions by using *context-sensitive formatting rules*, and avoids low-level instructions by using *idioms*, solving the aforementioned two problems [37].

Program alternatives To enable alternatives, we record the original candidate program set for each subexpression in the chosen program. Since it is represented as a VSA, we can easily retrieve a subspace of alternatives for each program subexpression, and apply the domain-specific ranking function on them. The top 5 alternatives are presented to the user.

6.2.3 Conversational Clarification

Conversational Clarification selects examples based on different outputs produced by generated programs. Each synthesis step produces a VSA of ambiguous programs that are consistent with the given examples. Conversational Clarification iteratively replaces the subexpressions of the top-ranked program with its top k alternatives from the VSA. The clarifying question for the

user is based on the *first discrepancy* between the outputs of the currently selected program P and an alternative program P'. Such a discrepancy can have three possible manifestations:

- The outputs of P and P' match until P selects a region r, which does not intersect any selection of P'. This leads to the question "Should r be highlighted or not?"
- The outputs of P and P' match until P' selects a region r', which does not intersect any selection of P. This leads to the question "Should r' have been highlighted?"
- The outputs of P and P' match until P selects a region r, P' selects a different region r', and r intersects r'. This leads to the question "Should r or r' be highlighted?"

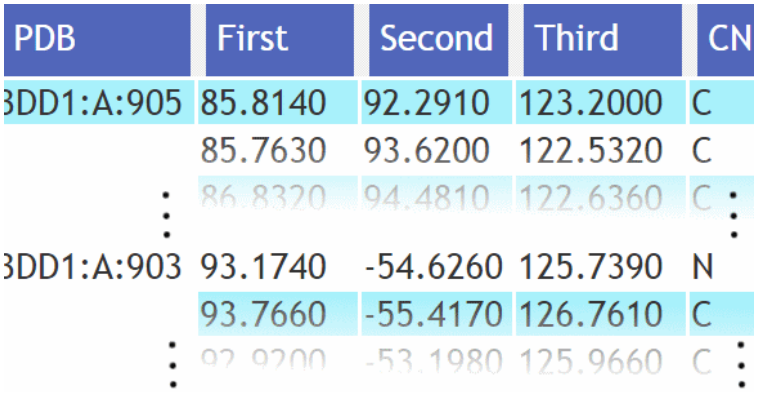
For better usability (and faster convergence), we merge the three question types into one, and ask the user "What should be highlighted: r_1, r_2, or nothing?" Selecting r_1 or r_2 would mark the selected region as a positive example. Selecting "nothing" would mark both r_1 and r_2 as negative examples. After selecting an option, we convert the choice into one or more examples, and invoke a new synthesis process.

6.2.4 Evaluation

To evaluate the usefulness of our novel interaction models for disambiguation, we conducted a user study in the data wrangling domain. In particular, we address three research questions for PBE:

- RQ1: Do Program Navigation and Conversational Clarification contribute to correctness?
- RQ2: Which of Program Navigation and Conversational Clarification is perceived more useful for data extraction?
- RQ3: Do our novel interaction models help alleviate typical distrust in PBE systems?

User study design Because typical data wrangling tasks can be solved without any programming skills, we performed a within-subject study over an heterogeneous population of 29 people: 4 women aged between 19 and 24 and 25 men aged between 19 and 34. Their programming experience ranged from none (a 32-year man doing extraction tasks several times a month), less than 5 years (8 people), less than 10 (9), less than 15 (8) to less than



PDB	First	Second	Third	CN
3DD1:A:905	85.8140	92.2910	123.2000	C
	85.7630	93.6200	122.5320	C
⋮	86.8320	94.4810	122.6360	C ⋮
3DD1:A:903	93.1740	-54.6260	125.7390	N
	93.7660	-55.4170	126.7610	C
⋮	92.9200	-53.1980	125.9660	C ⋮

Figure 6.5: Bioinformatic log: Result sample.

20 (3). They reported performing data extraction tasks never (4 people), several times a year (7), several times a month (11), several times a week (3) up to every day (2).

We selected 3 files containing several ambiguities these users have to find out and to resolve. We chose these files among anonymized files provided by our customers. Our choice was also motivated by repetitive tasks, where extraction programs are meant to be reused on other similar files. The three files are the following:

1. **Bank listing.** List of bank addresses and capital grouped by state. The postal code can be ambiguous.
2. **Amazon research.** The text of the search results on Amazon for the query "chair". The data is visually structured as a list of records, but contains spacing and noise.
3. **Bioinformatic log.** A log of numerical values obtained from five experiments, from bioinformatics research (Figures 6.5 and 6.6).

After a brief tutorial, we ask users to perform extraction on the three files. In order to measure the trust they give to the extraction, we do not tell them if their extraction is correct or not. For each extraction task, we provide a result sample (such as Figure 6.5). Users then manipulate Playground to generate the entire output table corresponding to that task.

To answer RQ1, we select a number of representative values across all fields for each task, and we automatically measure how many of them were incorrectly highlighted. These values

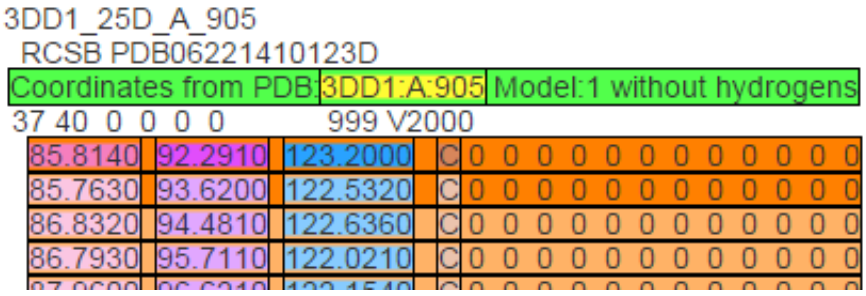


Figure 6.6: Highlighting in the Input Text View for obtaining Figure 6.5.

were selected by running Playground sessions in advance ourselves and observing insightful checkpoints that require attention. In total, we selected 6 values for task #1, 13 for task #2 and 12 for task #3. We do not notify users about their errors. This metric has more meaning than if we recorded all errors. As an illustration, a raw error measurement in the third task for a user forgetting about the third main record would yield more than 140 errors. Our approach returns 2 errors, one for the missing record, and one for another ambiguity that needed to be checked but could not. This makes error measurement comparable across tasks.

To measure the impact of Program Navigation and Conversational Clarification interaction models independently, we set up three interface environments:

Basic Interface (BI). This environment enables only the PBE interaction model. It includes the following UI features: the labeling interface for mouse-triggered highlighting, the label menu to rename labels, to switch between them and the Output tab.

BI + Program Navigation (BI + PN). Besides PBE, this interface enables the Program Navigation interaction model, which includes the Program Viewer tab and its features.

BI + Conversational Clarification (BI + CC). Besides PBE, this environment enables the Conversational Clarification interaction model, which includes the Disambiguation tab.

To compensate the learning curve effects when comparing the usefulness of various interaction models, we set up the environments in three configurations A, B, and C, shown in

	Tasks			
Configuration	**1. Bank**	**2. Amazon**	**3. Bio log**	**# of users**
A	BI + PN	BI + CC	BI	8
B	BI	BI + PN	BI + CC	12
C	BI + CC	BI	BI + PN	9

Table 6.1: Configurations of the user study of user interaction models in Playground.

Table 6.1. Each configuration has the same order of files/tasks, but we chose three environment permutations. As we could not force people to finish the study, the number of users per environment is not perfectly balanced.

To answer RQ2 and RQ3, we asked the participants about the perceived usefulness of our novel interaction models, and the confidence about the extraction of each file, using a Likert scale from 1 to 7, 1 being the least useful/confident:

1. How confident were you in the final extraction result?
2. After training, would you trust Playground to work correctly on another similar file if you do not provide any examples?
3. How easy was it for you to complete this task?

Results We analyzed the data both from the logs collected by the UI instrumentation, and from the initial and final surveys.

RQ1: Do Program Navigation and Conversational Clarification contribute to correctness?

Yes. We have found significant reduction of number of errors with each of these new interaction models (see Figure 6.7). Our new interaction models reduce the error rate in data extraction without any negative effect on the users' extraction speed. To obtain this result, we applied the Wilcoxon rank-sum test on the instrumentation data. More precisely,

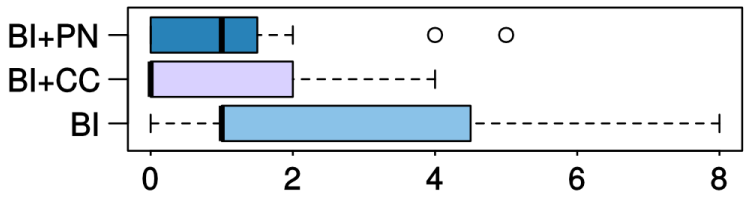


Figure 6.7: Distribution of error counts across the three environments in the user study of data wrangling in Playground. Both Conversational Clarification (CC) and Program Navigation (PN) significantly decrease the number of errors.

users in BI + CC ($W = 78.5$, $p = 0.01$) and BI + PN ($W = 99.5$, $p = 0.06$) performed better than BI, with no significant difference between the two of them ($W = 94$, *n.s.*). There was also no statistically significant difference between the completion time in BI and completion time in BI + CC ($W = 178.5$, *n.s.*) or BI + PN ($W = 173$, *n.s.*).

RQ2: Which of Program Navigation and Conversational Clarification is perceived more useful for data extraction?

Conversational Clarification is perceived more useful than Program Navigation (see Figures 6.8a and 6.8b). Comparing the user-reported usefulness between the Conversational Clarification and the Program Navigation, on a scale from 1 (not useful at all) to 7 (extremely useful), the Conversational Clarification has a mean score of 5.4 ($\sigma = 1.50$) whereas the Program Navigation has 4.2 ($\sigma = 2.12$) Only 4 users out of 29 score Program Navigation more useful than Conversational Clarification, whereas Conversational Clarification is scored more useful by 15 users.

RQ3: Do our novel interaction models help alleviate typical distrust in PBE systems?

Yes for Conversational Clarification. The tasks finished with Conversational Clarification obtained a higher confidence score compared to those without ($W = 181.5$, $p = 0.07$). No significant difference was found for Program Navigation ($W = 152.5$, *n.s.*). Regarding the trust our users would have if they ran the learned program on other data, we did not find

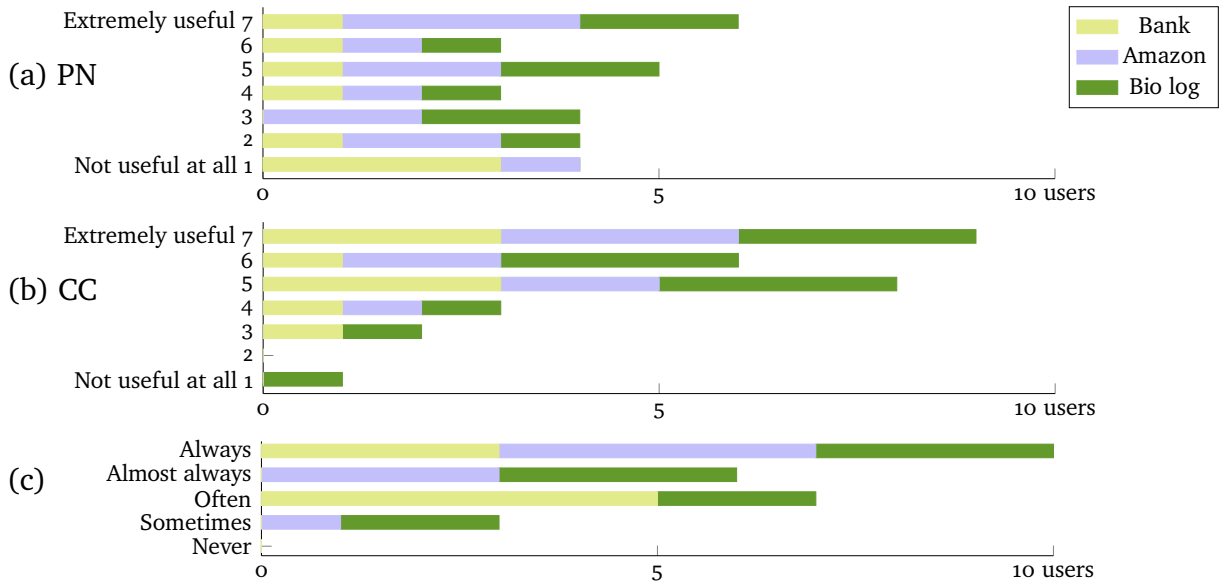


Figure 6.8: User-reported: **(a)** usefulness of Program Navigation, **(b)** usefulness of Conversational Clarification, **(c)** correctness of one of the choices of Conversational Clarification.

any significant differences for Conversational Clarification ($W = 146$, *n.s.*) and Program Navigation ($W = 161$, *n.s.*) over only BI.

Regarding the question "How often would you use Playground, compared to other extraction tools?", on a Likert scale from 1 (never) to 5 (always), 4 users answered 5, 17 answered 4, 3 answered 3, and the remaining 4 answered 2 or less. Furthermore, all would recommend Playground to others. When asked how excited would they be to have such a tool on a scale from 1 to 5, 8 users answered 5, and 15 answered 4.

The users' trust is supported by data: Perceived correctness is negatively correlated with number of errors (Spearman $\rho = -0.25$, $p = 0.07$). Although we asked them to make sure the extraction is correct and never told them they did errors, users making more errors (thus unseen) reported to be less sure about the extraction. However, there is no significant correlation between number of errors made and the programming experience mapped between 0 and 4 ($\rho = -0.09$, *n.s.*).

6.2.5 Conclusion

The two key takeaways of this user study is that **(a)** any user interaction model (Program Navigation or Conversational Clarification) reduces the number of errors in the completed tasks and improves the users' confidence in the results (as compared to example-only workflow); **(b)** Conversational Clarification is perceived as more useful thanks to its proactivity and clarity. Based on these observations, PROSE now includes a more general version of Conversational Clarification (called *feedback-based synthesis*) as a key capability for any DSL. The next section presents its design and applications.

6.3 Feedback-Based Synthesis

As we discussed above, an end user often does not have a clear understanding of the full spec, and may suffer significant cognitive load with this task. At every iteration, the current candidate program set $\mathcal{S}_i$ contains thousands of ambiguous programs, and humans struggle with reasoning about possible ambiguities in intent specification. In contrast, the synthesis system can analyze ambiguities in $\mathcal{S}_i$ and derive the most efficient way to resolve them by proactively soliciting concrete knowledge from the user. This observation introduces a new actor in the traditional learner-user interaction process, which we call *the hypothesizer*.

Figure 6.9 shows our envisioned workflow. Formally, any *constraint type* used in PBE (e.g. example, prefix, output type) states a *property* on a subset of the DSL. Given a program set $\mathcal{S}_i$, the hypothesizer deduces possible properties that best disambiguate among programs in $\mathcal{S}_i$. Any such property is convertible to a Boolean or multiple-choice question q, which the hypothesizer asks the user. Any response r for q is convertible to a concrete constraint ψ, which begins a new iteration of synthesis.

Such *feedback-based* interaction has several major benefits. First, it reduces the cognitive load on the user: instead of analyzing the program's behavior, she only answers concrete questions. Second, it significantly reduces the number of synthesis iterations thanks to the

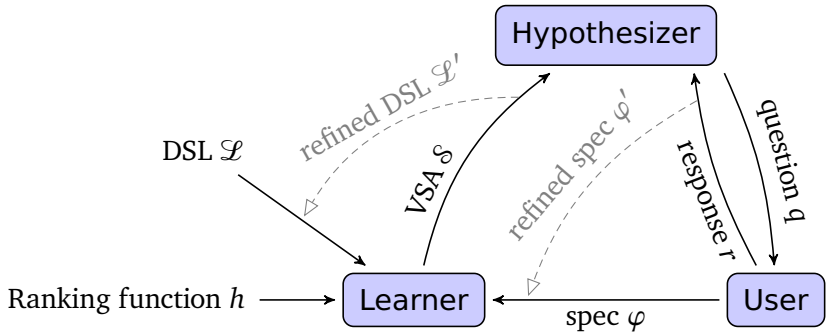


Figure 6.9: Learner-user communication in interactive PBE. At each iteration, the hypothesizer takes from the learner the VSA $\mathcal{S}$ of the programs consistent with the current spec φ, and performs two automatic optimizations: (i) it converts the VSA into a refined DSL $\mathscr{L}'$ to be used at the next iteration instead of the original DSL $\mathscr{L}$, and (ii) it constructs the most effective clarifying question q to ask the user about the ambiguous candidates in $\mathcal{S}$, and converts the user's response r into the corresponding refined spec φ'.

hypothesizer's insight into the program set $\mathcal{S}$ and its choice of disambiguating questions. Finally, feedback and proactiveness increases the user's confidence and trust in the system.

Correctness-burden balance The crucial part of the hypothesizer's job is evaluation of disambiguation effectiveness of every potential question. Our prototype implementation of Conversational Clarification in Playground always requested its clarification on the first available discrepancy. However, this is neither the fastest nor the most user-friendly experience:

- On one hand, a system might ask for an additional example every time there exists an alternative candidate program that disagrees with the current most likely candidate on some row. In this case, the learning session is guaranteed to converge to the desired program if one exists, typically within 10-20 iterations.

  Despite the attractive guarantee, this approach is undesirable because a good ranking function often picks the intended program after 1-5 examples. In such a case, all

subsequent iterations only eliminate implausible candidates without actually changing the result. The user spends a lot of cognitive effort answering superfluous questions.

- On another hand, the system might stop asking disambiguating questions too early. In this case, the learned program may be incorrect but the user will only notice it if she discovers a discrepancy manually.

In machine learning terminology, a superfluous disambiguating question is a *false positive* (it is generated but not strictly required to learn the correct program), and a missing disambiguating question is a *false negative* (it is required but not generated). Our Playground study [37] included only the most conservative disambiguating algorithm, which always converged to zero ambiguities. As a result, it always generated false positive questions.

To minimize false positives, a disambiguating algorithm must intelligently pick the discrepancies presented to the user. Well-chosen discrepancies will eliminate implausible candidate programs from consideration faster. To minimize false negatives, the algorithm must intelligently estimate the likelihood of the currently chosen candidate program. A good estimation should ensure that any correct (or at least plausible) program is ranked higher than all incorrect (implausible) programs.

6.3.1 Problem Definition

Let $\mathcal{L}$ be a DSL. Let Ψ be a set of top-level *constraint types* supported by the synthesizer and witness functions for $\mathcal{L}$. For each constraint type $\psi \in \Psi$ we associate a *descriptive question* q such that a response r for this question directly constitutes an instance of ψ. We denote such a constraint as $\Psi(r)$. Questions q can be Boolean (usually in ternary logic) or multiple-choice. We denote the set of possible responses for q as $R(q)$.

Example 10. An *example constraint* "output = v" corresponds to a multiple-choice question "Is the desired output on an input σ equal to v_1, v_2, ..., or v_k?" A response to this question constitutes an example constraint "output = v_i" for the chosen i.

procedure DISAMBIGUATE(Candidate programs $\mathcal{S}$, current spec φ)

Analyze the ambiguities in $\mathcal{S}$ w.r.t. φ.
Let Q be a set of questions that may resolve ambiguity in $\mathcal{S}$
// Compare the disambiguation scores of all questions
$q^* \leftarrow \underset{q \in Q}{\text{argmax}}\, \mathsf{ds}(q, \mathcal{S}, \varphi)$
if $\mathsf{ds}(q^*, \mathcal{S}, \varphi) <$ threshold T **then**
    break
else
    Present the question q^* to the user
    Let r be the user's response to q
    Let ψ be the response r converted into a constraint
    return ψ to the learner and invoke a new round of synthesis

Figure 6.10: The hypothesizer's proactive disambiguation algorithm.

A *datatype constraint* "output : τ" corresponds to a Boolean question "Is the desired program a computation of type τ? Yes (always), no (never), or unknown (maybe)."

Questions, like constraints, can be domain-specific. In FlashFill, a *relevance constraint* states "an input v_i must/must not appear in the program." It corresponds to a Boolean question "Should the input v_i be used? Yes (always), no (never), or unknown (maybe)."

Figure 6.10 shows the disambiguation algorithm of the hypothesizer. Given a VSA $\mathcal{S}$ of current candidate programs and the current iteration's spec φ, the job of the hypothesizer is to analyze $\mathcal{S}$ and pick the best question to resolve ambiguities in $\mathcal{S}$. If $\mathcal{S}$ has no ambiguities, or if the hypothesizer is not confident in the effectiveness of potential questions, it considers the current candidate program $P^* = \mathsf{Top}_h(\mathcal{S}, 1)$ correct and does not ask any questions.

6.3.2 Disambiguation Score

To evaluate a question's effectiveness, the hypothesizer is parameterized with a *disambiguation score* function $\mathsf{ds}(q, \mathcal{S}, \varphi)$. Higher disambiguation scores correspond to more effective questions q – that is, constraints generated by answering q eliminate more incorrect programs from $\mathcal{S}$.

Since the hypothesizer cannot predict the user's response, $\mathsf{ds}(q, \mathcal{S}, \varphi)$ must represent *potential effectiveness* of q for any possible outcome.

Disambiguation score functions may be domain-specific or general-purpose. In our evaluation, we found different functions to perform well for different DSLs. In this section, I present two efficient *general-purpose* disambiguation score function. One of them is independent of the current iteration's spec φ but takes into account the ranking scores of alternative candidate programs in $\mathcal{S}$. Another one is independent of any domain-specific details of the DSL (including its ranking function), but may be harder to compute.

Ranking-based disambiguation The *ranking-based disambiguation score* function prefers a question that promotes higher-ranked programs:

$$\mathsf{ds}_{\mathsf{R}}(q, \mathcal{S}, \varphi) \stackrel{\text{def}}{=} \min_{r \in R(q)} \max_{P \in \mathcal{S}_r} h(P)$$

where $\mathcal{S}_r$ is a set projection $\mathsf{Filter}\,(\mathcal{S}, \Psi(r))$, and h is a ranking function provided with the DSL. In other words, $\mathsf{ds}_{\mathsf{R}}(q, \mathcal{S}, \varphi)$ is higher if every response for the question q leads to a higher-ranked alternative program among the candidates that are consistent with it.

This disambiguation score can be efficiently evaluated for many constraint types. For instance, calculating $\mathsf{ds}_{\mathsf{R}}(q, \mathcal{S}, \varphi)$ for *example constraints* amounts to clustering $\mathcal{S}$ and comparing the top-ranked programs across all clusters. Alternatively, we can quickly compute a good approximation to $\mathsf{ds}_{\mathsf{R}}(q, \mathcal{S}, \varphi)$ by randomly sampling k programs from the VSA and considering only their outputs.

Entropy-based disambiguation The *entropy-based disambiguation score* prefers a question that resolves more uncertainty in the VSA:

$$\mathsf{ds}_{\mathsf{E}}(q, \mathcal{S}) = E\left\{\mathcal{S}_r \mid r \in R(q)\right\}$$

where each $\mathcal{S}_r$ is a *projection* of $\mathcal{S}$ onto the corresponding constraint for the response r:

$$\mathcal{S}_r = \mathsf{Filter}\,(\mathcal{S},\ \Psi(r))$$

and $E(\mathcal{S}_1, \ldots, \mathcal{S}_k)$ is the *entropy of sizes* of its parameters:

$$E(\mathcal{S}_1, \ldots, \mathcal{S}_k) = -\sum_{i=1}^{k} \frac{|\mathcal{S}_i|}{Z} \log \frac{|\mathcal{S}_i|}{Z}, \qquad Z = |\mathcal{S}_1| + \ldots + |\mathcal{S}_k|$$

In other words, $\mathsf{ds}_{\mathsf{E}}(q, \mathcal{S})$ is higher if possible responses for q induce a high-entropy partitioning of $\mathcal{S}$ (i.e., they split $\mathcal{S}$ into fairly even partitions).

Example 11. If q is a question for an *example constraint*, its set of responses $R(q)$ includes all possible outputs $r_1, \ldots, r_k$ of the programs currently in $\mathcal{S}$. The corresponding constraint $\Psi(r_j)$ for each response is the example constraint "output = r_j". Thus, the projections $\mathsf{Filter}(\mathcal{S}, \Psi(r_1)), \ldots, \mathsf{Filter}(\mathcal{S}, \Psi(r_k))$ are simply clusters in the partitioning $\mathcal{S}/_{\sigma}$ of $\mathcal{S}$ w.r.t. the current input σ. Each of the clusters contains only the programs that evaluate to the same output r_j.

Hence, $\mathsf{ds}_{\mathsf{E}}(q, \mathcal{S})$ in this case is equal to the entropy of the clustering $\mathcal{S}/_{\sigma}$. Given a VSA $\mathcal{S}$, the hypothesizer simply computes its clustering on the current input σ, computes the entropy of this clustering, and uses it to decide the potential effectiveness of asking a multiple-choice disambiguation question.

For example, consider a VSA of 10 programs that produce 2 different outputs. $R(q)$ then has two options, one for each possible output. If the cluster sizes corresponding to these outputs are $\{5; 5\}$, then the entropy of this clustering is high, and any response to q will eliminate a significant fraction of the candidate programs. In contrast, if the cluster sizes are $\{9; 1\}$, then the entropy of this clustering is low, and a response may or may not be efficient in eliminating incorrect programs.

Example 12. For a ternary Boolean question q (e.g., a datatype question) the response set $R(q) = \{$"yes", "no", "unknown"$\}$. The first two responses split $\mathcal{S}$ into two disjoint partitions. The third response does not provide any new information, so its corresponding projection is the entire $\mathcal{S}$, which contributes 0 to the entropy summation. Therefore, $\mathsf{ds}_{\mathsf{E}}(q, \mathcal{S})$ is equal to the entropy of the two-way split of $\mathcal{S}$ into programs satisfying q and all others.

As compared to ranking-based disambiguation score, the entropy-based score is independent of a ranking function and therefore DSL-agnostic. However, it also has its drawbacks, such as reliance on the clustering operation $\mathcal{S}/_{\sigma}$, which may be expensive for large VSAs.

6.3.3 Evaluation

We use the PROSE-based implementation of FlashFill to evaluate feedback-based synthesis on a set of 457 text transformation tasks. It uses *example* constraint questions. In order to efficiently generate these questions, we sample 2000 programs from the VSA and cluster on those, assigning the disambiguation score ds_{R} as described above. We chose 2000 to achieve a good balance between performance and having a high probability of including at least one program from every large cluster.

In the *baseline setting*, the user provides the earliest incorrect row as the next example at each iteration. In the *feedback-driven setting*, the system instead proactively asks the user disambiguating questions on selected input rows until the disambiguation score falls below the threshold T. We set $T = 0.47$ as the mean of the score distribution over our tasks.

We evaluate FlashFill's feedback on two dimensions: *cognitive burden* and *correctness*. Cognitive burden is defined as the number of rows the user has to *read and verify* in the process. In the baseline setting, it is the number of examples + the number of correct rows before the first discrepancy that are verified at each iteration. In the feedback-driven setting, it is the number of questions answered.

Correctness is a combination of *false positives* and *false negatives*. False positive questions occur when FlashFill keeps asking questions after the program is already correct. False negative questions occur when FlashFill stops before it finds a correct program.

In correctness evaluation, only 2/457 tasks completed incorrectly (i.e., with false negatives). The majority of tasks (342/457) finish with the same number of examples, and the number of false positives in the rest never exceeds 4 (specifically, 90 tasks with 1 false positive, 22 with 2, and 1 task with 4 false positives).

	Feedback-driven				
Baseline	**1**	**2**	**3**	**4**	**5**
1	241	50	16		1
2	75	30	4		
3	20	7	2		
4		5	3		
5		1			
6	1	1			

Table 6.2: Number of rows inspected in the baseline and the feedback-driven settings for the evaluation of a feedback-driven FlashFill interaction with a ranking-based disambiguation score. The data is presented as a histogram: for example, there were **20/457** scenarios such that the baseline setting required **3** iterations to complete the task, and the feedback-based setting required **1** iteration. Empty cells represent the value of 0.

Table 6.2 compares the cognitive burden of both settings. It shows the histogram distribution of our tasks for each pair of verified row counts in baseline and feedback-driven settings. The baseline setting often requires the user to inspect more rows (that is, the numbers *below* the diagonal in Table 6.2 are larger than the numbers above it).

6.4 Incremental Synthesis

In this section, I present the final enhancement we explored in the context of interactive program synthesis – *incremental synthesis*. In conventional synthesis, the learner uses the refined spec φ' at each iteration to synthesize a new valid program set $\mathcal{S} \models \varphi'$. All the information accumulated in the synthesis session is contained in φ' as a conjunction of provided examples, counterexamples, and more general constraints. Typically in PBE, the learner takes it all into account by solving a fresh synthesis problem, searching in the DSL $\mathcal{L}$ for programs that are consistent with all constraints in φ'. As the size of φ' grows with each

iteration, this synthesis problem becomes more complex, thereby slowing down the search process [22].

Our key observation here is that *the program set $\mathcal{S}_i$ learned at iteration i can be transformed into a new DSL $\mathscr{L}'$, which will become the search space for synthesis in iteration $i+1$ instead of $\mathscr{L}$*. Notice that every refined spec φ' imposes an *additional* restriction on the desired program. Thus, an i^{th} program set $\mathcal{S}_i$ must be a subset of the program set $\mathcal{S}_{i-1}$, learned at the previous iteration.

We developed an efficient procedure for transforming a VSA $\mathcal{S}_i$ into a DSL definition $\mathscr{L}_{i+1}$, which replaces $\mathscr{L}$ in the next iteration of synthesis. This replacement achieves two significant speedups. First, the size of $\mathscr{L}_i$ is monotonically decreasing, and quickly becomes many orders of magnitude smaller than $\mathscr{L}$. Second, at each iteration i we are only searching for programs consistent with the latest introduced constraint ψ_i, since the DSL $\mathscr{L}_i$ by construction only contains programs that satisfy previous constraints $\psi_1, \ldots, \psi_{i-1}$.

As part of our envisioned learner-user interaction process, the conversion of $\mathcal{S}_i$ into $\mathscr{L}_{i+1}$ is also the job of the hypothesizer. In Figure 6.9, it is shown as an automatic feedback with a "refined DSL $\mathscr{L}'$", leading back into the learner.

6.4.1 VSA Conversion

Recall that in Section 4.4, we have established that a VSA $\mathcal{S}$ over a DSL $\mathscr{L}$ is isomorphic to a context-free grammar of a sub-DSL $\mathscr{L}' \subset \mathscr{L}$. Figure 6.11 shows an algorithm for translating $\mathcal{S}$ into a grammar of $\mathscr{L}'$. It performs an isomorphic graph translation, converting VSA unions ($\cup$) into CFG alternatives ($N := N_1 \mid N_2$), VSA joins ($F_{\bowtie}$) into CFG operator rules ($N := F(\ldots)$), and explicit program sets into CFG terminals annotated with their possible values.

Note that as a subset of $\mathscr{L}$, the new DSL $\mathscr{L}'$ does not introduce any new operators. Thus, all witness functions for $\mathscr{L}$ are still applicable for synthesis in $\mathscr{L}'$. Moreover, $\mathscr{L}'$ is finite and its terminals are annotated with explicit sets of permitted values, which allows fast learning of constants for any spec type simply via set filtering.

function VsaToDsl(VSA $\mathcal{S}$)

1: Let V be a set of fresh nonterminals, one per each non-leaf node in $\mathcal{S}$
2: Let Σ be a set of fresh terminals, one per each leaf node in $\mathcal{S}$
3: *// We write* $\mathsf{sym}(\mathcal{S}') \in V \cup \Sigma$ *to denote the corresponding fresh symbol for a node* $\mathcal{S}'$ *from* $\mathcal{S}$
3: Productions $R \leftarrow \emptyset$
4: *// Create "symbol := symbol" productions for all union nodes*
5: **for all** union nodes $\mathcal{S}' = \mathsf{U}(\mathcal{S}_1, \ldots, \mathcal{S}_k)$ in $\mathcal{S}$ **do**
6: $\quad R \leftarrow R \cup \{\mathsf{sym}(\mathcal{S}') := \mathsf{sym}(\mathcal{S}_i) \mid i = 1 \ldots k\}$
7: *// Create operator productions for all join nodes*
8: **for all** join nodes $\mathcal{S}' = F_{\bowtie}(\mathcal{S}_1, \ldots, \mathcal{S}_k)$ in $\mathcal{S}$ **do**
9: $\quad R \leftarrow R \cup \{\mathsf{sym}(\mathcal{S}') := F(\mathsf{sym}(\mathcal{S}_1), \ldots, \mathsf{sym}(\mathcal{S}_k))\}$
10: *// Annotate terminal symbols with values extracted from leaf nodes*
11: **for all** leaf nodes $\mathcal{S}' = \{P_1, \ldots, P_k\}$ in $\mathcal{S}$ **do**
12: $\quad$ Annotate in Σ that $\mathsf{sym}(\mathcal{S}') \in \{P_1, \ldots, P_k\}$
13: **return** the context-free grammar $G = \langle V, \Sigma, R, \mathsf{sym}(\mathcal{S})\rangle$

Figure 6.11: An algorithm for translating a VSA $\mathcal{S}$ of programs in a DSL $\mathscr{L}$ into an isomorphic grammar for a sub-DSL $\mathscr{L}' \subset \mathscr{L}$.

6.4.2 Constraint Resolution

Deductive synthesis relies on existence of witness functions, which backpropagate constraints top-down through the grammar. Every witness function is defined for a particular *constraint type* ψ that it is able to decompose. While some generic operators allow efficient backpropagation procedures for common constraint types (see, e.g. [9, 33]), most witness functions are domain-specific.

We have identified a useful set of constraint types that occur in various PBE domains and often permit efficient witness functions. We broadly classify these constraints in three

categories depending on their *descriptive power*, with different incremental synthesis techniques required for each category.

Decomposable constraints: constructively *define* a subset of the DSL by the means of deduction through witness functions. In other words, properties of the program's output that they describe are narrow enough to enable deductive reasoning. For instance:

- *Example constraint*: "output = v",
- *Membership constraint*: "output $\in \{v_1, v_2, v_3\}$",
- *Prefix constraint*: "output = $[v_1, v_2, \ldots]$",
- *Subset/subsequence constraint*: "output $\sqsupseteq [v_1, v_2, v_3]$".

Locally refining constraints: do not define a DSL subset on their own, but can be used to *refine* an existing set in a witness function for some DSL operator(s). For instance:

- *Datatype constraint*: "output: τ". Eliminates all top-level programs rooted at any type-incompatible DSL operators.
- *Provenance constraint*: describes the desired construction method for some parts of the output. For example, in FlashFill it may take form "substring $[i : j]$ of the output example o_k is extracted from location ℓ of the corresponding input σ_k", or "substring $[i : j]$ of the output example o_k is a date value, formatted as "`YYYY-MM-DD`"". Allows simple domain-specific elimination of invalid subprograms.
- *Relevance constraint*: marks inputs or parts of the input as *required* or *irrelevant*. Eliminates all programs that do not use any required parts or reference any irrelevant parts.

Globally refining constraints: do not define a DSL subset on their own and do not permit any efficient local refining logic in witness functions. They can only be satisfied by filtering an existing program set on the topmost level of the DSL (i.e., by projecting the set on the constraint). For instance:

- *Negative example constraint:* "output $\neq v$",
- *Negative membership constraint:* "output $\notin v$".

Given a program set $\mathcal{S}$ and a new constraint ψ, we filter $\mathcal{S}$ w.r.t. ψ differently depending on the category of ψ.

- If ψ is decomposable, it seeds a new full round of deductive synthesis, which may narrow down the set unpredictably. We convert the set $\mathcal{S}$ into an isomorphic DSL VsaToDsl($\mathcal{S}$), and use it as a search space for a new synthesis round with a spec consisting of a single constraint ψ.

- If ψ is locally refining, it is only relevant to select witness functions. Suppose these witness functions backpropagate specs for the operator $F(N_1, \ldots, N_k)$.

  We first identify occurrences of F in $\mathcal{S}$. Each node of kind $F_{\bowtie}(\mathcal{S}_1, \ldots, \mathcal{S}_k)$ in $\mathcal{S}$ has been constructed during the top-down grammar traversal in a previous iteration of deductive synthesis as a solution to some intermediate synthesis subproblem $\mathsf{Learn}(F(N_1, \ldots, N_k), \varphi_F)$. To enable incrementality, *we keep references to all intermediate specs φ_F that were produced at this level in the previous synthesis iteration*.

  We now repeat the learning for F on all retained subproblems, but with their previous specs φ_F conjoined with the new constraint ψ. The witness functions for F take ψ into account and produce potentially more refined specs for $N_1, \ldots, N_k$. These new specs are decomposable (since they were produced by witness functions), and thus initiate new rounds of incremental synthesis on VsaToDsl($\mathcal{S}_1$), …, VsaToDsl($\mathcal{S}_k$). The results replace $\mathcal{S}_1, \ldots, \mathcal{S}_k$ in $\mathcal{S}$ without affecting the rest of the set.

- If ψ is globally refining, it cannot be efficiently resolved by inspecting $\mathcal{S}$ or invoking witness functions. Thus, we have to compute the projection $\mathsf{Filter}(\mathcal{S}, \psi)$ at the top level. An efficient implementation of the projection operation computes the clustering $\mathcal{S}/_\sigma$ on an input σ from the constraint ψ. All programs in the same cluster $\mathcal{S}_k$ produce the same

output v_k on σ. If the constraint ψ only references the output of the desired program (as all globally refining constraints do), then all programs in $\mathcal{S}_k$ either satisfy ψ or not. Thus, we simply take a union of clusters where the corresponding outputs v_k satisfy ψ.

We note that for many globally refining constraints this operation is trivial once the clustering is computed. For example, a negative example constraint "output $\neq v$" eliminates at most 1 cluster – the one where $v_k = v$, if one exists.

This incremental learning algorithm can be expressed as an instance of interactive synthesis formalism from Problem 2. For that, set Σ_i to be a tuple of $\mathcal{S}_i$ (required to become the search space at the next iteration) and all intermediate specs produced by witness functions (required to resolve locally refining constraints).

Theorem 7. *If the underlying one-shot learning algorithm $\mathcal{A}$ for $\mathcal{L}$ is complete, then* ***(a)*** *the incremental learning $\widehat{\mathcal{A}}$ is complete, and* ***(b)*** *at each iteration i the result $\mathcal{S}_i$ of incremental learning is equal to the result of cumulative one-shot learning* $\mathsf{Learn}_{\mathcal{A}}(N_i,\ \varphi_0 \wedge \ldots \wedge \varphi_i)$.

Proof. Follows from the fact that $\mathcal{S}_i$ are monotonically non-increasing and from the axioms of interactive synthesis in Problem 2. □

6.4.3 Evaluation

We evaluate the incremental synthesis algorithm in the context of the FlashFill DSL. For this case study, we picked all the scenarios which required the user to provide two or more examples to learn a correct program, from among our collected FlashFill scenarios. All the data reported in this subsection is obtained by repeating each experiment ten times and averaging the results after discarding outliers.

Figure 6.12 summarizes the results of our evaluation. It plots the *speedup* obtained by incremental algorithm over the non-incremental algorithm for each scenario for the FlashFill DSL. These were computed by dividing the execution time of the non-incremental algorithm by the execution time of the incremental algorithm for each scenario. We observe that almost

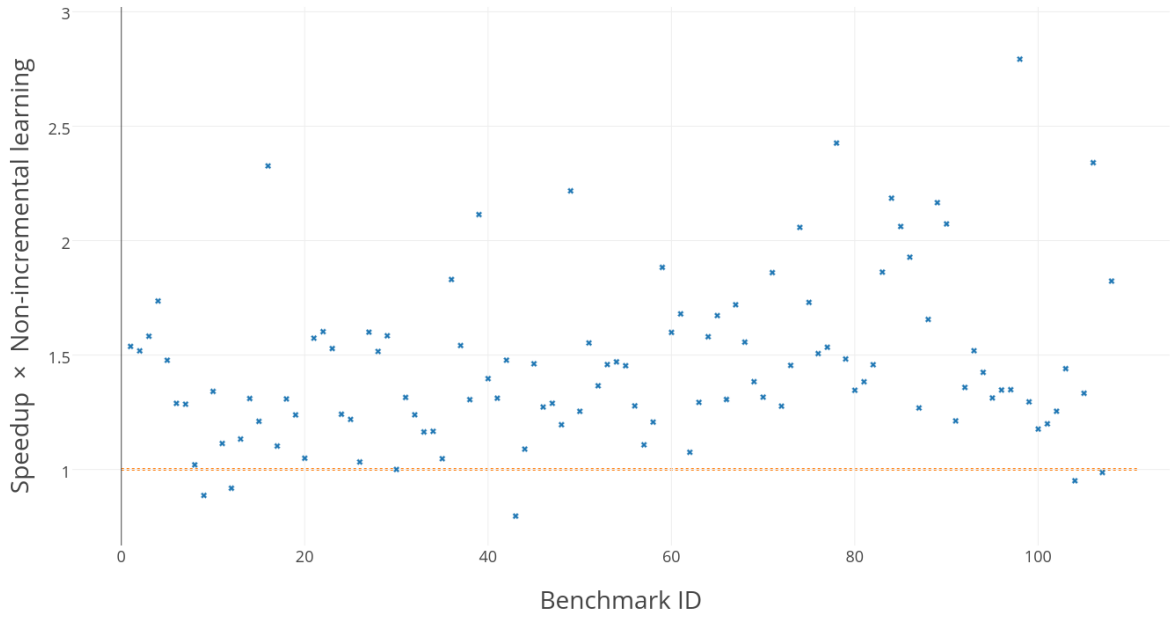


Figure 6.12: Speedups obtained by the incremental synthesis algorithm vs. the non-incremental algorithm. Values higher above the $y = 1$ line (where the runtimes are equal) are better.

all the speedup values are greater than one, with the exceptions being extremely short-running scenarios as mentioned earlier. Further, the incremental algorithm achieves a geometric mean speedup of 1.42 over the non-incremental algorithm, across the 108 scenarios considered for the FlashFill DSL. Despite the relatively small number of learning iterations, our evaluation demonstrates that incrementality yields significant performance improvements.

Chapter 7

CONCLUSIONS AND FUTURE WORK

This work presents a new universal framework for programming by examples (PBE) that is parameterized by a rich domain-specific language (DSL) and uses deductive search to efficiently construct a set of programs consistent with a given inductive specification. In the thesis statement, I set out to show that this proposed methodology generalizes most of the prior approaches to PBE. In Chapters 3 to 5, I introduce, explore, and apply this methodology, building a single unification to a disparate family of state-of-the-art techniques in the field of PBE. Moreover, as Chapter 5 demonstrates, these technologies after a reimplementation on top of our PROSE framework become more maintainable, extensible, and amenable to mass-market deployment. The fact that a PBE-based technology can now be implemented on top of one common scaffolding significantly lowers the entry barrier to developing a new such technology. As a result, PROSE allowed us to create a family of new PBE-based tools in various data wrangling and manipulation domains, which by now have been deployed in numerous industrial applications and re-used by external academic collaborators.

As we discovered in Chapter 6, after a PBE-based technology is made available as part of a mass-market application, the demands of superb user experience give birth to multiple unique research challenges, which do not arise as prominently at a smaller scale. The main such challenge is disambiguation of user intent. While omnipresent at any scale, it is particularly challenging when the underlying DSL is a large real-life data wrangling language and the task requires transforming or extracting a gigabyte-sized dataset. In such a setting, the user cannot examine the entire dataset or all the outputs of the currently learned candidate program. As a result, intent disambiguation, previously mostly reactive, has to become proactive. At each iteration of an interactive synthesis session the system must seek the user's feedback about

the currently learned program, suggest additional clarifying examples, and be transparent about its current understanding of the intent. As our experiments show, such user interaction models significantly improve the user's confidence and trust in the system, as well as decrease the number of errors in the completed tasks.

Chapter 6 presents our initial exploration of the interactive synthesis space. I aim to improve the interaction process in two ways: (a) reducing the number of refining iterations (thereby converging toward the right program faster and improving the user's confidence in the results), and (b) incrementally leveraging the results of past learning iterations to boost the performance of the synthesis engine (thereby shortening the entire session). I present our domain-agnostic implementation of feedback-based and incremental synthesis on top of the PROSE framework, which makes them immediately available to any derived PBE technology. The initial results show significant improvements in both the number of synthesis iterations and the learning time of a single iteration on a diverse collection of real-life scenarios.

7.1 Future Horizons

The PROSE framework makes a significant step toward democratizing program synthesis and making it a true industrial commodity, available to any domain expert without a required background in formal methods. However, this step does not yet bring us all the way to this goal. Developing a PBE technology on top of PROSE still requires non-trivial insight about both the underlying DSL and the synthesis process. Specifically,

- the designer must manually explore a large dataset of practical tasks to envision a concise but expressible enough DSL to cover all of them,
- the designer must implement witness functions for any non-standard operators in this DSL (which may be non-trivial for the operators that are not easily invertible),
- the designer must specify a ranking function to disambiguate among the programs in the DSL (which is tedious and typically requires taking into account numerous corner cases), and

- the designer must maintain all three aforementioned components manually, extending them to cover new tasks in the domain, to improve the synthesizer's performance, or to simplify the interaction process.

To truly democratize program synthesis, we should address these barriers, either via intelligent automation, or via computer-aided tooling. Below, I identify several specific problems in this space that we are currently investigating.

Learning to rank The most time-consuming part of developing a PROSE-based PBE technology is writing a ranking function [44]. Its nature (typically fuzzy, hardly interpretable, and full of corner cases) suggests that stochastic techniques such as deep learning should be effective at automatically learning a ranking function given a dataset of completed tasks. In 2015, Singh and Gulwani [54] successfully demonstrated such a technique on the original implementation of FlashFill. We are currently incorporating this capability in the core of the PROSE framework.

Ideally, we would like to make the training procedure domain-agnostic and thus potentially applicable to any PROSE-based DSL. However, this would eliminate a whole class of important features for the ranking functions – *data-based features*. Our recent experiments show that an effective ranking function cannot be defined on a program AST alone, it must also take into consideration the task spec provided to the synthesis engine (i.e. program inputs and outputs) [8]. Hence, the final architecture will necessarily include both domain-agnostic and domain-specific components, and designing it in an accessible way is an open question.

Ranking the search space The witness functions of PROSE implicitly define a search space for the programs that are consistent with a given task spec. This search space is structured as a highly branching DAG. As the branching factor of this DAG grows, so does the duration of the learning process (as we showed in Section 5.6). Because deductive search must independently explore all the branches to guarantee completeness (i.e. to find a conformant program if one exists), such branching becomes a bottleneck in the synthesis process.

While the ranking function on its own does not help to prioritize or eliminate any branches (since it only ranks complete programs), the ranking principle may nevertheless help us. Recently, Balog et al. [3] showed that a deep learning system trained with randomly generated DSL programs can learn a probability distribution over the DSL operators that improves the performance of an enumerative program synthesizer by several orders of magnitude. A natural extension of this methodology to deductive synthesis is learning *search tactics*. A search tactic is a probability distribution over the branches deduced by a witness function, which helps the system prioritize those branches that are more likely to contain a relevant subexpression for the given task spec. For example, in real-life FlashFill scenarios it is rare that two logically different subexpressions of a `Concat` program construct different consecutive parts of the same token in the output string (e.g. a number). Thus, a common effective search tactic suggests prioritizing the branches that split the output string only at token boundaries. Although such tactics are currently specified manually by the DSL designers, they look easily learnable given our large compiled dataset of real-life synthesis tasks.

Multi-modal specifications Input-output examples are a natural specification choice for end users, who typically do not know formal logic or even programming. However, in some cases (e.g. SQL or CSS synthesis) even specifying a full example may be overly cumbersome. A better means of specification in those cases is *natural language* – either via describing the desired program completely, or via augmenting other specifications by keyword-based queries. Although prior work in program synthesis by natural language has required significant engineering [15], recent advances in natural language processing (NLP), powered by deep learning, should help overcome this barrier [45].

Mixed-initiative DSL design Designing the right DSL for a PBE-based application remains in some ways an art form. On one hand, it must be expressive enough to cover the desired practical tasks, and generalize them enough to not overfit to the specific instances considered by the designer. On the other hand, it must be concise and modular enough to enable efficient

program synthesis via deductive search. Currently, PROSE does not offer the designer any help in this process beyond straightforward syntax validation. To improve this, we are considering a mixed-initiative tool that would analyze the DSL currently specified by the domain expert, and evaluate it w.r.t. the specified design goals (such as the desired task space coverage).

BIBLIOGRAPHY

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–8, 2013.

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

[3] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016.

[4] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 218–228, 2015.

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB standard – version 2.5, 2010.

[6] Tantek Celik, Elika J. Etemad, Daniel Glazman, Ian Hickson, Peter Linss, and John Williams. Selectors level 3. W3C recommendation. *World Wide Web Consortium*, page 76, 2011.

[7] Yves Chauvin and David E. Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology Press, 1995.

[8] Kevin Ellis and Sumit Gulwani. Learning to learn programs from examples: Going beyond program structure. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, May 2017.

[9] John Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[10] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *ACM SIGPLAN Notices*, volume 51, pages 802–815. ACM, 2016.

[11] M.I. Gorinova, Karl Prince, Sallyanne Meakins, Alain Vuylsteke, Matthew Jones, and A.F. Blackwell. The end-user programming challenge of data wrangling. In *27th Annual Workshop of Psychology of Programming Interest Group (PPIG)*, 2016.

[12] C. Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 219–240, 1969.

[13] Sumit Gulwani. Dimensions in program synthesis. In *The 12th International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 13–24. ACM, 2010.

[14] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL)*, volume 46, pages 317–330, 2011.

[15] Sumit Gulwani and Mark Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 803–814. ACM, 2014.

[16] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[17] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, July 1961. ISSN 0001-0782. doi: 10.1145/366622.366647.

[18] Wilfrid Hodges. *A shorter model theory*. Cambridge University Press, 1997.

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

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

[21] Susmit Jha and Sanjit A. Seshia. A Theory of Formal Synthesis via Inductive Learning. *Acta Informatica*, 2017.

[22] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, volume 1, pages 215–224. IEEE, 2010.

[23] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*, pages 3363–3372. ACM, 2011.

[24] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. Enterprise data analysis and visualization: An interview study. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2917–2926, 2012.

[25] Susumu Katayama. MagicHaskeller on the Web: Automated programming as a service. In *Haskell Symposium*, 2013.

[26] Dileep Kini and Sumit Gulwani. FlashNormalize: Programming by examples for text normalization. Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI), 2015.

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

[28] Ali Sinan Koksal, Yewen Pu, Saurabh Srivastava, Rastislav Bodik, Jasmin Fisher, and Nir Piterman. Synthesis of biological models from mutation experiments. In *Proceedings of the 40th ACM Symposium on Principles of Programming Languages (POPL)*, 2013.

[29] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Software synthesis procedures. *Communications of the ACM*, 55(2):103–111, 2012.

[30] Yasuo Kuniyoshi, Masayuki Inaba, and Hirochika 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.

[31] Tessa A. Lau. Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*, 2008.

[32] Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *Proceedings of the 17th International Conference on Machine Learning (ICML)*, pages 527–534, 2000.

[33] Vu Le and Sumit Gulwani. FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, page 55. ACM, 2014.

[34] David D. Lewis and William A. Gale. A sequential algorithm for training text classifiers. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 3–12, New York, NY, USA, 1994. Springer-Verlag New York, Inc. ISBN 0-387-19889-X.

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

[36] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[37] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th ACM Symposium on User Interface Software and Technology (UIST)*, 2015.

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

[39] John Morcos, Ziawasch Abedjan, Ihab Francis Ilyas, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. DataXFormer: An interactive data transformation tool. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 883–888. ACM, 2015.

[40] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[41] Stephen Owen. Advanced Parsing with ConvertFrom-String. https://foxdeploy.com/2015/01/13/walkthrough-part-two-advanced-parsing-with-convertfrom-string/, 2015. [Online; accessed July 4, 2016].

[42] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 522–538. ACM, 2016.

[43] Oleksandr Polozov and Sumit Gulwani. FlashMeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 107–126, 2015.

[44] Oleksandr Polozov and Sumit Gulwani. Program synthesis in the industrial world: Inductive, incremental, interactive. In *5th Workshop on Synthesis (SYNT)*, 2016.

[45] Chris Quirk, Raymond J Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd annual meeting of the Association for Computational Linguistics (ACL)*, pages 878–888, 2015.

[46] Mohammad Raza and Sumit Gulwani. Automated data extraction using predictive program synthesis. In *28th AAAI Conference on Artificial Intelligence*, pages 882–890, 2017.

[47] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, pages 404–415. IEEE Press, 2017.

[48] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *ACM SIGPLAN Notices*, volume 43, pages 159–169. ACM, 2008.

[49] Tobias Scheffer, Christian Decomain, and Stefan Wrobel. Active hidden markov models for information extraction. In *Proceedings of the 4th International Conference on Advances in Intelligent Data Analysis*, pages 309–318, London, UK, UK, 2001. Springer-Verlag.

[50] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 305–316. ACM, 2013.

[51] Burr Settles. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–114, 2012.

[52] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification (CAV)*, pages 634–651. Springer, 2012.

[53] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *Proceedings of the VLDB*, 5(8):740–751, 2012.

[54] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. Computer-Aided Verification (CAV), 2015.

[55] Armando Solar-Lezama. *Program synthesis by sketching*. ProQuest, 2008.

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

[57] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 287–296. ACM, 2013.

[58] Richard J. Waldinger and Richard C. T. Lee. PROW: A step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 241–252, 1969.

[59] John Walkenbach. The research behind Flash Fill. http://spreadsheetpage.com/index.php/blog/the_research_behind_flash_fill/, 2012. [Online; accessed July 4, 2016].

[60] Steven A. Wolfman, Tessa Lau, Pedro Domingos, and Daniel S. Weld. Mixed initiative interfaces for learning tasks: Smartedit talks back. In *Proceedings of the 6th International Conference on Intelligent User Interfaces (IUI)*, pages 167–174, New York, NY, USA, 2001. ACM. ISBN 1-58113-325-1. doi: 10.1145/359784.360332.

[61] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. A colorful approach to text processing by example. In *Proceedings of the 26th ACM Symposium on User Interface Software and Technology (UIST)*, pages 495–504. ACM, 2013.