UNIVERSITY OF CALIFORNIA
Los Angeles

Data-Driven Learning of Invariants and Specifications

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Saswat Padhi

2020

ABSTRACT OF THE DISSERTATION

Data-Driven Learning of Invariants and Specifications

by

Saswat Padhi
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2020
Professor Todd D. Millstein, Chair

Although the program verification community has developed several techniques for analyzing software and formally proving their correctness, these techniques are too sophisticated for end users and require significant investment in terms of time and effort. In this dissertation, I present techniques that help programmers easily formalize the initial requirements for verifying their programs — *specifications* and *inductive invariants*. The proposed techniques leverage ideas from program synthesis and statistical learning to automatically generate these formal requirements from readily available program-related data, such as test cases, execution traces etc. I detail three of these data-driven learning techniques – FlashProfile and PIE for specification learning, and LoopInvGen for invariant learning.

I conclude with some principles for building robust synthesis engines, which I learned while refining the aforementioned techniques. Since program synthesis is a form of function learning, it is perhaps unsurprising that some of the fundamental issues in program synthesis have also been explored in the machine learning community. I study one particular phenomenon — *overfitting*. I present a formalization of overfitting in program synthesis, and discuss two mitigation strategies, inspired by existing techniques.

The dissertation of Saswat Padhi is approved.

Adnan Darwiche

Sumit Gulwani

Miryung Kim

Jens Palsberg

Todd D. Millstein, Committee Chair

University of California, Los Angeles

2020

To my parents ... for everything

To the hackers who taught me so much, and made our software secure
To those upholding truth, honesty, and integrity across our institutions
To the heroes who keep up the fight for privacy, liberty, and sovereignty

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

It is the constant support and guidance of several kind people that has helped me achieve my career and success in graduate school. First, I wish to thank my advisor, Todd Millstein, for everything he has taught me during the years — from developing a good research taste to effectively communicating ideas. I am grateful to Todd for being extremely flexible with projects that I wanted to pursue, and for always being open to collaborations. I am also thankful to my initial advisor, Miryung Kim, for allowing me the opportunity to move with her (from UT Austin) to UCLA, and join Todd's research group.

This journey would not have been the same without the amazing researchers I met at Microsoft Research — Sumit Gulwani, Alex Polozov, Rahul Sharma, and Benjamin Zorn. I feel very fortunate to have had the chance to work with them and learn from them. I cannot thank them enough for all the help, the unwavering support, and for always believing in my ideas and encouraging me. I would especially like to thank Alex and Rahul for their candid advice, and above all, for being awesome mentors and great friends.

I also owe a debt of appreciation to my coauthors and collaborators for all the insightful discussions and constructive criticism — it has been as much fun as it has been educational. I would also like to thank my committee members and colleagues, who have reviewed drafts of my papers, listened patiently to my ideas, and have given helpful feedback.

Words fall short when expressing my gratitude to my brother, my parents, and grandparents — for their unending love and absolute faith in me, for words of encouragement, and for dealing with my mood swings and days (sometimes weeks) of radio silence. Many thanks to my friends in Los Angeles and Seattle as well: Aayush, Ashutosh, Aishwarya, Akshay, Brandon, Dat, Gulzar, Kirti, Krishna, Parthe, Pradeep, Pratik, Siva, and Tianyi, who made this journey joyful and memorable. Special thanks to Siva for teaching me about networks!

Finally, I would like to thank UCLA for this opportunity to learn and grow, and Microsoft Research for supporting and recognizing my research with a PhD fellowship.

VITA

2011, 2013	Teaching Assistant ("Computer Programming and Utilization" course), Computer Science and Engineering Department, IIT Bombay.
2012	Intern, Institut für Informationssysteme, TU Braunschweig, Germany.
2013	Intern, Google LLC, Mountain View, California.
2014	Teaching Assistant ("Abstraction and Paradigms in Programming" course), Computer Science and Engineering Department, IIT Bombay.
2014	B. Tech. in Computer Science and Engineering with Honors, IIT Bombay.
2014, 2016	Teaching Assistant ("Programming Languages" course), Computer Science Department, UCLA.
2016	Intern, PROSE, Microsoft Corp., Redmond, Washington.
2017	Intern, Microsoft Research, Redmond, Washington.
2018	FLoC Olympic Games Medal in Invariant-Synthesis Track of SyGuS-Comp.
2017 – 2018	Research Software Development Engineer (part-time), Microsoft Research, Redmond, Washington.
2018 – 2019	Intern, Microsoft Research, Bengaluru, India.
2019 – Present	Organizing Committee Member, SyGuS Competition.
2017 – 2019	Microsoft Research PhD Fellow.
2019 – 2020	UCLA Dissertation-Year Fellow.
2020	Distinguished Paper Award, PLDI 2020.

PUBLICATIONS

Saswat Padhi, Rahul Sharma, and Todd Millstein. "Data-Driven Precondition Inference with Learned Features." In Proceedings of the 37$^{\text{th}}$ ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 42-56. 2016.

Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. "FlashProfile: A Framework for Synthesizing Data Profiles." In Proceedings of ACM on Programming Languages (PACMPL), 2 OOPSLA, pp. 150:1-28. 2018.

Saswat Padhi, Todd Millstein, Aditya Nori, and Rahul Sharma. "Overfitting in Synthesis: Theory and Practice." In Proceedings of the 31$^{\text{st}}$ International Conference on Computer-Aided Verification (CAV), pp. 315-334. 2019.

Anders Miltner, **Saswat Padhi**, Todd Millstein, and David Walker. "Data-Driven Inference of Representation Invariants." In Proceedings of the 41$^{\text{st}}$ ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 2020. *To Appear.*

CHAPTER 1

Introduction

Due to their unmatched efficacy, computing devices have overrun almost every aspect of our lives, from transportation systems to power grid controls to healthcare applications. We are truly approaching the age of *ubiquitous computing*, a future envisioned by Mark Weiser [Wei91], with computing systems being inseparable from our daily lives, while simultaneously making them disappear [LY02]. However, with our ever increasing dependence on these systems, it is all the more important to ensure their reliability. Today, software defects are more than a mere annoyance — every defect exposes a potential vulnerability, which may be exploited by malicious attackers. We have already witnessed several *software horror stories* [Huc04, Neu86] that had major negative impact on our social and economic growth. And even today, we continue to discover major vulnerabilities in widely used mature software, such as the HeartBleed vulnerability in OpenSSL [DKA14], the ShellShock vulnerability in Unix Bash [Inc14], the BadLock security flaw in Samba file server [Inc16], and the Ghost vulnerability in the GNU C library (glibc) [Inc15].

Attempts at reasoning about software "to make sure it is correct" date back to 1949, when Alan Turing presented formal proofs for two properties of a program for computing factorials [Tur49]. *Program verification* techniques provide strong guarantees on software, by formally proving the desired properties on them: functional correctness, guaranteed termination, memory safety and so on. However, a proof for undecidability of the halting problem [Tur37], also due to Turing, essentially implies that it is impossible to design a system which automatically verifies arbitrary non-trivial properties of programs! Nevertheless,

significant progress has been made towards manual and automated solutions for helping programmers build reliable systems [DKW08, BH14]. Computer scientists have already shown remarkable achievements in building complex systems with verified guarantees, such as compilers [Ler06, MSG09], OS kernels [KEH09], and web browsers [JTL12], etc.

However, building software with proven reliability guarantees is still a niche domain for computer scientists, and remains largely unapproachable for the software development community. Several surveys since the last two decades [HS02, WLB09, Huu13, BH14] have reported that in spite of the success of technologies for formally verifying software, they are yet to see widespread adoption as a part of the conventional software development cycle. Only a few small pockets in the industrial software community, those dealing with critical systems within very specific domains, routinely use verification techniques.

A major hindrance to building reliable software is the colossal cost, both in terms of time and human effort, involved in formalizing its intended behavior [HS02, BH14]. Sections 1.1 and 1.2 provide an overview of two main approaches for ensuring reliability of software, and concretely describe the limitations of state-of-the-art techniques. The key challenge in leveraging these techniques is providing the desired *program invariants*: properties that always hold at certain program points for all executions. Manually constructing these invariants is not only onerous, but also highly error-prone. I propose automatically learning them, and in Section 1.3 I overview some techniques that I have developed.

1.1 Post-hoc Validation

The earliest, and most popular approach for guaranteeing the reliability of software involves validating it post-hoc, i.e. after it is built. Depending on the rigor of the validation technique, they can be classified into the two categories — **(a)** verification, and **(b)** testing — which are described in subsequent subsections.

1.1.1 Formal Verification

This class of techniques formally proves a desired property on a program, thereby providing strong guarantees on its behavior over all possible executions. The theoretical underpinnings for software verification originate from the concept of *program logics*, which provide a meaning to a program by viewing it as a mathematical entity. Floyd-Hoare logic [Flo67, Hoa69], and Dijkstra's predicate transformer semantics [Dij75] provide logical frameworks for rigorously reasoning about the meaning of programs. Program logics, along with efficient decision procedures [NO79, BCD11, DB08], model checkers [CE82, CGP99, CBR01], and automated theorem provers [CAB86, Gor88, ORS92] have established the foundations for modern verification techniques [DKW08, BH14].

However, although we have achieved algorithmic breakthroughs in formal verification, the proposed techniques are quite sophisticated, and require significant investment in terms of time and effort, even for the highly-skilled. Proof assistants such as Coq [CH84, BC13], HOL [Gor88], and PVS [ORS92] require developers to write much of the correctness proofs for their implementation, which can be machine-verified. While some existing technologies, such as Spec# [BLS04], VCC [DMS09], Dafny [Lei10], and Why3 [BFM11], alleviate this by automating the proof generation, they still require the developer to furnish the following two key requirements for verification:

- A *formal specification* detailing the intended behavior precisely, in terms of desired invariants over the expected inputs, and the output. Constructing an accurate specification is quite burdensome, even for experts. As pointed out in a recent study [HS02], "*crafting a correct specification (especially one using an obscure formal system) is often much more difficult than writing the program to be proved (even one written in an obscure programming language).*"

- *Inductive invariants* for reasoning about unbounded control flows in the implementation, such as loops, recursive functions, distributed protocols, etc. First proposed by Tony

Hoare [Hoa69], inductive invariants (similar to inductive hypotheses) allow for using mathematical induction in proving properties over a potentially unbounded number of executions. Providing these is even harder than writing correct specifications, since they are fundamentally implementation-dependent and require formal reasoning of the implementation along with the specification.

Both these requirements enforce invariants on the program behavior at various points in the program, which aids formal reasoning. However, existing tools offer little or no support in helping developers provide these. Besides a high competence in formal logic required to initially model them, the ever-changing requirements for real software also necessitate the ability to rapidly update them. Although there has been a increase in interest for helping developers formulate these, existing techniques are either too restrictive, or often generate low quality candidates. Sections 2.4 and 3.6 present detailed comparisons with prior art.

1.1.2 Software Testing

Naturally, due to the significant impediments in using formal verification techniques, a majority of the software development community have instead adopted *software testing*. Testing validates the correctness of an implementation over a small subset of the possible program executions, and therefore only provides a weak guarantee on the program behavior. Several approaches have been recently proposed to improve the coverage of tests over complex real-life programs. A particularly noteworthy approach is *concolic testing* [SMA05, GKS05, CDE08], which is a hybrid approach between software verification and testing.

However, the primary objective of software testing is limited to identifying bugs, rather than rigorously proving the correctness of programs. As Dijkstra has famously said [DDH72], "*Program testing can be used to show the presence of bugs, but never to show their absence!*". The reader may refer to a recent survey [OR14] for details on the current trends in software testing. These techniques can complement software verification.

1.2 Correctness by Construction

An alternative to post-hoc software validation is synthesizing software that is correct by construction. The once "dream" [MW79] of computers automatically synthesizing a program from description of the desired behavior, is now getting closer to becoming a reality. "*Program Synthesis is the task of discovering an executable program from user intent expressed in the form of some constraints.*" [Gul10] Program synthesis approaches can also be similarly classified into two categories — **(a)** formal synthesis, and **(b)** programming by examples — based on the rigor of the technique. I overview these techniques in the following subsections.

1.2.1 Formal Synthesis

Since as early as 1980s [MW79, MW80], several techniques have been proposed for synthesizing programs from an intent expressed as a formal description of the functionality. Probably unsurprisingly, there is a strong parallel between formal verification, and formal synthesis — both require a formal specification, and inductive invariants. While a formal specification is sufficient for synthesizing loop-free programs [KMP10, JGS10], inductive invariants for loopy programs must either be manually provided [MW79, MW80, KKK13], or are implicitly expressed by constraining the structure of loops [STB06, SGF10, ABJ13]. As previously discussed in the context of formal verification, the key challenge lies in formulating the accurate specifications and inductive invariants for the desired program, which is often as hard as writing a correct implementation.

1.2.2 Programming by Examples

With the goal of addressing the difficulty in formulating a complete formal specification, there have been several recent attempts at synthesizing intended programs from incomplete specifications, such as a small set of input/output examples [CH93, Lie01, Gul11, AGK13], or more generally, an incomplete set of constraints [PG15]. These approaches, referred to as

programming by example, have achieved tremendous success and popularity, especially in the domain of *data wrangling*, due to their unparalleled efficiency and accuracy at many common data-processing tasks [PG16].

However, this approach suffers from similar drawbacks as software testing. Although the techniques exhibit a high accuracy at many common repetitive tasks, the synthesized program is unreliable beyond the provided examples for challenging tasks. This unreliability has been a target of much criticism recently [Wal13, MSG15]. The root cause for this, is the inevitable ambiguity in predicting the desired program. While examples are easier to provide than a full specification, they do not accurately constrain the set of possible programs.

1.3 Thesis Statement and Contributions

My thesis is that, *it is possible to drastically reduce the manual effort involved in ensuring software reliability, by providing developers with automated techniques for formulating the appropriate invariants for their programs using information from concrete program executions, such as test inputs, execution traces etc. that are typically much easier to provide.*

I present several such solutions in this work, all of which are guided by four key principles, which I abbreviate as the PAIR principles for designing verification systems:

Predictive: proactively suggesting likely invariants

Adaptive: allowing developers to extend the techniques with their domain knowledge

Interactive: supporting refinement of the suggested invariants towards the desired one

Rigorous: providing well-defined guarantees on the invariants, despite being speculative

The main goal of my research so far has been to help developers generate likely invariants for their programs using readily-available program-related data. Both specifications and inductive invariants essentially enforce certain invariants on the behavior at various points in the program, which are necessary to formally prove satisfaction of the desired properties over

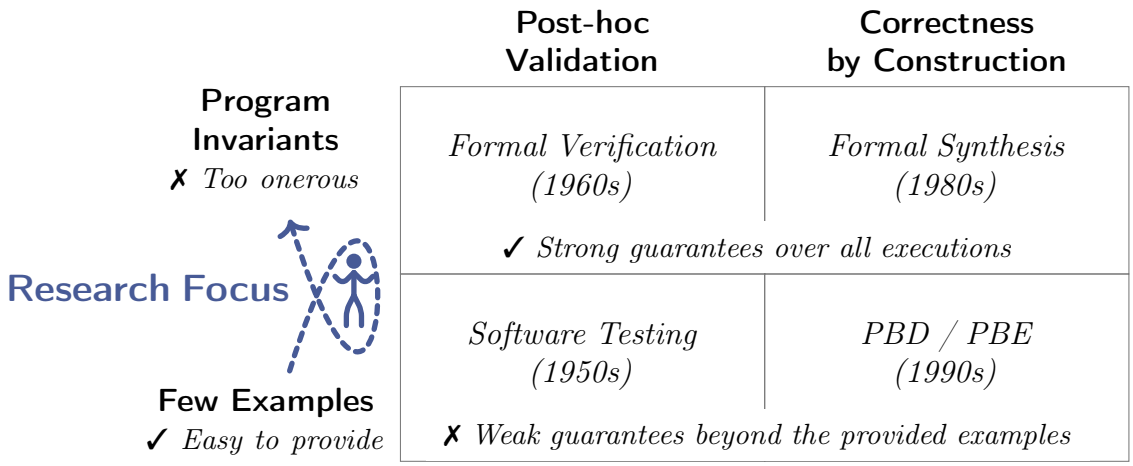


Figure 1.1: Approaches for developing reliable software, and my research focus relative to them.

all executions. However, formulating accurate program invariants manually is too onerous for developers, and relying on a few input/output examples, which are easy to provide, only yields weak guarantees. Inferring these invariants would not only simplify the task of formally verifying their implementation, but also allow for more accurate synthesis of the desired programs. Figure 1.1 positions my contributions with respect to the state of the art.

During my doctoral program, I have made significant progress on the proposed goals. In this thesis, I present three techniques based on the Pair principles, instantiated as PIE and LoopInvGen [PSM16] and FlashProfile [PJP18], which suggest program invariants from a set of tests, or program inputs respectively.

Chapter 2⟩ *Learning Program Invariants from Test Executions* [PSM16]

In this chapter, I present a technique for learning likely program invariants from a set of test executions. This approach can be used for inferring both likely specifications and inductive invariants. I demonstrate that this technique finds errors in documented specifications for OCaml library functions, and allows for a more expressive and less onerous form of automated program verification.

Chapter 3⟩ *Learning Input Specifications from Sample Inputs* [PJP18]

I present a technique for learning input specifications for large-scale data analysis programs. These specifications capture syntactic patterns in the provided input data and succinctly summarize them in form of a *data profile*. A profile is useful for identifying the input formats to be handled, when writing (or synthesizing) data processing applications that deal with large real-life datasets. I demonstrate that availability of data profiles enable improvements in PBE-based synthesizers, which traditionally only relied on input/output examples.

Chapter 4⟩ *Mitigating Overfitting in Synthesis-based Learning* [PMN19]

I demonstrate that existing example-driven synthesis techniques are prone to *overfitting* — they explain the given data well, but do not generalize well beyond it. I present a formal notion of overfitting for the example-driven program synthesis setting, and prove that overfitting is inevitable in program synthesis. I propose two mitigation techniques, inspired by existing techniques in machine learning literature, and show that they significantly improve the state-of-the-art synthesizers.

CHAPTER 2

Learning Program Invariants

In this chapter I extend the data-driven paradigm for *precondition inference*: given a piece of code C along with a predicate Q, the goal is to produce a predicate P whose satisfaction on entry to C is sufficient to ensure that Q holds after C is executed. Data-driven approaches to precondition inference [SCI08, GDV15] employ a machine learning algorithm to separate a set of "good" test inputs (which cause Q to be satisfied) from a set of "bad" ones (which cause Q to be falsified). Therefore, these techniques are quite general: they can infer candidate preconditions regardless of the complexity of C and Q, which must simply be executable.

A key limitation of existing data-driven inference algorithms, however, is the need to provide a set of *features*, which are predicates over the inputs to C (*e.g.* `x > 0`). The learner then searches for a boolean combination of these features that separates the set G of "good" inputs from the set B of "bad" inputs. Existing data-driven precondition inference approaches [SCI08, GDV15] require a fixed set of features to be specified in advance. If these features are not sufficient to separate G and B, the approaches must either **(a)** fail to produce a precondition, **(b)** produce a precondition that is insufficient (satisfying some "bad" inputs), or **(c)** produce a precondition that is overly strong (falsifying some "good" inputs).

In contrast, I show how to iteratively *learn* useful features on demand as part of the precondition inference process, thereby eliminating the problem of feature selection. I have implemented this approach in a tool called PIE (Precondition Inference Engine). Suppose that at some point PIE has produced a set F of features that is not sufficient to separate G and B. Observe that in this case there must be at least one pair of tests that *conflict*: the

tests have identical valuations to the features in F but one test is in G and the other is in B. This leads to a clear criterion for *feature learning*: the goal is to learn a new feature to add to F that resolves a given set of conflicts. PIE employs a form of search-based program synthesis [STB06, SGF10, AGK13] for this purpose, since it can automatically synthesize rich expressions over arbitrary data types. Once all conflicts are resolved in this manner, the boolean learner is guaranteed to produce a precondition that is both sufficient and necessary for the given set of tests.

In addition to making data-driven precondition inference less onerous and more expressive, the feature learning approach naturally applies to other forms of data-driven invariant inference that employ positive and negative examples. To demonstrate this, I have built a novel data-driven algorithm, named LoopInvGen, for inferring provably correct loop invariants. LoopInvGen uses PIE as a subroutine to generate candidate invariants, thereby learning features on demand through conflict resolution. In contrast, all prior data-driven loop invariant inference techniques require a fixed set or template of features to be specified in advance [GLM14, GNM16, SGH13b, SA14, JKW10, KJD10].

I have implemented PIE for OCaml libraries and LoopInvGen for C++ programs. I use these implementations to show and evaluate two distinct uses cases for PIE.[1]

First, PIE can be used in the "black box" setting to aid programmer understanding of third-party code. For example, suppose a programmer wants to understand the conditions under which a given library function throws an exception. PIE can automatically produce a likely precondition for an exception to be thrown, which is guaranteed to be both sufficient and necessary over the set of test inputs that were considered. I evaluate this use case by inferring likely preconditions for the functions in several widely used OCaml libraries. The inferred preconditions match the English documentation in the vast majority of cases and in two cases identify behaviors that are absent from the documentation.

[1] PIE is now exposed as a library within LoopInvGen: https://github.com/SaswatPadhi/LoopInvGen. The original benchmarks and results are still available at https://github.com/SaswatPadhi/PIE.

Second, PIE-based loop invariant inference can be used in the "white box" setting, in conjunction with the standard weakest precondition computation [DDD76], to automatically verify that a program meets its specification. My C++ implementation is able to automatically verify several benchmark programs used in the evaluation of three recent approaches to loop invariant inference [DDL13, SA14, GNM16]. These programs require loop invariants involving both linear and non-linear arithmetic as well as operations on strings. The only prior techniques that have demonstrated such generality require a fixed set or template of features to be specified in advance.

The rest of the chapter is structured as follows. Section 2.1 overviews PIE and LoopInvGen informally by example, and Section 2.2 describes these algorithms precisely. Section 2.3 presents an extensive experimental evaluation, and Section 2.4 compares the presented techniques with prior work in this area.

2.1 Overview

This section describes PIE through a running example. The `sub` function in the `String` module of the OCaml standard library takes a string `s` and two integers `i1` and `i2` and returns a substring of the original one. A caller of `sub` must provide appropriate arguments for a crash-free execution, or else an `Invalid_argument` exception is raised. PIE can be used to automatically infer a predicate that characterizes the set of valid arguments.

My OCaml implementation of precondition inference using PIE takes three inputs: a function `f` of type `'a -> 'b`; a set T of test inputs of type `'a`, which can be generated using any desired method; and a postcondition Q, which is simply a function of type `'a -> 'b result -> bool`. A `'b result` either has the form `Ok v` where `v` is the result value of type `'b` from `f`, or `Exn e` where `e` is the exception thrown by `f`. By executing `f` on each test input in T to obtain a result and then executing Q on each input-result pair, T is partitioned into a set G of "good" inputs that cause Q to be satisfied and a set B of "bad" inputs that

cause Q to be falsified. Finally, PIE is given the sets G and B, with the goal to produce a predicate that separates them. In this running example, the function `f` is `String.sub` and the postcondition Q is the following:

```
fun arg res ->
    match res with
    | Exn (Invalid_argument _) -> false
    | _ -> true
```

As I show in Section 2.3, when given many random inputs generated by the `qcheck` library,[2] PIE-based precondition inference can automatically produce the following precondition for `String.sub` to terminate normally:

```
i1 >= 0 && i2 >= 0 && i1 + i2 <= (length s)
```

Though in this running example the precondition is conjunctive, PIE infers arbitrary conjunctive normal form (CNF) formulas. For example, if the postcondition above is negated, *i.e.* to detect an `Invalid_argument` exception, then PIE will produce the following complementary precondition:

```
i1 < 0 || i2 < 0 ||  i1 + i2 > (length s)
```

2.1.1 Data-Driven Precondition Inference

This subsection reviews the data-driven approach to precondition inference [SCI08, GDV15] in the context of PIE. For purposes of this running example, assume that we are given only the eight test inputs for `sub` that are listed in the first column of Figure 2.1. The induced set G of "good" inputs that cause `String.sub` to terminate normally and set B of "bad" inputs that cause `sub` to raise an exception are shown in the last column of the figure.

[2] https://github.com/c-cube/qcheck

Tests	Features				Set
	i1 < 0	i1 > 0	i2 < 0	i2 > 0	
("pie", 0, 0)	F	F	F	F	G
("pie", 0, 1)	F	F	F	T	G
("pie", 1, 0)	F	T	F	F	G
("pie", 1, 1)	F	T	F	T	G
("pie", -1, 0)	T	F	F	F	B
("pie", 1, -1)	F	T	T	F	B
("pie", 1, 3)	F	T	F	T	B
("pie", 2, 2)	F	T	F	T	B

Figure 2.1: An example of data-driven precondition inference.

Like prior data-driven approaches, PIE separates G and B by reduction to the problem of *learning a boolean formula from examples* [SCI08, GDV15]. This reduction requires a set of *features*, which are predicates on the program inputs that will be used as building blocks for the inferred precondition. As detailed later, PIE's key innovation is the ability to automatically learn features on demand, although PIE also allows users to provide an optional initial set of features.

Suppose that PIE is given the four features shown along the top row of Figure 2.1. Then each test input induces a *feature vector* of boolean values that results from evaluating each feature on that input. For example, the first test induces the feature vector `<F,F,F,F>`. Each feature vector is now interpreted as an assignment to a set of four boolean variables, and the goal is to learn a propositional formula over these variables that satisfies all feature vectors from G and falsifies all feature vectors from B.

There are many algorithms for learning boolean formulas by example. PIE uses a simple but effective *probably approximately correct* (PAC) algorithm that can learn an arbitrary conjunctive normal form (CNF) formula and is biased toward small formulas [KVV94]. The resulting precondition is guaranteed to be both sufficient and necessary for the given test inputs, but there are no guarantees for other inputs.

2.1.2 Feature Learning via Program Synthesis

At this point in the running example, there is a problem: there is no boolean function on the current set of features that is consistent with the given examples! This situation occurs exactly when two test inputs *conflict*: they induce identical feature vectors, but one test is in G while the other is in B. For example, in Figure 2.1 the tests `("pie",1,1)` and `("pie",1,3)` conflict; therefore no boolean function over the given features can distinguish between them.

Prior data-driven approaches to precondition inference require a fixed set of features to be specified in advance. Therefore, whenever two tests conflict they must produce a precondition that violates at least one test. The approach of Sankaranarayanan *et al.* [SCI08] learns a decision tree using the ID3 algorithm [Qui86], which minimizes the total number of misclassified tests. The approach of Gehr *et al.* [GDV15] strives to produce sufficient preconditions and so returns a precondition that falsifies all "bad" tests while minimizing the misclassification of "good" tests.

In the running example, both prior approaches will produce a predicate equivalent to the following one, which misclassifies one "good" test:

```
!(i1 < 0) && !(i2 < 0) && !((i1 > 0) && (i2 > 0))
```

This precondition captures the actual lower-bound requirements on `i1` and `i2`. However, it includes an upper-bound requirement that is both overly restrictive, requiring at least one of `i1` and `i2` to be zero, and insufficient (for some unobserved inputs), since it is satisfied by erroneous inputs such as `("pie",0,5)`. Further, using more tests does not help. On a test suite with full coverage of the possible "good" and "bad" feature vectors, an approach that falsifies all "bad" tests must require both `i1` and `i2` to be zero, obtaining sufficiency but ruling out almost all "good" inputs. The ID3 algorithm will produce a decision tree that is larger than the original one, due to the need for more case splits over the features, and this tree will be either overly restrictive, insufficient, or both.

In contrast to these approaches, PIE uses a form of automatic *feature learning*, which augments the set of features in a targeted manner on demand. The key idea is to leverage the fact that there is a clear criterion for new features — they must resolve conflicts. Therefore, PIE first generates new features to resolve any conflicts, and it then uses the approach described in Section 2.1.1 to produce a precondition that is consistent with all tests.

Let a *conflict group* be a set of tests that induce the same feature vector and that participate in a conflict (*i.e.* at least one test is in G and one is in B). PIE's feature learner uses a form of search-based program synthesis [AGK13, FCD15] to generate a feature that resolves all conflicts in a given conflict group. Given a set of constants and operations for each type of data in the tests, the feature learner enumerates candidate boolean expressions in order of increasing size until it finds one that separates the "good" and "bad" tests in the given conflict group. The feature learner is invoked repeatedly until all conflicts are resolved.

In Figure 2.1, three tests induce the same feature vector and participate in a conflict. Thus, the feature learner is given these three input-output examples: `(("pie",1,1), T)`, `(("pie",1,3), F)`, and `(("pie",2,2), F)`. Various predicates are consistent with these examples, including the "right" one `i1 + i2 <= (length s)` and less useful ones like `i1 + i2 != 4`. However, overly specific predicates are less likely to resolve a conflict group that is sufficiently large; the small conflict group in the running example is due to the use of only eight test inputs. Further, existing synthesis engines bias against such predicates by assigning constants a larger "size" than variables [AGK13].

PIE with feature learning is *strongly convergent*: if there exists a predicate that separates G and B and is expressible in terms of the constants and operations given to the feature learner, then PIE will eventually (ignoring resource limitations) find such a predicate. PIE's search space is limited to predicates that are expressible in the "grammar" given to the feature learner. However, each type typically has a standard set of associated operations, which can be provided once and reused across many invocations of PIE. For each such invocation, feature learning automatically searches an unbounded space of expressions in order to produce

targeted features. For example, the feature `i1 + i2 <= (length s)` for `String.sub` in the running example is automatically constructed from the operations `+` and `<=` on integers and `length` on strings, obviating the need for users to manually craft this feature in advance.

The feature learning approach could itself be used to perform precondition inference in place of PIE, given all tests rather than only those that participate in a conflict. However, as detailed in Section 2.3 the separation of feature learning and boolean learning is critical for scalability. The search space for feature learning is exponential in the maximum feature size, so attempting to synthesize entire preconditions can quickly hit resource limitations. PIE avoids this problem by decomposing precondition inference into two subproblems: **(a)** generating rich features over arbitrary data types, and **(b)** generating a rich boolean structure over a fixed set of black-box features.

2.1.3 Feature Learning for Loop Invariant Inference

The feature learning approach also applies to other forms of data-driven invariant inference that employ positive and negative examples, and hence can have conflicts. To illustrate this, I have built a novel algorithm called LoopInvGen for inferring loop invariants that are sufficient to prove that a program meets its specification. The algorithm employs PIE as a subroutine, thereby learning features on demand as described above. In contrast, all prior data-driven loop invariant inference techniques require a fixed set or template of features to be specified in advance [GLM14, GNM16, SGH13b, SA14, JKW10, KJD10].

```
string sub(string s, int i1, int i2) {
  assume(i1 >= 0 && i2 >= 0 &&
         i1+i2 <= s.length());
  int i = i1;
  string r = "";
  while (i < i1+i2) {
    assert(i >= 0 && i < s.length());
    r = r + s.at(i);
    i = i + 1;
  }
  return r;
}
```

Figure 2.2: A C++ implementation of `sub`.

To continue the running example, suppose that I have inferred a likely precondition for the `sub` function to execute without error and want to verify its correctness for the C++ implementation of `sub` shown in Figure 2.2.[3] As is standard, I use the function `assume(`P`)` to encode the precondition; executions that do not satisfy P are silently ignored. I would like to automatically prove that the assertion inside the `while` loop never fails (which implies that the subsequent access `s.at(i)` is within bounds). However, doing so requires an appropriate loop invariant to be inferred, which involves both integer and string operations. To my knowledge, the only previous technique that has been demonstrated to infer such invariants employs a random search over a fixed set of features [SA14].

In contrast, the LoopInvGen algorithm can infer an appropriate loop invariant without being given any features as input. The algorithm is inspired by the HOLA loop invariant inference engine, a purely static analysis that employs logical abduction via quantifier elimination to generate candidate invariants [DDL13]. LoopInvGenuses a similar approach but does not require the logic of invariants to support quantifier elimination and instead leverages PIE to generate candidates. HOLA's abduction engine generates multiple candidates, and HOLA performs a backtracking search over them. PIE instead generates a single precondition, but I show how to iteratively augment the set of tests given to PIE in order to refine its result. I have implemented LoopInvGen for C++ programs.

The LoopInvGen algorithm has three main components. First, a standard program verifier V for *loop-free* programs: given code C along with a precondition P and postcondition Q, V generates the formula $P \Rightarrow WP(C, Q)$, where WP denotes the weakest precondition [DDD76]. Then V checks validity of this formula by querying an SMT solver that supports the necessary logical theories, which either indicates validity or provides a counterexample.

Second, PIE and the verifier V are used to build the VPreGen algorithm for generating provably sufficient preconditions for loop-free programs, via counterexample-driven refine-

[3] Note that `+` is overloaded as both addition and string concatenation in C++.

ment [CGJ00]. Given code C, a postcondition Q, and test sets G and B, VPreGen invokes PIE to generate a candidate precondition P. If the verifier V proves the sufficiency of P for C and Q, then we terminate. Otherwise, the counterexample from the verifier is incorporated as a new test in the set B, and the process iterates. PIE's feature learning automatically expands the search space of preconditions whenever a new test creates a conflict.

Finally, the LoopInvGen algorithm iteratively invokes VPreGen to produce candidate loop invariants until it finds one that is sufficient to verify the given program. I illustrate LoopInvGen in the running example, where the inferred loop invariant $I(i, i_1, i_2, r, s)$ must satisfy the following three properties:

1. The invariant should hold when the loop is first entered:

$$(i_1 \geq 0 \wedge i_2 \geq 0 \wedge i_1 + i_2 \leq s.\texttt{length()} \wedge \ i = i_1 \wedge r = \text{“”} \ \Rightarrow \ I(i, i_1, i_2, r, s)$$

2. The invariant should be inductive:

$$I(i, i_1, i_2, r, s) \wedge i < i_1 + i_2 \ \Rightarrow \ I(i + 1, i_1, i_2, r + s.\texttt{at}(i), s)$$

3. The invariant should be strong enough to prove the assertion:

$$I(i, i_1, i_2, r, s) \wedge i < i_1 + i_2 \ \Rightarrow \ 0 \leq i < s.\texttt{length()}$$

My example involves both linear arithmetic and string operations, so the program verifier V must use an SMT solver that supports both theories, such as Z3-Str [ZZG13] or CVC4 [LRT14].

To generate an invariant satisfying the above properties, LoopInvGen first asks VPreGen to find a precondition to ensure that the assertion will not fail in the following program, which represents the third constraint above:

```
assume(i < i1 + i2);
assert(0 <= i && i < s.length());
```

Given a sufficiently large set of test inputs, VPreGen generates the following precondition, which is simply a restatement of the assertion itself:

```
0 <= i && i < s.length()
```

While this *candidate* invariant is guaranteed to satisfy the third constraint, an SMT solver can show that it is not inductive. Therefore VPreGen is used again to iteratively strengthen the candidate invariant until it is inductive. For example, in the first iteration, VPreGen is asked to infer a precondition to ensure that the assertion will not fail in the following program:

```
assume(0 <= i && i < s.length());
assume(i < i1 + i2);
r = r + s.at(i);
i = i+1;
assert(0 <= i && i < s.length());
```

This program corresponds to the second constraint above, but with I replaced by the current candidate invariant. VPreGen generates the precondition `i1+i2 <= s.length()` for this program, which is conjoined to the current candidate to obtain a new candidate invariant:

```
0 <= i && i < s.length() && i1+i2 <= s.length()
```

This candidate is inductive, so the iteration stops.

Finally, the verifier is asked if the candidate satisfies the first constraint above. In this case it does, so a valid loop invariant has been found, thereby proving that the code's assertion will never fail. If instead the verifier provides a counterexample, then this counterexample is incorporated as a new test input and the entire process of finding a loop invariant restarts.

2.2 Algorithms

In this section I describe data-driven precondition inference and loop invariant inference algorithms in more detail.

2.2.1 Precondition Inference

Figure 2.3 presents the algorithm for precondition generation using PIE, which I call PreGen. We are given a code snippet C, which is assumed not to make any internal non-deterministic choices, and a postcondition Q, such as an assertion. We are also given a set of test inputs T for C, which can be generated by any means, for example a fuzzer, a symbolic execution engine, or manually written unit tests. The goal is to infer a precondition P such that the execution of C results in a state satisfying Q if and only if it begins from a state satisfying P. In other words, we require a predicate P that satisfies the Hoare triple $\{P\}C\{Q\}$. PreGen guarantees that P will be both sufficient and necessary on the given set of tests T but makes no guarantees for other inputs.

PreGen(C: Code, Q: Predicate, T: Tests) : Predicate
Returns: A precondition that holds over all tests in T

1: Tests (G, B) := partitionTests(C,Q,T)
2: **return** PIE(G,B)

Figure 2.3: Algorithm for precondition generation.

The function partitionTests in Figure 2.3 executes the tests in T in order to partition them into a sequence G of "good" tests, which cause C to terminate in a state that satisfies Q, and a sequence B of "bad" tests, which cause C to terminate in a state that falsifies Q (line 1). The precondition is then obtained by invoking PIE, which is discussed next.

Figure 2.4 describes the overall structure of PIE, which returns a predicate that is consistent with the given set of tests. The initial set F of features is empty, though my implementation optionally accepts an initial set of features from the user (not shown in the

figure). For example, such features could be generated based on the types of the input data, the branch conditions in the code, or by leveraging some knowledge of the domain.

Regardless, PIE then iteratively performs the loop on lines 2 – 9. First it creates a *feature vector* for each test in G and B (lines 3 and 4). The i^{th} element of the sequence V^+ is a sequence that stores the valuation of the features on the i^{th} test in G. Formally, $V^+ = \textsc{makeFV}(F, G) \iff \forall i, j.\, (V_i^+)_j = F_j(G_i)$. Here I use the notation S_k to denote the k^{th} element of the sequence S, and $F_j(G_i)$ denotes the boolean result of evaluating feature F_j on test G_i. V^- is created in an analogous manner given the set B.

PIE(G: Tests, B: Tests)

Returns: A predicate P such that $P(t)$ for all $t \in G$ and $\neg P(t)$ for all $t \in B$

1: Features $F := \{\}$
2: **repeat**
3: FeatureVectors $V^+ :=$ makeFV(F,G)
4: FeatureVectors $V^- :=$ makeFV(F,B)
5: Conflict $X :=$ getConflict(V^+, V^-, G, B)
6: **if** $X \neq$ `None` **then**
7: $F := F \cup$ FeatureLearn(X)
8: **until** $X =$ `None`
9: $\phi :=$ BoolLearn(V^+, V^-)
10: **return** substitute(F, ϕ)

Figure 2.4: The core PIE algorithm.

A feature vector v is said to be a *conflict* if it appears in both V^+ and V^-, *i.e.* $\exists i, j. V_i^+ = V_j^- = v$. The function getConflict returns `None` if there are no conflicts. Otherwise it selects one conflicting feature vector v and returns a pair of sets $X = (X^+, X^-)$, where X^+ is a subset of G whose associated feature vector is v and X^- is a subset of B whose associated feature vector is v. Next PIE invokes the feature learner on X, which uses a form of program synthesis to produce a new feature f such that $\forall t \in X^+.\, f(t)$ and $\forall t \in X^-.\, \neg f(t)$. This new feature is added to the set F of features, thus resolving the conflict.

The above process iterates, identifying and resolving conflicts until there are no more. PIE then invokes the function BoolLearn, which learns a propositional formula ϕ over $|F|$ variables such that $\forall v \in V^+.\, \phi(v)$ and $\forall v \in V^-.\, \neg\phi(v)$. Finally, the precondition is created by substituting each feature for its corresponding boolean variable in ϕ.

Discussion Before describing the algorithms for feature learning and boolean learning, Note some important aspects of the overall algorithm. First, like prior data-driven approaches, PreGen and PIE are very general. The only requirement on the code C in Figure 2.3 is that it be executable, in order to partition T into the sets G and B. The code itself is not even an argument to the function PIE. Therefore, PreGen can infer preconditions for any code, regardless of how complex it is. For example, the code can use idioms that are hard for automated constraint solvers to analyze, such as non-linear arithmetic, intricate heap structures with complex sharing patterns, reflection, and native code. Indeed, the source code itself need not even be available. The postcondition Q similarly must simply be executable and so can be arbitrarily complex.

Second, PIE can be viewed as a hybrid of two forms of precondition inference. Prior data-driven approaches to precondition inference [SCI08, GDV15] perform boolean learning but lack feature learning, which limits their expressiveness and accuracy. On the other hand, a feature learner based on program synthesis [STB06, SGF10, AGK13] can itself be used as a precondition inference engine without boolean learning, but the search space grows exponentially with the size of the required precondition. PIE uses feature learning only to resolve conflicts, leveraging the ability of program synthesis to generate expressive features over arbitrary data types, and then uses boolean learning to scalably infer a concise boolean structure over these features.

Due to this hybrid nature of PIE, a key parameter in the algorithm is the maximum number c of conflicting tests to allow in the conflict group X at line 5 in Figure 2.4. If the conflict groups are too large, then too much burden is placed on the feature learner, which limits scalability. For example, a degenerate case is when the set of features is empty, in which case all tests induce the empty feature vector and are in conflict. Therefore, if the set of conflicting tests that induce the same feature vector has a size greater than c, a random subset of size c is chosen to provide to the feature learner. In Section 2.3 different values for c are empirically evaluated.

Feature Learning Figure 2.5 describes my approach to feature learning. The algorithm is a simplified version of the ESCHER program synthesis tool [AGK13], which produces functional programs from examples. Like ESCHER, it requires a set of *operations* for each type of input data, which are used as building blocks for synthesized features. By default, FEATURELEARN includes operations for primitive types and for lists. For example, integer operations include `0` (a nullary operation), `+`, and `<=`, while list operations include `[]`, `::`, and `length`. Users can easily add their own operations, for these as well as other types of data.

FEATURELEARN(X^+: Tests, X^-: Tests)

Returns: A feature f such that $f(t)$ for all $t \in X^+$ and $\neg f(t)$ for all $t \in X^-$

1: Operations O := GETOPERATIONS()
2: Integer $i := 1$
3: **loop**
4: Features F := FEATURESOFSIZE(i, O)
5: **if** $\exists f \in F. (\forall t \in X^+. f(t) \wedge \forall t \in X^-. \neg f(t))$ **then**
6: **return** f
7: $i := i + 1$

Figure 2.5: The feature learning algorithm.

Given this set of operations, FEATURELEARN simply enumerates all possible features in order of the size of their abstract syntax trees. Before generating features of size $i + 1$, it checks whether any feature of size i completely separates the tests in X^+ and X^-; if so, that feature is returned. The process can fail to find an appropriate feature, either because no such feature over the given operations exists or because resource limitations are reached; either way, this causes the PIE algorithm to fail.

Despite the simplicity of this algorithm, it works well in practice, as I show in Section 2.3. Enumerative synthesis is a good match for learning features, since it biases toward small features, which are likely to be more general than large features and so helps to prevent against overfitting. Further, the search space is significantly smaller than that of traditional program synthesis tasks, since features are simple expressions rather than arbitrary programs. For example, the algorithm does not attempt to infer control structures such as conditionals, loops, and recursion, which is a technical focus of much program-synthesis research [AGK13, FCD15].

Boolean Function Learning I use a standard algorithm for learning a small CNF formula that is consistent with a given set of boolean feature vectors [KVV94]; it is described in Figure 2.6. Recall that a CNF formula is a conjunction of *clauses*, each of which is a disjunction of *literals*. A literal is either a propositional variable or its negation. The algorithm returns a CNF formula over a set $x_1, \ldots, x_n$ of propositional variables, where n is the size of each feature vector (line 1). The algorithm first attempts to produce a 1-CNF formula (*i.e.* a conjunction), and it increments the maximum clause size k iteratively until a formula is found that is consistent with all feature vectors. Since BoolLearn is only invoked once all conflicts have been removed (see Figure 2.4), this process is guaranteed to succeed eventually. Given a particular value of k, the learning algorithm first generates a set C of all clauses of size k or smaller over $x_1, \ldots, x_n$ (line 4), implicitly representing the conjunction of these clauses. In line 5, all clauses that are inconsistent with at least one of the "good" feature vectors (*i.e.* one of the vectors in V^+) are removed from C. A clause c is inconsistent with a "good" feature vector v if v falsifies c, formally denoted as:

BoolLearn(V^+: Feature Vectors, V^-: Feature Vectors)
Returns: A formula ϕ such that $\phi(v)$ for all $v \in V^+$ and $\neg\phi(v)$ for all $v \in V^-$

1: Integer n := size of each vector in V^+ and V^-
2: Integer $k := 1$
3: **loop**
4: Clauses C := allClausesUpToSize(k, n)
5: C := filterInconsistentClauses(C, V^+)
6: C := greedySetCover(C, V^-)
7: **if** $C \neq$ `None` **then return** C
8: $k := k + 1$

Figure 2.6: The boolean function learning algorithm.

$$\forall 1 \leq i \leq n.(x_i \in c \Rightarrow v_i = \texttt{false}) \wedge (\neg x_i \in c \Rightarrow v_i = \texttt{true})$$

At line 6, C is the strongest k-CNF formula that is consistent with all "good" feature vectors.

Finally, line 6 weakens C while still falsifying all of the "bad" feature vectors (*i.e.* the vectors in V^-). In particular, the goal is to identify a minimal subset C' of C where for each $v \in V^-$, there exists $c \in C'$ such that v falsifies c. This problem is equivalent to the classic *minimum set cover* problem, which is NP-complete. Therefore, the GREEDYSETCOVER function on line 6 uses a standard heuristic for that problem, iteratively selecting the clause that is falsified by the most "bad" feature vectors that remain, until all such feature vectors are "covered." This process will fail to cover all "bad" feature vectors if there is no k-CNF formula consistent with V^+ and V^-, in which case k is incremented; otherwise the resulting set C is returned as the CNF formula.

Because the boolean learner treats features as black boxes, this algorithm is unaffected by their sizes. Rather, the search space is $O(n^k)$, where n is the number of features and k is the maximum clause size, and in practice k is a small constant. Though I have found this algorithm to work well in practice, there are many other algorithms for learning boolean functions from examples. As long as they can learn arbitrary boolean formulas, then I expect that they would also suffice.

Properties As described above, the precondition returned by PIE is guaranteed to be both necessary and sufficient for the given set of test inputs. Furthermore, PIE is *strongly convergent*: if there exists a predicate that separates G and B and is expressible in terms of the constants and operations given to the feature learner, then PIE will eventually (ignoring resource limitations) find and return such a predicate.

To see why PIE is strongly convergent, note that FEATURELEARN (Figure 2.5) performs an exhaustive enumeration of possible features. By assumption a predicate that separates G and B is expressible in the language of the feature learner, and that predicate also separates any sets X^+ and X^- of conflicting tests, since they are respectively subsets of G and B. Therefore each call to FEATURELEARN on line 7 in Figure 2.4 will eventually succeed, reducing the number of conflicting tests and ensuring that the loop at line 2 eventually terminates. At

that point, there are no more conflicts, so there is some CNF formula over the features in F that separates G and B, and the boolean learner will eventually find it.

2.2.2 Loop Invariant Inference

As described in Section 2.1.3, my loop invariant inference engine relies on an algorithm VPreGen that generates provably sufficient preconditions for loop-free code. The VPreGen algorithm is shown in Figure 2.7. In the context of loop invariant inference (see below), VPreGen will always be passed a set of "good" tests to use and will start with no "bad" tests, so I specialize the algorithm to that setting. The VPreGen algorithm assumes the existence of a verifier for loop-free programs. If the verifier can prove the sufficiency of a candidate precondition P generated by PIE (lines 3-4), it returns `None` and we are done. Otherwise the verifier returns a counterexample t, which has the property that $P(t)$ is true but executing C on t ends in a state that falsifies Q. Therefore t is added to the set B of "bad" tests and the algorithm iterate.

VPreGen(C: Code, Q: Predicate, G: Tests)
Returns: A precondition P such that $P(t)$ for all t in G and $\{P\}C\{Q\}$ holds

1: Tests $B := \{\}$
2: **repeat**
3: $\quad P :=$ PIE(G,B)
4: $\quad t :=$ verify(P,C,Q)
5: $\quad B := B \cup \{t\}$
6: **until** $t =$ `None`
7: **return** P

Figure 2.7: Verified precondition generation.

The LoopInvGen algorithm for loop invariant inference is shown in Figure 2.8. For simplicity, I restrict the presentation to code snippets of the form

$$C = \texttt{assume}\ P\ ;\ \texttt{while}\ E\ \{C_{\text{LF}}\}\ ;\ \texttt{assert}\ Q$$

LoopInvGen(C: Code, T: Tests)
Returns: A loop invariant that is sufficient to verify that C's assertion never fails.

Require: $C = \texttt{assume}\ P\ ;\ \texttt{while}\ E\ \{C_{\text{LF}}\}\ ;\ \texttt{assert}\ Q$

1: $G :=$ loopHeadStates(C, T)
2: **loop**
3: $\quad I :=$ VPreGen([assume $\neg E$], Q, G)
4: $\quad$ **while** $\neg\{I \wedge E\}C_{\text{LF}}\{I\}$ **do**
5: $\quad\quad I' :=$ VPreGen([assume $I \wedge E; C_{\text{LF}}$], I, G)
6: $\quad\quad I := I \wedge I'$
7: $\quad t :=$ Valid($P \Rightarrow I$)
8: $\quad$ **if** $t =$ None **then return** I
9: $\quad G := G \cup$ loopHeadStates(C, $\{t\}$)

Figure 2.8: Loop invariant inference using PIE.

where C_{LF} is loop-free. My implementation also handles code with multiple and nested loops, by iteratively inferring invariants for each loop encountered in a backward traversal of the program's control-flow graph.

The goal of LoopInvGen is to infer a loop invariant I which is sufficient to prove that the Hoare triple $\{P\}(\texttt{while}\ E\{C_{\text{LF}}\})\{Q\}$ is valid. In other words, the invariant I must satisfy the following three constraints:

$$\begin{array}{rcl} P & \Rightarrow & I \\ \{I \wedge E\} & C_{\text{LF}} & \{I\} \\ I \wedge \neg E & \Rightarrow & Q \end{array}$$

Given a test suite T for C, LoopInvGen first generates a set of tests for the loop by logging the program state every time the loop head is reached (line 1). In other words, if $\vec{x}$

denotes the set of program variables the following instrumented version of C is executed on each test in T:

$$\texttt{assume}\ P\ ;\ \texttt{log}\ \vec{x}\ ;\ \texttt{while}\ E\ \{C_{\text{LF}}\ ;\ \texttt{log}\ \vec{x}\}\ ;\ \texttt{assert}\ Q$$

If the Hoare triple $\{P\}(\texttt{while}\ E\{C_{\text{LF}}\})\{Q\}$ is valid, then all test executions are guaranteed to pass the assertion, so all logged program states will belong to the set G of passing tests. If a test fails the assertion then no valid loop invariant exists, and the algorithm aborts (not shown in the figure).

With this new set G of tests, LoopInvGen first generates a candidate invariant that meets the third constraint above by invoking VPreGen on line 3. The inner loop (lines 4-7) then strengthens I until the second constraint is met. If the generated candidate also satisfies the first constraint (line 8), then a sufficient invariant has been found. Otherwise a counterexample t satisfying $P \wedge \neg I$ is obtained, which is used to collect new program states as additional tests (line 12), and the process iterates. The verifier for loop-free code is used on lines 3 (inside VPreGen), 4 (to check the Hoare triple), and 5 (inside VPreGen), and the underlying SMT solver is used on line 8 (the validity check).

Note the interplay of strengthening and weakening in the LoopInvGen algorithm. Each iteration of the inner loop strengthens the candidate invariant until it is inductive. However, each iteration of the outer loop uses a larger set G of passing tests. Because PIE is guaranteed to return a precondition that is consistent with all tests, the larger set G has the effect of weakening the candidate invariant. In other words, candidates get strengthened, but if they become stronger than P in the process then they will be weakened in the next iteration of the outer loop.

Properties Both the VPreGen and LoopInvGen algorithms are *sound*: VPreGen(C, Q, G) returns a precondition P such that $\{P\}C\{Q\}$ holds, and LoopInvGen(C, T) returns

a loop invariant I that is sufficient to prove that $\{P\}(\texttt{while}\ E\ \{C_{\text{LF}}\})\{Q\}$ holds, where $C = \texttt{assume}\ P\ ;\ \texttt{while}\ E\ \{C_{\text{LF}}\}\ ;\ \texttt{assert}\ Q$. However, neither algorithm is guaranteed to return the weakest such predicate.

VPreGen(C, Q, G) is *strongly convergent*: if there exists a precondition P that is expressible in the language of the feature learner such that $\{P\}C\{Q\}$ holds and $P(t)$ holds for each $t \in G$, then VPreGen will eventually find such a precondition.

To see why, first note that by assumption each test in G satisfies P, and since $\{P\}C\{Q\}$ holds, each test that will be put in B at line 5 in Figure 2.7 falsifies P (since each such test causes Q to be falsified). Therefore P is a separator for G and B, so each call to PIE at line 3 terminates due to the strong convergence result described earlier. Suppose P has size s. Then each call to PIE from VPreGen will generate features of size at most s, since P itself is a valid separator for any set of conflicts. Further, each call to PIE produces a logically distinct precondition candidate, since each call includes a new test in B that is inconsistent with the previous candidate. Since the feature learner has a finite number of operations for each type of data, there are a finite number of features of size at most s and so also a finite number of logically distinct boolean functions in terms of such features. Hence eventually P or another sufficient precondition will be found.

LoopInvGen is not strongly convergent: it can fail to terminate even when an expressible loop invariant exists. First, the iterative strengthening loop (lines 4-7 of Figure 2.8) can generate a VPreGen query that has no expressible solution, causing VPreGen to diverge. Second, an adversarial sequence of counterexamples from the SMT solver (line 9 of Figure 2.8) can cause LoopInvGen's outer loop to diverge. Nonetheless, my experimental results below indicate that the algorithm performs well in practice.

2.3 Evaluation

I have evaluated PIE's ability to infer preconditions for black-box OCaml functions and LoopInvGen's ability to infer sufficient loop invariants for verifying C++ programs.

2.3.1 Precondition Inference

Experimental Setup I have implemented the PreGen algorithm described in Figure 2.3 in OCaml. I use PreGen to infer preconditions for all of the first-order functions in three OCaml modules: `List` and `String` from the standard library, and `BatAvlTree` from the widely used `batteries` library.[4] My test generator and feature learner do not handle higher-order functions. For each function, I generate preconditions under which it raises an exception. Further, for functions that return a list, string, or tree, I generate preconditions under which the result value is empty when it returns normally. Similarly, for functions that return an integer (boolean) I generate preconditions under which the result value is `0` (`false`) when the function returns normally. A recent study finds that roughly 75% of manually written specifications are predicates like these, which relate to the presence or absence of data [SDC14].

For feature learning I use a simplified version of the Escher program synthesis tool [AGK13] that follows the algorithm described in Figure 2.5. Escher already supports operations on primitive types and lists; I augment it with operations for strings (*e.g.* `get`, `has`, `sub`) and AVL trees (*e.g.* `left_branch`, `right_branch`, `height`). For the set T of tests, I generate random inputs of the right type using the `qcheck` OCaml library. Analogous to the *small scope hypothesis* [Jac12], which says that "small inputs" can expose a high proportion of program errors, I find that generating many random tests over a small domain exposes a wide range of program behaviors. For my tests I generate random integers in the range $[-4, 4]$, lists of length at most 5, trees of height at most 5 and strings of length at most 12.

[4] http://batteries.forge.ocamlcore.org

In total I attempt to infer preconditions for 101 function-postcondition pairs. Each attempt starts with no initial features and is allowed to run for at most one hour and use up to 8 GB of memory. Two key parameters to my algorithm are the number of tests to use and the maximum size of conflict groups to provide the feature learner. Empirically I have found 6400 tests and conflict groups of maximum size 16 to provide good results (see below for an evaluation of other values of these parameters).

Results Under the configuration described above, PreGen generates correct preconditions in 87 out of 101 cases. By "correct" I mean that the precondition fully matches the English documentation, and possibly captures actual behaviors not reflected in that documentation. The latter happens for two `BatAvlTree` functions: the documentation does not mention that `split_leftmost` and `split_rightmost` will raise an exception when passed an empty tree.

Table 2.1 shows some of the more interesting preconditions that PreGen inferred, along with the number of synthesized features for each. For example, it infers an accurate precondition for `String.index_from(`s`,`i`,`c`)`, which returns the index of the first occurrence of character c in string s after position i, through a rich boolean combination of arithmetic and string functions. As another example, PreGen automatically discovers the definition of a balanced tree, since `BatAvlTree.create` throws an exception if the resulting tree is not balanced. Prior approaches to precondition inference [SCI08, GDV15] can only capture these preconditions if they are provided with exactly the right features (*e.g.* `height`$(t_1) >$ (`height`$(t_2) + 1)$) in advance, while PreGen learns the necessary features on demand.

The 14 cases that either failed due to time or memory limits or that produce an incorrect or incomplete precondition were of three main types. The majority (10 out of 14) require universally quantified features, which are not supported by my feature learner. For example, `List.flatten(`l`)` returns an empty list when each of the inner lists of l is empty. In a few cases the inferred precondition is incomplete due to my use of small integers as test inputs. For example, I do not infer that `String.make(`i`,`c`)` throws an exception if i is greater

Case	Postcondition	Learned Features
`String` module functions		
`set`(s,i,c)	throws exception	3
	$(i < 0) \vee (\texttt{len}(s) \leq i)$	
`sub`(s,i_1,i_2)	throws exception	3
	$(i_1 < 0) \vee (i_2 < 0) \vee\ (i_1 > \texttt{len}(s) - i_2)$	
`index`(s,c)	result $= 0$	2
	$\texttt{has}(\texttt{get}(s, 0), c)$	
`index_from`(s,i,c)	throws exception	4
	$(i < 0) \vee (i > \texttt{len}(s))\ \vee \neg\ \texttt{has}(\texttt{sub}(s, i, \texttt{len}(s) - i), c)$	
`List` module functions		
`nth`(l, n)	throws exception	2
	$(0 > n) \vee (n \geq \texttt{len}(l))$	
`append`(l_1, l_2)	`empty`(result)	2
	$\texttt{empty}(l_1) \wedge \texttt{empty}(l_2)$	
`BatAvlTree` module functions		
`create` (t_1, v, t_2)	throws exception	6
	$\texttt{height}(t_1) > (\texttt{height}(t_2) + 1) \vee \texttt{height}(t_2) > (\texttt{height}(t_1) + 1)$	
`concat`(t_1, t_2)	`empty`(result)	2
	$\texttt{empty}(t_1) \wedge \texttt{empty}(t_2)$	

Table 2.1: A sample of inferred preconditions for OCaml library functions.

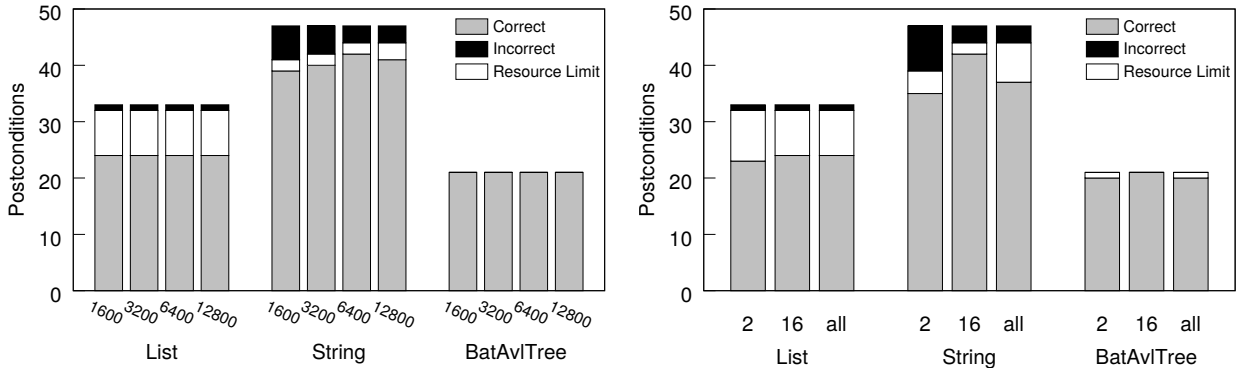


Figure 2.9: Comparison of PIE configurations. The *left* plot shows the effect of different numbers of tests. The *right* plot shows the effect of different conflict group sizes.

than `Sys.max_string_length`. Finally, a few cases produce erroneous specifications for list functions that employ physical equality, *e.g.* `List.memq`. My tests for lists only use primitives as elements, so they cannot distinguish physical from structural equality.

Configuration Parameter Sensitivity I also evaluated PIE's sensitivity to the number of tests and the maximum conflict group size. The left plot in Figure 2.9 shows the results with varied numbers of tests (and conflict group size of 16). In general, the more tests I use, the more correct my results. However, with 12,800 tests I incur one additional case that hits resource limits due to the extra overhead involved.

The right plot in Figure 2.9 shows the results with varied conflict group sizes (and 6400 tests). On the one hand, I can give the feature learner only a single pair of conflicting tests at a time. As the figure shows, this leads to more cases hitting resource limits and producing incorrect results versus a conflict group size of 16, due to the higher likelihood of synthesizing overly specific features. On the other hand, the feature learner can be given all conflicting tests at once. When starting with no initial features, all tests are in conflict, so this strategy requires the feature learner to synthesize the entire precondition. As the figure shows, this approach hits resource limitations more often versus a conflict group size of 16.

	Size of Features	Number of Features	
		EAGER	PIE
`Min`	2	13	1
Q_1	3	29.25	1
Q_2	4	55	1
Q_3	5.25	541.50	2
`Max`	13	18051	5
`Mean`	4.54	1611.80	1.50
`SDev`	2.65	4055.50	0.92

Table 2.2: Comparison of PIE with an approach that uses eager feature learning. The size of a feature is the number of nodes in its abstract syntax tree. Each Q_i indicates the i^{th} quartile, computed independently for each column.

For example, this approach fails to generate the preconditions for `String.index_from` and `BatAvlTree.create` shown in Table 2.1. Further, in the cases that do succeed, the average running time and memory consumption are 11.7 second and 309 MB, as compared to only 1.8 seconds and 66 MB when the conflict group size is 16.

Comparison With Eager Feature Learning PIE generates features *lazily* as necessary to resolve conflicts. An alternative approach is to use Escher up front to eagerly generate every feature for a given program up to some maximum feature size s. These features can then simply all be passed to the boolean learner. To evaluate this approach, I instrumented PIE to count the number of candidate features that were generated by ESCHER each time it was called.[5] For each call to PIE, the maximum such number across all calls to ESCHER is a lower bound, and therefore a best-case scenario, for the number of features that would need to be passed to the boolean learner in an eager approach. It is a lower bound for two

[5] All candidates generated by ESCHER are both type-correct and exception-free, *i.e.*they do not throw exceptions on any test inputs.

reasons. First, we are assuming that the user can correctly guess the maximum size s of features to generate in order to produce a precondition that separates the "good" and "bad" tests. Second, Escher stops generating features as soon as it finds one that resolves the given conflicts, so in general there will be many features of size s that are not counted.

Table 2.2 shows the results for the 52 cases in my experiment above where PIE produces a correct answer and at least one feature is generated. Notably, the minimum number of features generated in the eager approach (13) is more than double the maximum number of features selected in my approach (5). Nonetheless, for functions that require only simple preconditions, eager feature learning is reasonably practical. For example, 25% of the preconditions (`Min` to Q_1 in the table) require 29 or fewer features. However, the number of features generated by eager feature learning grows exponentially with their maximum size. For example, the top 25% of preconditions (from Q_3 to `Max`) require a minimum of 541 features to be generated and a maximum of more than $18,000$. Since boolean learning is in general super-linear in the number n of features (the algorithm I use is $O(n^k)$ where k is the maximum clause size), I expect an eager approach to hit resource limits as the preconditions become more complex.

2.3.2 Loop Invariants for C++ Code

I have implemented the loop invariant inference procedure described in Figure 2.8 for C++ code as a CLANG tool.[6] As mentioned earlier, my implementation supports multiple and nested loops. I have also implemented a verifier for loop-free programs using the CVC4 [BCD11] and Z3-STR [ZZG13] SMT solvers, which support several logical theories including both linear and non-linear arithmetic and strings. I employ both solvers because their support for both non-linear arithmetic and strings is incomplete, causing some queries to fail to terminate. I therefore run both solvers in parallel for two minutes and fail if neither returns a result in that time.

[6] http://clang.llvm.org/docs/LibTooling.html

Case	Calls to Solvers	Calls to Escher	Sizes of Invariants	Analysis Time
HOLAbenchmarks [DDL13]				
01	7	3	11	21
02	43	17	15	27
03	31	22	3,7,15	46
04	4	2	7	18
05	7	3	11	23
06	51	26	9,9	54
07	111	45	19	116
08	4	2	7	18
09	27	18	3,15,7,22	40
10	14	7	28	21
11	21	18	15	22
12	33	19	13,15	45
13	46	30	33	54
14	9	9	31	22
15	26	27	33	39
16	9	10	11	22
17	22	17	15,7	31
18	6	5	15	20
19	18	14	27	32
20	61	24	33	115
21	23	10	19	23
22	16	11	13	22
23	10	7	11	21
24	29	19	1,7,11	40
25	83	47	11,19	142
26	90	32	9,9,9	71
27	32	20	7,3,7	44
28	7	2	3,3	20
29	66	19	11,11	47
30	18	12	35	29
31	-	-	-	-
32	-	-	-	-
33	-	-	-	-
34	30	20	37	25
35	5	4	11	18
36	128	36	11,15,19,11	113
37	13	11	19	22
38	44	38	29	36
39	10	5	11	20
40	30	24	19,17	40
41	17	11	15	27
42	25	15	50	37
43	4	2	7	19
44	14	14	20	26
45	60	33	11,9,9	64
46	12	5	21	24

Case	Calls to Solvers	Calls to Escher	Sizes of Invariants	Analysis Time
Linear arithmetic benchmarks from ICE [GNM16]				
afnp	12	7	11	22
cegar1	16	11	12	30
cegar2	13	11	19	23
cggmp	34	22	143	32
countud	6	4	9	17
dec	4	2	3	17
dillig01	7	3	11	19
dillig03	14	7	15	29
dillig05	7	3	11	21
dillig07	8	4	11	21
dillig12	32	18	13,11	44
dillig15	31	35	55	42
dillig17	24	22	19,11	31
dillig19	19	13	31	32
dillig24	29	18	1,3,11	40
dillig25	57	31	11,19	74
dillig28	7	2	3,3	19
dtuc	9	2	3,3	22
fig1	4	2	7	17
fig3	7	5	7	17
fig9	7	2	7	19
formula22	12	11	16	25
formula25	11	5	15	21
formula27	23	5	19	25
inc2	4	2	7	18
inc	4	2	7	17
loops	19	12	7,7	28
sum1	16	14	21	22
sum3	6	1	3	20
sum4c	41	21	38	32
sum4	6	3	9	17
tacas6	9	8	11	22
trex1	6	2	7,1	19
trex3	9	7	7	23
w1	4	2	7	17
w2	-	-	-	-
Non-linear arithmetic benchmarks from ICE [GNM16]				
multiply	25	19	15	41
sqrt	-	-	-	-
square	11	7	5	24
String benchmarks [SA14]				
a	66	22	110	45
b	6	5	4	10
c	7	4	8	11
d	7	3	9	11

Table 2.3: Experimental results for LoopInvGen. An invariant's size is the number of nodes in its abstract syntax tree. The analysis time is in seconds.

(HOLA) 07
$I : (b = 3i - a) \wedge (n > i \vee b = 3n - a)$
(HOLA) 22
$I : (k = 3y) \wedge (x = y) \wedge (x = z)$
(ICE linear) dillig12
$I_1 : (a = b) \wedge (t = 2s \vee \texttt{flag} = 0) \qquad I_2 : (x \leq 2) \wedge (y < 5)$
(ICE linear) sum1
$I : (i = sn + 1) \wedge (sn = 0 \vee sn = n \vee n \geq i)$
(Strings) c
$I : \texttt{has}(r, \text{“}a\text{”}) \wedge (\texttt{len}(r) > i)$
(ICE non-linear) multiply
$I : (s = y * j) \wedge (x > j \vee s = x * y)$

Table 2.4: A sample of inferred invariants for C++ benchmarks.

I use the same implementation and configuration for PIE as in the previous experiment. To generate the tests I employ an initial set of 256 random inputs of the right type. As described in Section 2.2.2, the algorithm then captures the values of all variables whenever control reaches the loop head, and I retain at most 6400 of these states.

I evaluate my loop invariant inference engine on multiple sets of benchmarks; the results are shown in Table 2.3 and a sample of the inferred invariants is shown in Table 2.4. First, I have used LoopInvGen on all 46 of the benchmarks that were used to evaluate the HOLA loop invariant engine [DDL13]. These benchmarks require loop invariants that involve only the theory of linear arithmetic. Table 2.3 shows each benchmark's name from the original benchmark set, the number of calls to the SMT solvers, the number of calls to the feature learner, the size of the generated invariant, and the running time of LoopInvGen in seconds. LoopInvGen succeeds in inferring invariants for 43 out of 46 HOLA benchmarks, including three benchmarks which HOLA's technique cannot handle (cases 15, 19, and 34).

By construction, these invariants are sufficient to ensure the correctness of the assertions in these benchmarks. The three cases on which LoopInvGen fails run out of memory during PIE's CNF learning phase.

Second, I have used LoopInvGen on 39 of the benchmarks that were used to evaluate the ICE loop invariant engine [GNM16, GLM14]. The remaining 19 of their benchmarks cannot be evaluated with LoopInvGen because they use language features that my program verifier does not support, notably arrays and recursion. As shown in Table 2.3, I succeed in inferring invariants for 35 out of the 36 ICE benchmarks that require linear arithmetic. LoopInvGen infers the invariants fully automatically and with no initial features, while ICE requires a fixed template of features to be specified in advance. The one failing case is due to a limitation of the current implementation — I treat boolean values as integers, which causes PIE to consider many irrelevant features for such values.

I also evaluated LoopInvGen on the three ICE benchmarks whose invariants require non-linear arithmetic. Doing so simply required allowing the feature learner to generate non-linear features; such features were disabled for the above tests due to the SMT solvers' limited abilities to reason about non-linear arithmetic. LoopInvGen was able to generate sufficient loop invariants for two out of the three benchmarks. My approach fails on the third benchmark because both SMT solvers fail to terminate on a particular query. However, this is a limitation of the solvers rather than of LoopInvGen; indeed, if I vary the conflict-group size, which leads to different SMT queries, then LoopInvGensucceeds on this benchmark.

Third, I have evaluated my approach on the four benchmarks whose invariants require both arithmetic and string operations that were used to evaluate another recent loop invariant inference engine [SA14]. As shown in Table 2.3, my approach infers loop invariants for all of these benchmarks. The prior approach [SA14] requires both a fixed set of features and a fixed boolean structure for the desired invariants, neither of which is required by my approach.

Finally, I ran all of the above experiments again, but with PIE replaced by my program-synthesis-based feature learner. This version succeeds for only 61 out of the 89 benchmarks.

Further, for the successful cases, the average running time is 567 s and 1895 MB of memory, versus 28 s and 128 MB (with 573 MB peak memory usage) for the PIE-based approach.

2.4 Related Work

I compare my work against three forms of specification inference in the literature. First, there are several existing approaches to inferring preconditions given a piece of code and a postcondition. Closest to PIE are the prior data-driven approaches. Sankaranarayanan *et al.* [SCI08] uses a decision-tree learner to infer preconditions from good and bad examples. Gehr *et al.* also uses a form of boolean learning from examples, in order to infer conditions under which two functions commute [GDV15]. As discussed in Section 2.1, the key innovation of PIE over these works is its support for on-demand feature learning, instead of requiring a fixed set of features to be specified in advance. In addition to eliminating the problem of feature selection, PIE's feature learning ensures that the produced precondition is both sufficient and necessary for the given set of tests, which is not guaranteed by the prior approaches. Recently Astorga *et al.* have also proposed learning *stateful preconditions* (*i.e.* those that not only constrain primitive-type inputs but also non-primitive-type object states) [AMS19]. Their approach guarantees *maximality* with respect to a given test generator. Astorga *et al.* have also proposed a precondition learning approach [ASX18] that uses symbolic analysis. However, unlike PIE, both of these approaches use a small fixed set of features, and require white-box access to the source code.

There are also several static approaches to precondition inference. These techniques can provide provably sufficient (or provably necessary [CCF13]) preconditions. However, unlike data-driven approaches, they all require the source code to be available and statically analyzable. The standard weakest precondition computation infers preconditions for loop-free programs [DDD76]. For programs with loops, a backward symbolic analysis with search heuristics can yield preconditions [CC77a, CFS09]. Other approaches leverage properties

of particular language paradigms [Gia98], require logical theories that support quantifier elimination [Moy08, DD13], and employ counterexample-guided abstraction refinement (CEGAR) [CGJ00] with domain-specific refinement heuristics [SK13, SS14]. Finally, some static approaches to precondition inference target specific program properties, such as predicates about the heap structure [LSR07, CDO11] or about function equivalence [KLR10].

Second, I have shown how PIE can be used to build a novel data-driven algorithm LoopInvGen for inferring loop invariants that are sufficient to prove that a program meets its specification. Several prior data-driven approaches exist for this problem [JKW10, KJD10, GLM14, GNM16, KPW15, SNA12, SGH13b, SA14, SSC15, GFM14]. As above, the key distinguishing feature of LoopInvGen relative to this work is its support for feature learning. Other than one exception [SNA12], which uses support vector machines (SVMs) [CV95] to learn new numerical features, all prior works employ a fixed set or template of features. In addition, some prior approaches can only infer restricted forms of boolean formulas [SNA12, SGH13b, SA14, SSC15], while LoopInvGen learns arbitrary CNF formulas. The ICE approach [GLM14] requires a set of "implication counterexamples" in addition to good and bad examples, which necessitates new algorithms for learning boolean formulas [GNM16]. In contrast, LoopInvGen can employ any off-the-shelf boolean function learner. Unlike LoopInvGen, ICE is strongly convergent [GLM14]: it restricts invariant inference to a finite set of candidate invariants that is iteratively enlarged using a dovetailing strategy that eventually covers the entire search space. Finally, several other data-driven invariant inference approaches have been proposed recently that use neural network learning [SDR18], interpolation between forward and backward reachability [LSX17], and solving the more general constrained-horn-clause (CHC) systems [END18, ZMJ18].

There are also many static approaches to invariant inference. The HOLA [DDL13] loop invariant generator is based on an algorithm for logical abduction [DD13]; I employed a similar technique to turn PIE into a loop invariant generator. HOLA requires the underlying logic of invariants to support quantifier elimination, while LoopInvGen has no such restriction.

Standard invariant generation tools that are based on abstract interpretation [CH78, CC77a], constraint solving [GMR09, CSS03], or probabilistic inference [GJ07] require the number of disjunctions to be specified manually. Other approaches [FL10, FR94, GR07, GSV08, SIS06, MR05, GIB12] can handle disjunctions but restrict their number via trace-based heuristics, custom built abstract domains, or widening. In contrast, LoopInvGen places no *a priori* bound on the number of disjunctions.

Third, there has been prior work on data-driven inference of specifications given only a piece of code as input. For example, Daikon [EPG07] generates likely invariants at various points within a given program. Other work leverages Daikon to generate candidate specifications and then uses an automatic program verifier to validate them, eliminating the ones that are not provable [NE02a, NE02b, SDC14]. As above, these approaches employ a fixed set or template of features. Unlike precondition inference and loop invariant inference, which require more information from the programmer (*e.g.* a postcondition), general invariant inference has no particular goal and so no notion of "good" and "bad" examples. Hence these approaches cannot obtain counterexamples to refine candidate invariants and cannot use my conflict-based approach to learn features.

Finally, the work of Cheung *et al.* [CSM12], like PIE, combines machine learning and program synthesis, but for a very different purpose: to provide event recommendations to users of social media. They use the Sketch system [STB06] to generate a set of recommendation functions that each classify all test inputs, and then they employ SVMs to produce a linear combination of these functions. PIE instead uses program synthesis for feature learning, and only as necessary to resolve conflicts, and then it uses machine learning to infer boolean combinations of these features that classify all test inputs.

2.5 Applications and Extensions

Since its publication in 2016 [PSM16], this automatic on-demand feature learning approach has been shown to be useful in several other applications across many different domains.

The ALIVE-INFER work by Menendez *et al.* [MN17] utilizes our feature learning approach for inferring sufficient preconditions for peephole optimizations. Unlike PIE, which uses SMT-Lib [BST10] as the target language for its synthesizer, ALIVE-INFER uses the ALIVE DSL [LMN15] as the target language for preconditions. This tool has been used to improve the compiler optimization passes within the LLVM compiler infrastructure [LA04], and is being used by the LLVM developers to implement provably correct compiler optimizations.

Recent work by Barbosa *et al.* [BRL19] also uses an on-demand feature learning approach similar to PIE for inferring sufficient loop invariants. However, unlike our approach, which uses the PAC learning algorithm to combine features to a candidate invariant, their approach uses decision tree learning.

Beillahi *et al.* [BCE20] have also used PIE for formally verifying smart contracts written for blockchain systems. Their work uses SOLIDITY [Woo14b] as the target language for the synthesizer. SOLIDITY is the most widely used programming language for smart contracts that run on the ETHEREUM [Woo14a] virtual machine. They also extend PIE to work with *symbolic examples* in addition to concrete examples.

CHAPTER 3

Learning Input Specifications

In modern data science, most real-life datasets lack high-quality metadata — they are often incomplete, erroneous, and unstructured [DS13]. This severely impedes data analysis, even for domain experts. For instance, a merely preliminary task of *data wrangling* (importing, cleaning, and reshaping data) consumes 50 – 80% of the total analysis time [Loh14]. Prior studies show that high-quality metadata not only help users clean, understand, transform, and reason over data, but also enable advanced applications, such as compression, indexing, query optimization, and schema matching [AGN15]. Traditionally, data scientists engage in *data gazing* [May07] — they manually inspect small samples of data, or experiment with aggregation queries to get a bird's-eye view of the data. Naturally, this approach does not scale to modern large-scale datasets [AGN15].

Data profiling is the process of generating small but useful metadata (typically as a succinct summary) for the data [AGN15]. In this work, I focus on syntactic profiling, *i.e.* learning structural patterns that summarize the data. A syntactic profile is a disjunction of regex-like patterns that describe all of the syntactic variations in the data. Each pattern succinctly describes a specific variation, and is defined by a sequence of *atomic patterns* or *atoms*, such as digits or letters.

While existing tools, such as Microsoft SQL Server Data Tools (Microsoft SSDT) [Cor17d], and Ataccama [Cor17a] allow pattern-based profiling, they generate a single profile that cannot be customized. In particular, **(a)** they use a small predetermined set of atoms, and

Reference ID
ISBN:␣1-158-23466-X
not_available
doi:␣10.1016/S1387-7003(03)00113-8
⋮
PMC9473786
ISBN:␣0-006-08903-1
doi:␣␣10.13039/100005795
PMC9035311
⋮
PMC5079771
ISBN:␣2-287-34069-6

(a) Sample data

- `W_W` (5)
- `W: N.N/LN-N(N)N-D` (11)
- `W: D-N-N-L` (34)
- `W: N.N/N` (110)
- `W: D-N-N-D` (267)
- `WN` (1024)

Classes: [L]etter, [W]ord, [D]igit, [N]umber

(b) Profile from Ataccama One

- `doi:␣+10\.\d\d\d\d\d/\d+` (110)
- `.*` (113)
- `ISBN:␣0-\d\d\d-\d\d\d\d\d\d-\d` (204)
- `PMC\d`$^{+}$ (1024)

(c) Profile from Microsoft SSDT

- “not_available” (5)
- “doi:” ␣$^{+}$ “10.1016/” U D^{4} “-” D^{4} “(” D^{2} “)” D^{5} “-” D (11)
- “ISBN:” ␣ D “-” D^{3} “-” D^{5} “-X” (34)
- “doi:” ␣$^{+}$ “10.13039/” D^{+} (110)
- “ISBN:” ␣ D “-” D^{3} “-” D^{5} “-” D (267)
- “PMC” D^{7} (1024)

Classes: [U]ppercase, [D]igit

Superscripts indicate repetition of atoms.
Constant strings are surrounded by quotes.

(d) Default profile from FlashProfile

Table 3.1: Profiles for a set of references[7]— number of matches for each pattern is shown on the right

do not allow users to supply custom atoms specific to their domains, and **(b)** they provide little support for controlling granularity, *i.e.* the number of patterns in the profile.

I present a novel application of program synthesis techniques to addresses these two key issues. The implementation, FlashProfile, supports custom user-defined atoms that may encapsulate arbitrary pattern-matching logic, and also allows users to interactively control the granularity of generated profiles, by providing desired bounds on the number of patterns.

A Motivating Example Table 3.1(a) shows a fragment of a dataset containing a set of references in various formats, and its profiles generated by Ataccama(in Table 3.1(b)), Microsoft SSDT (in Table 3.1(c)), and the tool FlashProfile (in Table 3.1(d)). Syntactic profiles expose rare variations that are hard to notice by manual inspection of the data,

[7] The full dataset is available at https://github.com/SaswatPadhi/FlashProfileDemo/tree/master/motivating_example.json.

- “not_available” (5)
- “doi:” ␣$^{+}$ ⟨DOI⟩ (121)
- “ISBN:” ␣ ⟨ISBN10⟩ (301)
- “PMC” D^{7} (1024)

(a) The auto-suggested profile

- “not_available” (5)
- “doi:” ␣$^{+}$ “10.1016/” U D^{4} “-” D^{4} “(” D^{2} “)” D^{5} “-” D (11)
- “doi:” ␣$^{+}$ “10.13039/” D^{+} (110)
- “ISBN:” ␣ ⟨ISBN10⟩ (301)
- “PMC” D^{7} (1024)

(b) A refined profile on requesting 5 patterns

Figure 3.1: Custom atoms,[8] and refinement of profiles with FlashProfile

or from simple statistical properties such as distribution of string lengths. For example, Ataccama reveals a suspicious pattern “W_W”, which matches less than 0.5% of the dataset. Microsoft SSDT, however, groups this together with other less frequent patterns into a “.*” pattern. Since Microsoft SSDT does not provide a way of controlling the granularity of the profile, a user would be unable to further refine the “.*” pattern. FlashProfile shows that this pattern actually corresponds to missing entries, which read “not_available”.

For this dataset, although Ataccama suggests a profile of the same granularity as from FlashProfile, the patterns in the profile are overly general. For instance, it may not be immediately obvious that the pattern W: D-N-N-L maps to ISBNs in the dataset. FlashProfile further qualifies the W (word) to the constant “ISBN”, and restricts the [N]umber patterns to D^{3} (short for Digit$^{\times 3}$) and D^{5}, and the final [L]etter to the constant “X”.

FlashProfile also allows users familiar with their domains to define custom patterns, that cluster data in ways that are specific to the domain. For example, the two patterns for “doi” in Table 3.1(d) are vastly different — one contains letters and parentheses, whereas the other contains only digits. However, grouping them together makes the profile more readable, and helps spot outliers differing from the expected patterns. Figure 3.1(a) shows a profile suggested by FlashProfile when provided with two custom atoms: ⟨DOI⟩ and ⟨ISBN10⟩,[8]

[8] ⟨DOI⟩ is defined as the regex `10.\d{4,9}/[-._;()/:A-Z0-9a-z]+`.
⟨ISBN10⟩ is defined as the regex `\d-\d{3}-\d{5}-[0-9Xx]`.

with appropriate costs. Users may refine the profile to observe more specific variations within the DOIs and ISBNs. On requesting one more pattern, FlashProfile unfolds ⟨DOI⟩, since the DOIs are *more dissimilar* to each other than ISBNs, and produces the profile shown in Figure 3.1(b).

Key Challenges A key barrier to allowing custom atoms is the large search space for the desirable profiles. Prior tools restrict their atoms to letters and digits, followed by simple upgrades such as sequences of digits to numbers, and letters to words. However, this simplistic approach is not effective in the presence of several overlapping atoms and complex pattern-matching semantics. Moreover, a naïve exhaustive search over all profiles is prohibitively expensive. Every substring might be generalized in multiple ways into different atoms, and the search space grows exponentially when composing patterns as sequences of atoms, and a profile as a disjunction of patterns.

One approach to classifying strings into matching patterns might be to construct decision trees or random forests [Bre01] with features based on atoms. However features are typically defined as predicates over entire strings, whereas atoms match specific substrings and may match multiple times within a string. Moreover, the location of an atomic match within a string depends on the lengths of the preceding atomic matches within that string. Therefore, this approach seems intractable since generating features based on atoms leads to an exponential blow up.

Instead, I propose to address the challenge of learning a profile by first *clustering* [XW05] — partitioning the dataset into *syntactically similar* clusters of strings and then learning a succinct pattern describing each cluster. This approach poses two key challenges: **(a)** efficiently learning patterns for a given cluster of strings over an arbitrary set of atomic patterns provided by the user, and **(b)** defining a suitable notion of *pattern-based similarity* for clustering, that is aware of the user-specified atoms. For instance, as I show in the motivating example (Table 3.1 and Figure 3.1), the clustering must be sensitive to the presence of ⟨DOI⟩ and

⟨ISBN10⟩ atoms. Traditional character-based similarity measures over strings [GF13] are ineffective for imposing a clustering that is susceptible to high-quality explanations using a given set of atoms.

Proposed Technique I address both the aforementioned challenges by leveraging recent advances in *inductive program synthesis* [GPS17] — an approach for learning programs from incomplete specifications, such as input-output examples for the desired program.

First, to address challenge **(1)**, I present a novel domain-specific language (DSL) for patterns, and define a specification over a given set of strings. Users are also allowed to augment this DSL with custom atoms defined as a regex, or constant string, or a string predicate (detailed in § 3.3.1) together with a *static cost* (detailed in § 3.3.3). I then describe an efficient synthesis procedure for learning patterns that are consistent with the specification, and a cost function to select compact patterns that are not overly general, out of all patterns that are consistent with a given cluster of strings.

Second, the cost function for patterns induces a natural *syntactic dissimilarity measure* over strings, which is the key to addressing challenge **(2)**. I consider two strings to be similar if both can be described by a low-cost pattern. Strings requiring overly general / complex patterns are considered dissimilar. Typical clustering algorithms require computation of all pairwise dissimilarities [XW05]. However, in contrast to standard clustering scenarios, computing dissimilarity for a pair of strings not only gives us a numeric measure, but also a pattern for them. This allows for practical performance optimizations. In particular, I present a strategy to approximate dissimilarity computations using a small set of carefully sampled patterns.

To summarize, I present a framework for syntactic profiling based on clustering, that is parameterized by a pattern learner and a cost function. Figure 3.2 outlines this interaction model. In the default mode, users simply provide their dataset. Additionally, they may control the performance *vs.* accuracy trade-off, define custom atoms, and provide bounds on

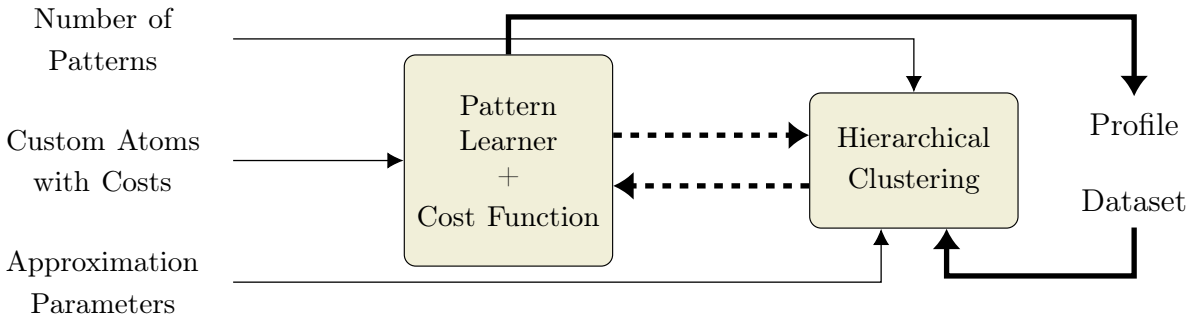


Figure 3.2: FLASHPROFILE's interaction model: *thick* edges denote input and output to the system, *dashed* edges denote internal communication, and *thin* edges denote optional parameters.

the number of patterns. To enable efficient refinement of profiles based on the given bounds, I construct a *hierarchical clustering* [XW05, Section IIB] that may be *cut* at a suitable height to extract the desired number of clusters.

Evaluation I have implemented the proposed technique as FLASHPROFILE using PROSE [Cor17e], also called FLASHMETA [PG15], a state-of-the-art inductive synthesis framework. I evaluate the technique on 75 publicly available datasets collected from online sources.[9] Over 153 tasks, FLASHPROFILE achieves a median profiling time of 0.7s, 77% of which fall under 2s. I show a thorough analysis of the optimizations, and a comparison with state-of-the-art tools.

Applications in PBE Systems The benefits of syntactic profiles extend beyond data understanding. An emerging technology, programming by examples (PBE) [Lie01, GPS17], provides end users with powerful semi-automated alternatives to manual data wrangling. For instance, they may use a tool like FLASH FILL [Gul11], a popular PBE system for data transformations within Microsoft EXCEL and Azure ML Workbench [Cor17c, Cor17b]. However, a key challenge to the success of PBE is finding a representative set of examples that best discriminates the desired program from a large space of possible programs [MSG15].

[9] Datasets are available at: https://github.com/SaswatPadhi/FlashProfileDemo/tree/master/tests.

Typically users provide the desired outputs over the first few entries, and FLASH FILL then synthesizes the *simplest generalization* over them. However, this often results in incorrect programs, if the first few entries are not representative of the various formats present in the entire dataset [MSG15].

Instead, a syntactic profile can be used to select a representative set of examples from syntactically dissimilar clusters. I tested 163 scenarios where FLASH FILL requires more than one input-output example to learn the desired transformation. In 80% of them, the examples belong to different syntactic clusters identified by FLASHPROFILE. Moreover, I show that a *profile-guided interaction model* for FLASH FILL, which I detail in § 3.5, is able to complete 86% of these tasks requiring the *minimum* number of examples. Instead of the user having to select a representative set of examples, the dissimilarity measure allows for proactively requesting the user to provide the desired output on an entry that is *most discrepant* with respect to those previously provided.

In summary, I make the following major contributions:

(§ 3.1) I formally define syntactic profiling as a problem of clustering of strings, followed by learning a succinct pattern for each cluster.

(§ 3.2) I show a hierarchical clustering technique that uses pattern learning to measure dissimilarity of strings, and give performance optimizations that further exploit the learned patterns.

(§ 3.3) I present a novel DSL for patterns, and give an efficient synthesis procedure with a cost function for selecting desirable patterns.

(§ 3.4) I evaluate FLASHPROFILE's performance and accuracy on large real-life datasets, and provide a detailed comparison with state-of-the-art tools.

(§ 3.5) I present a profile-guided interaction model for FLASH FILL, and show that data profiles may aid PBE systems by identifying a representative set of inputs.

3.1 Overview

I use *dataset* to denote a set of strings. Formally, a syntactic profile is defined as below:

Definition 3.1 (Syntactic Profile)**.** Given a dataset $\mathcal{S}$ and a desired number k of patterns, syntactic profiling involves learning **(a)** a partitioning $\mathcal{S}_1 \sqcup \ldots \sqcup \mathcal{S}_k = \mathcal{S}$, and **(b)** a set of patterns $\{P_1, \ldots, P_k\}$, where each P_i is an expression that describes the strings in $\mathcal{S}_i$. I call the disjunction of these patterns $\widetilde{P} = P_1 \vee \ldots \vee P_k$ a *syntactic profile* of $\mathcal{S}$, which describes all the strings in $\mathcal{S}$.

The goal of syntactic profiling is to learn a set of patterns that summarize a given dataset, but is neither too specific nor too general (to be practically useful). For example, the dataset itself is a trivial overly specific profile, whereas the regex "`.*`" is an overly general one. I propose a technique that leverages the following two key subcomponents to generate and rank profiles:[10]

- a pattern learner $\mathcal{L}\colon 2^{\mathbb{S}} \to 2^{\mathscr{L}}$, which generates a set of patterns over an arbitrary pattern language $\mathscr{L}$, that are consistent with a given dataset.
- a cost function $\mathcal{C}\colon \mathscr{L} \times 2^{\mathbb{S}} \to \mathbb{R}_{\geq 0}$, which quantifies the suitability of an arbitrary pattern (in the same language $\mathscr{L}$) with respect to the given dataset.

Using $\mathcal{L}$ and $\mathcal{C}$, I can quantify the suitability of clustering a set of strings together. More specifically, I can define a minimization objective $\mathcal{O}\colon 2^{\mathbb{S}} \to \mathbb{R}_{\geq 0}$ that indicates an aggregate cost of a cluster. I can now define an *optimal syntactic profile* that minimizes $\mathcal{O}$ over a given dataset $\mathcal{S}$:

Definition 3.2 (Optimal Syntactic Profile)**.** Given a dataset $\mathcal{S}$, a desired number k of patterns, and access to a pattern learner $\mathcal{L}$, a cost function $\mathcal{C}$ for patterns, and a minimization objective $\mathcal{O}$ for partitions, I define: **(a)** the optimal partitioning $\widetilde{\mathcal{S}}_{opt}$ as one that minimizes

[10] I denote the set of strings as $\mathbb{S}$, the set of non-negative reals as $\mathbb{R}_{\geq 0}$, and the power set of a set X as 2^X.

the objective function $\mathcal{O}$ over all partitions, and **(b)** the optimal syntactic profile $\widetilde{P}_{opt}$ as the disjunction of the least-cost patterns describing each partition in $\widetilde{\mathcal{S}}_{opt}$. Formally,

$$\widetilde{\mathcal{S}}_{opt} \stackrel{\text{def}}{=} \underset{\substack{\{\mathcal{S}_1,\ldots,\mathcal{S}_k\} \\ \text{s.t. } \mathcal{S}=\bigsqcup_{i=1}^{k}\mathcal{S}_i}}{\operatorname{argmin}} \sum_{i=1}^{k} \mathcal{O}(\mathcal{S}_i) \qquad \text{and} \qquad \widetilde{P}_{opt} \stackrel{\text{def}}{=} \bigvee_{\mathcal{S}_i \in \widetilde{\mathcal{S}}_{opt}} \underset{P \in \mathcal{L}(\mathcal{S}_i)}{\operatorname{argmin}} \mathcal{C}(P, \mathcal{S}_i)$$

Ideally, I would define the aggregate cost of a partition as the minimum cost incurred by a pattern that describes it entirely. This is captured by $\mathcal{O}(\mathcal{S}_i) \stackrel{\text{def}}{=} \min_{P \in \mathcal{L}(\mathcal{S}_i)} \mathcal{C}(P, \mathcal{S}_i)$. However, with this objective, computing the optimal partitioning $\widetilde{\mathcal{S}}_{opt}$ is intractable in general. For an arbitrary learner $\mathcal{L}$ and cost function $\mathcal{C}$, this would require exploring all k-partitionings of the dataset $\mathcal{S}$.[11] Instead, I use an objective that is tractable and works well in practice — the aggregate cost of a cluster is given by the maximum cost of describing any two strings belonging to the cluster, using the best possible pattern. Formally, this objective is $\widehat{\mathcal{O}}(\mathcal{S}_i) \stackrel{\text{def}}{=} \max_{x,y \in \mathcal{S}_i} \min_{P \in \mathcal{L}(\{x,y\})} \mathcal{C}(P, \{x, y\})$. This objective is inspired by the *complete-linkage* criterion [SSS48], which is widely used in clustering applications across various domains [JMF99]. To minimize $\widehat{\mathcal{O}}$, it suffices to only compute the costs of describing (at most) $|\mathcal{S}|^2$ pairs of strings in $\mathcal{S}$.

I outline the main algorithm PROFILE in Figure 3.3. It is parameterized by an arbitrary learner $\mathcal{L}$ and cost function $\mathcal{C}$. PROFILE accepts a dataset $\mathcal{S}$, the bounds $[m, M]$ for the desired number of patterns, and a sampling factor θ that decides the efficiency *vs.* accuracy trade-off. It returns the generated

func PROFILE$_{\langle\mathcal{L},\mathcal{C}\rangle}$($\mathcal{S}$: String[ ], m: Int, M: Int, θ: Real)
output: $\widetilde{P}$, a partitioning of $\mathcal{S}$ with the associated patterns for each partition, s.t. $m \leq |\,\widetilde{P}\,| \leq M$

1: $\widetilde{P} \leftarrow \{\}$
2: $H \leftarrow$ BUILDHIERARCHY$_{\langle\mathcal{L},\mathcal{C}\rangle}(\mathcal{S}, M, \theta)$
3: **for all** $X \in$ PARTITION(H, m, M) **do**
4: $\quad\langle$Pattern: P, Cost: $c\rangle \leftarrow$ LEARNBESTPATTERN$_{\langle\mathcal{L},\mathcal{C}\rangle}(X)$
5: $\quad\widetilde{P} \leftarrow \widetilde{P} \cup \{\langle$Data: X, Pattern: $P\rangle\}$
6: **return** $\widetilde{P}$

Figure 3.3: The main profiling algorithm

[11] The number of ways to partition a set $\mathcal{S}$ into k non-empty subsets is given by Stirling numbers of the second kind [GKP94], $\left\{{|\mathcal{S}| \atop k}\right\}$. When $k \ll |\mathcal{S}|$, the asymptotic value of $\left\{{|\mathcal{S}| \atop k}\right\}$ is given by $\frac{k^{|\mathcal{S}|}}{k!}$.

partitions paired with the least-cost patterns describing them: $\{\langle \mathcal{S}_1, P_1\rangle, \ldots, \langle \mathcal{S}_k, P_k\rangle\}$, where $m \leq k \leq M$.

At a high level, first the dataset is partitioned using a syntactic dissimilarity measure induced by the cost of patterns. For large enough θ, all $O(|\mathcal{S}|^2)$ pairwise dissimilarities are computed to generate the partitioning $\widetilde{\mathcal{S}}_{opt}$ that minimizes $\widehat{\mathcal{O}}$. However, many large real-life datasets have a very small number of syntactic clusters, and $\widetilde{\mathcal{S}}_{opt}$ can be very closely approximated by sampling only a few pairwise dissimilarities. In line 1, BuildHierarchyis invoked to construct a hierarchy H over $\mathcal{S}$ with accuracy controlled by θ. The hierarchy H is then *cut* at a certain height to obtain k clusters by calling Partition in line 2 — if $m \neq M$, k is heuristically decided based on the quality of clusters obtained at various heights. Finally, using LearnBestPattern, a pattern P is learned for each cluster X, and is added to the profile $\widetilde{P}$.

In the following subsections, I explain the two main components: **(a)** BuildHierarchy for building a hierarchical clustering, and **(b)** LearnBestPattern for pattern learning.

3.1.1 Pattern-Specific Clustering

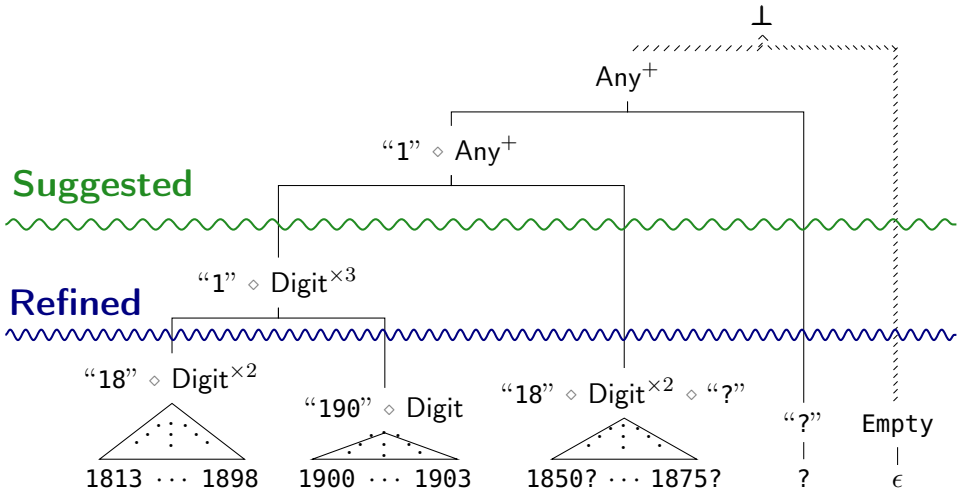
BuildHierarchy uses an agglomerative hierarchical clustering (AHC) [XW05, Section IIB] to construct a hierarchy (also called a *dendrogram*) that depicts a nested grouping of the given collection of strings, based on their syntactic similarity. Figure 3.5 shows such a hierarchy over an incomplete and inconsistent dataset containing years, using the default set of atoms listed in Figure 3.4. Once constructed, a hierarchy may be split at a suitable height to extract clusters of desired granularity, which enables a natural form of refinement — supplying a desired number of clusters. In contrast,

Lower $[a\text{–}z]$	Bin $[01]$
Upper $[A\text{–}Z]$	Digit $[0\text{–}9]$
⟨TitleCaseWord⟩ Upper ⋄ Lower$^+$	Hex $[a\text{–}fA\text{–}F0\text{–}9]$
Alpha $[a\text{–}zA\text{–}Z]$	AlphaDigit $[a\text{–}zA\text{–}Z0\text{–}9]$
␣ $\backslash s$	AlphaDigitSpace $[a\text{–}zA\text{–}Z0\text{–}9\backslash s]$
DotDash $[.\text{–}]$	Punct $[.,:?/\text{–}]$
AlphaDash $[a\text{–}zA\text{–}Z\text{–}]$	Symb $[\text{–}.,:?/@\#\$\%\&\cdots]$
AlphaSpace $[a\text{–}zA\text{–}Z\backslash s]$	Base64 $[a\text{–}zA\text{–}Z0\text{–}9{+}\backslash{=}]$

Figure 3.4: Default atoms in FlashProfile, with the corresponding regex.

Year
1900
1877
ϵ
1860
?
1866
ϵ
1893
⋮
1888 ?
1872

(a) Dataset [12]



(b) A hierarchy based on default atoms from Figure 3.4

Figure 3.5: A hierarchy with suggested and refined clusters: Leaf nodes represent strings, and internal nodes are labelled with patterns describing the strings below them. Atoms are concatenated using " ⋄ ". A dashed edge denotes the absence of a pattern that describes the strings together.

flat clustering methods like k-MEANS [Mac67] generate a fixed partitioning within the same time complexity. In Figure 3.5(b), I show a heuristically suggested split with 4 clusters, and a refined split on a request for 5 clusters. A key challenge to clustering is defining an appropriate pattern-specific measure of dissimilarity over strings, as I show below.

Example 3.1. Consider the pairs: $p = \{$"1817","1813?"$\}$ and $q = \{$"1817","1907"$\}$. Selecting the pair that is syntactically *more similar* is ambiguous, even for humans. The answer depends on the user's application — it may make sense to either cluster homogeneous strings (containing only digits) together, or to cluster strings with a longer common prefix together.

A natural way to resolve this ambiguity is to allow users to express their application-specific preferences by providing custom atoms, and then to make the clustering algorithm sensitive to the available atoms. Therefore I desire a dissimilarity measure that incorporates

[12] Linda K. Jacobs, The Syrian Colony in New York City 1880-1900. http://bit.ly/LJacobs

the user-specified atoms, and yet remains efficiently computable, since typical clustering algorithms compute dissimilarities between all pairs of strings [XW05].

Syntactic Dissimilarity The key insight is to leverage program synthesis techniques to efficiently learn patterns describing a given set of strings, and induce a dissimilarity measure using the learned patterns — overly general or complex patterns indicate a high degree of syntactic dissimilarity.

In § 3.2.1, I formally define the dissimilarity measure η, as the minimum cost incurred by any pattern for describing a given pair of strings, using a specified pattern learner $\mathcal{L}$ and cost function $\mathcal{C}$. I evaluate the η measure in § 3.4.1, and demonstrate that for estimating syntactic similarity it is superior to classical character-based measures [GF13], and simple machine-learned models such as random forests based on intuitive features.

Adaptive Sampling and Approximation While η captures a high-quality syntactic dissimilarity, with it, each pairwise dissimilarity computation requires learning and scoring of patterns, which may be expensive for large real datasets. To allow end users to quickly generate approximately correct profiles for large datasets, I present a two-stage sampling technique.

1. At the top-level, FlashProfile employs a Sample−Profile−Filter cycle: a small subset of the data is sampled, profiled, and is used to filter out data that is described by the profile.

2. While profiling each sample, the BuildHierarchyalgorithm approximates some pairwise dissimilarities using previously seen patterns.

Wnd users are allowed to control the degree of approximations using two optional parameters.

The key insight that, unlike typical clustering scenarios, computing dissimilarity between a pair of strings gives us more than just a measure — one also learns a pattern. This pattern

can be tested on other pairs to approximate their dissimilarity, which is typically faster than learning new patterns. The technique is inspired by counter-example guided inductive synthesis (CEGIS) [STB06]. CEGIS was extended to sub-sampling settings by Raychev *et al.* [RBV16], but they synthesize a single program and require the outputs for all inputs. In this work, the learned profile is a disjunction of several programs, the outputs for which over a dataset, *i.e.* the partitions, are unknown a priori.

Example 3.2. The pattern "PMC" $\diamond$ Digit$^{\times 7}$ learned for the pair {"PMC2536233", "PMC4901429"}, also describes the string pair {"PMC4901429", "PMC2395569"}, and may be used to accurately estimate their dissimilarity without invoking learning again.

However this sampling needs to be performed carefully for accurate approximations. Although the pattern "1" $\diamond$ Digit$^{\times 3}$ learned for {"1901", "1875"}, also describes {"1872", "1875"}, there exists another pattern "187" $\diamond$ Digit, which indicates a much lower syntactic dissimilarity. I propose an adaptive algorithm for sampling patterns based on previously observed patterns and strings in the dataset. The sampling and approximation algorithms are detailed in § 3.2.2 and § 3.2.3 respectively.

3.1.2 Pattern Learning via Program Synthesis

An important aspect of the clustering-based approach to profiling, described in Section 3.1.1, is its generality. It is agnostic to the specific language $\mathscr{L}$ in which patterns are expressed, as long as appropriate pattern learner $\mathcal{L}$ and cost function $\mathcal{C}$ are provided for $\mathscr{L}$.

func LearnBestPattern$_{\langle \mathcal{L},\mathcal{C} \rangle}$($\mathcal{S}$: String[])
output: The least-cost pattern and its cost, for $\mathcal{S}$
1: $V \leftarrow \mathcal{L}(\mathcal{S})$
2: **if** $V = \{\}$ **then return** ⟨Pattern: $\perp$, Cost: ∞⟩
3: $P \leftarrow \text{argmin}_{P \in V}\ \mathcal{C}(P, \mathcal{S})$
4: **return** ⟨Pattern: P, Cost: $\mathcal{C}(P, \mathcal{S})$⟩

Figure 3.6: Learning the best pattern for a dataset

LearnBestPattern, listed in Figure 3.6, first invokes $\mathcal{L}$ to learn a set V of patterns each of which describes all strings in $\mathcal{S}$. If pattern learning fails,[13] in line 2, the special pattern $\perp$ is returned together with a very high cost ∞. Otherwise, the pattern that has the minimum cost using $\mathcal{C}$ w.r.t. $\mathcal{S}$ is returned. LearnBestPattern is used during clustering to compute pairwise dissimilarity and finally compute the least-cost patterns for clusters.

A natural approach to learning patterns is *inductive program synthesis* [GPS17], which generalizes a given specification to desired programs over a domain-specific language (DSL). I propose a rich DSL for patterns, and present an efficient inductive synthesizer for it.

Language for Patterns The DSL $\mathscr{L}_{\mathsf{FP}}$ is designed to support efficient synthesis using existing technologies while still being able to express rich patterns for practical applications. A pattern is an arbitrary sequence of atomic patterns (atoms), each containing low-level logic for matching a sequence of characters. A pattern $P \in \mathscr{L}_{\mathsf{FP}}$ *describes* a string s, *i.e.* $P(s) = \texttt{true}$, iff the atoms in P match contiguous non-empty substrings of s, ultimately matching s in its entirety. FlashProfile uses a default set of atoms listed in Figure 3.4, which may be augmented with new regular expressions, constant strings, or ad hoc functions. I formally define the language $\mathscr{L}_{\mathsf{FP}}$ in Section 3.3.1.

Pattern Synthesis The inductive synthesis problem for pattern learning is: given a set of strings $\mathcal{S}$, learn a pattern $P \in \mathscr{L}_{\mathsf{FP}}$ such that $\forall\, s \in \mathcal{S}.\, P(s) = \texttt{true}$. The learner $\mathcal{L}_{\mathsf{FP}}$ decomposes the synthesis problem for P over the strings in $\mathcal{S}$ into synthesis problems for individual atoms in P over appropriate substrings. However, a naïve approach of tokenizing each string to (exponentially many) sequences of atoms, and computing their intersection is simply impractical. Instead, $\mathcal{L}_{\mathsf{FP}}$ computes the intersection incrementally at each atomic match, using a novel decomposition technique.

[13] Pattern learning may fail, for example, if the language $\mathscr{L}$ is too restrictive and no pattern can describe the given strings.

$\mathcal{L}_{\mathsf{FP}}$ is implemented using PROSE [Cor17e, PG15], a state-of-the-art inductive synthesis framework. PROSE requires DSL designers to define the logic for decomposing a synthesis problem over an expression to those over its subexpressions, which it uses to automatically generate an efficient synthesizer for their DSL. I detail the synthesis procedure in Section 3.3.2.

Cost of Patterns Once a set of patterns has been synthesized, a variety of strategies may be used to identify the most desirable one. The cost function $\mathcal{C}_{\mathsf{FP}}$ is inspired by *regularization* [Tik63] techniques that are heavily used in statistical learning to construct generalizations that do not overfit to the data. $\mathcal{C}_{\mathsf{FP}}$ decides a trade-off between two opposing factors: **(a)** *specificity:* prefer a pattern that is not general, and **(b)** *simplicity:* prefer a compact pattern that is easy to interpret.

Example 3.3. The strings {"`Male`", "`Female`"} are matched by the patterns $\mathsf{Upper} \diamond \mathsf{Lower}^+$, and $\mathsf{Upper} \diamond \mathsf{Hex} \diamond \mathsf{Lower}^+$. Although the latter is more specific, it is overly complex. On the other hand, the pattern Alpha^+ is simpler and easier to interpret, but is overly general.

To this end, each atom in $\mathscr{L}_{\mathsf{FP}}$ has a fixed *static cost* similar to fixed *regularization hyperparameters* used in machine learning [Bis06], and a dataset-driven *dynamic weight*. The cost of a pattern is the weighted sum of the cost of its constituent atoms. In § 3.3.3, I detail the cost function $\mathcal{C}_{\mathsf{FP}}$, and provide some guidelines on assigning static costs for custom atoms.

3.2 Hierarchical Clustering

I now detail the clustering-based approach for generating syntactic profiles and show practical optimizations for fast approximately-correct profiling. In § 3.2.1 – § 3.2.4, I explain these in the context of a small chunk of data drawn from a large dataset. In § 3.2.5, I then discuss how profile large datasets by chunking it and combining the profiles generated for the chunks.

Recall that the first step in PROFILE is to build a hierarchical clustering over the data. The BUILDHIERARCHY procedure, listed in Figure 3.7(a), constructs a hierarchy H over

func BUILDHIERARCHY$_{\langle \mathcal{L},\mathcal{C} \rangle}$($\mathcal{S}$: String[], M: Int, θ: Real)
output: A hierarchical clustering over $\mathcal{S}$

1: $D \leftarrow$ SAMPLEDISSIMILARITIES$_{\langle \mathcal{L},\mathcal{C} \rangle}(\mathcal{S}, \lceil \theta\, M \rceil)$
2: $A \leftarrow$ APPROXDMATRIX$(\mathcal{S}, D)$
3: **return** $AHC(\mathcal{S}, A)$

(a) Building an approximately-correct hierarchy[14]

func AHC($\mathcal{S}$: String[], A: String $\times$ String $\mapsto$ Real)
output: A hierarchy (as nested sets) over $\mathcal{S}$

1: $H \leftarrow \{\{\, s \,\} \mid s \in \mathcal{S}\}$
2: **while** $|\, H \,| > 1$ **do**
3: $\quad \langle X, Y \rangle \leftarrow \operatorname{argmin}_{X,Y \in H}\, \widehat{\eta}\,(X, Y \mid A)$
4: $\quad H \leftarrow \big(H \setminus \{X, Y\}\big) \cup \big\{\{\, X, Y \,\}\big\}$
5: **return** H

(b) A standard algorithm for AHC

Figure 3.7: Algorithms for pattern-similarity-based hierarchical clustering of string datasets

a given dataset $\mathcal{S}$, with parameters M and θ. M is the maximum number of clusters in a desired profile. The *pattern sampling factor* θ decides the performance *vs.* accuracy trade-off while constructing the hierarchy H.

Henceforth, I use *pair* to denote a pair of strings. In line 1 of BUILDHIERARCHY, the pairwise dissimilarities (the best patterns and their costs) are sampled for a small set (based on the θ factor) of string pairs. Specifically, out of all $O(|\mathcal{S}|^2)$ pairs within $\mathcal{S}$, dissimilarities for only $O(\theta M|\mathcal{S}|)$ pairs are adaptively sampled by calling SAMPLEDISSIMILARITIES, and the learned patterns are cached in D. I formally define the dissimilarity measure in § 3.2.1, and describe SAMPLEDISSIMILARITIES in § 3.2.2. The cache D is then used by APPROXDMATRIX, in line 2, to complete the dissimilarity matrix A over $\mathcal{S}$, using approximations wherever necessary. I describe these approximations in § 3.2.3. Finally, a standard agglomerative hierarchical clustering (AHC) [XW05, Section IIB] is used to construct a hierarchy over $\mathcal{S}$ using the matrix A.

3.2.1 Syntactic Dissimilarity

I formally define the syntactic dissimilarity measure as follows:

[14] $\lceil x \rceil$ denotes the *ceiling* of x, *i.e.* $\lceil x \rceil = \min\{m \in \mathbb{Z} \mid m \geq x\}$.

Definition 3.3 (Syntactic Dissimilarity)**.** For a given pattern learner $\mathcal{L}$ and a cost function $\mathcal{C}$ over an arbitrary language of patterns $\mathscr{L}$, I define the syntactic dissimilarity between strings $x, y \in \mathbb{S}$ as the minimum cost incurred by a pattern in $\mathscr{L}$ to describe them together, *i.e.*

$$\eta(x, y) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x = y \\ \infty & \text{if } x \neq y \bigwedge V = \{\} \\ \min\limits_{P \in V} \mathcal{C}\big(P, \{x, y\}\big) & \text{otherwise} \end{cases}$$

where $V = \mathcal{L}\big(\{x, y\}\big) \subseteq \mathscr{L}$ is the set of patterns that describe strings x and y, and ∞ denotes a high cost for a failure to describe x and y together using patterns learned by $\mathcal{L}$.

The following example shows some candidate patterns and their costs encountered during dissimilarity computation for various pairs. The actual numbers depend on the pattern learner and cost function used, in this case FlashProfile's $\mathcal{L}_{\mathsf{FP}}$ and $\mathcal{C}_{\mathsf{FP}}$, which I describe in § 3.3. However, this example highlights the desirable properties for a natural measure of syntactic dissimilarity.

Example 3.4. For three pairs, I show the shortcomings of classical character-based similarity measures. I compare the Levenshtein distance (LD) [Lev66] for these pairs against the pattern-based dissimilarity η computed with the default atoms from Figure 3.4. On the right, I also show the least-cost pattern, and two other randomly sampled patterns that describe the pair.

First, I compare two dates both using the same syntactic format "`YYYY-MM-DD`":

(a) `1990-11-23` / `2001-02-04` $\quad \mathsf{LD} = 8 \quad \textit{vs.} \quad \eta = 4.96 \left\{ \begin{array}{r|l} \textit{4.96} & \mathsf{Digit}^{\times 4} \diamond \text{“-”} \diamond \mathsf{Digit}^{\times 2} \diamond \text{“-”} \diamond \mathsf{Digit}^{\times 2} \\ \textit{179.9} & \mathsf{Hex}^{+} \diamond \mathsf{Symb} \diamond \mathsf{Hex}^{+} \diamond \text{“-”} \diamond \mathsf{Hex}^{+} \\ \textit{46482} & \mathsf{Digit}^{+} \diamond \mathsf{Punct} \diamond \mathsf{Any}^{+} \end{array} \right.$

Syntactically, these dates are very similar — they use the same delimiter "-", and have same width for the numeric parts. The best pattern found by FlashProfile captures exactly

these features. However, Levenshtein distance for these dates is higher than the following dates which uses a different delimiter and a different order for the numeric parts:

(b)

1990-11-23
29/05/1923

$\mathsf{LD} = 5 \quad \textit{vs.} \quad \eta = 30.2 \left\{ \begin{array}{r|l} \textit{30.2} & \mathsf{Digit}^{+} \diamond \mathsf{Punct} \diamond \mathsf{Digit}^{\times 2} \diamond \mathsf{Punct} \diamond \mathsf{Digit}^{+} \\ \textit{318.6} & \mathsf{Digit}^{+} \diamond \mathsf{Symb} \diamond \mathsf{Digit}^{+} \diamond \mathsf{Symb} \diamond \mathsf{Digit}^{+} \\ \textit{55774} & \mathsf{Digit}^{+} \diamond \mathsf{Punct} \diamond \mathsf{Any}^{+} \end{array} \right.$

The Levenshtein distance is again lower for the following pair containing a date and an ISBN code:

(c)

1990-11-23
899-2119-33-X

$\mathsf{LD} = 7 \quad \textit{vs.} \quad \eta = 23595 \left\{ \begin{array}{r|l} \textit{23595} & \mathsf{Digit}^{+} \diamond \text{“-”} \diamond \mathsf{Digit}^{+} \diamond \text{“-”} \diamond \mathsf{Any}^{+} \\ \textit{55415} & \mathsf{Digit}^{+} \diamond \mathsf{Punct} \diamond \mathsf{Any}^{+} \\ \textit{92933} & \mathsf{Any}^{+} \end{array} \right.$

The same trend is also observed for Jaro-Winkler [Win99], and other measures based on edit distance [GF13]. Whereas these measures look for exact matches on characters, pattern-based measures have the key advantage of being able to generalize substrings to atoms.

3.2.2 Adaptive Sampling of Patterns

Although η accurately captures the syntactic dissimilarity of strings over an arbitrary language of patterns, it requires pattern learning and scoring for every pairwise dissimilarity computation, which is computationally expensive. While this may not be a concern for non-realtime scenarios, such as profiling large datasets on cloud-based datastores, I provide a tunable parameter to end users to be able to generate approximately correct profiles for large datasets in real time.

Besides a numeric measure of dissimilarity, computing η over a pair also generates a pattern that describes the pair. Since the patterns generalize substrings to atoms, often the patterns learned for one pair also describe many other pairs. I aim to sample a subset of patterns that are likely to be sufficient for constructing a hierarchy accurate until M levels, *i.e.* $1 \leq k \leq M$ clusters extracted from this hierarchy should be identical to k clusters extracted from a hierarchy constructed without approximations. The SampleDissimilarities algorithm,

func SAMPLEDISSIMILARITIES$_{\langle\mathcal{L},\mathcal{C}\rangle}(\mathcal{S}$: String[ ], $\widehat{M}$: Int)

output: A dictionary mapping $O(\widehat{M}|\mathcal{S}|)$ pairs of strings from $\mathcal{S}$, to the best pattern describing each pair and its cost

1: $D \leftarrow \{\}$; $a \leftarrow$ a random string in $\mathcal{S}$; $\rho \leftarrow \{a\}$
2: **for** $i \leftarrow 1$ **to** $\widehat{M}$ **do**
3: **for all** $b \in \mathcal{S}$ **do**
4: $D[a, b] \leftarrow$ LEARNBESTPATTERN$_{\langle\mathcal{L},\mathcal{C}\rangle}(\{a, b\})$
▶ Pick the most dissimilar string w.r.t. strings already in ρ.
5: $a \leftarrow \text{argmax}_{x \in \mathcal{S}} \min_{y \in \rho} D[y, x]$.Cost
6: $\rho \leftarrow \rho \cup \{a\}$
7: **return** D

Figure 3.8: Adaptively sampling a small set of patterns

shown in Figure 3.8, is inspired by the seeding technique of k-MEANS++ [AS07]. Instead of computing all pairwise dissimilarities for pairs in $\mathcal{S} \times \mathcal{S}$, I compute the dissimilarities for pairs in $\rho \times \mathcal{S}$, where set ρ is a carefully selected small set of *seed strings*. The patterns learned during this process are likely to be sufficient for accurately estimating the dissimilarities for the remaining pairs.

SAMPLEDISSIMILARITIES takes a dataset $\mathcal{S}$ and a factor $\widehat{M}$, and it samples dissimilarities for $O(\widehat{M}|\mathcal{S}|)$ pairs. It iteratively selects a set ρ containing $\widehat{M}$ strings that are most dissimilar to each other. Starting with a random string in ρ, in each iteration, at line 6, it adds the string $x \in \mathcal{S}$ such that it is as dissimilar as possible, even with its most-similar neighbor in ρ. In the end, the set D only contains dissimilarities for pairs in $\mathcal{S} \times \rho$, computed at line 5. Recall that, $\widehat{M}$ is controlled by the pattern sampling factor θ. In line 1 of BUILDHIERARCHY (in Figure 3.7(a)), $\widehat{M}$ is set to $\lceil \theta M \rceil$.

Since the user may request up to at most M clusters, θ must be at least 1.0, so that at least one seed string is picked from each cluster to ρ. Then, computing the dissimilarities with

all other strings in the dataset would ensure a good distribution of patterns that describe intra– and inter–cluster dissimilarities, even for the finest granularity clustering with M clusters.

Example 3.5. Consider the dataset containing years in Figure 3.5(a). Starting with a random string, say "`1901`"; the set ρ of seed strings grows as shown below, with increasing $\widehat{M}$. At each step, NN (nearest neighbor) shows the new string added to ρ paired with its most similar neighbor.

$\widehat{M} = 2 \mid NN = \langle \epsilon, \text{“}\texttt{1901}\text{”} \rangle \qquad \rho = \{\text{“}\texttt{1901}\text{”}, \epsilon\}$

$\widehat{M} = 3 \mid NN = \langle \text{“}\texttt{?}\text{”}, \text{“}\texttt{1901}\text{”} \rangle \qquad \rho = \{\text{“}\texttt{?}\text{”}, \text{“}\texttt{1901}\text{”}, \epsilon\}$

$\widehat{M} = 4 \mid NN = \langle \text{“}\texttt{1875?}\text{”}, \text{“}\texttt{1901}\text{”} \rangle \qquad \rho = \{\text{“}\texttt{1875?}\text{”}, \text{“}\texttt{?}\text{”}, \text{“}\texttt{1901}\text{”}, \epsilon\}$

$\widehat{M} = 5 \mid NN = \langle \text{“}\texttt{1817}\text{”}, \text{“}\texttt{1875?}\text{”} \rangle \qquad \rho = \{\text{“}\texttt{1817}\text{”}, \text{“}\texttt{1875?}\text{”}, \text{“}\texttt{?}\text{”}, \text{“}\texttt{1901}\text{”}, \epsilon\}$

$\widehat{M} = 6 \mid NN = \langle \text{“}\texttt{1898}\text{”}, \text{“}\texttt{1817}\text{”} \rangle \qquad \rho = \{\text{“}\texttt{1898}\text{”}, \text{“}\texttt{1817}\text{”}, \text{“}\texttt{1875?}\text{”}, \text{“}\texttt{?}\text{”}, \text{“}\texttt{1901}\text{”}, \epsilon\}$

3.2.3 Dissimilarity Approximation

Now I present the technique for completing a dissimilarity matrix over a dataset $\mathcal{S}$, using the patterns sampled from the previous step. Note that, for a large enough value of the pattern sampling factor, *i.e.* $\theta \geq {}^{|\mathcal{S}|}/_{M}$, our technique would sample all pairwise dissimilarities and no approximation would be necessary. For smaller values of θ, the patterns learned while computing η over $\rho \times \mathcal{S}$ are used to approximate the remaining pairwise dissimilarities in $\mathcal{S} \times \mathcal{S}$. The key observation here is that, testing whether a pattern describes a string is typically much faster than learning a new pattern.

The ApproxDMatrix procedure, listed in Figure 3.9, uses the dictionary D of patterns from SampleDissimilarities to generate a matrix A of all pairwise dissimilarities over $\mathcal{S}$. Lines 7 and 8 show the key approximation steps for a pair $\{x, y\}$. In line 7, the patterns in D are tested to select a subset V of them containing only those which describe both x and y. Then their new costs relative to $\{x, y\}$ is computed in line 8 to select the least pattern cost

func ApproxDMatrix$_{\langle\mathcal{L},\mathcal{C}\rangle}$($\mathcal{S}$: String[ ], D: String $\times$ String $\mapsto$ Pattern $\times$ Real)
output: A matrix A of all pairwise dissimilarities over strings in $\mathcal{S}$

1: $A \leftarrow \{\}$
2: **for all** $x \in \mathcal{S}$ **do**
3: **for all** $y \in \mathcal{S}$ **do**
4: **if** $x = y$ **then** $A[x, y] \leftarrow 0$
5: **else if** $\langle x, y\rangle \in D$ **then** $A[x, y] \leftarrow D[x, y]$.Cost
6: **else**
▶ Select the least cost pattern that describes x and y.
7: $V \leftarrow \{P \mid \langle$Pattern: P, Cost: $\cdot\rangle \in D \bigwedge P(x) \bigwedge P(y)\}$
8: **if** $V \neq \{\}$ **then** $A[x, y] \leftarrow \min_{P \in V} \mathcal{C}(P, \{x, y\})$
9: **else**
▶ Compute $\eta(s, y)$, and store the learned pattern.
10: $D[x, y] \leftarrow$ LearnBestPattern$_{\langle\mathcal{L},\mathcal{C}\rangle}(\{x, y\})$
11: $A[x, y] \leftarrow D[x, y]$.Cost
12: **return** A

Figure 3.9: Approximating a complete dissimilarity matrix

as an approximation of $\eta(x, y)$. If V turns out to be empty, *i.e.* no sampled pattern describes both x and y, then, in line 10, LearnBestPattern is called to compute $\eta(x, y)$. The new pattern is also added to D for use in future approximations.

Although $\theta = 1.0$ ensures that at least one seed string is picked from each final cluster, in practice a θ that is slightly greater than 1.0 works better. This results in sampling a few more seed strings, and ensures a better distribution of patterns in D at the cost of a negligible performance overhead. In practice, it rarely happens that no sampled pattern describes a new pair (at line 9, Figure 3.9), since seed patterns for inter-cluster string pairs are usually overly general, as I show in the example below.

Example 3.6. Consider a dataset $\mathcal{S} =$ {"`07-jun`", "`aug-18`", "`20-feb`", "`16-jun`", "`20-jun`"}. Assuming $M = 2$ and $\theta = 1.0$ (*i.e.* $\widehat{M} = 2$), suppose I start with the string "`20-jun`".

Then, following the SampleDissimilaritiesalgorithm shown in Figure 3.8, I would select $\rho = \{$"`20-jun`","`aug-18`"$\}$, and would sample the following seed patterns into D based on patterns defined over the default atoms (listed in Figure 3.4) and constant string literals:

(a) D["`20-jun`","`07-jun`"] $\mapsto$ Digit$^{\times 2}$ $\diamond$ "`-jun`", and

(b) D["`20-jun`","`20-feb`"] $\mapsto$ "`20-`" $\diamond$ Lower$^{\times 3}$,

(c) D["`20-jun`","`16-jun`"] $\mapsto$ Digit$^{\times 2}$ $\diamond$ "`-jun`", and

(d) D["`20-jun`","`aug-18`"], D["`aug-18`","`07-jun`"], D["`aug-18`","`20-feb`"], D["`aug-18`","`16-jun`"] $\mapsto$ AlphaDigit^{+} $\diamond$ "`-`" $\diamond$ AlphaDigit^{+}.

Next, I estimate η("`16-jun`","`20-feb`") using these patterns. None of *(a)* — *(c)* describe the pair, but *(d)* does. However, it is overly general compared to the least-cost pattern, Digit$^{\times 2}$ $\diamond$ "`-`" $\diamond$ Lower$^{\times 3}$.

As in the case above, depending on the expressiveness of the pattern language, for a small θ the sampled patterns may be too specific to be useful. With a slightly higher $\theta = 1.25$, *i.e.* $\widehat{M} = \lceil \theta M \rceil = 3$, I would also select "`07-jun`" as a seed string in ρ, and sample the desired while computing D["`07-jun`","`20-feb`"]. I evaluate the impact of θ on performance and accuracy in § 3.4.2.

3.2.4 Hierarchy Construction and Splitting

Once a dissimilarity matrix has been computed, a standard agglomerative hierarchical clustering (AHC) [XW05, Section IIB] algorithm is used, as outlined in Figure 3.7(b). Note that AHC is not parameterized by $\mathcal{L}$ and $\mathcal{C}$, since it does not involve learning or scoring of patterns any more.

Starting with each string in a singleton set (leaf nodes of the hierarchy), the least-dissimilar pair of sets are iteratively merged, until only a single set (root of the hierarchy) remains. AHC relies on a *linkage criterion* to estimate dissimilarity of sets of strings. I use the

classic complete-linkage (also known as further-neighbor linkage) criterion [SSS48], which has been shown to be resistant to outliers, and yield useful hierarchies in practical applications [JMF99].

Definition 3.4 (Complete-Linkage)**.** For a set $\mathcal{S}$ and a dissimilarity matrix A defined on $\mathcal{S}$, given two arbitrarily-nested clusters X and Y over a subset of entities in $\mathcal{S}$, the dissimilarity between their contents (the *flattened* sets $\overline{X}, \overline{Y} \subseteq \mathcal{S}$, respectively) is defined as:

$$\widehat{\eta}\,(X, Y \mid A) \stackrel{\text{def}}{=} \max_{x \in \overline{X}, y \in \overline{Y}} A[x, y]$$

Once a hierarchy has been constructed, the PROFILE algorithm (in Figure 3.3) invokes the PARTITION method (at line 2) to extract k clusters within the provided bounds $[m, M]$. If $m \neq M$, a heuristic based on the *elbow* (also called *knee*) method [HBV01] is used: between the top m^{th} and the M^{th} nodes, The hierarchy is split till the knee — a node below which the average intra-cluster dissimilarity does not vary significantly. A user may request $m = k = M$, in which case PARTITION simply splits the top k nodes of the hierarchy to generate k clusters.

3.2.5 Profiling Large Datasets

To scale the technique to large datasets, I now describe a second round of sampling. Recall that in SAMPLEDISSIMILARITIES, $O(\theta M|\mathcal{S}|)$ pairwise dissimilarities are sampled. Although θM is very small, $|\,\mathcal{S}\,|$ can still be quite large for real-life datasets. In order to address this, the PROFILE algorithm from Figure 3.3 is run on small chunks of the dataset, and the generated profiles are then combined.

func BIGPROFILE$_{\langle\mathcal{L},\mathcal{C}\rangle}$($\mathcal{S}$: String[], m: Int, M: Int, θ: Real, μ: Real)
output: A profile $\widetilde{P}$ that satisfies $m \leq |\,\widetilde{P}\,| \leq M$
1: $\widetilde{P} \leftarrow \{\}$
2: **while** $|\,\mathcal{S}\,| > 0$ **do**
3: $\quad X \leftarrow$ SAMPLERANDOM($\mathcal{S}, \lceil \mu M \rceil$)
4: $\quad \widetilde{P}' \leftarrow$ PROFILE$_{\langle\mathcal{L},\mathcal{C}\rangle}(X, m, M, \theta)$
5: $\quad \widetilde{P} \leftarrow$ COMPRESSPROFILE$_{\langle\mathcal{L},\mathcal{C}\rangle}(\widetilde{P} \cup \widetilde{P}', M)$
6: $\quad \mathcal{S} \leftarrow$ REMOVEMATCHINGSTRINGS($\mathcal{S}, \widetilde{P}$)
7: **return** $\widetilde{P}$

Figure 3.10: Profiling large datasets

func CompressProfile$_{\langle\mathcal{L},\mathcal{C}\rangle}(\widetilde{P}$: ref Profile, M: Int)
output: A compressed profile $\widetilde{P}$ that satisfies $|\,\widetilde{P}\,| \leq M$

1: **while** $|\,\widetilde{P}\,| > M$ **do**
► Compute the most similar partitions in the profile so far.
2: $\langle X, Y\rangle \leftarrow \underset{X,Y\in\widetilde{P}}{\operatorname{argmin}} \left[\text{LearnBestPattern}_{\langle\mathcal{L},\mathcal{C}\rangle}(X.\text{Data} \cup Y.\text{Data})\right].\text{Cost}$
► Merge partitions $\langle X, Y\rangle$, and update $\widetilde{P}$.
3: $Z \leftarrow X.\text{Data} \cup Y.\text{Data}$
4: $P \leftarrow \text{LearnBestPattern}_{\langle\mathcal{L},\mathcal{C}\rangle}(Z).\text{Pattern}$
5: $\widetilde{P} \leftarrow (\widetilde{P} \setminus \{X, Y\}) \cup \{\,\langle\text{Data: } Z, \text{Pattern: } P\rangle\,\}$
6: **return** $\widetilde{P}$

Figure 3.11: Limiting the number of patterns in a profile

I outline the BigProfile algorithm in Figure 3.10. This algorithm accepts a new *string sampling factor* $\mu \geq 1$, which controls the size of chunks profiled in each iteration. In § 3.4.3, I evaluate the impact of μ on performance and accuracy.

First, a random subset X of size $\lceil \mu M \rceil$ is selected from $\mathcal{S}$ in line 3. In line 4, a profile $\widetilde{P}'$ of X is obtained, and is merged with the global profile $\widetilde{P}$ in line 5. This loop is repeated with the remaining strings in $\mathcal{S}$ that do not match the global profile. For brevity, I elide the details of SampleRandom and RemoveMatchingStrings, which have straightforward implementations. Note that, while merging $\widetilde{P}$ and $\widetilde{P}'$ in line 5, it is possible to exceed the maximum number of patterns M. In Figure 3.11 I outline CompressProfilethat compresses a given profile to a given size bound. It accepts a profile $\widetilde{P}$ and shrinks it to at most M patterns. The key idea is to repeatedly merge the most similar pair of patterns in $\widetilde{P}$. However, the similarity between patterns cannot be computed directly. Instead, it is estimated using syntactic similarity of the associated data partitions. In line 2, the partitions $\langle X, Y\rangle$ which are the most similar, *i.e.* require the least cost pattern for describing them together, are identified. Then X and Y are merged to Z, a pattern describing Z is learned,

and $\widetilde{P}$ is updated by replacing X and Y with Z and its pattern. This process is repeated until the total number of patterns falls to M.

Theorem 3.1 (Termination)**.** *Over an arbitrary language $\mathscr{L}$ of patterns, assume an arbitrary learner $\mathcal{L}\colon 2^{\mathbb{S}} \to 2^{\mathscr{L}}$ and a cost function $\mathcal{C}\colon \mathscr{L} \times 2^{\mathbb{S}} \to \mathbb{R}_{\geq 0}$, such that for any finite dataset $\mathcal{S} \subset \mathbb{S}$, I have:* ***(a)*** *$\mathcal{L}(\mathcal{S})$ terminates and produces a finite set of patterns, and* ***(b)*** *$\mathcal{C}(P, \mathcal{S})$ terminates for all $P \in \mathscr{L}$. Then, the* BigProfile *procedure (Figure 3.10) terminates on any finite dataset $\mathcal{S} \subset \mathbb{S}$, for arbitrary valid values of the optional parameters m, M, θ and μ.*

Proof. Note that in BigProfile, the loop within lines 2 – 6 runs for at most $|\mathcal{S}|/\lceil \mu M \rceil$ iterations, since at least $\lceil \mu M \rceil$ strings are removed from $\mathcal{S}$ in each iteration. Therefore it is sufficient to show that Profile and CompressProfile terminate.

First, note that termination of LearnBestPattern immediately follows from **(a)** and **(b)**. Then, it is easy to observe that CompressProfile terminates as well: **(a)** the loop in lines 1 – 5 runs for at most $|\widetilde{P}| - M$ iterations, and **(b)** LearnBestPattern is invoked $O(|\widetilde{P}|^2)$ times in each iteration.

The Profile procedure (Figure 3.3) makes at most $O((\mu M)^2)$ calls to LearnBestPattern (Figure 3.6) to profile the $\lceil \mu M \rceil$ strings sampled in to X — at most $O\big((\mu M)^2\big)$ calls within BuildHierarchy (Figure 3.7(a)), and $O(M)$ calls to learn patterns for the final partitions. Depending on θ, BuildHierarchy may make many fewer calls to LearnBestPattern. However, it makes no more than 1 such call per pair of strings in X, to build the dissimilarity matrix. Therefore, Profile terminates as well. □

3.3 Pattern Synthesis

I now describe the specific pattern language, leaning technique and cost function used to instantiate the profiling technique as FlashProfile. I begin with a brief description the

pattern language in Section 3.3.1, present the pattern synthesizer in Section 3.3.2, and conclude with the cost function in Section 3.3.3.

3.3.1 The Pattern Language $\mathscr{L}_{\mathsf{FP}}$

Figure 3.12(a) shows the formal syntax for the pattern language $\mathscr{L}_{\mathsf{FP}}$. Each pattern $P \in \mathscr{L}_{\mathsf{FP}}$ is a predicate defined on strings, *i.e.* a function $P\colon \mathsf{String} \to \mathsf{Bool}$, which embodies a set of constraints over strings. A pattern P *describes* a given string s, *i.e.* $P(s) = \texttt{true}$, iff s satisfies all constraints imposed by P. Patterns in $\mathscr{L}_{\mathsf{FP}}$ are composed of atomic patterns:

Definition 3.5 (Atomic Pattern (or Atom))**.** An atom, $\alpha\colon \mathsf{String} \to \mathsf{Int}$ is a function, which given a string s, returns the length of the longest prefix of s that satisfies its constraints. Atoms only match non-empty prefixes. $\alpha(s) = 0$ indicates match failure of α on s.

The following four kinds of atoms are allowed in $\mathscr{L}_{\mathsf{FP}}$:

(a) *Constant Strings*: A Const_s atom matches only the string s as the prefix of a given string. For brevity, I denote $\mathsf{Const}_{\text{“}\texttt{str}\text{”}}$ as simply “`str`” throughout the text.

(b) *Regular Expressions*: A RegEx_r atom returns the length of the longest prefix of a given string, that is matched by the regex r.

(c) *Character Classes*: A Class_c^0 atom returns the length of the longest prefix of a given string, which contains characters only from the set c. A Class_c^z atom with $z > 0$ further enforces a fixed-width constraint — the match $\mathsf{Class}_c^z(s)$ fails if $\mathsf{Class}_c^0(s) \neq z$, otherwise it returns z.

(d) *Arbitrary Functions*: A Funct_f atom uses the function f that may contain arbitrary logic, to match a prefix p of a given string and returns $|p|$.

---

[15] $a \circ b$ denotes the concatenation of strings a and b, and $r \triangleright x$ denotes that the regex r matches the string x in its entirety.

Pattern $P[s] := \texttt{Empty}(s)$

$\quad\mid P[\texttt{SuffixAfter}(s, \alpha)]$

Atom $\alpha := \mathsf{Class}_c^z \mid \mathsf{RegEx}_r$

$\quad\mid \mathsf{Funct}_f \mid \mathsf{Const}_s$

$c \in$ power set of characters
$f \in$ functions String $\to$ Int
$r \in$ regular expressions
$s \in$ set of strings $\mathbb{S}$
$z \in$ non-negative integers

(a) Syntax of $\mathscr{L}_{\mathsf{FP}}$ patterns.

$$\frac{}{\texttt{Empty}(\epsilon) \ \Downarrow \ \texttt{true}}$$

$$\frac{s = s_0 \circ s_1 \qquad \alpha(s) = |s_0| > 0}{\texttt{SuffixAfter}(s, \alpha) \ \Downarrow \ s_1}$$

$$\frac{}{\mathsf{Funct}_f(s) \ \Downarrow \ f(s)}$$

$$\frac{|s| > 0 \qquad s_0 = s \circ s_1}{\mathsf{Const}_s(s_0) \ \Downarrow \ |s|}$$

$$\frac{L = \{n \in \mathbb{N} \mid r \rhd s[0:n]\}}{\mathsf{RegEx}_r(s) \ \Downarrow \ \max L}$$

$$\frac{s = s_0 \circ s_1 \qquad \forall x \in s_0 : x \in c \qquad s_1 = \epsilon \vee s_1[0] \notin c}{\mathsf{Class}_c^0(s) \ \Downarrow \ |s_0|}$$

$$\frac{s = s_0 \circ s_1 \qquad \forall x \in s_0 : x \in c \qquad |s_0| = z > 0 \qquad s_1 = \epsilon \vee s_1[0] \notin c}{\mathsf{Class}_c^z(s) \ \Downarrow \ z}$$

(b) Big-step semantics for $\mathscr{L}_{\mathsf{FP}}$ patterns: We use the judgement $E \Downarrow v$ to indicate that the expression E evaluates to a value v.

Figure 3.12: Formal syntax and semantics of our DSL $\mathscr{L}_{\mathsf{FP}}$ for defining syntactic patterns over strings[15]

Note that, although both constant strings and character classes may be expressed as regular expressions, having separate terms for them has two key benefits:

- As shown in the next subsection, all constant strings are *automatically inferred*, and some character class atoms (namely, those having a fixed-width). This is unlike regular expression or function atoms, which are not inferred and must be provided a priori.
- These atoms may leverage more efficient matching logic and do not require regular expression matching in its full generality. Constant string atoms use equality checks for characters, and character class atoms use set membership checks.

I list the default set of atoms provided with FlashProfile, in Figure 3.4. Users may extend this set with new atoms from any of the aforementioned kinds.

Example 3.7. The atom Digit is Class_D^1 with $D = \{0, \ldots, 9\}$. I write Class_D^0 as Digit^+, and Class_D^n as $\mathsf{Digit}^{\times n}$ for clarity. Note that, $\mathsf{Digit}^{\times 2}$ matches "`04/23`" but not "`2017/04`", although Digit^+ matches both, since the longest prefix matched, "`2017`", has length $4 \neq 2$.

Definition 3.6 (Pattern)**.** A pattern is simply a sequence of atoms. The pattern `Empty` denotes an empty sequence, which only matches the empty string ϵ. I use the concatenation

operator " $\diamond$ " for sequencing atoms. For $k > 1$, the sequence $\alpha_1 \diamond \alpha_2 \diamond \ldots \diamond \alpha_k$ of atoms defines a pattern that is realized by the $\mathscr{L}_{\mathsf{FP}}$ expression:

$$\texttt{Empty(SuffixAfter(}\cdots\texttt{SuffixAfter(}s, \alpha_1\texttt{)}\cdots, \alpha_k\texttt{))},$$

which matches a string s, iff

$$s \neq \epsilon \bigwedge \forall i \in \{1, ..., k\} : \alpha_i(s_i) > 0 \bigwedge s_{k+1} = \epsilon,$$

where $s_1 \stackrel{\text{def}}{=} s$ and $s_{i+1} \stackrel{\text{def}}{=} s_i[\alpha_i(s_i)\texttt{:}]$ is the remaining suffix of the string s_i after matching atom α_i.

Throughout this section, I use $s[i]$ to denote the i^{th} character of s, and $s[i\texttt{:}j]$ denotes the substring of s from the i^{th} character, until the j^{th}. I omit j to indicate a substring extending until the end of s. In $\mathscr{L}_{\mathsf{FP}}$, the $\texttt{SuffixAfter(}s, \alpha\texttt{)}$ operator computes $s[\alpha(s)\texttt{:}]$, or fails with an error if $\alpha(s) = 0$. I also show the formal semantics of patterns and atoms in $\mathscr{L}_{\mathsf{FP}}$, in Figure 3.12(b).

Note that, atoms are forbidden from matching empty substrings. This reduces the search space by an exponential factor, since an empty string may trivially be inserted between any two characters within a string. However, this does not affect the expressiveness of the final profiling technique, since a profile uses a disjunction of patterns. For instance, the strings matching a pattern $\alpha_1 \diamond (\epsilon \mid \alpha_2) \diamond \alpha_3$ can be clustered into those matching $\alpha_1 \diamond \alpha_3$ and $\alpha_1 \diamond \alpha_2 \diamond \alpha_3$.

3.3.2 Synthesis of $\mathscr{L}_{\mathsf{FP}}$ Patterns

The pattern learner $\mathcal{L}_{\mathsf{FP}}$ uses inductive program synthesis [GPS17] for synthesizing patterns that describe a given dataset $\mathcal{S}$ using a specified set of atoms $\mho$. For the convenience of end users, I automatically *enrich* their specified atoms by including: **(a)** all possible Const atoms,

and **(b)** all possible fixed-width variants of all Class atoms specified by them. The learner $\mathcal{L}_{\mathsf{FP}}$ is instantiated with these enriched atoms derived from $\mho$, denoted as $\widehat{\mho}$:

$$\begin{aligned} \widehat{\mho} \stackrel{\text{def}}{=} \mho \;\cup\; & \{\mathsf{Const}_s \mid s \in \mathbb{S}\} \\ \cup\; & \{\mathsf{Class}_c^z \mid \mathsf{Class}_c^0 \in \mho \bigwedge z \in \mathbb{N}\} \end{aligned} \tag{3.1}$$

Although $\widehat{\mho}$ is very large, as I describe below, the learner $\mathcal{L}_{\mathsf{FP}}$ efficiently explores this search space, and also provides a completeness guarantee on patterns possible over $\widehat{\mho}$.

I build on top of PROSE [Cor17e], a state-of-the-art inductive program synthesis library, which implements the FlashMeta [PG15] framework. PROSEuses *deductive reasoning* — reducing a problem of synthesizing an expression to smaller synthesis problems for its subexpressions, and provides a robust framework with efficient algorithms and data-structures for this. The key contribution towards $\mathcal{L}_{\mathsf{FP}}$ are efficient *witness functions* [PG15, §5.2] that enable PROSE to carry out the deductive reasoning over $\mathscr{L}_{\mathsf{FP}}$.

An inductive program synthesis task is defined by: **(a)** a *domain-specific language* (DSL) for the target programs, which in the case is $\mathscr{L}_{\mathsf{FP}}$, and **(b)** a *specification* [PG15, §3.2] (spec) that defines a set of constraints over the output of the desired program. For learning patterns over a collection $\mathcal{S}$ of strings, I define a spec φ, that simply requires a learned pattern P to describe all given strings, *i.e.* $\forall s \in \mathcal{S}\colon P(s) = \texttt{true}$. I formally write this as:

$$\varphi \stackrel{\text{def}}{=} \bigwedge_{s \in \mathcal{S}} [s \rightsquigarrow \texttt{true}]$$

I provide a brief overview of the deductive synthesis process here, and refer the reader to FlashMeta [PG15] for a detailed discussion. In a deductive synthesis framework, we are required to define the logic for reducing a spec for an expression to specs for its subexpressions. The reduction logic for specs, called witness functions [PG15, §5.2], is domain-specific, and depends on the semantics of the DSL. Witness functions are used to recursively reduce the specs to terminal symbols in the DSL. PROSE uses a succinct data structure [PG15, §4]

to track the valid solutions to these specs at each reduction and generate expressions that satisfy the initial spec. For $\mathscr{L}_{\mathsf{FP}}$, I describe the logic for reducing the spec φ over the two kinds of patterns: `Empty` and $P\,[\texttt{SuffixAfter(}s, \alpha\texttt{)}]$. For brevity, I elide the pseudocode for implementing the witness functions — their implementation is straightforward, based on the reductions I describe below.

For `Empty(`s`)` to satisfy a spec φ, *i.e.* describe all strings $s \in \mathcal{S}$, each string s must indeed be ϵ. No further reduction is possible since s is a terminal. Only $\forall s \in \mathcal{S}\colon\ s = \epsilon$ is checked, and `Empty(`s`)` is rejected if $\mathcal{S}$ contains at least one non-empty string.

The second kind of patterns for non-empty strings, $P\,[\texttt{SuffixAfter(}s, \alpha\texttt{)}]$, allows for complex patterns that are a composition of an atom α and a pattern P. The pattern $P\,[\texttt{SuffixAfter(}s, \alpha\texttt{)}]$ contains two unknowns: **(a)** an atom α that matches a non-empty prefix of s, and **(b)** a pattern P that matches the remaining suffix $s[\alpha(s)\texttt{:}\,]$. Again, note that this pattern must match all strings $s \in \mathcal{S}$. Naïvely considering all possible combinations of α and P leads to an exponential blow up.

First note that for a fixed α the candidates for P can be generated recursively by posing a synthesis problem similar to the original one, but over the suffix $s[\alpha(s)\texttt{:}\,]$ instead of each string s. This reduction style is called a *conditional witness function* [PG15, §5.2], and generates the following spec for P assuming a fixed α:

$$\varphi_\alpha \stackrel{\text{def}}{=} \bigwedge_{s \in \mathcal{S}} \left[s[\alpha(s)\texttt{:}\,] \rightsquigarrow \texttt{true}\right] \tag{3.2}$$

However, naïvely creating φ_α for all possible values of α is infeasible, since the set $\widehat{\mathfrak{O}}$ of atoms is unbounded. Instead, I consider only those atoms (called *compatible atoms*) that match some non-empty prefix for *all* strings in $\mathcal{S}$, since ultimately the pattern needs to describe all strings. Prior pattern-learning approaches [RH01, Sin16] learn complete patterns for each individual string, and then compute their intersection to obtain patterns consistent with the entire dataset. In contrast, this approach computes the set of atoms that are

compatible with the entire dataset at each step, which allows us to generate this intersection in an incremental manner.

Definition 3.7 (Compatible Atoms)**.** Given a universe $\mathfrak{V}$ of atoms, I say a subset $A \subseteq \mathfrak{V}$ is compatible with a dataset $\mathcal{S}$, denoted as $A \propto \mathcal{S}$, if all atoms in A match each string in $\mathcal{S}$, *i.e.*

$$A \propto \mathcal{S} \quad \text{iff} \quad \forall\, \alpha \in A :\ \forall\, s \in \mathcal{S} :\ \alpha(s) > 0$$

I say that a compatible set A of atoms is *maximal* under the given universe $\mathfrak{V}$, denoted as $A = \max_{\propto}^{\mathfrak{V}}[\mathcal{S}]$ iff $\forall\, X \subseteq \mathfrak{V} :\ X \propto \mathcal{S} \Rightarrow X \subseteq A$.

Example 3.8. Consider a dataset with Canadian postal codes: $\mathcal{S} = \{$"`V6E3V6`", "`V6C2S6`", "`V6V1X5`", "`V6X3S4`"$\}$. With $\mathfrak{V}$ = the default atoms (listed in Figure 3.4), I obtain the enriched set $\widehat{\mathfrak{V}}$ using Equation (3.1). Then, the maximal set of atoms compatible with $\mathcal{S}$ under $\widehat{\mathfrak{V}}$, *i.e.* $\max_{\propto}^{\widehat{\mathfrak{V}}}[\mathcal{S}]$ contains 18 atoms, such as "`V6`", "`V`", Upper, Upper^{+}, AlphaSpace, $\text{AlphaDigit}^{\times 6}$ *etc.*

For a given universe $\mathfrak{V}$ of atoms and a dataset $\mathcal{S}$, the GetMaxCompatibleAtoms method outlined in Figure 3.13 is invoked to efficiently compute the set $\Lambda = \max_{\propto}^{\widehat{\mathfrak{V}}}[\mathcal{S}]$, where $\widehat{\mathfrak{V}}$ denotes the enriched set of atoms based on $\mathfrak{V}$ given by Equation (3.1). Starting with $\Lambda = \mathfrak{V}$, in line 1, atoms that are not compatible with $\mathcal{S}$, *i.e.* fail to match at least one string $s \in \mathcal{S}$, are iteratively removed from Λ at line 4. At the same time, a hashtable C is maintained, which maps each Class atom to its width at line 6. C is used to enrich $\mathfrak{V}$ with fixed-width versions of Class atoms that are already specified in $\mathfrak{V}$. If the width of a Class atom is not constant over all strings in $\mathcal{S}$, it is removed from the hashtable C, at line 7. For each remaining Class atom α in C, a fixed-width variant is added for α to Λ. In line 8, RestrictWidth is invoked to generate the fixed-width variant for α with width $C[\alpha]$. Finally, Λ is also enriched with Const atoms — the longest common prefix L is computed across all strings, then every prefix of L is added to Λ, at line 12. Note that, GetMaxCompatibleAtoms does not explicitly compute the entire set $\widehat{\mathfrak{V}}$ of enriched atoms, but performs simultaneous pruning and enrichment on $\mathfrak{V}$ to ultimately compute their maximal compatible subset, $\Lambda = \max_{\propto}^{\widehat{\mathfrak{V}}}[\mathcal{S}]$.

func GetMaxCompatibleAtoms($\mathcal{S}$: String[], $\mho$: Atom[])
output: The maximal set of atoms that are compatible with $\mathcal{S}$

$C \leftarrow \{\}$; $\Lambda \leftarrow \mho$
for all $s \in \mathcal{S}$ **do**
  for all $\alpha \in \Lambda$ **do**
  ▶ Remove incompatible atoms.
    if $\alpha(s) = 0$ **then** Λ.Remove(α) ; C.Remove(α)
    else if $\alpha \in$ `Class` **then**
    ▶ Check if character class atoms maintain a fixed width.
      if $\alpha \notin C$ **then** $C[\alpha] \leftarrow \alpha(s)$
      else if $C[\alpha] \neq \alpha(s)$ **then** C.Remove(α)
▶ Add compatible fixed-width Class atoms.
for all $\alpha \in C$ **do** Λ.Add(RestrictWidth($\alpha, C[\alpha]$))
▶ Add compatible Const atoms.
$L \leftarrow$ LongestCommonPrefix($\mathcal{S}$)
Λ.Add(Const$_{L[0:1]}$, Const$_{L[0:2]}$, ..., Const$_L$)
return Λ

Figure 3.13: Computing the maximal set of compatible atoms

In essence, the problem of learning an expression $P\,[\texttt{SuffixAfter(}s, \alpha\texttt{)}]$ with spec φ is reduced to $\left|\max^{\widehat{\mho}}_{\propto}[\mathcal{S}]\right|$ subproblems for P with specs $\{\varphi_\alpha \mid \alpha \in \max^{\widehat{\mho}}_{\propto}[\mathcal{S}]\}$, where φ_α is as given by Equation (3.2), and $\widehat{\mho}$ denotes the enriched set of atoms derived from $\mho$ by Equation (3.1). Note that these subproblems are recursively reduced further, until all characters in each string are matched, and then synthesis terminates with `Empty`. Given this reduction logic as witness functions, PROSE performs these recursive synthesis calls efficiently, and finally combines the atoms to candidate patterns. I conclude this subsection with a comment on the soundness and completeness of $\mathcal{L}_{\mathsf{FP}}$.

Definition 3.8 (Soundness and $\mho$-Completeness)**.** A learner for $\mathscr{L}_{\mathsf{FP}}$ patterns is said to be *sound* if, for any dataset $\mathcal{S}$, every learned pattern P satisfies $\forall s \in \mathcal{S} : P(s) = \texttt{true}$.

A learner for $\mathscr{L}_{\mathsf{FP}}$, instantiated with a universe $\mathfrak{V}$ of atoms is said to be $\mathfrak{V}$-*complete* if, for any dataset $\mathcal{S}$, it learns every possible pattern $P \in \mathscr{L}_{\mathsf{FP}}$ over $\mathfrak{V}$ atoms that satisfy $\forall s \in \mathcal{S} : P(s) = \texttt{true}$.

Theorem 3.2 (Soundness and $\widehat{\mathfrak{V}}$-Completeness of $\mathcal{L}_{\mathsf{FP}}$)**.** *For an arbitrary set $\mathfrak{V}$ of user-specified atoms,* FlashProfile*'s pattern learner $\mathcal{L}_{FP}$ is sound and $\widehat{\mathfrak{V}}$-complete, where $\widehat{\mathfrak{V}}$ denotes the enriched set of atoms obtained by augmenting $\mathfrak{V}$ with constant-string and fixed-width atoms, as per Equation (3.1).*

Proof. Soundness is guaranteed since only compatible atoms are composed. $\widehat{\mathfrak{V}}$-completeness follows from the fact that always the *maximal* compatible subset of $\widehat{\mathfrak{V}}$ is considered. □

Due to the $\widehat{\mathfrak{V}}$-completeness of $\mathcal{L}_{\mathsf{FP}}$, once the set $\mathcal{L}_{\mathsf{FP}}(\mathcal{S})$ of patterns over $\mathcal{S}$ has been computed, a variety of cost functions may be used to select the most suitable pattern for $\mathcal{S}$ amongst all possible patterns over $\widehat{\mathfrak{V}}$, without having to invoke pattern learning again.

3.3.3 Cost of Patterns in $\mathscr{L}_{\mathsf{FP}}$

The cost function $\mathcal{C}_{\mathsf{FP}}$ assigns a real-valued score to each pattern $P \in \mathscr{L}_{\mathsf{FP}}$ over a given dataset $\mathcal{S}$, based on the structure of P and its behavior over $\mathcal{S}$. This cost function is used to select the most desirable pattern that represents the dataset $\mathcal{S}$. `Empty` is assigned a cost of 0 regardless of the dataset, since `Empty` can be the only pattern consistent with such datasets. For a pattern $P = \alpha_1 \diamond \ \ldots \ \diamond \alpha_k$, I define the cost $\mathcal{C}_{\mathsf{FP}}(P, \mathcal{S})$ with respect to a given dataset $\mathcal{S}$ as:

$$\mathcal{C}_{\mathsf{FP}}(P, \mathcal{S}) = \sum_{i=1}^{k} Q(\alpha_i) \cdot W(i, \mathcal{S} \mid P)$$

$\mathcal{C}_{\mathsf{FP}}$ balances the trade-off between a pattern's specificity and complexity. Each atom α in $\mathscr{L}_{\mathsf{FP}}$ has a statically assigned cost $Q(\alpha) \in (0, \infty]$, based on a priori bias for the atom. The

cost function $\mathcal{C}_{\mathsf{FP}}$ computes a sum over these static costs after applying a data-driven weight $W(i, \mathcal{S} \mid P) \in (0, 1)$:

$$W(i, \mathcal{S} \mid \alpha_1 \diamond \ \ldots \ \diamond \alpha_k) = \frac{1}{|\mathcal{S}|} \cdot \sum_{s \in \mathcal{S}} \frac{\alpha_i(s_i)}{|s|},$$

where $s_1 \stackrel{\text{def}}{=} s$ and $s_{i+1} \stackrel{\text{def}}{=} s_i[\alpha_i(s_i) :]$ denotes the remaining suffix of s_i after matching with α_i, as in Definition 3.6. This dynamic weight is an average over the fraction of length matched by α_i across $\mathcal{S}$. It gives a quantitative measure of how well an atom α_i generalizes over the strings in $\mathcal{S}$. With a sound pattern learner, an atomic match would never fail and $W(i, \mathcal{S} \mid P) > 0$ for all atoms α_i.

Example 3.9. Consider $\mathcal{S} = \{$"`Male`","`Female`"$\}$, that are matched by $P_1 = \mathsf{Upper} \diamond \mathsf{Lower}^+$, and $P_2 = \mathsf{Upper} \diamond \mathsf{Hex} \diamond \mathsf{Lower}^+$. Given FLASHPROFILE's static costs: $\{\mathsf{Upper} \mapsto 8.2,\ \mathsf{Hex} \mapsto 26.3,\ \mathsf{Lower}^+ \mapsto 9.1\}$, the costs for these two patterns shown above are:

$$\begin{aligned}
\mathcal{C}_{\mathsf{FP}}(P_1, \mathcal{S}) &= 8.2 \times \frac{1/4 + 1/6}{2} + 9.1 \times \frac{3/4 + 5/6}{2} = 8.9 \\
\mathcal{C}_{\mathsf{FP}}(P_2, \mathcal{S}) &= 8.2 \times \frac{1/4 + 1/6}{2} + 26.3 \times \frac{1/4 + 1/6}{2} + 9.1 \cdot \frac{2/4 + 4/6}{2} = 12.5
\end{aligned}$$

P_1 is chosen as best pattern, since $\mathcal{C}_{\mathsf{FP}}(P_1, \mathcal{S}) < \mathcal{C}_{\mathsf{FP}}(P_2, \mathcal{S})$.

Note that although $\mathsf{Hex}$is a more specific character class compared to $\mathsf{Upper}$and Lower, I assign it a higher static cost to avoid strings like "`face`" being described as $\mathsf{Hex}^{\times 4}$ instead of $\mathsf{Lower}^{\times 4}$. $\mathsf{Hex}^{\times 4}$ would be chosen over $\mathsf{Lower}^{\times 4}$ only if some other strings, such as "`f00d`", are found in the dataset, which cannot be described using $\mathsf{Lower}^{\times 4}$.

Static Cost (Q) for Atoms The learner $\mathcal{L}_{\mathsf{FP}}$ automatically assigns the static cost of a Const_s atom to be proportional to $1/|s|$, and the cost of a Class_c^z atom, with width $z \geq 1$, to be proportional to $Q(\mathsf{Class}_c^0)/z$. However, static costs for other kinds of atoms must be provided by the user.

Static costs for the default atoms, listed in Figure 3.4, were seeded with the values based on their estimated *size* — the number of strings the atom may match. Then they were penalized (e.g. the Hexatom) with empirically decided penalties to prefer patterns that are more *natural* to users. I describe the *quality* measure for profiles in § 3.4.2, which I have used to decide the penalties for the default atoms. In future, I plan to automate the process of penalizing atoms by designing a learning procedure which tries various perturbations to the seed costs to optimize profiling quality.

3.4 Evaluation

I now present experimental evaluation of the FlashProfile tool which implements the technique, focusing on the following key questions:

(§ 3.4.1) How well does the syntactic similarity measure capture similarity of real-life entities?

(§ 3.4.2) How accurate are the profiles? How do sampling and approximations affect them?

(§ 3.4.3) How fast is FlashProfile, with and without approximations, on real-life datasets?

(§ 3.4.4) Are the profiles natural and useful? How do they compare against existing tools?

Implementation I have implemented FlashProfile as a cross-platform C# library built using Microsoft PROSE [Cor17e]. It is now publicly available as part of the PROSE NuGet package.[16] All of the experiments were performed with PROSE 2.2.0 and .NET Core 2.0.0, on an Intel i7 3.60 GHz CPU with 32 GB RAM running 64-bit Ubuntu 17.10.

[16] FlashProfile has been publicly released as the `Matching.Text` module within the PROSE SDK. For more information, please see: https://microsoft.github.io/prose/documentation/matching-text/intro/.

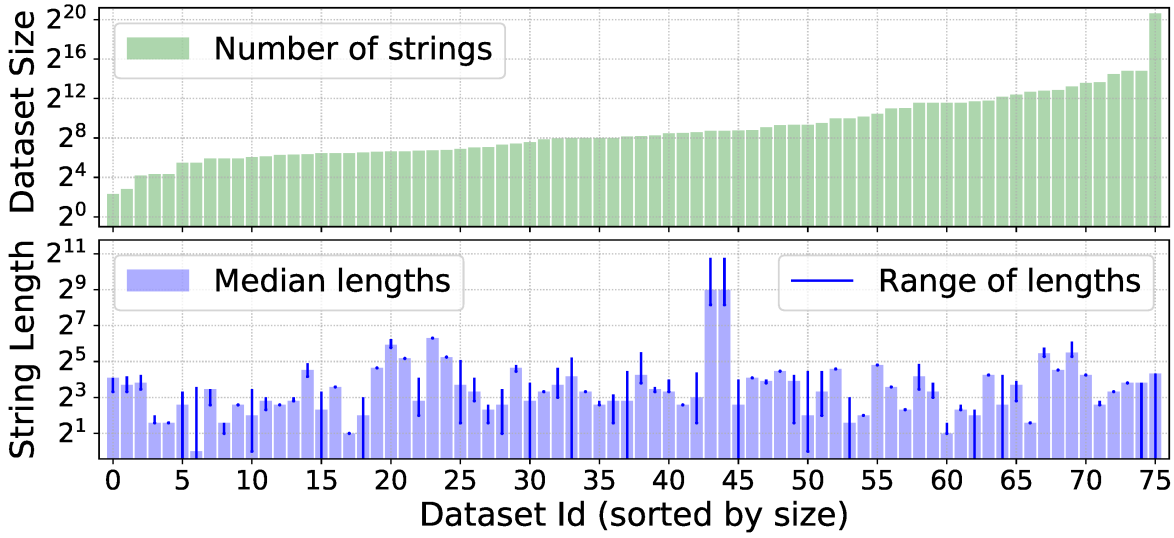


Figure 3.14: Number and length of strings across datasets[17]

Test Datasets I have collected 75 datasets from public sources,[17] spanning various domains such as names, postal codes, phone numbers, etc. Their sizes and the distribution of string lengths is shown in Figure 3.14. I sort them into three (overlapping) groups:

- CLEAN (25 datasets): Each of these datasets, uses a *single format* that is *distinct* from other datasets. I test syntactic similarity over them — strings from the same dataset must be labeled as similar.
- DOMAINS (63 datasets): These datasets belong to *mutually-distinct domains* but may exhibit multiple formats. I test the quality of profiles over them — a profile learned over fraction of a dataset should match rest of it, but should not be too general as to also match other domains.
- ALL (75 datasets): I test FLASHPROFILE's performance across all datasets.

[17] Datasets are available at: https://github.com/SaswatPadhi/FlashProfileDemo/tree/master/tests.

3.4.1 Syntactic Similarity

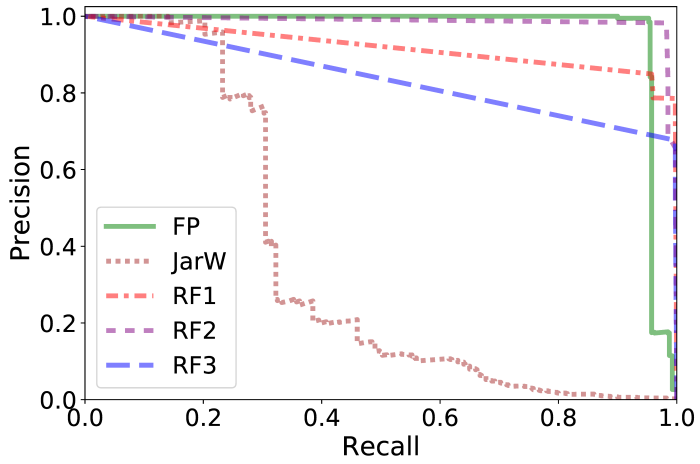
I evaluate the applicability of the dissimilarity measure from Definition 3.3, over real-life entities. From the CLEAN group, I randomly pick two datasets and select a random string from each. A good similarity measure should recognize when the pair is drawn from the same dataset by assigning them a lower dissimilarity value, compared to a pair from different datasets. For example, the pair {"`A. Einstein`", "`I. Newton`"} should have a lower dissimilarity value than {"`A. Einstein`", "`03/20/1998`"}. I instantiated FLASHPROFILE with only the default atoms listed in Figure 3.4 and tested 240400 such pairs. Figure 3.15 shows a comparison of the method against two simple baselines: **(a)** a character-wise edit-distance-based similarity measure (`JarW`), and **(b)** a machine-learned predictor (RF) over intuitive syntactic features.

For evaluation, I use the standard precision-recall (PR) [MRS08] measure. In the context, precision is the fraction of pairs that truly belongs to the same dataset, out of all pairs that are labeled to be "similar" by the predictor. Recall is the fraction of pairs retrieved by the predictor, out of all pairs truly drawn from same datasets. By varying the threshold for labelling a pair as "similar", I generate a PR curve and measure the area under the curve (AUC). A good similarity measure should exhibit high precision and high recall, and therefore have a high AUC.

First, observe that character-based measures [GF13] show poor AUC, and are not indicative of syntactic similarity. For instance, Levenshtein distance [Lev66], used within Google OPENREFINE [Inc10], exhibits a negligible AUC over the benchmarks. Although the Jaro-Winkler distance [Win99], indicated as `JarW` in Figure 3.15(a), shows a better AUC, it is quite low compared to both the and machine-learned predictors.

The second baseline is a standard random forest [Bre01] model RF using the syntactic features listed in Figure 3.15(b), such as difference in length, number of digits, etc. I train RF_1

[18] `len` returns string length, `begin`$\langle X \rangle$ checks if both strings begin with a character in X, `cnt`$\langle X \rangle$ counts occurrences of characters from X in a string, and $\Delta[f]$ computes $|f(s_1) - f(s_2)|^2$ for a pair of strings s_1 and s_2.



(a) Precision-Recall curves

Δ[len]
Δ[cnt⟨Digit⟩]
Δ[cnt⟨Lower⟩]
Δ[cnt⟨Upper⟩]
Δ[cnt⟨ ␣ ⟩]
Δ[cnt⟨“.”⟩]
Δ[cnt⟨“,”⟩]
Δ[cnt⟨“-”⟩]
begin⟨Upper⟩
begin⟨Lower⟩
begin⟨Digit⟩

(b) Features[18]

Predictor	FP	JarW	RF_1	RF_2	RF_3
AUC	96.28%	35.52%	91.73%	98.71%	76.82%

Figure 3.15: Similarity prediction accuracy of FlashProfile (FP) *vs.* a character-based measure (`JarW`), and random forests ($RF_{1...3}$) trained on different distributions

using $\sim 80,000$ pairs with $\left(1/25\right)^2 = 0.16\%$ pairs drawn from same datasets. In Figure 3.15(a) observe that the accuracy of RF is quite susceptible to changes in the distribution of the training data. RF_2 and RF_3 were trained with 0.64% and 1.28% pairs from same datasets, respectively. While RF_2 performs marginally better than the predictor, RF_1 and RF_3 perform worse.

3.4.2 Profiling Accuracy

I demonstrate the accuracy of FlashProfile along two dimensions:

- *Partitions:* The sampling and approximation techniques preserve partitioning accuracy
- *Descriptions:* The generated profiles are natural — not overly specific or general

For these experiments, I used FlashProfile with only the default atoms.

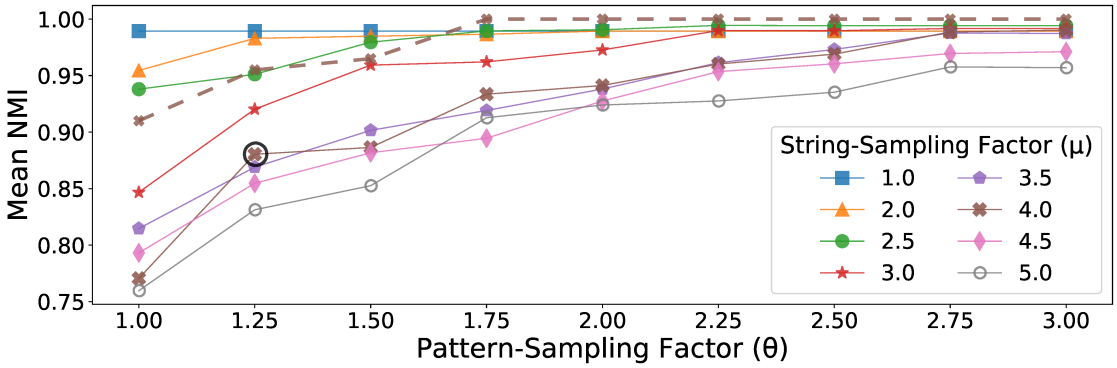


Figure 3.16: FLASHPROFILE's partitioning accuracy with different $\langle \mu, \theta \rangle$-configurations

Partitioning For each $c \in \{2, \ldots, 8\}$, I measure FLASHPROFILE's ability to repartition $256c$ strings — 256 strings collected from each of c randomly picked datasets from CLEAN. Over 10 runs for each c, I pick different sets of c files, shuffle the $256c$ strings, and invoke FLASHPROFILE to partition them into c clusters. For a fair distribution of strings across each run, I ignore one dataset from the CLEAN group which had much longer strings (> 1500 characters) compared to other datasets (10 – 100 characters). I experiment with different values of $1.0 \leq \mu \leq 5.0$ (*string-sampling factor*, which controls the size of chunks given to the core PROFILE method), and $1.0 \leq \theta \leq 3.0$ (*pattern-sampling factor*, which controls the approximation during hierarchical clustering).

I measure the precision of clustering using *symmetric uncertainty* [WFH17], which is a measure of normalized mutual information (NMI). An NMI of 1 indicates the resulting partitioning to be identical to the original clusters, and an NMI of 0 indicates that the final partitioning is unrelated to the original one. For each $\langle \mu, \theta \rangle$-configuration, I show the mean NMI of the partitionings over $10c$ runs (10 for each value of c), in Figure 3.16. The NMI improves with increasing θ, since I sample more dissimilarities, resulting in better approximations. However, the NMI drops with increasing μ, since more pairwise dissimilarities are approximated. Note that the number of string pairs increases quadratically with μ, but

reduces only linearly with θ. This is reflected in Figure 3.16 — for $\mu > 4.0$, the partitioning accuracy does not reach 1.0 even for $\theta = 3.0$. FLASHPROFILE's default configuration $\langle \mu = 4.0, \theta = 1.25 \rangle$, achieves a median NMI of 0.96 (mean 0.88) (indicated by a circled point). The dashed line indicates the median NMIs with $\mu = 4.0$. The median NMI is significantly higher than the mean, indicating the approximations were accurate in most cases. As I explain below in § 3.4.3, with $\langle \mu = 4.0, \theta = 1.25 \rangle$, FLASHPROFILE achieves the best performance *vs.* accuracy trade-off.

Descriptions I evaluate the suitability of the default profiles, by measuring their overall precision and recall. A natural profile should not be too specific — it should generalize well over the dataset (high true positives), but not beyond it (low false positives).

For each dataset in the DOMAINS, I profile a randomly selected 20% of its strings, and measure: **(a)** the fraction of the remaining dataset described by it, and **(b)** the fraction of an equal number of strings from other datasets, matched by it. Figure 3.17 summarizes the results. The lighter and darker shades indicate the fraction of true positives and false positives respectively. The white area at the top indicates the fraction of false negatives – the fraction of the remaining 80% of the dataset that is not described by the profile. The overall precision is 97.8%, and recall is 93.4%. The dashed line indicates a mean true positive rate of 93.2%, and the dotted line shows a mean false positive rate of 2.3%; across all datasets.

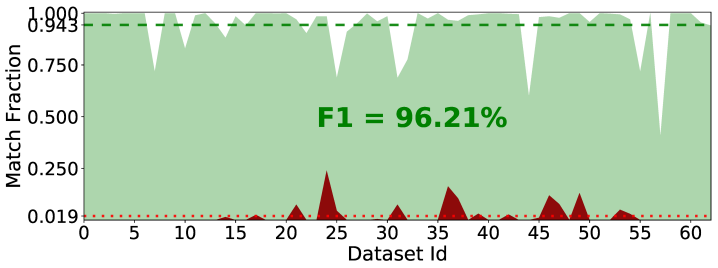
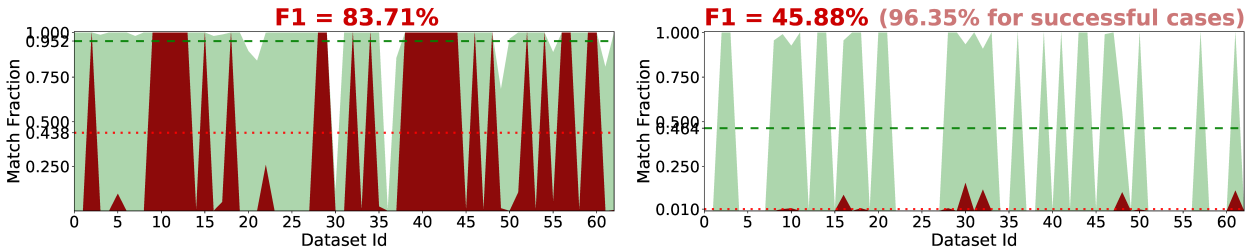


Figure 3.17: Quality of descriptions at $\langle \mu = 4.0, \theta = 1.25 \rangle$

I also perform similar quality measurements with Microsoft SSDT [Cor17d] and ATACCAMA [Cor17a]. I use "Column Pattern Profiling Tasks" feature within Microsoft SSDT with `PercentageDataCoverageDesired` = 100, and "Pattern Analysis" feature within the AT-



(a) Microsoft SSDT [Cor17d] **(b)** ATACCAMA [Cor17a]

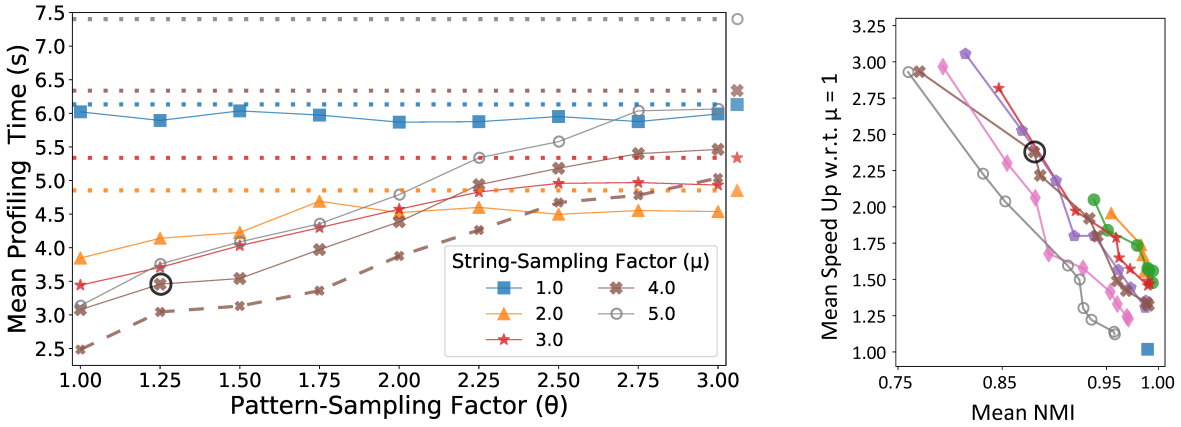
Figure 3.18: Quality of descriptions from current state-of-the-art tools

ACCAMA platform. I summarize the per-dataset description quality for Microsoft SSDT in Figure 3.18(a), and for ATACCAMA in Figure 3.18(b). A low overall F1 score is observed for both tools.

While Microsoft SSDT has a very high false positive rate, ATACCAMA has a high failure rate. For 27 out of 63 datasets, Microsoft SSDT generates "`.*`" as one of the patterns, and it fails to profile one dataset that has very long strings (up to 1536 characters). On the other hand, ATACCAMA fails to profile 33 datasets. But for the remaining 30 datasets, the simple atoms (digits, numbers, letters, words) used by ATACCAMA seem to work well — the profiles exhibit high precision and recall. Note that, this quantitative measure only captures the specificity of profiles, but not their readability. I present a qualitative comparison of profiles generated by these tools in Section 3.4.4.

3.4.3 Performance

I measure the mean profiling time with various $\langle \mu, \theta \rangle$-configurations, and summarize the findings in Figure 3.19(a). The dotted lines indicate profiling time without pattern sampling, *i.e.* $\theta \to \infty$, for different values of the μ factor. The dashed line shows the median profiling time for different values of θ with the default $\mu = 4.0$. I also show the performance-accuracy trade off in Figure 3.19(b) by measuring the mean speed up of each configuration w.r.t.



(a) Mean Profiling Time **(b)** Performance ∼ Accuracy

Figure 3.19: Impact of sampling on performance (using the same colors and markers as Figure 3.16)

$\langle \mu = 1.0, \theta = 1.0 \rangle$. I select the *Pareto optimal* point $\langle \mu = 4.0, \theta = 1.25 \rangle$ as FlashProfile's default configuration. It achieves a mean speed up of 2.3× over $\langle \mu = 1.0, \theta = 1.0 \rangle$, at a mean NMI of 0.88 (median = 0.96).

As one would expect, the profiling time increases with θ, due to sampling more patterns, which in turn result in more calls to $\mathcal{L}_{\text{FP}}$. The dependence of profiling time on μ however, is more interesting. Notice that with $\mu = 1$, the profiling time is *higher* than any other configurations, when pattern sampling is enabled, *i.e.* $\theta \neq \infty$ (solid lines). This is due to the fact that FlashProfile learns very specific profiles with $\mu = 1$ with very small samples of strings, which do not generalize well over the remaining data. This results in many Sample-Profile-Filter iterations. Also note that with pattern-sampling enabled, the profiling time decreases with μ until $\mu = 4.0$ as, and then increases as profiling larger samples of strings becomes expensive.

Finally, I evaluate FlashProfile's performance on end-to-end real-life profiling tasks on all 75 datasets, that have a mixture of clean and dirty datasets. Over 153 tasks – 76 for automatic profiling, and 77 for refinement, The median profiling time is roughly 0.7 s. With the default configuration, 77% of the requests are fulfilled within 2 seconds – 70% of

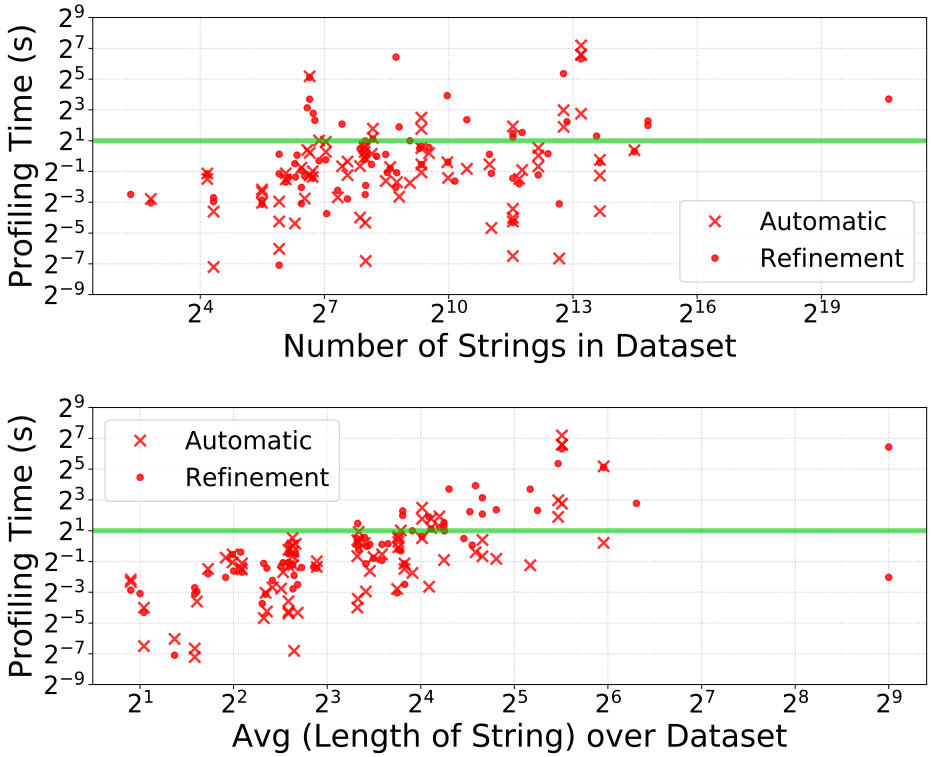


Figure 3.20: Performance of FLASHPROFILE over real-life datasets

automatic profiling tasks, and 83% of refinement tasks. In Figure 3.20 I show the variance of profiling times w.r.t. size of the datasets (number of strings in them), and the average length of the strings in the datasets (all axes being logarithmic). Observe that the number of string in the dataset doesn't have a strong impact on the profiling time. This is expected, since I only sample smaller chunks of datasets, and remove strings that are already described by the profile I have learned so far. I repeated this experiment with 5 dictionary-based custom atoms: ⟨DayName⟩, ⟨ShortDayName⟩, ⟨MonthName⟩, ⟨ShortMonthName⟩, ⟨US_States⟩, and noticed an increase of $\sim 0.02\,\text{s}$ in the median profiling time.

3.4.4 Comparison of Learned Profiles

[19] Dataset collected from a database of vendors across US and Canada: https://goo.gl/PGS2pL

Zip Code
99518
61021-9150
⋮
2645
83716
⋮
K0K 2C0
14480
S7K7K9

(a) Dataset

LDL DLD
LDLDLD
N-N
N

(b) AUTOTYPE

\w\w\w \w\w\w
\d\d\d\d\d
\d\d\d\d
.*

(c) Microsoft SSDT

U D U ␣ D U D
"61" D^3 "-" D^4
"S7K7K9"
D^+
ϵ

(d) FLASHPROFILE

U D U ␣ D U D
"61" D^3 "-" D^4
"S7K7K9"
D^5
D^4
ϵ

(e) FLASHPROFILE (6)

Most frequent pattern from POTTERSWHEEL= `int`

Table 3.5: Profiles for a dataset with zip codes[19]

I compare the profiles learned by FLASHPROFILE to the outputs from 3 state-of-the-art tools: **(a)** AUTOTYPE [Cor17a]: a dedicated profiling tool,, **(b)** Microsoft SSDT [Cor17d] a feature-rich IDE for database applications, and **(c)** POTTERSWHEEL [RH01]: a tool that detects the most frequent data pattern and predicts anomalies in data. Table 3.5 and Table 3.6 show the observed outputs. I list the profiles generated on requesting exactly k patterns against FLASHPROFILE$_k$. For brevity, I **(a)** omit the concatenation operator " $\diamond$ " between atoms, and **(b)** abbreviate Digit $\mapsto$ D, Upper $\mapsto$ U, AlphaSpace $\mapsto$ Π, AlphaDigitSpace $\mapsto$ Σ.

First, observe that Microsoft SSDT generates an overly general "`.*`" pattern in both cases. AUTOTYPE generates a very coarse grained profile in both cases, which although explains the pattern of special characters, does not say much about other syntactic properties, such as common prefixes, or fixed-length patterns. With FLASHPROFILE, one can immediately notice in Table 3.5(d), that "S7K7K9" is the only Canadian zip code which does not have a space in the middle, and that some US zip codes have 4 digits instead of 5 (probably the leading

[20] Dataset collected from https://portal.its.pdx.edu/fhwa

Routes
OR-213
I-5 N
I-405 S
OR-141
⋮
OR-99E
US-26 E
12348 N CENTER
US-217 S
⋮
I-84 E
US 26(SUNSET)
OR-224

(a) Dataset

N L W
W N (W)
W N (W W W)
W-N
W-NW
W-N W

(b) Ataccama

US-26 E
US-26 W
I-5 N
I-5 S
I-84 E
I-84 W
I-\d\d\d N
I-\d\d\d S
.*

(c) Microsoft SSDT

“12348 N CENTER”
“US 26(” Π^+ “)”
U^+ “-” Σ^+
ϵ

(d) FlashProfile

“12348 N CENTER”
“US 26(SUNSET)”
“US 26(MT HOOD HWY)”
U^+ “-” D^+
U^2 “-” D^2 U^+
U^+ “-” D^+ ␣ U^+
ϵ

(e) FlashProfile (7)

“US-30BY”	“12348 N CENTER”
ϵ	“US 26(SUNSET)”
U^+ “-” D^+	“OR-99” U^1
“I-” D^+ ␣ U^+	U^2 “-2” D^+ ␣ U^1
“US 26(MT HOOD HWY)”	

(f) FlashProfile (9)

“US-30BY”	“12348 N CENTER”
“I-5”	“US-26” ␣ U^1
“US-30”	“US 26(SUNSET)”
“OR-” D^+	“OR-99” U^1
“I-5” ␣ U^+	“I-” D^+ ␣ U^1
ϵ	“OR-217” ␣ U^1
“US 26(MT HOOD HWY)”	

(g) FlashProfile (13)

Most frequent pattern from PottersWheel= `IspellWord int space AllCapsWord`

Table 3.6: Profiles for a dataset containing US routes[20]

zero was lost while interpreting it as a number). Similarly, one can immediately observe that in Table 3.6(d), “`12348 N CENTER`” is not a route. Similarly the pattern “`US 26(`” Π^+ “`)`” indicates that it is the only entry with a space instead of a dash between the “`US`” and “`26`”.

In many real-life scenarios, simple statistical profiles are not enough for data understanding or validation. FlashProfile allows users to gradually drill into the data by requesting profiles with a desired granularity. Furthermore, they may also provide custom atoms for domain-specific profiling.

3.5 Applications in PBE Systems

In this section, I discuss how syntactic profiles can improve programming-by-example (PBE) [Lie01, GPS17] systems, which synthesize a desired program from a small set of input-output examples. For instance, given an example "`Albert Einstein`" ⇝ "`A.E.`", the system should learn a program that extracts the initials for names. Although many PBE systems exist today, most share criticisms on low usability and confidence in them [Lau09, MSG15].

Examples are an inherently under-constrained form of specifying the desired program behavior. Depending on the target language, a large number of programs may be consistent with them. Two major challenges faced by PBE systems today are: **(a)** obtaining a set of examples that accurately convey the desired behavior to limit the space of synthesized programs, and **(b)** ranking these programs to select the ones that are *natural* to users.

In a recent work, Ellis *et al.* [EG17] address **(b)** using data profiles. They show that incorporating profiles for input-output examples significantly improves ranking, compared to traditional techniques which only examine the structure of the synthesized programs. I show that data profiles can also address problem **(a)**. Raychev *et al.* [RBV16] have presented *representative data samplers* for synthesis scenarios, but they require the outputs for all inputs. In contrast, I show a novel interaction model for proactively requesting users to supply the desired outputs for syntactically different inputs, thereby providing a representative set of examples to the PBE system.

Significant Inputs Typically, users provide outputs for only the first few inputs of target dataset. However, if these are not representative of the entire dataset, the system may not learn a program that generalizes over other inputs. Therefore, I propose a novel interaction model that requests the user to provide the desired outputs for *significant* inputs, incrementally. A significant input is one that is syntactically the most dissimilar with all previously labelled inputs.

func OrderPartitions$_{\langle\mathcal{L},\mathcal{C}\rangle}(\widetilde{P}$: Profile)

output: A sequence of partitions $\langle\mathcal{S}_1, \ldots, \mathcal{S}_{|\widetilde{P}|}\rangle$ over $\mathcal{S}$

► Select with the partition that has the minimum cost.

1: $\rho \leftarrow \langle\big(\text{argmin}_{X\in\widetilde{P}}\ \mathcal{C}(X.\text{Pattern}, X.\text{Data})\big).\text{Data}\rangle$

2: **while** $|\,\rho\,| < |\,\widetilde{P}\,|$ **do**

► Pick the most dissimilar partition w.r.t.those in ρ.

3: $T \leftarrow \text{argmax}_{Z\in\widetilde{P}}\ \min_{X\in\rho}\big(\text{LearnBestPattern}_{\langle\mathcal{L},\mathcal{C}\rangle}(Z.\text{Data} \cup X)\big).\text{Cost}$

4: ρ.Append(T.Data)

5: **return** ρ

Figure 3.21: Ordering partitions by mutual dissimilarity

I start with a syntactic profile $\widetilde{P}$ for the input dataset and invoke the OrderPartitions function, listed in Figure 3.21, to order the partitions identified in $\widetilde{P}$ based on mutual dissimilarity, *i.e.* each partition $\mathcal{S}_i$ must be as dissimilar as possible with (its most-similar neighbor within) the partitions $\{\mathcal{S}_1, \ldots, \mathcal{S}_{i-1}\}$. It is a simple extension of the SampleDissimilarities procedure (Figure 3.8) to work with sets of strings instead of strings. I start with the partition that can be described with the minimum-cost pattern. Then, from the remaining partitions, I iteratively select the one that is most dissimilar to those previously selected. I define the dissimilarity between two partitions as the cost of the best (least-cost) pattern required to describe them together.

Once I have an ordered set of partitions, $\langle\mathcal{S}_1, \ldots, \mathcal{S}_{|\widetilde{P}|}\rangle$, I request the user to provide the desired output for a randomly selected input from each partition in order. Since PBE systems like Flash Fill are interactive, and start synthesizing programs right from the first example, the user can inspect and skip over inputs for which the output is correctly predicted by the synthesized program. After one cycle through all partitions, I restart from partition $\mathcal{S}_1$, and request the user to provide the output for a new random string in each partition.

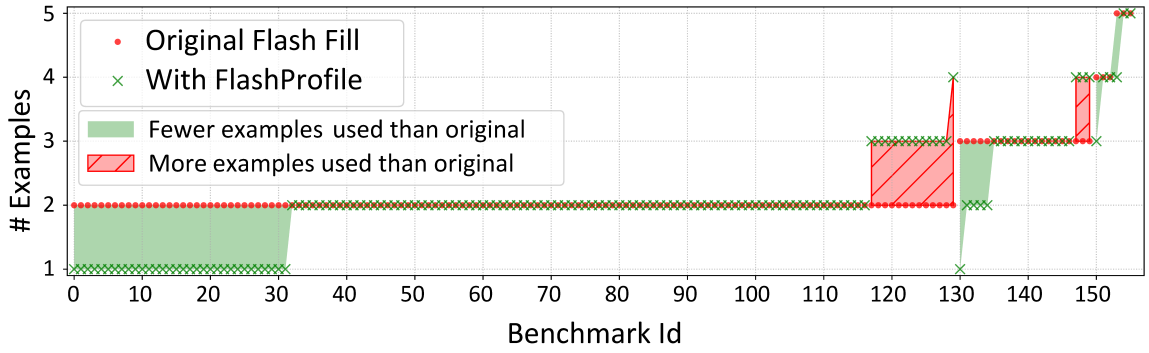


Figure 3.22: Examples needed with and without FlashProfile

I evaluate the proposed interaction model over 163 Flash Fill benchmarks[21] that require more than one example to learn the desired string-transformation program. Figure 3.22 compares the number of examples required originally, to that using the interaction model. Seven cases that timeout due to the presence of extremely long strings have been omitted.

Over the remaining 156 cases, Flash Fill **(a)** requires a single example per partition for 131 (=80%) cases, and **(b)** uses the minimal set[22] of examples to synthesize the desired program for 140(=86%) cases — 39 of which were improvements over Flash Fill. Thus, **(1)** validates the hypothesis that the partitions indeed identify representative inputs, and **(2)** further indicates that the interaction model is highly effective. Selecting inputs from partitions ordered based on mutual syntactic dissimilarity helps Flash Fill converge to the desired programs with fewer examples. Note that, these results are based on the default set of atoms. Designing custom atoms for string-transformation tasks, based on Flash Fill's semantics is also an interesting direction.

[21] These benchmarks are a superset of the original set of Flash Fill [Gul11] benchmarks, with many more real-world scenarios collected from customers using products powered by PROSE [Cor17e].

[22] By *minimal*, I mean that there is no smaller set of examples with which Flash Fill can synthesize the desired program.

Although the significant inputs scenario is similar to *active learning*, which is well studied in machine-learning literature [Han14], typical active-learning methods require hundreds of labeled examples. In contrast, PBE systems deal with very few examples [MSG15].

3.6 Related Work

There has been a line of work on profiling various aspects of datasets — Abedjan *et al.* [AGN15] present a recent survey. Traditional techniques for summarizing data target statistical profiles [CGH12], such as sampling-based aggregations [HNS95], histograms [Ioa03], and wavelet-based summaries [KM07]. However, pattern-based profiling is relatively underexplored, and is minimally or not supported by state-of-the-art data analysis tools [Inc10, Cor17d, Cor17a, Tri17].

To the knowledge, no existing approach learns syntactic profiles over an extensible language and allows refinement of generated profiles. I present a novel dissimilarity measure which is the key to learning refinable profiles over arbitrary user-specified patterns. Microsoft's SQL Server Data Tools (Microsoft SSDT) [Cor17d] learns rich regular expressions but is neither extensible not comprehensive. A dedicated profiling tool Ataccama [Cor17a] generates comprehensive profiles over a very small set of base patterns. Google OpenRefine [Inc10] does not learn syntactic profiles, but it allows clustering of strings using character-based similarity measures [GF13]. In § 3.4 I show that such measures do not capture syntactic similarity. While PottersWheel [RH01] does not learn a complete profile, it learns the most frequent data pattern over arbitrary user-defined *domains* that are similar to the atomic patterns.

Application-Specific Structure Learning There has been prior work on learning specific structural properties aimed at aiding data wrangling applications, such as data transformations [RH01, Sin16], information extraction [LKR08], and reformatting or text normal-

ization [KG15]. However, these approaches make specific assumptions regarding the target application, which do not necessarily hold when learning general purpose profiles. Although profiles generated by FlashProfile are primarily aimed at data understanding, in § 3.5 I show that they may aid PBE applications, such as Flash Fill [Gul11] for data transformation. Bhattacharya *et al.* [BHC15] also utilize hierarchical clustering to group together sensors used in building automation based on their tags. However, they use a fixed set of domain-specific features for tags and do not learn a pattern-based profile.

Grammar Induction Syntactic profiling is also related to the problem of learning regular expressions, or more generally grammars from a given set of examples. De la Higuera [Hig10] present a recent survey on this line of work. Most of these techniques, such as LStar [Ang87] and RPNI [OG92], assume availability of both positive and negative examples, or a membership oracle. Bastani *et al.* [BSA17] show that these techniques are either too slow or do not generalize well and propose an alternate strategy for learning grammars from positive examples. When a large number of negative examples are available, genetic programming has also been shown to be useful for learning regular expressions [Svi98, BDD12]. Finally, LearnPADS [FWZ08, ZFW12] also generates a syntactic description, but does not support refinement or user-specified patterns.

Program Synthesis The techniques for sampling-based approximation and finding representative inputs relate to prior work by Raychev *et al.* [RBV16] on synthesizing programs from noisy data. However, they assume a single target program and the availability of outputs for all inputs. In contrast, I synthesize a disjunction of several programs, each of which returns `true` only on a specific partition of the inputs, which is unknown a priori.

FlashProfile's pattern learner uses the PROSE library [Cor17e], which implements the FlashMeta framework [PG15] for inductive program synthesis, specifically programming-by-examples (PBE) [Lie01, GPS17]. PBE has been leveraged by recent works on automat-

ing repetitive text-processing tasks, such as string transformation [Gul11, Sin16], extraction [LG14], and format normalization [KG15]. However, unlike these applications, data profiling does not solicit any (output) examples from the user. I demonstrate a novel application of a supervised synthesis technique to solve an unsupervised learning problem.

CHAPTER 4

Overfitting in Program Synthesis

The previous chapters utilized different forms of program synthesis for generating invariants and data profiles from a user-specified grammar. In this chapter I take a step back and investigate a common challenge in scaling these synthesis techniques to complex grammars. Specifically, I study the impact of grammar expressiveness on the performance of grammar-based example-driven synthesis.

The *syntax-guided synthesis* (SyGuS) framework [ABJ13] provides a unified format to describe a program synthesis problem by supplying **(a)** a logical specification for the desired functionality, and **(b)** a grammar of allowed implementations. Given these two inputs, a SyGuS tool searches through the programs that are permitted by the grammar to generate one that meets the specification. Today, SyGuS is at the core of several state-of-the-art program synthesizers [END18, PSM16, ARU17, LCL17, LHA18], many of which compete annually in the SyGuS competition [Org14, AF19].

I demonstrate empirically that five state-of-the-art SyGuS tools are very sensitive to the choice of grammar. Increasing grammar expressiveness allows the tools to solve some problems that are unsolvable with less-expressive grammars. However, it also causes them to fail on many problems that the tools are able to solve with a less expressive grammar. I analyze the latter behavior both theoretically and empirically and present techniques that make existing tools much more robust in the face of increasing grammar expressiveness.

I restrict my investigation to a widely used approach [ASF18] to SyGuS called *counterexample-guided inductive synthesis* (CEGIS) [Sol13, §5]. In this approach, the synthesizer is composed

of a learner and an oracle. The learner iteratively identifies a candidate program that is consistent with a given set of examples (initially empty) and queries the oracle to either prove that the program is *correct*, *i.e.* meets the given specification, or obtain a counterexample that demonstrates that the program does not meet the specification. The counterexample is added to the set of examples for the next iteration. The iterations continue until a correct program is found or resource/time budgets are exhausted. The LoopInvGen technique presented in Chapter 2 utilized the CEGIS approach for inferring provably sufficient loop invariants for program verification.

Overfitting. To better understand the observed performance degradation, I instrumented one of these SyGuS tools (Section 4.1.2). I empirically observe that for a large number of problems, the performance degradation on increasing grammar expressiveness is often accompanied by a significant increase in the number of counterexamples required. Intuitively, as grammar expressiveness increases so does the number of *spurious* candidate programs, which satisfy a given set of examples but violate the specification. If the learner picks such a candidate, then the oracle generates a counterexample, the learner searches again, and so on.

In other words, increasing grammar expressiveness increases the chances for *overfitting*, a well-known phenomenon in machine learning (ML). Overfitting occurs when a learned function explains a given set of observations but does not generalize correctly beyond it. Since SyGuS is indeed a form of function learning, it is perhaps not surprising that it is prone to overfitting. However, I identify its specific source in the context of SyGuS — the spurious candidates induced by increasing grammar expressiveness — and show that it is a significant problem in practice. I formally define the *potential for overfitting* (Ω), in Definition 4.7, which captures the number of spurious candidates.

No Free Lunch. In the ML community, this tradeoff between expressiveness and overfitting has been formalized for various settings as *no-free-lunch* (NFL) theorems [SB14, §5.1].

Intuitively such a theorem says that for every learner there exists a function that cannot be efficiently learned, where efficiency is defined by the number of examples required. I prove corresponding NFL theorems for the CEGIS-based SyGuS setting (Theorems 4.1 and 4.2).

A key difference between the ML and SyGuS settings is the notion of *m-learnability*. In the ML setting, the learned function may differ from the true function, as long as this difference (expressed as an error probability) is relatively small. However, because the learner is allowed to make errors, it is in turn required to learn given an arbitrary set of m examples (drawn from some distribution). In contrast, the SyGuS learning setting is *all-or-nothing* — either the tool synthesizes a program that meets the given specification or it fails. Therefore, it would be overly strong to require the learner to handle an arbitrary set of examples.

Instead, I define a much weaker notion of m-learnability for SyGuS, which only requires that there *exist* a set of m examples for which the learner succeeds. Yet, my NFL theorem shows that even this weak notion of learnability can always be thwarted: given an integer $m \geq 0$ and an expressive enough (as a function of m) grammar, for every learner there exists a SyGuS problem that cannot be learned without access to more than m examples. I also prove that overfitting is inevitable with an expressive enough grammar (Theorems 4.3 and 4.4) and that the potential for overfitting increases with grammar expressiveness (Theorem 4.5).

Mitigating Overfitting. Inspired by *ensemble methods* [Die00] in ML, which aggregate results from multiple learners to combat overfitting (and underfitting), I propose PLearn — a black-box framework that runs multiple parallel instances of a SyGuS tool with different grammars. Although prior SyGuS tools run multiple instances of learners with different random seeds [JQS15, BCD11], to my knowledge, this is the first proposal to explore multiple grammars as a means to improve the performance of SyGuS. My experiments indicate that PLearn significantly improves the performance of five state-of-the-art SyGuS tools — CVC4 [RDK15, BCD11], EUSolver [ARU17], LoopInvGen [PSM16], SketchAC [JQS15, Sol13], and Stoch [ABJ13, III F].

However, running parallel instances of a synthesizer is computationally expensive. Hence, I also devise a white-box approach, called *hybrid enumeration*, that extends the enumerative synthesis technique [AGK13] to efficiently interleave exploration of multiple grammars in a single SyGuS instance. I have implemented hybrid enumeration within LoopInvGen[23] and show that the resulting single-threaded learner, HE+LoopInvGen, has negligible overhead but achieves performance comparable to that of PLearn for LoopInvGen. Moreover, HE+LoopInvGen significantly outperforms the winner [PSM17] of the invariant-synthesis (`Inv`) track of 2018 SyGuS competition [AF19] — a variant of LoopInvGen specifically tuned for the competition — including a 5× mean speedup and solving two SyGuS problems that no tool in the competition could solve.

Contributions. In summary, I present the following contributions:

(§ 4.1) I empirically observe that, in many cases, increasing grammar expressiveness degrades performance of existing SyGuS tools due to *overfitting*.

(§ 4.2) I formally define overfitting and prove *no-free-lunch* theorems for the SyGuS setting, which indicate that overfitting with increasing grammar expressiveness is a fundamental characteristic of SyGuS.

(§ 4.3) I propose two mitigation strategies – **(a)** a black-box technique that runs multiple parallel instances of a synthesizer, each with a different grammar, and **(b)** a single-threaded enumerative technique, called *hybrid enumeration*, that interleaves exploration of multiple grammars.

(§ 4.4) I show that incorporating these mitigating measures in existing tools significantly improves their performance.

[23] The implementation and benchmarks are available at `https://github.com/SaswatPadhi/LoopInvGen`.

4.1 Motivation

In this section, I first present empirical evidence that existing SyGuS tools are sensitive to changes in grammar expressiveness. Specifically, I show that as I increase the expressiveness of the provided grammar, every tool starts failing on some benchmarks that it was able to solve with less-expressive grammars. I then investigate one of these tools in detail.

4.1.1 Grammar Sensitivity of SyGuS Tools

I evaluated 5 state-of-the-art SyGuS tools that use very different techniques:

- SketchAC [JQS15] extends the Sketch synthesis system [Sol13] by combining both explicit and symbolic search techniques.
- Stoch [ABJ13, III F] performs a stochastic search for solutions.
- EUSolver [ARU17] combines enumeration with unification strategies.
- Reynolds *et al.* [RDK15] extend CVC4 [BCD11] with a refutation-based approach.
- LoopInvGen [PSM16] combines enumeration and Boolean function learning.

$\langle b \rangle \models$ `true` | `false` | ⟨`Bool` variables⟩
| `(not` b`)` | `(or` b b`)` | `(and` b b`)`
$\langle i \rangle \models$ ⟨`Int` constants⟩ | ⟨`Int` variables⟩

▶ Additional rule in Equalities grammar:
$\langle b \rangle \models^{+}$ `(=` i i`)`

▶ Additional rules in Intervals grammar:
$\langle b \rangle \models^{+}$ `(>` i i`)` | `(>=` i i`)`
| `(<` i i`)` | `(<=` i i`)`

▶ Additional rules in Octagons grammar:
$\langle i \rangle \models^{+}$ `(+` i i`)` | `(-` i i`)`

▶ Additional rule in Polyhedra grammar:
$\langle i \rangle \models^{+}$ $(*_S\ i\ i)$

▶ Additional rule in Polynomials grammar:
$\langle i \rangle \models^{+}$ $(*_N\ i\ i)$

▶ Additional rule in Peano grammar:
$\langle i \rangle \models^{+}$ `(div` i i`)` | `(mod` i i`)`

Figure 4.1: Grammars of quantifier-free predicates over integers [24]

I ran these five tools on 180 invariant-synthesis benchmarks, which I describe in Section 4.4. I ran the benchmarks with each of the six grammars of quantifier-free predicates, which are shown in Figure 4.1. These grammars correspond to widely used abstract domains in the analysis of integer-manipulating programs — Equalities, Intervals [CC77b],

[24] We use the $\models$ operator to append new rules to previously defined nonterminals.

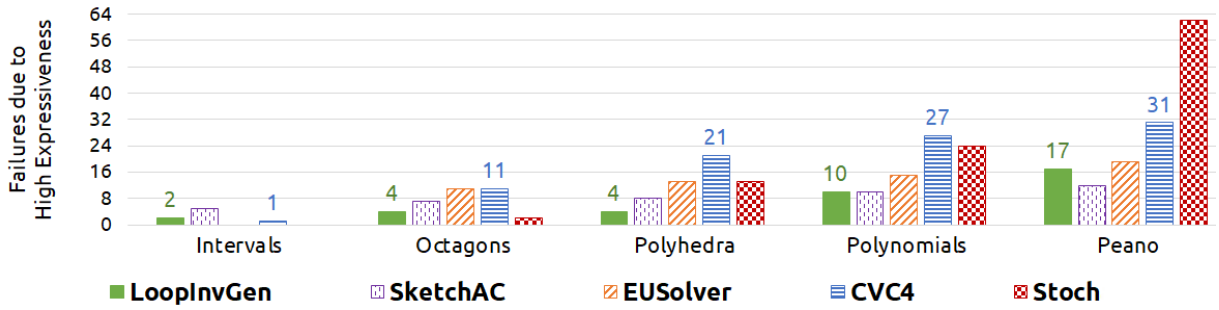


Figure 4.2: For each grammar, each tool, the ordinate shows the number of benchmarks that *fail* with the grammar but are solvable with a less-expressive grammar.

Octagons [Min06], Polyhedra [CH78], algebraic expressions (Polynomials) and arbitrary integer arithmetic (Peano) [Pea88]. The $*_S$ operator denotes scalar multiplication, *e.g.*`(*s 2 x)`, and $*_N$ denotes nonlinear multiplication, *e.g.*`(*n x y)`.

In Figure 4.2, I report my findings on running each benchmark on each tool with each grammar, with a 30-minute wall-clock timeout. For each ⟨tool, grammar⟩ pair, the y-axis shows the number of failing benchmarks that the same tool is able to solve with a less-expressive grammar. Observe that, for each tool, the number of such failures increases with the grammar expressiveness. For instance, introducing the scalar multiplication operator ($*_S$) causes CVC4 to fail on 21 benchmarks that it is able to solve with Equalities ($^4/_{21}$), Intervals ($^{18}/_{21}$), or Octagons ($^{10}/_{21}$). Similarly, adding nonlinear multiplication causes LoopInvGen to fail on 10 benchmarks that it can solve with a less-expressive grammar.

4.1.2 Evidence for Overfitting

To better understand this phenomenon, I instrumented LoopInvGen [PSM16] to record the candidate expressions that it synthesizes and the number of CEGIS iterations (called *rounds* henceforth). I compare each pair of successful runs of each of my 180 benchmarks on

	Increase (↑)	Unchanged (=)	Decrease (↓)
Expressiveness$^{\uparrow}$ ∧ Time$^{\uparrow}$ → Rounds$^{?}$	27 %	67 %	6 %
Expressiveness$^{\uparrow}$ ∧ Rounds$^{\uparrow}$ → Time$^{?}$	79 %	6 %	15 %

Table 4.1: Observed correlation between synthesis time and number of rounds, upon increasing grammar expressiveness, with LoopInvGen [PSM16] on 180 benchmarks

distinct grammars.[25] In 65 % of such pairs, performance degradation is observed with the more expressive grammar. I also report the correlation between performance degradation and number of rounds for the more expressive grammar in each pair in Table 4.1.

In 67 % of the cases with degraded performance upon increased grammar expressiveness, the number of rounds remains unaffected — indicating that this slowdown is mainly due to a larger search space. However, there is significant evidence of performance degradation due to *overfitting* as well. Note an increase in the number of rounds for 27 % of the cases with degraded performance. Moreover, I notice performance degradation in 79 % of all cases that required more rounds on increasing grammar expressiveness.

Thus, a more expressive grammar not only increases the search space, but also makes it more likely for LoopInvGen to overfit — select a spurious expression, which the oracle rejects with a counterexample, hence requiring more rounds. In the remainder of this section, I demonstrate this overfitting phenomenon on the verification problem shown in Figure 4.3, an example by Gulwani *et al.* [GJ07], which is the `fib_19` benchmark in the `Inv` track of SyGuS-Comp 2018 [AF19].

1: **assume** $(0 \leq n \ \wedge \ 0 \leq m \leq n)$
2: **assume** $(x = 0 \ \wedge \ y = m)$
3: **while** $(x < n)$ **do**
4: $\quad x \leftarrow x + 1$
5: $\quad$ **if** $(x > m)$ **then** $y \leftarrow y + 1$
6: **assert** $(y = n)$

Figure 4.3: The `fib_19` benchmark [GJ07]

[25] I ignore failing runs since they require an unknown number of rounds.

For Figure 4.3, an inductive invariant must be strong enough to prove that the assertion on line 6 always holds. In the SyGuS setting, the invariant is a predicate $\mathcal{I} : \mathbb{Z}^4 \to \mathbb{B}$ defined on a symbolic state $\sigma = \langle m, n, x, y \rangle$, that satisfies $\forall \sigma.\, \varphi(\mathcal{I}, \sigma)$ for the specification φ:[26]

$$\begin{aligned}
\varphi(\mathcal{I}, \sigma) \stackrel{\text{def}}{=} & \big(0 \leq n \wedge 0 \leq m \leq n \wedge x = 0 \wedge y = m\big) \implies \mathcal{I}(\sigma) && \text{(precondition)} \\
& \wedge \forall \sigma'.\, \big(\mathcal{I}(\sigma) \wedge T(\sigma, \sigma')\big) \implies \mathcal{I}(\sigma') && \text{(inductiveness)} \\
& \wedge \big(x \geq n \wedge \mathcal{I}(\sigma)\big) \implies y = n && \text{(postcondition)}
\end{aligned}$$

where $\sigma' = \langle m', n', x', y' \rangle$ denotes the new state after one iteration, and T is a transition relation that describes the loop body:

$$\begin{aligned}
T(\sigma, \sigma') \stackrel{\text{def}}{=} & \ (x < n) \ \wedge \ (x' = x + 1) \ \wedge \ (m' = m) \ \wedge \ (n' = n) \\
& \wedge \big[\, (x' \leq m \wedge y' = y) \ \vee \ (x' > m \wedge y' = y + 1) \,\big]
\end{aligned}$$

In Table 4.2(a), I report the performance of LoopInvGen on `fib_19` (Figure 4.3) with my six grammars (Figure 4.1). It succeeds with all but the least-expressive grammar. However, as grammar expressiveness increases, the number of rounds increase significantly — from 19 rounds with Intervals to 88 rounds with Peano.

LoopInvGen converges to the *exact same* invariant with both Polyhedra and Peano but requires 30 more rounds in the latter case. Tables 4.2(b) and 4.2(c) show some expressions synthesized with Polyhedra and Peano respectively. These expressions are solutions to intermediate subproblems — the final invariant is a conjunction of a subset of these expressions [PSM16, §3.2]. The expressions generated with the Peano grammar are quite complex and unlikely to generalize well. Peano's extra expressiveness leads to more spurious candidates, increasing the chances of overfitting and making the benchmark harder to solve.

[26] I use the symbols $\mathbb{B}$, $\mathbb{N}$, and $\mathbb{Z}$ to denote the sets of all Boolean values, all natural numbers (positive integers), and all integers respectively.

Increasing expressiveness →

Equalities	Intervals	Octagons	Polyhedra	Polynomials	Peano
×	0.32 s	2.49 s	2.48 s	55.3 s	68.0 s
FAIL	(19 rounds)	(57 rounds)	(57 rounds)	(76 rounds)	(88 rounds)

(a) Synthesis time and number of CEGIS iterations (rounds) with various grammars

16: $(x \geq n) \vee (x + 1 < n) \vee (m \geq x \wedge m = y)$

28: $(x = y) \vee (y + m - n = x) \vee (x + 2 < n)$

57: $(m = y) \vee (x \geq m \wedge x \geq y)$

(b) Sample predicates with Polyhedra

16: $(x \geq n) \vee (x + 1 < n) \vee (2y = n) \vee (y\,(m - 1) = m)$

28: $(y = 1) \vee (y = 0) \vee (m < 1) \vee (x^2 y > 1)$

57: $(x + 1 \geq n) \vee (x + 2 < n) \vee ((m - n)(x - y) = 1)$

(c) Sample predicates with Peano

Solution in both grammars: $(n \geq y) \wedge (y \geq x) \wedge ((m = y) \vee (x \geq m \wedge x \geq y))$

Table 4.2: Performance of LoopInvGen [PSM16] on the `fib_19` benchmark (Figure 4.3). In **(b)** and **(c)**, we show predicates generated at various rounds (numbered in **bold**).

4.2 SyGuS Overfitting in Theory

In this section, first I formalize the *counterexample-guided inductive synthesis* (CEGIS) approach [Sol13] to SyGuS, in which examples are iteratively provided by a verification oracle. I then state and prove *no-free-lunch* theorems, which show that there can be no optimal learner for this learning scheme. Finally, I formalize a natural notion of *overfitting* for SyGuS and prove that the potential for overfitting increases with grammar expressiveness.

4.2.1 Preliminaries

I borrow the formal definition of a SyGuS problem from prior work [ABJ13]:

Definition 4.1 (SyGuS Problem)**.** Given a background theory $\mathbb{T}$, a function symbol $f : X \to Y$, and two constraints on f: **(a)** a semantic constraint, also called a *specification*, $\varphi(f, x)$ over the vocabulary of $\mathbb{T}$ along with f and a symbolic input x, and **(b)** a syntactic constraint, also called a *grammar*, given by a (possibly infinite) set $\mathcal{E}$ of expressions over the vocabulary of the theory $\mathbb{T}$; find an expression $e \in \mathcal{E}$ such that the formula $\forall x \in X.\, \varphi(e, x)$ is valid modulo $\mathbb{T}$.

I denote this SyGuS problem as $\langle f_{X \to Y} \,|\, \varphi, \mathcal{E} \rangle_{\mathbb{T}}$ and say that it is *satisfiable* iff there exists such an expression e, *i.e.* $\exists\, e \in \mathcal{E}.\, \forall x \in X.\, \varphi(e, x)$. I call e a *satisfying expression* for this problem, denoted as $e \models \langle f_{X \to Y} \,|\, \varphi, \mathcal{E} \rangle_{\mathbb{T}}$.

Recall, I focus on a common class of SyGuS learners, namely those that learn from examples. First I define the notion of input-output (IO) examples that are consistent with a SyGuS specification:

Definition 4.2 (Input-Output Example)**.** Given a specification φ defined on $f : X \to Y$ over a background theory $\mathbb{T}$, I call a pair $\langle x, y \rangle \in X \times Y$ an input-output (IO) example for φ, denoted as $\langle x, y \rangle \mathrel{\approx\!\!\!\models}_{\mathbb{T}} \varphi$ iff it is satisfied by some valid interpretation of f within $\mathbb{T}$, *i.e.*

$$\langle x, y \rangle \mathrel{\approx\!\!\!\models}_{\mathbb{T}} \varphi \;\stackrel{\text{def}}{=}\; \exists\, e_* \in \mathbb{T}.\, e_*(x) = y \;\wedge\; \big(\forall x \in X.\, \varphi(e_*, x)\big)$$

The next two definitions respectively formalize the two key components of a CEGIS-based SyGuS tool: the verification oracle and the learner.

Definition 4.3 (Verification Oracle)**.** Given a specification φ defined on a function $f : X \to Y$ over theory $\mathbb{T}$, a verification oracle $\mathcal{O}_\varphi$ is a partial function that given an expression e, either returns $\perp$ indicating $\forall x \in X.\, \varphi(e, x)$ holds, or gives a counterexample $\langle x, y \rangle$ against e, denoted as $e \rightsquigarrow\!\!\!\!\times_{\varphi} \langle x, y \rangle$, such that

$$e \rightsquigarrow\!\!\!\!\times_{\varphi} \langle x, y \rangle \;\stackrel{\text{def}}{=}\; \neg\, \varphi(e, x) \;\wedge\; e(x) \neq y \;\wedge\; \langle x, y \rangle \mathrel{\approx\!\!\!\models}_{\mathbb{T}} \varphi$$

I omit φ from the notations $\mathcal{O}_\varphi$ and $\rightsquigarrow\!\!\!\!\times_\varphi$ when it is clear from the context.

Definition 4.4 (CEGIS-based Learner)**.** A CEGIS-based learner $\mathcal{L}^{\mathcal{O}}(q, \mathcal{E})$ is a partial function that given an integer $q \geq 0$, a set $\mathcal{E}$ of expressions, and access to an oracle $\mathcal{O}$ for a specification φ defined on $f : X \to Y$, queries $\mathcal{O}$ at most q times and either fails with $\perp$ or generates an expression $e \in \mathcal{E}$. The *trace*

$$\left[e_0 \rightsquigarrow\!\!\!\!\times \langle x_0, y_0\rangle,\ \ldots,\ e_{p-1} \rightsquigarrow\!\!\!\!\times \langle x_{p-1}, y_{p-1}\rangle,\ e_p\right] \qquad \text{where } 0 \leq p \leq q$$

summarizes the interaction between the oracle and the learner. Each e_i denotes the i^{th} candidate for f and $\langle x_i, y_i\rangle$ is a counterexample e_i, *i.e.*

$$\big(\forall j < i.\, e_i(x_j) = y_j \wedge \varphi(e_i, x_j)\big) \ \wedge\ \big(e_i \rightsquigarrow\!\!\!\!\times_\varphi \langle x_i, y_i\rangle\big)$$

Note that I have defined oracles and learners as (partial) functions, and hence as *deterministic*. In practice, many SyGuS tools are deterministic and this assumption simplifies the subsequent theorems. However, I expect that these theorems can be appropriately generalized to randomized oracles and learners.

4.2.2 Learnability and No Free Lunch

In the machine learning (ML) community, the limits of learning have been formalized for various settings as *no-free-lunch* theorems [SB14, §5.1]. Here, I provide a natural form of such theorems for CEGIS-based SyGuS learning.

In SyGuS, the learned function must conform to the given grammar, which may not be fully expressive. Therefore I first formalize grammar expressiveness:

Definition 4.5 (***k***-Expressiveness)**.** Given a domain X and range Y, a grammar $\mathcal{E}$ is said to be k-expressive iff $\mathcal{E}$ can express exactly k distinct $X \to Y$ functions.

A key difference from the ML setting is my notion of *m-learnability*, which formalizes the number of examples that a learner requires in order to learn a desired function. In the ML setting, a function is considered to m-learnable by a learner if it can be learned using an *arbitrary* set of m i.i.d. examples (drawn from some distribution). This makes sense in the ML setting since the learned function is allowed to make errors (up to some given bound on the error probability), but it is much too strong for the *all-or-nothing* SyGuS setting.

Instead, I define a much weaker notion of m-learnability for CEGIS-based SyGuS, which only requires that there *exist* a set of m examples that allows the learner to succeed. The following definition formalizes this notion.

Definition 4.6 (CEGIS-based $\boldsymbol{m}$-Learnability)**.** Given a SyGuS problem $\mathsf{S} = \langle f_{X \to Y} \mid \varphi, \mathcal{E} \rangle_{\mathbb{T}}$ and an integer $m \geq 0$, I say that S is m-learnable by a CEGIS-based learner $\mathcal{L}$ iff there exists a verification oracle $\mathcal{O}$ under which $\mathcal{L}$ can learn a satisfying expression for S with at most m queries to $\mathcal{O}$, *i.e.*$\exists\, \mathcal{O} : \mathcal{L}^{\mathcal{O}}(m, \mathcal{E}) \models \mathsf{S}$.

Finally I state and prove the no-free-lunch (NFL) theorems, which make explicit the tradeoff between grammar expressiveness and learnability. Intuitively, given an integer m and an expressive enough (as a function of m) grammar, for every learner there exists a SyGuS problem that cannot be solved without access to at least $m+1$ examples. This is true despite my weak notion of learnability.

Put another way, as grammar expressiveness increases, so does the number of examples required for learning. On one extreme, if the given grammar is 1-expressive, *i.e.* can express exactly one function, then all satisfiable SyGuS problems are 0-learnable — no examples are needed because there is only one function to learn — but there are *many* SyGuS problems that cannot be satisfied by this function. On the other extreme, if the grammar is $|Y|^{|X|}$-expressive, *i.e.* can express all functions from X to Y, then for every learner there exists a SyGuS problem that requires *all* $|X|$ *examples* in order to be solved.

Below I first present the NFL theorem for the case when the domain X and range Y are finite. I then generalize to the case when these sets may be countably infinite.

Theorem 4.1 (NFL in CEGIS-based SyGuS on Finite Sets)**.** *Let X and Y be two arbitrary finite sets, $\mathbb{T}$ be a theory that supports equality, $\mathcal{E}$ be a grammar over $\mathbb{T}$, and m be an integer such that $0 \leq m < |X|$. Then, either:*

- *$\mathcal{E}$ is not k-expressive for any $k > \sum_{i=0}^{m} \frac{|X|!\ |Y|^i}{(|X|-i)!}$, or*
- *for every CEGIS-based learner $\mathcal{L}$, there exists a satisfiable SyGuS problem $\mathsf{S} = \langle f_{X \to Y} \,|\, \varphi, \mathcal{E} \rangle_{\mathbb{T}}$ such that S is not m-learnable by $\mathcal{L}$. Moreover, there exists a different CEGIS-based learner for which S is m-learnable.*

Proof. First, note that there are $t = \sum_{i=0}^{m} \frac{|X|!\ |Y|^i}{(|X|-i)!}$ distinct traces (sequences of counterexamples) of length at most m over X and Y. Now, consider some CEGIS-based learner $\mathcal{L}$, and suppose $\mathcal{E}$ is k-expressive for some $k > t$. Then, since the learner can deterministically choose at most t candidates for the t traces, there must be at least one function f that is expressible in $\mathcal{E}$, but does not appear in the trace of $\mathcal{L}^{\mathcal{O}}(m, \mathcal{E})$ for any oracle $\mathcal{O}$.

Let e be an expression in $\mathcal{E}$ that implements the function f. Then, we can define the specification $\varphi(f, x) \stackrel{\text{def}}{=} f(x) = e(x)$ and the SyGuS problem $\mathsf{S} = \langle f_{X \to Y} \,|\, \varphi, \mathcal{E} \rangle_{\mathbb{T}}$. By construction, S is satisfiable since $e \models \mathsf{S}$, but we have that $\mathcal{L}^{\mathcal{O}}(m, \mathcal{E}) \not\models \mathsf{S}$ for all oracles $\mathcal{O}$. So, by Definition 4.6, we have that S is not m-learnable by $\mathcal{L}$.

However, we can construct a learner $\mathcal{L}'$ such that S is m-learnable by $\mathcal{L}'$. We construct $\mathcal{L}'$ such that $\mathcal{L}'$ always produces e as its first candidate expression for any trace. The result then follows by Definition 4.6. □

Theorem 4.2 (NFL in CEGIS-based SyGuS on Countably Infinite Sets)**.** *Let X be an arbitrary countably infinite set, Y be an arbitrary finite or countably infinite set, $\mathbb{T}$ be a theory that supports equality, $\mathcal{E}$ be a grammar over $\mathbb{T}$, and m be an integer such that $m \geq 0$. Then, either:*

- *$\mathcal{E}$ is not k-expressive for any $k > \aleph_0$, where $\aleph_0 \stackrel{\text{def}}{=} |\mathbb{N}|$, or*
- *for every CEGIS-based learner $\mathcal{L}$, there exists a satisfiable SyGuS problem $\mathsf{S} = \langle f_{X \to Y} \,|\, \varphi, \mathcal{E} \rangle_{\mathbb{T}}$ such that S is not m-learnable by $\mathcal{L}$. Moreover, there exists a different CEGIS-based learner for which S is m-learnable.*

Proof. Consider some CEGIS-based learner $\mathcal{L}$, and suppose $\mathcal{E}$ is k-expressive for some $k > \aleph_0$. Note that there are $\sum_{i=0}^{m} \frac{|X|!\ |Y|^i}{(|X|-i)!}$ distinct traces of length at most m over X and Y. Let us overapproximate each $\frac{|X|!\ |Y|^i}{(|X|-i)!}$ as $(|X|\,|Y|)^m$, and thus the number of distinct traces as $(m+1)\,(|X|\,|Y|)^m$. We have two cases for Y:

1. Y is finite *i.e.* $|X| = \aleph_0$ and $|Y| < \aleph_0$. Then, the number of distinct traces is at most $(m+1)\,(|X|\,|Y|)^m = (\aleph_0\,|Y|)^m = \aleph_0$. Or,
2. Y is countably infinite *i.e.* $|X| = |Y| = \aleph_0$. Then, the number of distinct traces is at most $(m+1)\,(|X|\,|Y|)^m = (\aleph_0\,\aleph_0)^m = \aleph_0$.

Thus, the number of distinct traces is at most $\aleph_0$, *i.e.* countably infinite. Since the number of distinct functions $k > \aleph_0$, the claim follows using a construction similar to the proof of Theorem 4.1. □

4.2.3 Overfitting

Last, I relate the above theory to the notion of *overfitting* from ML. In the context of SyGuS, overfitting can potentially occur whenever there are multiple candidate expressions that are consistent with a given set of examples. Some of these expressions may not generalize to satisfy the specification, but the learner has no way to distinguish among them (using just the given set of examples) and so can "guess" incorrectly. I formalize this idea through the following measure:

Definition 4.7 (Potential for Overfitting)**.** Given a problem $\mathsf{S} = \langle f_{X \to Y} \,|\, \varphi, \mathcal{E} \rangle_{\mathbb{T}}$ and a set Z of IO examples for φ, I define the potential for overfitting Ω as the number of expressions in $\mathcal{E}$ that are consistent with Z but do not satisfy S, *i.e.*

$$\Omega(\mathsf{S}, Z) \stackrel{\text{def}}{=} \begin{cases} \left|\left\{e \in \mathcal{E} \;\mid\; e \not\models \mathsf{S} \;\wedge\; \forall \langle x, y \rangle \in Z \colon e(x) = y\right\}\right| & \forall z \in Z \colon z \not\approx_{\mathbb{T}} \varphi \\ \bot \qquad \text{(undefined)} & \text{otherwise} \end{cases}$$

Intuitively, a zero potential for overfitting means that overfitting is not possible on the given problem with respect to the given set of examples, because there is no spurious candidate. A positive potential for overfitting means that overfitting is possible, and higher values imply more spurious candidates and hence more potential for a learner to choose the "wrong" expression.

The following theorems connect my notion of overfitting to the earlier NFL theorems by showing that overfitting is inevitable with an expressive enough grammar.

Theorem 4.3 (Overfitting in SyGuS on Finite Sets)**.** *Let X and Y be two arbitrary finite sets, m be an integer such that $0 \leq m < |X|$, $\mathbb{T}$ be a theory that supports equality, and $\mathcal{E}$ be a k-expressive grammar over $\mathbb{T}$ for some $k > \frac{|X|!\ |Y|^m}{m!\ (|X|-m)!}$. Then, there exists a satisfiable SyGuS problem $\mathsf{S} = \langle f_{X \to Y} \,|\, \varphi, \mathcal{E} \rangle_{\mathbb{T}}$ such that $\Omega(\mathsf{S}, Z) > 0$, for every set Z of m IO examples for φ.*

Proof. First, note that there are $t = \frac{|X|!\ |Y|^m}{m!\ (|X|-m)!}$ distinct ways of constructing a set of m IO examples, over X and Y. Now, suppose $\mathcal{E}$ is k-expressive for some $k > t$. Then, there must be at least one function f that is expressible in $\mathcal{E}$, but every set of m IO examples that f is consistent with is also satisfied by some other expressible function.

Let e be an expression in $\mathcal{E}$ that implements the function f. Then, we can define the specification $\varphi(f, x) \stackrel{\text{def}}{=} f(x) = e(x)$ and the SyGuS problem $\mathsf{S} = \langle f_{X \to Y} \,|\, \varphi, \mathcal{E} \rangle_{\mathbb{T}}$. The claim then immediately follows from Definition 4.7. □

Theorem 4.4 (Overfitting in SyGuS on Countably Infinite Sets)**.** *Let X be an arbitrary countably infinite set, Y be an arbitrary finite or countably infinite set, $\mathbb{T}$ be a theory that supports equality, and $\mathcal{E}$ be a k-expressive grammar over $\mathbb{T}$ for some $k > \aleph_0$. Then, there exists a satisfiable SyGuS problem $\mathsf{S} = \langle f_{X \to Y} \,|\, \varphi, \mathcal{E} \rangle_{\mathbb{T}}$ such that $\Omega(\mathsf{S}, Z) > 0$, for every set Z of m IO examples for φ.*

Proof. Let us overapproximate the number of distinct ways of constructing a set of m IO examples, $\frac{|X|!\ |Y|^m}{m!\ (|X|-m)!}$ as $(|X|\,|Y|)^m$. Using cardinal arithmetic, as shown in the the proof of Theorem 4.2, this number is always at most $\aleph_0$. Then the claim follows using a construction similar to the proof of Theorem 4.3. □

Finally, it is straightforward to show that as the expressiveness of the grammar provided in a SyGuS problem increases, so does its potential for overfitting.

Theorem 4.5 (Overfitting Increases with Expressiveness)**.** *Let X and Y be two arbitrary sets, $\mathbb{T}$ be an arbitrary theory, $\mathcal{E}_1$ and $\mathcal{E}_2$ be grammars over $\mathbb{T}$ such that $\mathcal{E}_1 \subseteq \mathcal{E}_2$, φ be an arbitrary specification over $\mathbb{T}$ and a function symbol $f \colon X \to Y$, and Z be a set of IO examples for φ. Then, I have*

$$\Omega\big(\langle f_{X \to Y} \,|\, \varphi, \mathcal{E}_1 \rangle_{\mathbb{T}}, Z\big) \;\leq\; \Omega\big(\langle f_{X \to Y} \,|\, \varphi, \mathcal{E}_2 \rangle_{\mathbb{T}}, Z\big)$$

Proof. If $\mathcal{E}_1 \subseteq \mathcal{E}_2$, then for any set $Z \subseteq X \times Y$ of IO examples, we have

$$\{e \in \mathcal{E}_1 \mid \forall \langle x, y \rangle \in Z \colon e(x) = y\} \;\subseteq\; \{e \in \mathcal{E}_2 \mid \forall \langle x, y \rangle \in Z \colon e(x) = y\}$$

The claim immediately follows from this observation and Definition 4.7. □

4.3 Mitigating Overfitting

Ensemble methods [Die00] in machine learning (ML) are a standard approach to reduce overfitting. These methods aggregate predictions from several learners to make a more

function PLEARN($\mathcal{T}$: Synthesis Tool, $\langle f_{X \to Y} \mid \varphi, \mathcal{E} \rangle_{\mathbb{T}}$: Problem, $\mathcal{E}_{1 \ldots p}$: Subgrammars)

▶ Requires: $\forall\, \mathcal{E}_i \in \mathcal{E}_{1 \ldots p}\colon\ \mathcal{E}_i \subseteq \mathcal{E}$

1 **parallel for** $i \leftarrow 1, \ldots, p$ **do**

2 $\quad \mathsf{S}_i \leftarrow \langle f_{X \to Y} \mid \varphi, \mathcal{E}_i \rangle_{\mathbb{T}}$

3 $\quad e_i \leftarrow \mathcal{T}(\mathsf{S}_i)$

4 $\quad$ **if** $e_i \neq \bot$ **then return** e_i

5 **return** $\bot$

Figure 4.4: The PLEARN framework for SyGuS tools.

accurate prediction. In this section I propose two approaches, inspired by ensemble methods in ML, for mitigating overfitting in SyGuS. Both are based on the key insight from Section 4.2.3 that synthesis over a subgrammar has a smaller potential for overfitting as compared to that over the original grammar.

4.3.1 Parallel SyGuS on Multiple Grammars

My first idea is to run multiple parallel instances of a synthesizer on the same SyGuS problem but with grammars of varying expressiveness. This framework, called PLEARN, is outlined in Figure 4.4. It accepts a synthesis tool $\mathcal{T}$, a SyGuS problem $\langle f_{X \to Y} \mid \varphi, \mathcal{E} \rangle_{\mathbb{T}}$, and subgrammars $\mathcal{E}_{1 \ldots p}$,[27] such that $\mathcal{E}_i \subseteq \mathcal{E}$. The **parallel for** construct creates a new thread for each iteration. The loop in PLEARN creates p copies of the SyGuS problem, each with a different grammar from $\mathcal{E}_{1 \ldots p}$, and dispatches each copy to a new instance of the tool $\mathcal{T}$. PLEARN returns the first solution found or $\bot$ if none of the synthesizer instances succeed.

Since each grammar in $\mathcal{E}_{1 \ldots p}$ is subsumed by the original grammar $\mathcal{E}$, any expression found by PLEARN is a solution to the original SyGuS problem. Moreover, from Theorem 4.5 it is immediate that PLEARN indeed reduces overfitting.

[27] I use the shorthand $\mathrm{X}_{1,\ldots,n}$ to denote the sequence $\langle \mathrm{X}_1, \ldots, \mathrm{X}_n \rangle$.

Theorem 4.6 (PLEARN Reduces Overfitting)**.** *Given a SyGuS problem* $\mathsf{S} = \langle f_{X \to Y} \,|\, \varphi, \mathcal{E} \rangle_{\mathbb{T}}$*, if* PLEARN *is instantiated with* S *and subgrammars* $\mathcal{E}_{1 \ldots p}$ *such that* $\forall\, \mathcal{E}_i \in \mathcal{E}_{1 \ldots p} \colon \mathcal{E}_i \subseteq \mathcal{E}$*, then for each* $\mathsf{S}_i = \langle f_{X \to Y} \,|\, \varphi, \mathcal{E}_i \rangle_{\mathbb{T}}$ *constructed by* PLEARN*, I have that* $\Omega(\mathsf{S}_i, Z) \leq \Omega(\mathsf{S}, Z)$ *on any set* Z *of IO examples for* φ*.*

A key advantage of PLEARN is that it is agnostic to the synthesizer's implementation. Therefore, existing SyGuS learners can immediately benefit from PLEARN, as I demonstrate in Section 4.4.1. However, running p parallel SyGuS instances can be prohibitively expensive, both computationally and memory-wise. The problem is worsened by the fact that many existing SyGuS tools already use multiple threads, *e.g.* the SKETCHAC [JQS15] tool spawns 9 threads. This motivates my *hybrid enumeration* technique described next, which is a novel synthesis algorithm that interleaves exploration of multiple grammars in a single thread.

4.3.2 Hybrid Enumeration

Hybrid enumeration extends the *enumerative synthesis* technique, which enumerates expressions within a given grammar in order of size and returns the first candidate that satisfies the given examples [AGK13]. My goal is to simulate the behavior of PLEARN with an enumerative synthesizer in a single thread. However, a straightforward interleaving of multiple PLEARN threads would be highly inefficient because of redundancies – enumerating the same expression (which is contained in multiple grammars) multiple times. Instead, I propose a technique that **(a)** enumerates each expression at most once, and **(b)** reuses previously enumerated expressions to construct larger expressions.

To achieve this, I extend a widely used [AGK13, PGG14, FMV17] synthesis strategy, called *component-based synthesis* [JGS10], wherein the grammar of expressions is induced by a set of components, each of which is a typed operator with a fixed arity. For example, the grammars shown in Figure 4.1 are induced by integer components (such as `1`, `+`, `mod`, `=`, etc.)

and Boolean components (such as `true`, `and`, `or`, etc.). Below, I first formalize the grammar that is implicit in this synthesis style.

Definition 4.8 (Component-Based Grammar)**.** Given a set $\mathscr{C}$ of typed components, I define the *component-based* grammar $\mathcal{E}$ as the set of all expressions formed by well-typed component application over $\mathscr{C}$, *i.e.*

$$\mathcal{E} \;=\; \{\, c(e_1, \ldots, e_a) \quad | \quad (c : \tau_1 \times \cdots \times \tau_a \to \tau) \in \mathscr{C} \;\wedge\; e_{1\ldots a} \subset \mathcal{E} \\ \wedge \quad e_1 : \tau_1 \;\wedge\; \cdots \;\wedge\; e_a : \tau_a \,\}$$

where $e : \tau$ denotes that the expression e has type τ.

I denote the set of all components appearing in a component-based grammar $\mathcal{E}$ as $\mathsf{components}(\mathcal{E})$. Henceforth, I assume that $\mathsf{components}(\mathcal{E})$ is known (explicitly provided by the user) for each $\mathcal{E}$. I also use $\mathsf{values}(\mathcal{E})$ to denote the subset of nullary components (variables and constants) in $\mathsf{components}(\mathcal{E})$, and $\mathsf{operators}(\mathcal{E})$ to denote the remaining components with positive arities.

The closure property of component-based grammars significantly reduces the overhead of tracking which subexpressions can be combined together to form larger expressions. Given a SyGuS problem over a grammar $\mathcal{E}$, hybrid enumeration requires a sequence $\mathcal{E}_{1\ldots p}$ of grammars such that each $\mathcal{E}_i$ is a component-based grammar and that $\mathcal{E}_1 \subset \cdots \subset \mathcal{E}_p \subseteq \mathcal{E}$. Next, I explain how the subset relationship between the grammars enables efficient enumeration of expressions.

Given grammars $\mathcal{E}_1 \subset \cdots \subset \mathcal{E}_p$, observe that an expression of size k in $\mathcal{E}_i$ may only contain subexpressions of size $\{1, \ldots, (k-1)\}$ belonging to $\mathcal{E}_{1\ldots i}$. This allows us to enumerate expressions in an order such that each subexpression e is synthesized (and cached) before any expressions that have e as a subexpression. I call an enumeration order that ensures this property a *well order*.

Definition 4.9 (Well Order)**.** Given arbitrary grammars $\mathcal{E}_{1...p}$, I say that a strict partial order $\lhd$ on $\mathcal{E}_{1...p} \times \mathbb{N}$ is a well order iff

$$\forall\, \mathcal{E}_a, \mathcal{E}_b \in \mathcal{E}_{1...p} : \ \forall\, k_1, k_2 \in \mathbb{N} : \ [\, \mathcal{E}_a \subseteq \mathcal{E}_b \ \wedge \ k_1 < k_2 \,] \implies (\mathcal{E}_a, k_1) \lhd (\mathcal{E}_b, k_2)$$

Motivated by Theorem 4.5, my implementation of hybrid enumeration uses a particular well order that incrementally increases the expressiveness of the space of expressions. For a rough measure of the expressiveness (Definition 4.5) of a pair $(\mathcal{E}, k)$, *i.e.* the set of expressions of size k in a given component-based grammar $\mathcal{E}$, I simply overapproximate the number of syntactically distinct expressions as $|\mathsf{components}(\mathcal{E})|^k$. To see that this induces a well order, I first prove the following lemma:

Lemma 4.7. *Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be two arbitrary component-based grammars. Then, if $\mathcal{E}_1 \subseteq \mathcal{E}_2$, it must also be the case that* $\mathsf{components}(\mathcal{E}_1) \subseteq \mathsf{components}(\mathcal{E}_2)$*, where* $\mathsf{components}(\mathcal{E}_i)$ *denotes the set of all components appearing in $\mathcal{E}_i$.*

Proof. Let $\mathscr{C}_1 = \mathsf{components}(\mathcal{E}_1)$, $\mathscr{C}_2 = \mathsf{components}(\mathcal{E}_2)$, and $\mathcal{E}_1 \subseteq \mathcal{E}_2$. Suppose $\mathscr{C}_1 \not\subseteq \mathscr{C}_2$. Then, there must be at least one component c such that $c \in \mathscr{C}_1 \setminus \mathscr{C}_2$. By definition of $\mathsf{components}(\mathcal{E}_1)$, the component c must appear in at least one expression $e \in \mathcal{E}_1$. However, since $c \notin \mathscr{C}_2$, it must be the case that $e \notin \mathcal{E}_2$, thus contradicting $\mathcal{E}_1 \subseteq \mathcal{E}_2$. Hence, our assumption $\mathscr{C}_1 \not\subseteq \mathscr{C}_2$ must be false. □

Theorem 4.8. *Let $\mathcal{E}_{1...p}$ be component-based grammars and $\mathscr{C}_i = \mathsf{components}(\mathcal{E}_i)$. Then, the following strict partial order $\lhd_*$ on $\mathcal{E}_{1...p} \times \mathbb{N}$ is a well order*

$$\forall\, \mathcal{E}_a, \mathcal{E}_b \in \mathcal{E}_{1...p} : \ \forall\, m, n \in \mathbb{N} : \ (\mathcal{E}_a, m) \lhd_* (\mathcal{E}_b, n) \iff |\mathscr{C}_a|^m < |\mathscr{C}_b|^n$$

Proof. Let $\mathcal{E}_a$ and $\mathcal{E}_b$ be two component-based grammars in $\mathcal{E}_{1...p}$. By Lemma 4.7, we have that $\mathcal{E}_a \subseteq \mathcal{E}_b \implies \mathsf{components}(\mathcal{E}_1) \subseteq \mathsf{components}(\mathcal{E}_2)$. The claim then immediately follows from Definition 4.9. □

function HENUM($\langle f_{X\to Y} \mid \varphi, \mathcal{E} \rangle_{\mathbb{T}}$: Problem, $\mathcal{E}_{1\ldots p}$: Grammars, $\lhd$: WO, q: Max. Size)

▶ Requires: component-based grammars $\mathcal{E}_1 \subset \cdots \subset \mathcal{E}_p \subseteq \mathcal{E}$ and v as the input variable

$C \leftarrow \{\}$
for $i \leftarrow 1$ **to** p **do**
  $V \leftarrow$ **if** $i = 1$ **then** values($\mathcal{E}_1$) **else** [values($\mathcal{E}_i$) $\setminus$ values($\mathcal{E}_{i-1}$)]
  for each $(e : \tau) \in V$ **do**
    $C[i, 1][\tau] \leftarrow C[i, 1][\tau] \cup \{e\}$
    if $\forall x \in X : \varphi(\lambda v.\, e, x)$ **then return** $\lambda v.\, e$
$R \leftarrow$ SORT($\lhd,\ \mathcal{E}_{1\ldots p} \times \{2, \ldots, q\}$)
for $i \leftarrow 1$ **to** $|\, R \,|$ **do**
  $(\mathcal{E}_j, k) \leftarrow R[i]$
  for $l \leftarrow 1$ **to** j **do**
    $O \leftarrow$ **if** $l = 1$ **then** operators($\mathcal{E}_1$) **else** [operators($\mathcal{E}_l$) $\setminus$ operators($\mathcal{E}_{l-1}$)]
    for each $(o : \tau_1 \times \cdots \times \tau_a \to \tau) \in O$ **do**
      $L \leftarrow$ DIVIDE($a,\ k-1,\ l,\ j,\ \langle\rangle$)
      for each $\langle (x_1, y_1), \ldots, (x_a, y_a) \rangle \in L$ **do**
        for each $e_{1\ldots a} \in C[x_1, y_1][\tau_1] \times \cdots \times C[x_a, y_a][\tau_a]$ **do**
          $e \leftarrow o(e_1, \ldots, e_a)$
          $C[j, k][\tau] \leftarrow C[j, k][\tau] \cup \{e\}$
          if $\forall x \in X : \varphi(\lambda v.\, e, x)$ **then return** $\lambda v.\, e$
return $\bot$

Figure 4.5: *Hybrid enumeration* to combat overfitting in SyGuS

function DIVIDE(a: Arity, q: Size, l: Op. Level, j: Expr. Level, α: Accumulated Args.)

▶ Requires: $1 \leq a \leq q \ \wedge \ l \leq j$

1 **if** $a = 1$ **then**

2 **if** $l = j \ \vee \ \exists \langle x, y \rangle \in \alpha \colon x = j$ **then return** $\{(1, q) \diamond \alpha, \ldots, (j, q) \diamond \alpha\}$

3 **return** $\{(j, q) \diamond \alpha\}$

4 $L = \{\}$

5 **for** $u \leftarrow 1$ **to** j **do**

6 **for** $v \leftarrow 1$ **to** $(q - a + 1)$ **do**

7 $L \leftarrow L \ \cup$ DIVIDE$(a - 1, \ q - v, \ l, \ j, \ (u, v) \diamond \alpha)$

8 **return** L

Figure 4.6: An algorithm to divide a given size budget among subexpressions [28]

I now describe the main hybrid enumeration algorithm, which is listed in Figure 4.5. The HENUM function accepts a SyGuS problem $\langle f_{X \to Y} \,|\, \varphi, \mathcal{E} \rangle_{\mathbb{T}}$, a set $\mathcal{E}_{1 \ldots p}$ of component-based grammars such that $\mathcal{E}_1 \subset \cdots \subset \mathcal{E}_p \subseteq \mathcal{E}$, a well order $\lhd$, and an upper bound $q \geq 0$ on the size of expressions to enumerate. In lines 4 – 8, I first enumerate all values and cache them as expressions of size one. In general $C[j, k][\tau]$ contains expressions of type τ and size k from $\mathcal{E}_j \setminus \mathcal{E}_{j-1}$. In line 9 I sort (grammar, size) pairs in some total order consistent with $\lhd$. Finally, in lines 10 – 20, I iterate over each pair $(\mathcal{E}_j, k)$ and each operator from $\mathcal{E}_{1 \ldots j}$ and invoke the DIVIDE procedure (Figure 4.6) to carefully choose the operator's argument subexpressions ensuring **(a)** *correctness* — their sizes sum up to $k - 1$, **(b)** *efficiency* — expressions are enumerated at most once, and **(c)** *completeness* — all expressions of size k in $\mathcal{E}_j$ are enumerated.

The DIVIDE algorithm generates a set of locations for selecting arguments to an operator. Each location is a pair (x, y) indicating that any expression from $C[x, y][\tau]$ can be an argument, where τ is the argument type required by the operator. DIVIDE accepts an arity a for an

[28] I use $\diamond$ as the `cons` operator for sequences, *e.g.* $x \diamond \langle y, z \rangle = \langle x, y, z \rangle$.

operator o, a size budget q, the index l of the least-expressive grammar containing o, the index j of the least-expressive grammar that should contain the constructed expressions of the form $o(e_1, \ldots, e_a)$, and an accumulator α that stores the list of argument locations. In lines 7 – 9, the size budget is recursively divided among $a-1$ locations. In each recursive step, the upper bound $(q-a+1)$ on v ensures that I have a size budget of at least $q-(q-a+1) = a-1$ for the remaining $a-1$ locations. This results in a call tree such that the accumulator α at each leaf node contains the locations from which to select the last $a-1$ arguments, and we are left with some size budget $q \geq 1$ for the first argument e_1. Finally in lines 4 – 5, I carefully select the locations for e_1 to ensure that $o(e_1, \ldots, e_a)$ has not been synthesized before — either $o \in \mathsf{components}(\mathcal{E}_j)$ or at least one argument belongs to $\mathcal{E}_j \setminus \mathcal{E}_{j-1}$.

I conclude this section by stating some desirable properties satisfied by HEnum. I first define the following notion of $\boldsymbol{j^k}$-Uniqueness and prove a few helper lemmas, then finally present and prove some properties of HEnum.

Definition 4.10 ($\boldsymbol{j^k}$-Uniqueness)**.** Given grammars $\mathcal{E}_1 \subseteq \cdots \subseteq \mathcal{E}_p$, we say that an expression e of size k is j^k-unique with respect to $\mathcal{E}_{1\ldots p}$ if it is contained in $\mathcal{E}_j$ but not in $\mathcal{E}_{(j-1)}$. We define $\mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j$ as the maximal such set of expressions, *i.e.*

$$\mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j \stackrel{\text{def}}{=} \left\{ e \in \mathcal{E}_j \;\mid\; \mathsf{size}(e) = k \;\wedge\; e \notin \mathcal{E}_{(j-1)} \right\}$$

Lemma 4.9. *Given grammars $\mathcal{E}_1 \subseteq \cdots \subseteq \mathcal{E}_p$, for any distinct pairs (j,k) and (j',k') the sets $\mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j$ and $\mathcal{U}[\mathcal{E}_{1\ldots p}]^{k'}_{j'}$ must be disjoint,* i.e.

$$\forall\; j,k,j',k' \colon\; j \neq j' \vee k \neq k' \implies \mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j \cap \mathcal{U}[\mathcal{E}_{1\ldots p}]^{k'}_{j'} = \{\}$$

Proof. When $k \neq k'$, it is straightforward to show that $\mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j \cap \mathcal{U}[\mathcal{E}_{1\ldots p}]^{k'}_{j'} = \{\}$, since an expression cannot be of size k and k' at the same time.

We now prove the claim for the case when $j \neq j'$ by contradiction. Suppose there exists an expression $e \in \mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j \cap \mathcal{U}[\mathcal{E}_{1\ldots p}]^k_{j'}$. Without loss of generality, assume $j > j'$, and therefore $\mathcal{E}_j \supseteq \mathcal{E}_{j'}$. But then, by Definition 4.10, it must be the case that $e \notin \mathcal{E}_{j'}$ and thus $e \notin \mathcal{U}[\mathcal{E}_{1\ldots p}]^k_{j'}$. Therefore, our assumption that $\mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j \cap \mathcal{U}[\mathcal{E}_{1\ldots p}]^k_{j'} \neq \{\}$ must be false. □

Lemma 4.10. *Let $\mathcal{E}_1 \subseteq \cdots \subseteq \mathcal{E}_p$ be p component-based grammars. Then, for any expression $o(e_1, \ldots, e_a) \in \mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j$, if the operator o belongs to $\mathsf{operators}(\mathcal{E}_q)$ such that $q < j$, at least one argument must belong to $\mathcal{E}_j$ but not $\mathcal{E}_{(j-1)}$,* i.e.

$$o \in \mathsf{operators}(\mathcal{E}_q) \wedge q < j \implies \exists\, e \in e_{1\ldots a} \colon e \in \mathcal{E}_j \wedge e \notin \mathcal{E}_{(j-1)}$$

Proof. Consider an arbitrary expression $e_* = o(e_1, \ldots, e_a) \in \mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j$ such that $o \in \mathsf{operators}(\mathcal{E}_q) \wedge q < j$. Suppose $\left[\forall\, e \in e_{1\ldots a} \colon e \notin \mathcal{E}_j \vee e \in \mathcal{E}_{(j-1)}\right]$. Then, for any argument subexpression e, we have the following three possibilities:

- ✗ $e \notin \mathcal{E}_j \wedge e \in \mathcal{E}_{(j-1)}$ is impossible since $\mathcal{E}_{(j-1)} \subseteq \mathcal{E}_j$.
- ✗ $e \notin \mathcal{E}_j \wedge e \notin \mathcal{E}_{(j-1)}$ is also impossible, by Definition 4.8, due to the closure property of component-based grammars.
- ✗ $e \in \mathcal{E}_j \wedge e \in \mathcal{E}_{(j-1)}$ must be false for at least one argument subexpression. Otherwise, since $o \in \mathsf{operators}(\mathcal{E}_{(j-1)})$ and $\mathcal{E}_{(j-1)}$ is closed under operator application by Definition 4.8, $e_* \in \mathcal{E}_{(j-1)}$ must be true. However, by Definition 4.10, we have that $e_* \in \mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j \implies e_* \notin \mathcal{E}_{(j-1)}$.

Therefore, our assumption $\left[\forall\, e \in e_{1\ldots a} \colon e \notin \mathcal{E}_j \vee e \in \mathcal{E}_{(j-1)}\right]$ must be false. □

Lemma 4.11. *Let $\mathcal{E}_0 = \{\}$ and $\mathcal{E}_1 \subseteq \cdots \subseteq \mathcal{E}_p$ be p component-based grammars. Then, for any $l \geq 1$ and any operator $o \in \mathsf{operators}(\mathcal{E}_l) \setminus \mathsf{operators}(\mathcal{E}_{l-1})$ of arity a,* DIVIDE$(a, k-1, l, j, \langle\rangle)$

generates the following set L of all possible distinct locations for selecting the arguments for o such that $o(e_1, \ldots, e_a) \in \mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j$:

$$L = \Big\{ \big\langle (j_1, k_1), \ldots, (j_a, k_a) \big\rangle \;\Big|\; o(e_1, \ldots, e_a) \in \mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j \\ \wedge \;\; \forall\, 1 \leq i \leq a \colon e_i \in \mathcal{U}[\mathcal{E}_{1\ldots p}]^{k_i}_{j_i} \Big\}$$

Proof. In lines 7 – 9 of Figure 4.6, the top-level DIVIDE$(a, k-1, l, j, \langle\rangle)$ call first recursively creates a call tree of height $a-1$ such that the accumulator α at each leaf node contains the locations for selecting the last $a-1$ arguments from. Since u in line 7 ranges over $\{1, \ldots, j\}$ and v in line 8 ranges over $\{1, \ldots, (k-2)\}$, the call tree must be exhaustive by construction. Concretely, the values of α at the the leaf nodes must capture every possible sequence of $a-1$ locations, $\big\langle (j_1, k_1), \ldots, (j_{(a-1)}, k_{(a-1)}) \big\rangle$, such that $k_1 + \cdots + k_{(a-1)} \leq k-2$.

Finally at the leaf nodes of the call tree, lines 4 – 5 are triggered to select locations for the first argument. The naïve approach of simply assigning the remaining size to each grammar in $\mathcal{E}_{1\ldots j}$ would be exhaustive, but may lead to enumerating other expressions $o(e_1, \ldots, e_a) \notin \mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j$ when $l < j$. Therefore, we check if $l < j$ and no location (x, y) in α satisfies $x = j$, in which case we assign the remaining size to only $\mathcal{E}_j$ in line 5. Lemma 4.10 shows that this check is sufficient to guarantee that we only enumerate expressions in $\mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j$. □

Theorem 4.12 (HENUM is Complete)**.** *Given a SyGuS problem $\mathcal{S} = \langle f_{X \to Y} \,|\, \varphi, \mathcal{E} \rangle_{\mathbb{T}}$, let $\mathcal{E}_{1\ldots p}$ be component-based grammars over theory $\mathbb{T}$ such that $\mathcal{E}_1 \subset \cdots \subset \mathcal{E}_p = \mathcal{E}$, $\lhd$ be a well order on $\mathcal{E}_{1\ldots p} \times \mathbb{N}$, and $q \geq 0$ be an upper bound on size of expressions. Then,* HENUM$(\mathcal{S}, \mathcal{E}_{1\ldots p}, \lhd, q)$ *will eventually find a satisfying expression if there exists one with size $\leq q$.*

Proof. First, we observe that every expression $e \in \mathcal{E}$ must belong to *some* maximal set of j^k-unique expressions with respect to $\mathcal{E}_{1\ldots p}$:

$$\forall\, e \in \mathcal{E} \colon\; \exists\, j \in \{1, \ldots, p\} \colon\; \exists\, k \in \{1, \ldots, q\} \colon\; e \in \mathcal{U}[\mathcal{E}_{1\ldots p}]^k_j$$

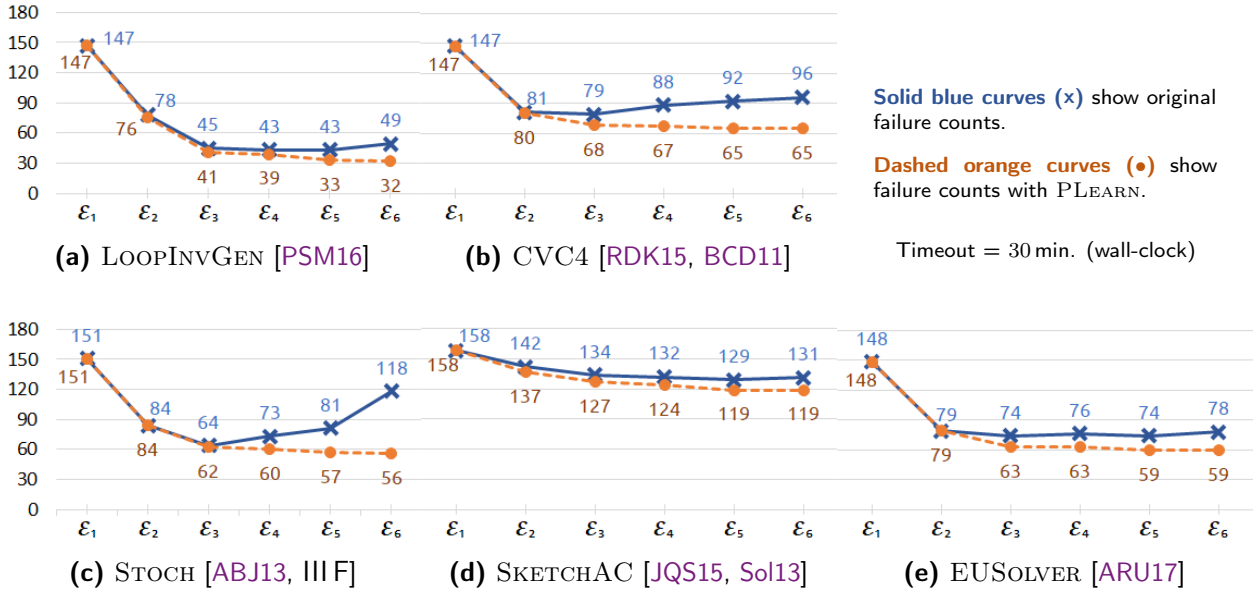
We show that $C[j,k]$ in HENUM (Figure 4.5) stores $\mathcal{U}[\mathcal{E}_{1\ldots p}]_j^k$, into various $C[j,k][\tau]$ lists based on the expression type τ. Since HENUM computes $C[j,k]$ for each $j \in \{1,\ldots,p\}$ and each $k \in \{1,\ldots,q\}$, it must enumerate every expression in $\mathcal{E}$ with size at most q, and thus eventually find e.

The base cases $C[i,1] = \mathcal{U}[\mathcal{E}_{1\ldots p}]_i^1$ are straightforward. The inductive case follows from Lemma 4.11. For each $(j,k) \in \{1,\ldots,p\} \times \{1,\ldots,q\}$ and each operator in $\mathcal{E}_{1\ldots j}$, we invoke DIVIDE (Figure 4.6) to generate all possible locations for the operator's arguments such that the final expression is contained in $\mathcal{U}[\mathcal{E}_{1\ldots p}]_j^k$. Lines 16 – 20 in HENUM then populate $C[j,k]$ as $\mathcal{U}[\mathcal{E}_{1\ldots p}]_j^k$ by applying the operator to subexpressions of appropriate types drawn from these locations. □

Theorem 4.13 (HENUM is Efficient)**.** *Given a SyGuS problem* $\mathcal{S} = \langle f_{X\to Y} \mid \varphi, \mathcal{E}\rangle_{\mathbb{T}}$*, let* $\mathcal{E}_{1\ldots p}$ *be component-based grammars over theory* $\mathbb{T}$ *such that* $\mathcal{E}_1 \subset \cdots \subset \mathcal{E}_p \subseteq \mathcal{E}$*,* $\lhd$ *be a well order on* $\mathcal{E}_{1\ldots p} \times \mathbb{N}$*, and* $q \geq 0$ *be an upper bound on size of expressions. Then,* HENUM$(\mathcal{S}, \mathcal{E}_{1\ldots p}, \lhd, q)$ *will enumerate each distinct expression at most once.*

Proof. As shown in the proof of Theorem 4.12, $C[j,k]$ in HENUM (Figure 4.5) stores $\mathcal{U}[\mathcal{E}_{1\ldots p}]_j^k$. Then, by Lemma 4.9, we immediately have that all pairs $C[j,k]$ and $C[j',k']$ of synthesized expressions are disjoint when $j \neq j'$ or $k \neq k'$.

Furthermore, although each $C[j,k]$ is implemented as a list, we show that any two expressions within any $C[j,k]$ list must be syntactically distinct. The base cases $C[i,1]$ are straightforward. For the inductive case, observe that if each list $C[j_1,k_1],\ldots,C[j_a,k_a]$ only contains syntactically distinct expressions, then all tuples within $C[j_1,k_1]\times\cdots\times C[j_a,k_a]$ must also be distinct. Thus, if an operator o with arity a is applied to subexpressions drawn from the cross product, *i.e.* $\langle e_1,\ldots,e_a\rangle \in C[j_1,k_1] \times \cdots \times C[j_a,k_a]$, then all resulting expressions of the form $o(e_1,\ldots,e_a)$ must be syntactically distinct. Thus, by structural induction, we have that in any list $C[j,k]$ all contained expressions are syntactically distinct. □



(a) LoopInvGen [PSM16] **(b)** CVC4 [RDK15, BCD11]

(c) Stoch [ABJ13, III F] **(d)** SketchAC [JQS15, Sol13] **(e)** EUSolver [ARU17]

Figure 4.7: The number of failures on increasing grammar expressiveness, for state-of-the-art SyGuS tools, with and without the PLearnframework (Figure 4.4)

4.4 Experimental Evaluation

In this section I empirically evaluate PLearn and HEnum. My evaluation uses a set of 180 synthesis benchmarks,[29] consisting of all 127 official benchmarks from the `Inv` track of 2018 SyGuS competition [AF19] augmented with benchmarks from the 2018 Software Verification competition (SV-Comp) [Bey17] and challenging verification problems proposed in prior work [BDM18, BMS05]. All these synthesis tasks are defined over integer and Boolean values, and I evaluate them with the six grammars described in Figure 4.1. I have omitted benchmarks from other tracks of the SyGuS competition as they either require us to construct $\mathcal{E}_{1\ldots p}$ (Section 4.3) by hand or lack verification oracles. All experiments presented in this section were performed on an 8-core Intel® Xeon® E5 machine clocked at 2.30 GHz with 32 GB memory running Ubuntu® 18.04.

[29] All benchmarks are available at `https://github.com/SaswatPadhi/LoopInvGen`.

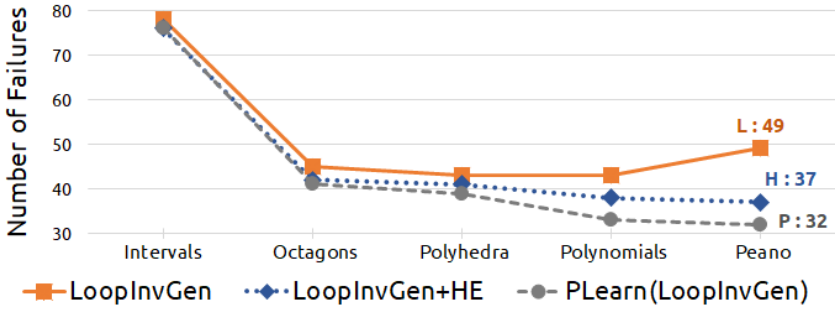
4.4.1 Robustness of PLearn

For five state-of-the-art SyGuS solvers – **(a)** LoopInvGen [PSM16], **(b)** CVC4 [RDK15, BCD11], **(c)** Stoch [ABJ13, III F], **(d)** SketchAC [JQS15, Sol13], and **(e)** EUSolver [ARU17] – I have compared the performance across various grammars, with and without the PLearn framework (Figure 4.4). In this framework, to solve a SyGuS problem with the p^{th} expressiveness level from my six integer-arithmetic grammars (see Figure 4.1), I run p independent parallel instances of a SyGuS tool, each with one of the first p grammars. For example, to solve a SyGuS problem with the Polyhedra grammar, I run four instances of a solver with the Equalities, Intervals, Octagons, and Polyhedra grammars. I evaluate these runs for each tool, for each of the 180 benchmarks and for each of the six expressiveness levels.

Figure 4.7 summarizes my findings. Without PLearn the number of failures initially decreases and then increases across all solvers, as grammar expressiveness increases. However, with PLearn the tools incur fewer failures at a given level of expressiveness, and there is a trend of *decreased* failures with increased expressiveness. Thus, I have demonstrated that PLearn is an effective measure to mitigate overfitting in SyGuS tools and significantly improve their performance.

4.4.2 Performance of Hybrid Enumeration

To evaluate the performance of hybrid enumeration, I augment an existing synthesis engine with HEnum(Figure 4.5). I modify my LoopInvGen tool [PSM16], which is the best-performing SyGuS synthesizer from Figure 4.7. Internally, LoopInvGen leverages Escher [AGK13], an enumerative synthesizer, which I replace with HEnum. I make no other changes to LoopInvGen. I evaluate the performance and resource usage of this solver, HE+LoopInvGen, relative to the original LoopInvGen with and without PLearn(Figure 4.4).



(a) Failures on increasing grammar expressiveness

Grammar	$\mathbf{M}\left[\frac{\tau[\mathbf{P}]}{\tau[\mathbf{H}]}\right]$	$\mathbf{M}\left[\frac{\tau[\mathbf{H}]}{\tau[\mathbf{L}]}\right]$
Equalities	1.00	1.00
Intervals	1.91	1.04
Octagons	2.84	1.03
Polyhedra	3.72	1.01
Polynomials	4.62	1.00
Peano	5.49	0.97

(b) Median(**M**) overhead

Figure 4.8: **L** = LoopInvGen, **H** = HE+LoopInvGen, **P** = PLearn(LoopInvGen). **H** is not only significantly robust against increasing grammar expressiveness, but it also has a smaller total-time cost (τ) than **P** and a negligible overhead over **L**.

Performance. In Figure 4.8(a), I show the number of failures across my six grammars for LoopInvGen, HE+LoopInvGen and LoopInvGen with PLearn, over my 180 benchmarks. HE+LoopInvGen has a significantly lower failure rate than LoopInvGen, and the number of failures decreases with grammar expressiveness. Thus, hybrid enumeration is a good proxy for PLearn.

Resource Usage. To estimate how computationally expensive each solver is, I compare their *total-time cost* (τ). Since LoopInvGen and HE+LoopInvGen are single-threaded, for them I simply use the wall-clock time for synthesis as the total-time cost. However, for PLearn with p parallel instances of LoopInvGen, I consider the total-time cost as p times the wall-clock time for synthesis.

In Figure 4.8(b), I show the median overhead (ratio of τ) incurred by PLearn over HE+LoopInvGen and HE+LoopInvGen over LoopInvGen, at various expressiveness levels. As I move to grammars of increasing expressiveness, the total-time cost of PLearn increases significantly, while the total-time cost of HE+LoopInvGen essentially matches that of LoopInvGen.

4.4.3 Competition Performance

Finally, I evaluate the performance of HE+LoopInvGen on the benchmarks from the `Inv` track of the 2018 SyGuS competition [AF19], against the official winning solver, which I denote LIG [PSM17] — a version of LoopInvGen [PSM16] that has been extensively tuned for this track. In the competition, there are some invariant-synthesis problems where the postcondition itself is a satisfying expression. LIG starts with the postcondition as the first candidate and is extremely fast on such programs. For a fair comparison, I added this heuristic to HE+LoopInvGen as well. No other change was made to HE+LoopInvGen.

LoopInvGen solves 115 benchmarks in a total of 2191 seconds whereas HE+LoopInvGen solves 117 benchmarks in 429 seconds, for a mean speedup of over 5×. Moreover, no entrants to the competition could solve [AF19] the two additional benchmarks (`gcnr_tacas08` and `fib_20`) that HE+LoopInvGen solves.

4.5 Related Work

The most closely related work investigates overfitting for verification tools [SNA14]. My work differs from theirs in several respects. First, I address the problem of overfitting in CEGIS-based synthesis. Second, I formally define overfitting and prove that all synthesizers must suffer from it, whereas they only observe overfitting empirically. Third, while they use cross-validation to combat overfitting in tuning a specific hyperparameter of a verifier, my approach is to search for solutions at different expressiveness levels.

The general problem of efficiently searching a large space of programs for synthesis has been explored in prior work. Lee *et al.* [LHA18] use a probabilistic model, learned from known solutions to synthesis problems, to enumerate programs in order of their likelihood. Other approaches employ type-based pruning of large search spaces [PKS16, OZ15]. These techniques are orthogonal to, and may be combined with, my approach of exploring grammar subsets.

The presented results are widely applicable to existing SyGuS tools, but some tools fall outside my purview. For instance, in programming-by-example (PBE) systems [GPS17, §7], the specification consists of a set of input-output examples. Since any program that meets the given examples is a valid satisfying expression, my notion of overfitting does not apply to such tools. However in a recent work, Inala *et al.* [IS17] show that incrementally increasing expressiveness can also aid PBE systems. They report that searching within increasingly expressive grammar subsets requires significantly fewer examples to find expressions that generalize better over unseen data. Other instances where the synthesizers can have a free lunch, *i.e.* always generate a solution with a small number of counterexamples, include systems that use grammars with limited expressiveness [JGS10, SGH13a, GT12].

This work falls in the category of formal results about SyGuS. In one such result, Jha *et al.* [JS17] analyze the effects of different kinds of counterexamples and of providing bounded versus unbounded memory to learners. Notably, they do not consider variations in "concept classes" or "program templates," which are precisely the focus of my study. Therefore, my results are complementary: I treat counterexamples and learners as opaque and instead focus on grammars.

CHAPTER 5

Conclusion

I have presented a number of techniques that I have developed to automatically infer likely program invariants using information from concrete program executions, such as examples, test cases, execution traces etc. The key enabling factor was a convergence of data-driven insights from machine learning community and symbolic reasoning from the formal methods community — a common theme across my thesis work. I show that the proposed techniques are not only faster [AF19] than the state-of-the-art techniques, but also result in a less onerous, more expressive workflow. Ultimately, these techniques are meant to aid end users in formulating the specifications and inductive invariants that are required to formally reason about correctness of their programs.

Beyond the techniques presented in this thesis, the proposed data-driven invariant learning strategy has also been shown to be useful for learning invariants in other domains. In a recent work, I have also explored automatically learning representation invariants [MPW20] for verifying data structure implementations. Menendez *et al.* have also utilized this approach for verifying compiler optimizations [MN17]. The FlashProfile [PJP18] technique has also been deployed by Microsoft as part of the widely used PROSE [Cor17e] SDK.

There are still a number challenges that need to be addressed before program verification is likely to see mass adoption. One of the most important ones is developing effective yet intuitive user-interaction models for verification tools. The techniques presented in this thesis work with concrete program states or input-output examples. However, providing concrete examples of internal program states can be challenging for complex programs. Alternative

interaction models, such as proactively requesting users to choose between discriminating examples, would be worthwhile to explore.

REFERENCES

[ABJ13] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK 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), 2013*, pp. 1–8. IEEE, 2013.

[AF19] Rajeev Alur, , Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. "SyGuS-Comp 2018: Results and Analysis." *arXiv preprint arXiv:1904.07146*, 2019.

[AGK13] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. "Recursive program synthesis." In *International Conference on Computer Aided Verification*, pp. 934–950. Springer, 2013.

[AGN15] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. "Profiling relational data: a survey." *The VLDB Journal*, **24**(4):557–581, 2015.

[AMS19] Angello Astorga, P Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. "Learning stateful preconditions modulo a test generator." In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 775–787, 2019.

[Ang87] Dana Angluin. "Learning regular sets from queries and counterexamples." *Information and computation*, **75**(2):87–106, 1987.

[ARU17] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling enumerative program synthesis via divide and conquer." In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 319–336. Springer, 2017.

[AS07] David Arthur and Vassilvitskii Sergei. "K-means++: The Advantages of Careful Seeding." In *18th annual ACM-SIAM symposium on Discrete algorithms (SODA), New Orleans, Louisiana*, pp. 1027–1035, 2007.

[ASF18] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. "Search-based program synthesis." *Communications of the ACM*, **61**(12):84–93, 2018.

[ASX18] Angello Astorga, Siwakorn Srisakaokul, Xusheng Xiao, and Tao Xie. "PreInfer: Automatic inference of preconditions via symbolic analysis." In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 678–689. IEEE, 2018.

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[BCD11] Clark Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "Cvc4." In *Computer aided verification*, pp. 171–177. Springer, 2011.

[BCE20] Sidi Mohamed Beillahi, Gabriela Ciocarlie, Michael Emmi, and Constantin Enea. "Behavioral simulation for smart contracts." In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 470–486, 2020.

[BDD12] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Marco Mauri, Eric Medvet, and Enrico Sorio. "Automatic generation of regular expressions from examples with genetic programming." In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pp. 1477–1478, 2012.

[BDM18] Dimitar Bounov, Anthony DeRossi, Massimiliano Menarini, William G Griswold, and Sorin Lerner. "Inferring loop invariants through gamification." In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pp. 1–13, 2018.

[Bey17] Dirk Beyer. "Software verification with validation of results." In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 331–349. Springer, 2017.

[BFM11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. "Why3: Shepherd your herd of provers." In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pp. 53–64, 2011.

[BH14] Bernhard Beckert and Reiner Hahnle. "Reasoning and verification: State of the art and current trends." *IEEE Intelligent Systems*, **29**(1):20–29, 2014.

[BHC15] Arka A Bhattacharya, Dezhi Hong, David Culler, Jorge Ortiz, Kamin Whitehouse, and Eugene Wu. "Automated metadata construction to support portable building applications." In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*, pp. 3–12, 2015.

[Bis06] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[BLS04] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. "The Spec# programming system: An overview." In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pp. 49–69. Springer, 2004.

[BMS05] Aaron R Bradley, Zohar Manna, and Henny B Sipma. "The polyranking principle." In *International Colloquium on Automata, Languages, and Programming*, pp. 1349–1361. Springer, 2005.

[Bre01] Leo Breiman. "Random forests." *Machine learning*, **45**(1):5–32, 2001.

[BRL19] Haniel Barbosa, Andrew Reynolds, Daniel Larraz, and Cesare Tinelli. "Extending enumerative function synthesis via SMT-driven classification." In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pp. 212–220. IEEE, 2019.

[BSA17] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. "Synthesizing program input grammars." *ACM SIGPLAN Notices*, **52**(6):95–110, 2017.

[BST10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The smt-lib standard: Version 2.0." In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, volume 13, p. 14, 2010.

[CAB86] RL Constable, SF Allen, HM Bromley, WR Cleaveland, JF Cremer, RW Harper, DJ Howe, TB Knoblock, NP Mendler, P Panangaden, et al. *Implementing mathematics*. Prentice-Hall, 1986.

[CBR01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. "Bounded model checking using satisfiability solving." *Formal methods in system design*, **19**(1):7–34, 2001.

[CC77a] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252. ACM, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. "Static determination of dynamic properties of generalized type unions." *ACM SIGOPS Operating Systems Review*, **11**(2):77–94, 1977.

[CCF13] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. "Automatic inference of necessary preconditions." In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pp. 128–148. Springer, 2013.

[CDE08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In *OSDI*, volume 8, pp. 209–224, 2008.

[CDO11] Cristiano Calcagno, Dino Distefano, Peter W O'hearn, and Hongseok Yang. "Compositional shape analysis by means of bi-abduction." *Journal of the ACM (JACM)*, **58**(6):1–66, 2011.

[CE82] Edmund Clarke and E Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic." *Logics of programs*, pp. 52–71, 1982.

[CFS09] Satish Chandra, Stephen J Fink, and Manu Sridharan. "Snugglebug: a powerful approach to weakest preconditions." *ACM Sigplan Notices*, **44**(6):363–374, 2009.

[CGH12] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. "Synopses for massive data: Samples, histograms, wavelets, sketches." *Foundations and Trends in Databases*, **4**(1–3):1–294, 2012.

[CGJ00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement." In *International Conference on Computer Aided Verification*, pp. 154–169. Springer, 2000.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[CH78] Patrick Cousot and Nicolas Halbwachs. "Automatic discovery of linear restraints among variables of a program." In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 84–96. ACM, 1978.

[CH84] Thierry Coquand, Gérard Huet, et al. "The Coq proof assistant.", 1984.

[CH93] Allen Cypher and Daniel Conrad Halbert. *Watch what I do: programming by demonstration*. MIT press, 1993.

[Cor17a] Ataccama Corp. "Ataccama One Platform.", 2017. https://www.ataccama.com/.

[Cor17b] Microsoft Corp. "Azure Machine Learning By-Example Data Transform.", 2017. https://www.youtube.com/watch?v=9KG0Sc2B2KI.

[Cor17c] Microsoft Corp. "Data Transformations "By Example" in the Azure ML Workbench.", 2017. https://blogs.technet.microsoft.com/machinelearning/2017/09/25/by-example-transformations-in-the-azure-machine-learning-workbench/.

[Cor17d] Microsoft Corp. "Microsoft SQL Server Data Tools (SSDT).", 2017.

[Cor17e] Microsoft Corp. "Program Synthesis using Examples SDK.", 2017. https://microsoft.github.io/prose/.

[CSM12] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. "Using program synthesis for social recommendations." In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pp. 1732–1736, 2012.

[CSS03] Michael A Colón, Sriram Sankaranarayanan, and Henny B Sipma. "Linear invariant generation using non-linear constraint solving." In *International Conference on Computer Aided Verification*, pp. 420–432. Springer, 2003.

[CV95] Corinna Cortes and Vladimir Vapnik. "Support-vector networks." *Machine learning*, **20**(3):273–297, 1995.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver." In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

[DD13] Isil Dillig and Thomas Dillig. "Explain: A tool for performing abductive inference." In *International Conference on Computer Aided Verification*, pp. 684–689. Springer, 2013.

[DDD76] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Etats-Unis Informaticien, and Edsger Wybe Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976.

[DDH72] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming*. Academic Press Ltd., 1972.

[DDL13] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. "Inductive invariant generation via abductive inference." In *Acm Sigplan Notices*, volume 48, pp. 443–456. ACM, 2013.

[Die00] Thomas G Dietterich. "Ensemble methods in machine learning." In *International workshop on multiple classifier systems*, pp. 1–15. Springer, 2000.

[Dij75] Edsger W Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs." *Communications of the ACM*, **18**(8):453–457, 1975.

[DKA14] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. "The matter of heartbleed." In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pp. 475–488. ACM, 2014.

[DKW08] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. "A survey of automated techniques for formal software verification." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **27**(7):1165–1178, 2008.

[DMS09] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. "VCC: Contract-based modular verification of concurrent C." In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pp. 429–430. IEEE, 2009.

[DS13] Xin Luna Dong and Divesh Srivastava. "Big data integration." In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pp. 1245–1248. IEEE, 2013.

[EG17] Kevin Ellis and Sumit Gulwani. "Learning to Learn Programs from Examples: Going Beyond Program Structure." In *IJCAI*, pp. 1638–1645, 2017.

[END18] P Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P Madhusudan. "Horn-ICE learning for synthesizing invariants and contracts." *Proceedings of the ACM on Programming Languages*, **2**(OOPSLA):1–25, 2018.

[EPG07] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. "The Daikon system for dynamic detection of likely invariants." *Science of Computer Programming*, **69**(1):35–45, 2007.

[FCD15] John K Feser, Swarat Chaudhuri, and Isil Dillig. "Synthesizing data structure transformations from input-output examples." *ACM SIGPLAN Notices*, **50**(6):229–239, 2015.

[FL10] Manuel Fähndrich and Francesco Logozzo. "Static contract checking with abstract interpretation." In *International Conference on Formal Verification of Object-Oriented Software*, pp. 10–30. Springer, 2010.

[Flo67] Robert W Floyd. "Assigning meanings to programs." *Mathematical aspects of computer science*, **19**(19-32):1, 1967.

[FMV17] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. "Component-based synthesis of table consolidation and transformation tasks from examples." *ACM SIGPLAN Notices*, **52**(6):422–436, 2017.

[FR94] Gilberto Filé and Francesco Ranzato. "Improving abstract interpretations by systematic lifting to the powerset." In *Proceedings of the 1994 International Symposium on Logic programming*, pp. 655–669, 1994.

[FWZ08] Kathleen Fisher, David Walker, and Kenny Q Zhu. "LearnPADS: automatic tool generation from ad hoc data." In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1299–1302. ACM, 2008.

[GDV15] Timon Gehr, Dimitar Dimitrov, and Martin Vechev. "Learning commutativity specifications." In *International Conference on Computer Aided Verification*, pp. 307–323. Springer, 2015.

[GF13] Wael H Gomaa and Aly A Fahmy. "A survey of text similarity approaches." *International Journal of Computer Applications*, **68**(13), 2013.

[GFM14] Juan Pablo Galeotti, Carlo A Furia, Eva May, Gordon Fraser, and Andreas Zeller. "Dynamate: Dynamically inferring loop invariants for automatic full functional verification." In *Haifa Verification Conference*, pp. 48–53. Springer, 2014.

[Gia98] Roberto Giacobazzi. "Abductive analysis of modular logic programs." *Journal of Logic and Computation*, **8**(4):457–483, 1998.

[GIB12] Khalil Ghorbal, Franjo Ivančić, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. "Donut domains: Efficient non-convex domains for abstract interpretation." In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pp. 235–250. Springer, 2012.

[GJ07] Sumit Gulwani and Nebojsa Jojic. "Program verification as probabilistic inference." *ACM SIGPLAN Notices*, **42**(1):277–289, 2007.

[GKP94] Ronald L Graham, Donald E Knuth, and Oren Patashnik. *Concrete Math. A Foundation for Computer Science*. Addison-Wesley, Reading, MA, 1994.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." In *ACM Sigplan Notices*, volume 40, pp. 213–223. ACM, 2005.

[GLM14] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. "ICE: A robust framework for learning invariants." In *International Conference on Computer Aided Verification*, pp. 69–87. Springer, 2014.

[GMR09] Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. "From tests to proofs." In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 262–276. Springer, 2009.

[GNM16] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. "Learning invariants using decision trees and implication counterexamples." In *ACM SIGPLAN Notices*, volume 51, pp. 499–512. ACM, 2016.

[Gor88] Michael JC Gordon. "HOL: A proof generating system for higher-order logic." In *VLSI specification, Verification and Synthesis*, pp. 73–128. Springer, 1988.

[GPS17] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. "Program synthesis." *Foundations and Trends® in Programming Languages*, **4**(1-2):1–119, 2017.

[GR07] Denis Gopan and Thomas Reps. "Guided static analysis." In *International Static Analysis Symposium*, pp. 349–365. Springer, 2007.

[GSV08] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. "Program analysis as constraint solving." *ACM SIGPLAN Notices*, **43**(6):281–292, 2008.

[GT12] Patrice Godefroid and Ankur Taly. "Automated synthesis of symbolic instruction encodings from I/O samples." *ACM SIGPLAN Notices*, **47**(6):441–452, 2012.

[Gul10] Sumit Gulwani. "Dimensions in program synthesis." In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pp. 13–24. ACM, 2010.

[Gul11] Sumit Gulwani. "Automating string processing in spreadsheets using input-output examples." In *ACM SIGPLAN Notices*, volume 46, pp. 317–330. ACM, 2011.

[Han14] Steve Hanneke et al. "Theory of disagreement-based active learning." *Foundations and Trends® in Machine Learning*, **7**(2-3):131–309, 2014.

[HBV01] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. "On clustering validation techniques." *Journal of intelligent information systems*, **17**(2-3):107–145, 2001.

[Hig10] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[HNS95] Peter J Haas, Jeffrey F Naughton, S Seshadri, and Lynne Stokes. "Sampling-based estimation of the number of distinct values of an attribute." In *VLDB*, volume 95, pp. 311–322, 1995.

[Hoa69] Charles Antony Richard Hoare. "An axiomatic basis for computer programming." *Communications of the ACM*, **12**(10):576–580, 1969.

[HS02] Brent Hailpern and Padmanabhan Santhanam. "Software debugging, testing, and verification." *IBM Systems Journal*, **41**(1):4–12, 2002.

[Huc04] Thomas Huckle. "Collection of Software Bugs.", 2004.

[Huu13] Ralf Huuck. "Formal verification, engineering and business value." *arXiv preprint arXiv:1301.0037*, 2013.

[Inc10] Google Inc. "OpenRefine: A free, open source, powerful tool for working with messy data.", 2010. http://openrefine.org/.

[Inc14] Red Hat Inc. "Shellshock vulnerability.", 2014.

[Inc15] Red Hat Inc. "Ghost vulnerability.", 2015.

[Inc16] Red Hat Inc. "Badlock Security flaw in Samba - CVE-2016-2118.", 2016.

[Ioa03] Yannis Ioannidis. "The history of histograms (abridged)." In *Proceedings 2003 VLDB Conference*, pp. 19–30. Elsevier, 2003.

[IS17] Jeevana Priya Inala and Rishabh Singh. "WebRelate: integrating web data with spreadsheets using examples." *Proceedings of the ACM on Programming Languages*, **2**(POPL):1–28, 2017.

[Jac12] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[JGS10] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. "Oracle-guided component-based program synthesis." In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 215–224. ACM, 2010.

[JKW10] Yungbum Jung, Soonho Kong, Bow-Yaw Wang, and Kwangkeun Yi. "Deriving invariants by algorithmic learning, decision procedures, and predicate abstraction." In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pp. 180–196. Springer, 2010.

[JMF99] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. "Data clustering: a review." *ACM computing surveys (CSUR)*, **31**(3):264–323, 1999.

[JQS15] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S Foster. "Adaptive concretization for parallel program synthesis." In *International Conference on Computer Aided Verification*, pp. 377–394. Springer, 2015.

[JS17] Susmit Jha and Sanjit A Seshia. "A theory of formal synthesis via inductive learning." *Acta Informatica*, **54**(7):693–726, 2017.

[JTL12] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. "Establishing browser security guarantees through formal shim verification." In *Proceedings of the 21st USENIX conference on Security symposium*, pp. 8–8. USENIX Association, 2012.

[KEH09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. "seL4: Formal verification of an OS kernel." In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 207–220. ACM, 2009.

[KG15] Dileep Kini and Sumit Gulwani. "Flashnormalize: Programming by examples for text normalization." In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[KJD10] Soonho Kong, Yungbum Jung, Cristina David, Bow-Yaw Wang, and Kwangkeun Yi. "Automatically inferring quantified loop invariants by algorithmic learning from simple templates." In *Asian Symposium on Programming Languages and Systems*, pp. 328–343. Springer, 2010.

[KKK13] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. "Synthesis modulo recursive functions." *Acm Sigplan Notices*, **48**(10):407–426, 2013.

[KLR10] Ming Kawaguchi, Shuvendu K Lahiri, and Henrique Rebelo. "Conditional equivalence." *Microsoft, MSR-TR-2010-119, Tech. Rep*, 2010.

[KM07] Panagiotis Karras and Nikos Mamoulis. "The Haar+ tree: a refined synopsis data structure." In *2007 IEEE 23rd International Conference on Data Engineering*, pp. 436–445. IEEE, 2007.

[KMP10] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. "Complete functional synthesis." *ACM Sigplan Notices*, **45**(6):316–329, 2010.

[KPW15] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. "Learning invariants using decision trees." *arXiv preprint arXiv:1501.04725*, 2015.

[KVV94] Michael J Kearns, Umesh Virkumar Vazirani, and Umesh Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86. IEEE, 2004.

[Lau09] Tessa Lau. "Why programming-by-demonstration systems fail: Lessons learned for usable ai." *AI Magazine*, **30**(4):65–65, 2009.

[LCL17] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. "S3: syntax-and semantic-guided repair synthesis via programming by examples." In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 593–604, 2017.

[Lei10] K Rustan M Leino. "Dafny: An automatic program verifier for functional correctness." In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 348–370. Springer, 2010.

[Ler06] Xavier Leroy. "Formal certification of a compiler back-end or: programming a compiler with a proof assistant." In *ACM SIGPLAN Notices*, volume 41, pp. 42–54. ACM, 2006.

[Lev66] Vladimir I Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals." In *Soviet physics doklady*, volume 10, pp. 707–710, 1966.

[LG14] Vu Le and Sumit Gulwani. "FlashExtract: a framework for data extraction by examples." In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 542–553, 2014.

[LHA18] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. "Accelerating search-based program synthesis using learned probabilistic models." *ACM SIGPLAN Notices*, **53**(4):436–449, 2018.

[Lie01] Henry Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[LKR08] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and HV Jagadish. "Regular expression learning for information extraction." In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pp. 21–30, 2008.

[LMN15] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. "Provably correct peephole optimizations with alive." In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 22–32, 2015.

[Loh14] Steve Lohr. "For big-data scientists,'janitor work'is key hurdle to insights." *New York Times*, **17**, 2014.

[LRT14] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. "A DPLL (T) theory solver for a theory of strings and regular expressions." In *International Conference on Computer Aided Verification*, pp. 646–662. Springer, 2014.

[LSR07] Tal Lev-Ami, Mooly Sagiv, Thomas Reps, and Sumit Gulwani. "Backward analysis for inferring quantified preconditions." *Tr-2007-12-01, Tel Aviv University*, 2007.

[LSX17] Shang-Wei Lin, Jun Sun, Hao Xiao, Yang Liu, David Sanán, and Henri Hansen. "FiB: Squeezing loop invariants by interpolation between forward/backward predicate transformers." In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 793–803. IEEE, 2017.

[LY02] Kalle Lyytinen and Youngjin Yoo. "Ubiquitous computing." *Communications of the ACM*, **45**(12):63–96, 2002.

[Mac67] James MacQueen et al. "Some methods for classification and analysis of multivariate observations." In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pp. 281–297. Oakland, CA, USA, 1967.

[May07] Arkady Maydanchik. *Data quality assessment*. Technics publications, 2007.

[Min06] Antoine Miné. "The octagon abstract domain." *Higher-order and symbolic computation*, **19**(1):31–100, 2006.

[MN17] David Menendez and Santosh Nagarakatte. "Alive-infer: Data-driven precondition inference for peephole optimizations in llvm." In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 49–63, 2017.

[Moy08] Yannick Moy. "Sufficient preconditions for modular assertion checking." In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pp. 188–202. Springer, 2008.

[MPW20] Anders Miltner, Saswat Padhi, David Walker, and Todd Millstein. "Data-driven inference of representation invariants." In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. (To Appear). ACM, 2020.

[MR05] Laurent Mauborgne and Xavier Rival. "Trace partitioning in abstract interpretation based static analyzers." In *European Symposium on Programming*, pp. 5–20. Springer, 2005.

[MRS08] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge university press, 2008.

[MSG09] Magnus O Myreen, Konrad Slind, and Michael JC Gordon. "Extensible proof-producing compilation." In *International Conference on Compiler Construction*, pp. 2–16. Springer, 2009.

[MSG15] 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 Annual ACM Symposium on User Interface Software & Technology*, pp. 291–301. ACM, 2015.

[MW79] Zohar Manna and Richard Waldinger. "Synthesis: dreams → programs." *IEEE Transactions on Software Engineering*, (4):294–328, 1979.

[MW80] Zohar Manna and Richard Waldinger. "A deductive approach to program synthesis." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **2**(1):90–121, 1980.

[NE02a] Jeremy W Nimmer and Michael D Ernst. "Automatic generation of program specifications." *ACM SIGSOFT Software Engineering Notes*, **27**(4):229–239, 2002.

[NE02b] Jeremy W Nimmer and Michael D Ernst. "Invariant inference for static checking: An empirical evaluation." *ACM SIGSOFT Software Engineering Notes*, **27**(6):11–20, 2002.

[Neu86] Peter G. Neumann. "The Risks Digest.", 1986.

[NO79] Greg Nelson and Derek C Oppen. "Simplification by cooperating decision procedures." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **1**(2):245–257, 1979.

[OG92] José Oncina and Pedro García. "Identifying regular languages in polynomial time." *Advances in Structural and Syntactic Pattern Recognition*, **5**(99-108):15–20, 1992.

[OR14] Alessandro Orso and Gregg Rothermel. "Software testing: a research travelogue (2000–2014)." In *Proceedings of the on Future of Software Engineering*, pp. 117–132. ACM, 2014.

[Org14] SyGuS-Comp Organizers. "The SyGuS Competition.", 2014. http://sygus.org/comp/.

[ORS92] Sam Owre, John M Rushby, and Natarajan Shankar. "PVS: A prototype verification system." In *International Conference on Automated Deduction*, pp. 748–752. Springer, 1992.

[OZ15] Peter-Michael Osera and Steve Zdancewic. "Type-and-example-directed program synthesis." *ACM SIGPLAN Notices*, **50**(6):619–630, 2015.

[Pea88] Giuseppe Peano. *Calcolo geometrico secondo l'Ausdehnungslehre di H. Grassmann: preceduto dalla operazioni della logica deduttiva*, volume 3. Fratelli Bocca, 1888.

[PG15] Oleksandr Polozov and Sumit Gulwani. "Flashmeta: A framework for inductive program synthesis." *ACM SIGPLAN Notices*, **50**(10):107–126, 2015.

[PG16] Oleksandr Polozov and Sumit Gulwani. "Program Synthesis in the Industrial World: Inductive, Incremental, Interactive." In *5th Workshop on Synthesis (SYNT)*, 2016.

[PGG14] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. "Test-driven synthesis." *ACM Sigplan Notices*, **49**(6):408–418, 2014.

[PJP18] Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. "FlashProfile: a framework for synthesizing data profiles." *Proceedings of the ACM on Programming Languages*, **2**(OOPSLA):150, 2018.

[PKS16] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. "Program synthesis from polymorphic refinement types." *ACM SIGPLAN Notices*, **51**(6):522–538, 2016.

[PMN19] Saswat Padhi, Todd Millstein, Aditya Nori, and Rahul Sharma. "Overfitting in synthesis: Theory and practice." In *International Conference on Computer Aided Verification*, pp. 315–334. Springer, 2019.

[PSM16] Saswat Padhi, Rahul Sharma, and Todd Millstein. "Data-driven precondition inference with learned features." In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 42–56. ACM, 2016.

[PSM17] Saswat Padhi, Rahul Sharma, and Todd Millstein. "LoopInvGen: A Loop Invariant Generator based on Precondition Inference." *arXiv preprint arXiv:1707.02029*, 2017.

[Qui86] J. Ross Quinlan. "Induction of decision trees." *Machine learning*, **1**(1):81–106, 1986.

[RBV16] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. "Learning programs from noisy data." *ACM SIGPLAN Notices*, **51**(1):761–774, 2016.

[RDK15] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. "Counterexample-guided quantifier instantiation for synthesis in SMT." In *International Conference on Computer Aided Verification*, pp. 198–216. Springer, 2015.

[RH01] Vijayshankar Raman and Joseph M Hellerstein. "Potter's wheel: An interactive data cleaning system." In *VLDB*, volume 1, pp. 381–390, 2001.

[SA14] Rahul Sharma and Alex Aiken. "From Invariant Checking to Invariant Inference Using Randomized Search." In *International Conference on Computer Aided Verification*, pp. 88–105. Springer, 2014.

[SB14] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[SCI08] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivančić, and Aarti Gupta. "Dynamic inference of likely data preconditions over predicates by tree learning." In *Proceedings of the 2008 international symposium on Software testing and analysis*, pp. 295–306. ACM, 2008.

[SDC14] Todd W Schiller, Kellen Donohue, Forrest Coward, and Michael D Ernst. "Case studies and tools for contract specifications." In *Proceedings of the 36th International Conference on Software Engineering*, pp. 596–607. ACM, 2014.

[SDR18] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. "Learning loop invariants for program verification." In *Advances in Neural Information Processing Systems*, pp. 7751–7762, 2018.

[SGF10] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. "From program verification to program synthesis." In *ACM Sigplan Notices*, volume 45, pp. 313–326. ACM, 2010.

[SGH13a] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. "A data driven approach for algebraic loop invariants." In *European Symposium on Programming*, pp. 574–592. Springer, 2013.

[SGH13b] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V Nori. "Verification as learning geometric concepts." In *International Static Analysis Symposium*, pp. 388–411. Springer, 2013.

[Sin16] Rishabh Singh. "Blinkfill: Semi-supervised programming by example for syntactic string transformations." *Proceedings of the VLDB Endowment*, **9**(10):816–827, 2016.

[SIS06] Sriram Sankaranarayanan, Franjo Ivančić, Ilya Shlyakhter, and Aarti Gupta. "Static analysis in disjunctive numerical domains." In *International Static Analysis Symposium*, pp. 3–17. Springer, 2006.

[SK13] Mohamed Nassim Seghir and Daniel Kroening. "Counterexample-guided precondition inference." In *European Symposium on Programming*, pp. 451–471. Springer, 2013.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C." In *ACM SIGSOFT Software Engineering Notes*, volume 30, pp. 263–272. ACM, 2005.

[SNA12] Rahul Sharma, Aditya V Nori, and Alex Aiken. "Interpolants as classifiers." In *International Conference on Computer Aided Verification*, pp. 71–87. Springer, 2012.

[SNA14] Rahul Sharma, Aditya V Nori, and Alex Aiken. "Bias-variance tradeoffs in program analysis." *ACM SIGPLAN Notices*, **49**(1):127–137, 2014.

[Sol13] Armando Solar-Lezama. "Program sketching." *International Journal on Software Tools for Technology Transfer*, **15**(5-6):475–495, 2013.

[SS14] Mohamed Nassim Seghir and Peter Schrammel. "Necessary and sufficient preconditions via eager abstraction." In *Asian Symposium on Programming Languages and Systems*, pp. 236–254. Springer, 2014.

[SSC15] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. "Conditionally correct superoptimization." *ACM SIGPLAN Notices*, **50**(10):147–162, 2015.

[SSS48] Thorvald Sørensen, TA Sørensen, TJ Sørensen, T SORENSEN, T Sorensen, TA Sorensen, and T Biering-Sørensen. "A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on Danish commons." 1948.

[STB06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. "Combinatorial sketching for finite programs." *ACM SIGOPS Operating Systems Review*, **40**(5):404–415, 2006.

[Svi98] Borge Svingen. "Learning regular languages using genetic programming." In *Proc. 3-rd Genetic Programming Conference*, pp. 374–376, 1998.

[Tik63] Andrei Nikolaevich Tikhonov. "On the solution of ill-posed problems and the method of regularization." In *Doklady Akademii Nauk*, volume 151, pp. 501–504. Russian Academy of Sciences, 1963.

[Tri17] Trifacta. "Trifacta Wrangler.", 2017. https://www.trifacta.com/.

[Tur37] Alan Mathison Turing. "On computable numbers, with an application to the Entscheidungsproblem." *Proceedings of the London mathematical society*, **2**(1):230–265, 1937.

[Tur49] Alan Mathison Turing. "Checking a large routine." In *Report of a Conference on High Speed Automatic Calculating Machines*, pp. 70–72. University Mathematical Laboratory, Cambridge, 06 1949.

[Wal13] John Walkenbach. *Excel 2013 formulas*. John Wiley & Sons, 2013.

[Wei91] Mark Weiser. "The computer for the 21st century." *Scientific american*, **265**(3):94–104, 1991.

[WFH17] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical Machine Learning Tools and Techniques, 4th Edition*. Morgan Kaufmann Series in Data Management Systems. Elsevier Science & Technology, 2017.

[Win99] William E Winkler. *State of statistical data editing and current research problems*. Bureau of the Census, 1999.

[WLB09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. "Formal methods: Practice and experience." *ACM computing surveys (CSUR)*, **41**(4):19, 2009.

[Woo14a] Daniel Davis Wood. "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER." 2014.

[Woo14b] Gavin Wood. "The Solidity Contract-Oriented Programming Language.", 2014.

[XW05] Rui Xu and Donald Wunsch. "Survey of clustering algorithms." *IEEE Transactions on neural networks*, **16**(3):645–678, 2005.

[ZFW12] Kenny Q Zhu, Kathleen Fisher, and David Walker. "Learnpads++: Incremental inference of ad hoc data formats." In *International Symposium on Practical Aspects of Declarative Languages*, pp. 168–182. Springer, 2012.

[ZMJ18] He Zhu, Stephen Magill, and Suresh Jagannathan. "A data-driven CHC solver." *ACM SIGPLAN Notices*, **53**(4):707–721, 2018.

[ZZG13] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. "Z3-str: A z3-based string solver for web application analysis." In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 114–124, 2013.