The Dissertation Committee for Xinyu Wang
certifies that this is the approved version of the following dissertation:

An Efficient Programming-by-Example Framework

Committee:

Isil Dillig, Supervisor

Greg Durrett

Keshav Pingali

Ranjit Jhala

Mayur Naik

An Efficient Programming-by-Example Framework

by

Xinyu Wang, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2019

Acknowledgments

I will always be in the debt of my advisor, Isil Dillig, for the support and guidance I have received over the past six years. Isil introduced me to the field of programming languages and taught me how to conduct scientific research hand in hand. She taught me how to crystallize a mess of ideas into a simple and precise solution and everything else to become a good researcher. Besides teaching me how to do good research, Isil also showed me how to be a supportive and caring advisor. She always has time for me whenever I need to talk. I truly feel extremely lucky to be her student and have the privilege to work with her during the past few years.

I also want to thank Rishabh Singh for helping me throughout my PhD. Rishabh was my mentor during an internship at Microsoft Research in 2015, where I did my first project on program synthesis with him. I fell in love with Rishabh's passion about research the first time I met him. He is always ready to listen to me and brainstorm ideas with me no matter how crazy they are. Thank you Risahbh for always being so encouraging.

I am also extremely grateful to Sumit Gulwani for his guidance during my PhD career. The first paper on program synthesis that I read thoroughly is his FlashFill paper, and I was extremely privileged to have him as another mentor during my MSR internship. This dissertation is largely influenced by those ideas in FlashFill, and I hope I was able to advance the state-of-the-art on top of that. Thank you Sumit for encouraging me to always focus on doing great work.

I would like to thank Mayur Naik and Ranjit Jhala for being on my dissertation committee and supporting me during my job search. I would also like to thank Keshav Pingali and Greg Durrett for being on my dissertation committee as well. I have recently started collaborating with Greg and it has been quite a refreshing experience. I have learned immensely from him.

I want to thank Yu Feng for teaching me the basics of program analysis and collaborating with me in a couple of projects during our first couple of years at UT. I will never forget the days and nights that we have been working together as well as what Austin looks like at 4 am in the morning. Thank you my dear friend! I wish you all the best and success one can hope for.

I was fortunate to get to collaborate with many great researchers: Greg Anderson, Jocelyn Chen, Isil Dillig, Thomas Dillig, Greg Durrett, Yu Feng, Sumit Gulwani, Calvin Lin, Ken McMillan, Hovav Shacham, Rishabh Singh, Shankara Pailoor, Yuepeng Wang, Navid Yaghmazadeh, and Xi Ye. I learned a lot about how to conduct good research and improve myself as a researcher and presenter while working with them. Thank y'all!

My colleagues in the UToPiA group made this long PhD journey much shorter and much more enjoyable, and I wholeheartedly thank them for that. The UToPiA family started with Yu and me (and also Isil, of course). Then, our hacker Oswaldo and awesome Navid joined. After that, we had Yuepeng, Kostas, Ruben, Valentin, and Jocob join the party. I will always miss the good old days that we play board games at Isil's house (and the pool parties). Now, our UToPiA family has many more members: Jia, Greg, Jiayi, Jocelyn, Jon, Rong, and Shankara. I really enjoy the past several years with all of you, and I will miss everyone of you. Ciao Utopians!

Besides Utopians, I also want to thank my many other friends at UT who made my everyday life full of joy: Bo, Chunzhi, Hangchen, Jian, Jianyu, Wenguang, Ye, Yuanzhong, Zhiting, and many others.

Finally, I would like to thank my parents who support me all the way until I finish my PhD. I dedicate this dissertation to you.

An Efficient Programming-by-Example Framework

Publication No. ____________________

Xinyu Wang, Ph.D.
The University of Texas at Austin, 2019

Supervisor: Isil Dillig

Due to the ubiquity of computing, programming has started to become an essential skill for an increasing number of people, including data scientists, financial analysts, and spreadsheet users. While it is well known that building any complex and reliable software is difficult, writing even simple scripts is challenging for novices with no formal programming background. Therefore, there is an increasing need for technology that can provide basic programming support to non-expert computer end-users.

Program synthesis, as a technique for generating programs from high-level specifications such as input-output examples, has been used to automate many real-world programming tasks in a number of application domains such as spreadsheet programming and data science. However, developing specialized synthesizers for these application domains is notoriously hard.

This dissertation aims to make the development of program synthesizers easier so that we can expand the applicability of program synthesis to more application domains. In particular, this dissertation describes a programming-by-example framework that is both *generic* and *efficient*. This framework can be applied broadly to automating tasks across different application domains. It is also efficient and achieves orders of magnitude improvement in terms of the synthesis speed compared to existing state-of-the-art techniques.

Table of Contents

Acknowledgments **iv**

Abstract **vii**

List of Tables **xi**

List of Figures **xii**

Chapter 1. Introduction **1**

Chapter 2. Program Synthesis using Finite Tree Automata **9**

2.1 Background on Finite Tree Automata 9

2.2 Program Synthesis using Finite Tree Automata 11

2.3 Implementation 16

2.4 Application 17

2.5 Evaluation 26

Chapter 3. Improving Efficiency using Abstract Interpretation **35**

3.1 Program Synthesis using Abstract Finite Tree Automata . . . 36

3.1.1 Abstractions 36

3.1.2 Abstract Finite Tree Automata 39

3.2 Program Synthesis using Abstraction Refinement 43

3.2.1 Algorithm Architecture 44

3.2.2 Constructing Incorrectness Proofs 48

3.3 A Working Example 56

3.4 Implementation 59

3.5 Applications 59

3.5.1 String Processing 60

3.5.2 Tensor Reshaping 62

3.6 Evaluation . . . 64
3.6.1 String Processing . . . 65
3.6.2 Tensor Reshaping . . . 71
3.6.3 Discussion . . . 75

Chapter 4. Learning Abstractions for Program Synthesis 76
4.1 Overview . . . 76
4.2 An Illustrative Example . . . 78
4.3 Overall Abstraction Learning Algorithm . . . 80
4.4 Synthesis of Predicate Templates . . . 82
4.5 Synthesis of Abstract Transformers . . . 87
4.5.1 Example Generation . . . 92
4.6 Evaluation . . . 95
4.6.1 Abstraction Learning . . . 96
4.6.2 Evaluating the Usefulness of Learned Abstractions . . . 98

Chapter 5. Related Work 100

Chapter 6. Conclusion 106

Appendices 107

Appendix A. Proofs of Theorems 108

Bibliography 110

Vita 122

List of Tables

List of Figures

1.1 Workflow of our synthesis framework. 4

2.1 A finite tree automaton example. 10
2.2 CFTA construction rules. 13
2.3 A CFTA example. 15
2.4 An example data completion task. 19
2.5 Data completion DSL syntax. 23
2.6 Data completion DSL semantics. 24
2.7 Data completion benchmark statistics. 27
2.8 Blaze vs. Prose in data completion domain. 29
2.9 Blaze vs. Sketch in data completion domain. 33

3.1 AFTA construction rules. 41
3.2 An AFTA example. 42
3.3 Top-level structure of our synthesis algorithm. 45
3.4 An incorrectness proof example. 47
3.5 Incorrectness proof construction algorithm. 49
3.6 Auxiliary EvalAbstract procedure used in Figure 3.5. . . . 49
3.7 Algorithm for finding a strengthening for the root. 51
3.8 Algorithm for finding a strengthening for nodes other than root. 51
3.9 A proof construction example. 54
3.10 An end-to-end working example. 57
3.11 String processing DSL. 60
3.12 Abstract transformers for string processing DSL. 61
3.13 Tensor reshaping DSL. 63
3.14 Abstract transformers for tensor reshaping DSL. 64
3.15 Blaze results for string processing domain. 68
3.16 Blaze vs. existing techniques for string processing domain. . 70

3.17 Blaze results for tensor reshaping domain. 73
3.18 Blaze vs. existing techniques for tensor reshaping domain. . . 74

4.1 Schematic overview of our abstraction learning technique. . . . 77
4.2 Overall learning algorithm. 81
4.3 A tree interpolation problem and a tree interpolant (underlined). 84
4.4 Algorithm for learning abstract domain using tree interpolation. 85
4.5 Algorithm for synthesizing abstract transformers. 90
4.6 Example generation for learning abstract transformers. 93
4.7 Training results of Atlas. 97
4.8 Improvement of Blaze* over Blaze†. 99

Chapter 1

Introduction

Program synthesis aims to automatically construct a program in a given programming language that satisfies a given specification. In particular, there has been significant interest in *example-guided* program synthesis where the specification is given as *input-output examples.* Such programming-by-example (PBE) techniques have been successfully used to automate many programming tasks that non-expert computer end-users struggle with, such as various data wrangling tasks [21, 56, 58, 15, 70, 8, 33] that arise in the context of data science and spreadsheets. Due to its potential to automate many programming tasks encountered by non-expert users, programming-by-example has now become a burgeoning research area.

Program synthesis is effectively a search problem: it searches for a program in the given programming language that satisfies the given specification. A key challenge in this area is how to deal with the enormous search space. Even if we restrict ourselves to small programs up to a fixed size over a small domain-specific language, the synthesizer may still need to explore a colossal number of programs before it finds one that satisfies the specification.

In programming-by-example, a common search space reduction technique exploits the observation that programs which yield the same output on the same input are indistinguishable with respect to the user-provided specification and therefore are deemed "equivalent" to each other. Based on this observation, many techniques use a canonical representation of a set of programs that share the same input-output behavior. For instance, enumeration-based techniques, such as Escher [2] and Transit [68], discard programs that yield the same output as a previously explored program. Similarly, synthesis algorithms in the Flash* family [21, 50, 33, 8], use a single node to represent all sub-programs that have the same input-output behavior. Thus, in all of these algorithms, the size of the search space is determined by the number of output values produced by the programs on the given inputs. While this search space reduction technique is quite generic, it is not sufficient. In order to synthesize non-trivial programs and be truly useful in practice, existing synthesis techniques also need to employ other *domain-specific* pruning strategies. For example, enumeration-based techniques, such as λ^2 [16] and MORPHEUS [15], use a set of built-in deduction rules to prune the search space. This improves the search efficiency dramatically, however, at the cost of restricting the applicability of these techniques to only those domains under consideration.

This dissertation describes a programming-by-example framework that is both *generic* and *efficient*: it can be applied in different application domains and at the same time achieves efficient synthesis. The key idea underlying this framework is to leverage *abstract interpretation*. Building on top of the insight in prior work [2, 68, 21, 50, 33] that we can reduce the size of the search space by exploiting commonalities in the *concrete* input-output behavior of programs, our approach considers two programs to belong to the same equivalence class if they produce the same *abstract* output on the same input. Specifically, the framework is parametrized with a domain-specific language (DSL) and its abstract semantics as well as a set of input-output examples. Starting from the input example, our algorithm symbolically executes programs in the DSL using their abstract semantics and merges any programs that produce the same abstract output into the same equivalence class. The algorithm then looks for a program whose *abstract behavior* is consistent with the user-provided examples. Because two programs that do not have the same input-output behavior in terms of their concrete semantics may have the same behavior in terms of their abstract semantics, our approach has the potential to reduce the search space in a more dramatic way.

While this *abstraction-based* approach is able to synthesize programs efficiently, one obvious implication is that the synthesized programs may now be *spurious*. That is, a program that is consistent with the provided examples with respect to the abstract semantics may not actually satisfy the examples according to the concrete semantics. Our synthesis algorithm iteratively eliminates such spurious programs by performing a form of *counterexample-guided abstraction refinement*. That is, starting with a (coarse) initial abstraction, we first find a program P that is consistent with the input-output examples with respect to the abstract semantics. If P also satisfies the examples according to the concrete semantics, our algorithm returns P as a solution. Otherwise, we *refine* the current abstraction, with the goal of ensuring that P (and hopefully many other spurious programs) are no longer consistent with the specification using the *new* abstraction. This refinement process continues until we either find a program that indeed satisfies the input-output examples (according to concrete semantics), or prove that no such DSL program exists.

Our framework can be instantiated for an application domain with a suitable domain-specific language and its abstract semantics. Its workflow is shown schematically in Figure 1.1. Note that for a new application domain, a *domain expert* needs to provide a DSL with its abstract semantics. From an *end-user*'s perspective, the only input is a set of input-output examples.

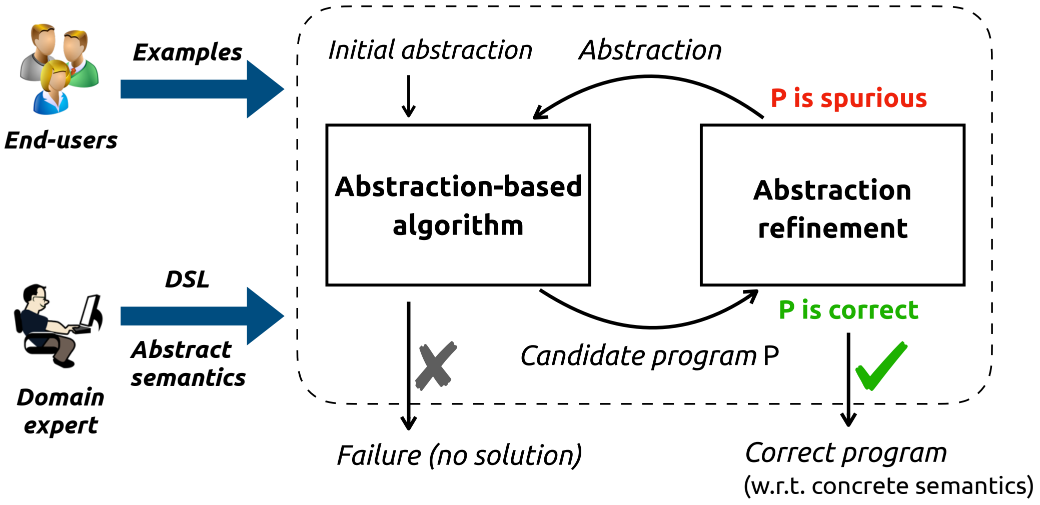


Figure 1.1: Workflow of our synthesis framework.

While this framework can be realized in different ways, our particular development is based on a novel synthesis methodology that uses *finite tree au-*

tomata (FTAs). In the simplest form (with no abstractions), this FTA-based technique takes as input a domain-specific language with *concrete semantics* and a set of input-output examples. It then constructs a finite tree automaton whose language correspond to exactly the set of programs that are consistent with the given examples. Finally, our FTA-based method ranks these programs and returns a "best" program as the result. While this approach can, in principle, be used to synthesize programs over a broad class of DSLs, it suffers from the same scalability issue as other techniques that also use concrete program semantics.

This dissertation further introduces the notion of *abstract finite tree automata* (AFTAs), which can be used to synthesize programs over the DSLs abstract semantics. Taking as input a domain-specific language with *abstract semantics* and a set of input-output examples, our AFTA method constructs a finite tree automaton whose language is exactly the set of programs that satisfy the given examples according to the DSL's *abstract semantics*. Therefore, after ranking, the synthesized program is consistent with the specification in terms of the DSLs abstract semantics. However, this program does not necessarily satisfy the specification according to the DSL's concrete semantics. In order to avoid synthesizing such a *spurious* program, our technique automatically refines the current abstraction by constructing a so-called *incorrectness proof*. Such a proof annotates the nodes of the abstract syntax tree representing a spurious program P with predicates that should be included in the *new* abstraction. Then, using this new abstraction, the AFTA constructed in the next iteration is guaranteed to reject P, alongside many other spurious programs accepted by the AFTA in the previous iteration.

We have implemented our proposed idea in a programming-by-example framework called BLAZE, which can be instantiated in different domains by providing a suitable domain-specific language with its corresponding abstract semantics. We have instantiated BLAZE in three different application domains, namely, data completion in data science, string processing in spreadsheets, and tensor reshaping in MATLAB. In particular, our benchmark suite consists of real-world programming tasks that are collected from the standard SYGUS data set and online help forums such as StackOverflow. For each application domain, we also compare BLAZE with existing state-of-the-art synthesis tools. Our experimental results show that BLAZE can successfully synthesize programs to automate many tasks that arise across different application domains, and it achieves orders of magnitude improvement in terms of the synthesis speed compared to existing techniques.

In summary, this dissertation makes the following contributions:

- We introduce a novel programming-by-example paradigm that consists of two components: an *abstraction-based synthesis* component that synthesizes programs with respect to an abstraction and an *abstraction refinement* component that refines the abstraction whenever it is not precise.

- We describe an abstraction-based synthesis technique that utilizes *finite tree automata* (FTAs). This technique constructs an FTA from the

DSL's semantics and input-output examples, and the FTA's language is guaranteed to be the set of programs that satisfy the given examples.

- We present an abstraction refinement technique that is based on constructing an *incorrectness proof.* We show how to construct such a proof for any spurious program and describe how to use it to refine the abstraction so that the same spurious program will not be synthesized again.

- We develop a generic and efficient programming-by-example framework, called Blaze, that can be instantiated in different application domains by providing a domain-specific language with its abstract semantics.

- We instantiate Blaze in three application domains, namely, data completion, string processing, and tensor reshaping. Our evaluation demonstrates that Blaze can successfully synthesize non-trivial programs and achieves significant improvement over existing techniques in terms of the synthesis speed.

- We propose a technique for learning abstractions that are useful for instantiating Blaze in a new domain. Our evaluation demonstrates that this technique learns abstractions that allow Blaze to achieve significantly better results compared to the manually crafted abstractions.

The rest of this dissertation is organized as follows. Chapter 2 presents a generic synthesis framework that is based on finite tree automata. Chapter 3 describes a technique that improves the efficiency of this synthesis framework by leveraging abstract interpretation. Chapter 4 further proposes a technique that automatically learns abstractions that are useful for instantiating our synthesis framework. Chapter 5 discusses related work and Chapter 6 concludes.

Chapter 2

Program Synthesis using Finite Tree Automata

This chapter presents a program synthesis algorithm that is based on finite tree automata (FTAs). We first give some background on FTAs. Then, we present a generic programming-by-example technique that is based on FTAs. Finally, we describe how to instantiate this technique to automate data completion tasks and present our experimental results.

2.1 Background on Finite Tree Automata

A *finite tree automaton* is a type of state machine that deals with tree-structured data. In particular, finite tree automata generalize standard finite word automata by accepting trees rather than words (strings).

Definition 2.1.1. **(FTA)** A (bottom-up) finite tree automaton (FTA) a tuple $\mathcal{A} = (Q, F, Q_f, \Delta)$ where Q is a set of states, F is an alphabet, $Q_f \subseteq Q$ is a set of final states, and Δ is a set of transitions (rewrite rules) of the form $f(q_1, \cdots, q_n) \rightarrow q$ where we have $q, q_1, \cdots, q_n \in Q$ and $f \in F$.

We assume that every symbol $f \in F$ is associated with an arity (rank), and we use the notation F_k to denote the function symbols of arity k. We view ground terms over alphabet F as trees such that a ground term t is accepted by an FTA if we can rewrite t to a state $q \in Q_f$ using rules in Δ. The language of an FTA $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, corresponds to the set of all ground terms that are accepted by $\mathcal{A}$.

Example 2.1.1. Consider the tree automaton $\mathcal{A}$ defined by states $Q = \{q_0, q_1\}$, $F = F_0 \cup F_1 \cup F_2$, $F_0 = \{0, 1\}$, $F_1 = \{\neg\}$, $F_2 = \{\wedge\}$, final states $Q_f = \{q_0\}$, and the following transitions Δ:

$$\begin{array}{llll} 1 \rightarrow q_1 & 0 \rightarrow q_0 & \wedge(q_0, q_0) \rightarrow q_0 & \wedge(q_0, q_1) \rightarrow q_0 \\ \neg(q_0) \rightarrow q_1 & \neg(q_1) \rightarrow q_0 & \wedge(q_1, q_0) \rightarrow q_0 & \wedge(q_1, q_1) \rightarrow q_1 \end{array}$$

This tree automaton accepts exactly those propositional logic formulas (without variables) that evaluate to *false*. As an example, Figure 2.1 shows the tree for formula $\neg(0 \wedge \neg 1)$ where each sub-term is annotated with its state on the right. This formula is *not* accepted by the tree automaton $\mathcal{A}$ because the rules in Δ "rewrite" the formula to state q_1, which is not a final state.

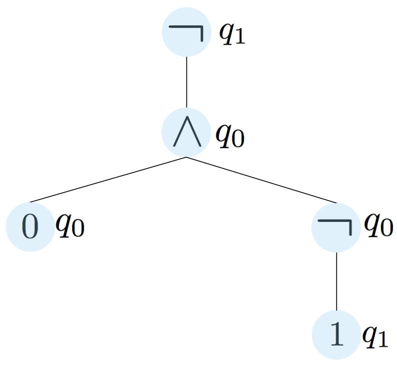


Figure 2.1: A finite tree automaton example.

2.2 Program Synthesis using Finite Tree Automata

In this section, we describe how to apply finite tree automata in the context of programming-by-example. Here, we consider a general setting where given a domain-specific language (DSL) and a set of input-output examples, we synthesize a program in the given DSL that satisfies the given examples.

Given a DSL and a set of examples, our key idea is to construct a finite tree automaton that represents the set of all DSL programs that are consistent with the examples. Specifically, the states of the FTA correspond to *concrete* values, and the transitions are obtained using the DSL's *concrete semantics*. We therefore refer to such tree automata as *concrete FTAs* (CFTAs).

To understand the construction of CFTAs, suppose that we are given a DSL with its syntax (defined by a context-free grammar G) and operational semantics as well as a set of input-output examples $\vec{e}$. We represent the input-output examples $\vec{e}$ as a vector where each element in $\vec{e}$ is of the form $e_{in} \rightarrow e_{out}$. We also write $\vec{e}_{in}$ (resp. $\vec{e}_{out}$) to represent the input (resp. output) examples. Without loss of generality, we assume that programs always take a single input x, as we can always represent multiple inputs as a list. Thus, the synthesized program is always of the form $\lambda x.S$, and S is defined by a context-free grammar $G = (T, N, P, s_0)$ where:

- T is a set of terminal symbols which includes the input variable x. We refer to terminals other than x as *constants*, and we use the notation T_C to denote these constants.

- N is a finite set of non-terminal symbols corresponding to sub-expressions in the DSL.

- P is a set of production rules of the form $s \to f(s_1, \cdots, s_n)$ where f is a built-in DSL function and $s, s_1, \cdots, s_n$ are symbols in the grammar.

- $s_0 \in N$ is the topmost non-terminal symbol (start symbol).

We can construct the CFTA for a DSL and a set of input-output examples using the rules shown in Figure 2.2. First, the alphabet of the CFTA consists of the built-in functions (operators) in the DSL. The states in the CFTA are of the form $q_s^{\vec{c}}$, where s is a symbol (terminal or non-terminal) in the grammar G and $\vec{c}$ is a vector of concrete values. Intuitively, the CFTA has a state $q_s^{\vec{c}}$ if symbol s can take values $\vec{c}$ for input examples $\vec{e}_{in}$. Similarly, the existence of a transition $f(q_{s_1}^{\vec{c_1}}, \cdots, q_{s_n}^{\vec{c_n}}) \to q_s^{\vec{c}}$ means that applying function f on the values $c_{1j}, \cdots, c_{nj}$ yields c_j. Hence, as mentioned earlier, transitions of the CFTA are constructed using the DSL's concrete semantics.

We now explain the rules from Figure 2.2 in more detail. The first rule, labeled Var, states that $q_x^{\vec{c}}$ is a state whenever x is the input variable and $\vec{c}$ is the input examples. The Const rule adds a state $q_t^{[\llbracket t \rrbracket, \cdots, \llbracket t \rrbracket]}$ for each constant t in the grammar. The next rule, called Final, indicates that $q_{s_0}^{\vec{c}}$ is a final state if s_0 is the start symbol and $\vec{c}$ is the output examples. The last rule, labeled Prod, processes each production $s \to f(s_1, \cdots, s_n)$ in the grammar and generates new states and transitions. Essentially, this rule states that, if symbol s_i can take value $\vec{c_i}$ (*i.e.*, there exists a state $q_{s_i}^{\vec{c_i}}$) and executing f on

$$\frac{\vec{c} = \vec{e}_{in}}{q_x^{\vec{c}} \in Q}\ (\textsc{Var}) \qquad \frac{t \in T_C \quad \vec{c} = \big[[\![t]\!], \cdots, [\![t]\!]\big] \quad |\vec{c}| = |\vec{e}|}{q_t^{\vec{c}} \in Q}\ (\textsc{Const}) \qquad \frac{q_{s_0}^{\vec{c}} \in Q \quad \vec{c} = \vec{e}_{out}}{q_{s_0}^{\vec{c}} \in Q_f}\ (\textsc{Final})$$

$$\frac{(s \to f(s_1, \cdots, s_n)) \in P \quad q_{s_1}^{\vec{c_1}} \in Q, \cdots, q_{s_n}^{\vec{c_n}} \in Q \quad c_j = [\![f(c_{1j}, \cdots, c_{nj})]\!] \quad \vec{c} = [c_1, \cdots, c_{|\vec{e}|}]}{q_s^{\vec{c}} \in Q, \quad \big(f(q_{s_1}^{\vec{c_1}}, \cdots, q_{s_n}^{\vec{c_n}}) \to q_s^{\vec{c}}\big) \in \Delta}\ (\textsc{Prod})$$

Figure 2.2: CFTA construction rules.

values $c_{1j}, \cdots, c_{nj}$ yields value c_j, then we also have a state $q_s^{\vec{c}}$ in the CFTA as well as a transition $f(q_{s_1}^{\vec{c}_1} \cdots, q_{s_n}^{\vec{c}_n}) \to q_s^{\vec{c}}$.

In general, the CFTA constructed using the rules from Figure 2.2 may have infinitely many states. As standard in synthesis literature [50, 62], we bound the size of the programs under consideration and search within a finite space [1]. In terms of the CFTA construction, this means that we add a state $q_s^{\vec{c}}$ only if the size of the smallest tree accepted by the automaton $(Q, F, \{q_s^{\vec{c}}\}, \Delta)$ is lower than the threshold. This ensures the number of states in our CFTA is finite and therefore our CFTA construction always terminates.

It can be shown that the language of the CFTA constructed from Figure 2.2 is exactly the set of abstract syntax trees (ASTs) of DSL programs that are consistent with the input-output examples.[2] Hence, once we construct such a CFTA, the synthesis task boils down to finding an AST that is accepted by the automaton. However, since there are typically many accepting ASTs, one

[1]The size of the search space is in general exponential to the bound.

[2]The proof can be found in Theorem 2.2.1 and Theorem 2.2.2 the appendix.

can use heuristics to identify a "best" AST (*i.e.*, program) that satisfies the input-output examples. For instance, one heuristic could be based on *Occam's razor* that always favors a program with the smallest size.

Example 2.2.1. To see how to construct CFTAs, let us consider a very simple toy DSL, whose syntax is given by the following context-free grammar that only contains two constants and allows addition and multiplication by constants:

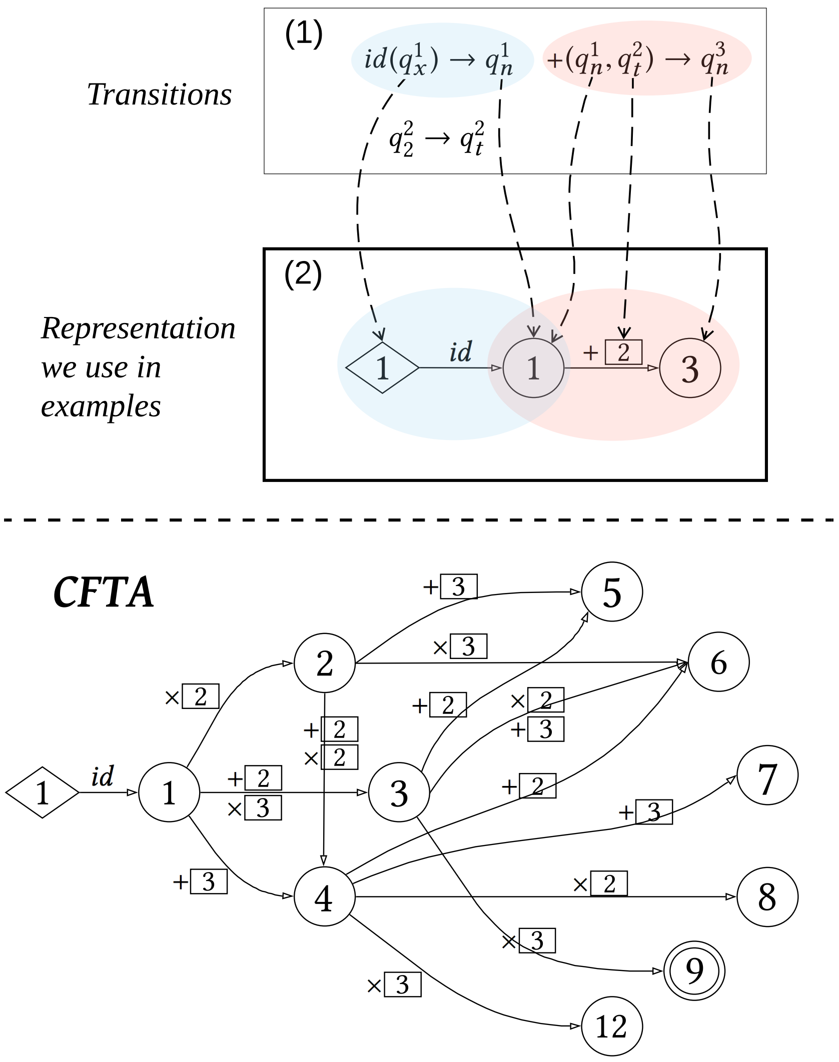
$$\begin{array}{rcl} n & := & id(x) \mid n + t \mid n \times t; \\ t & := & 2 \mid 3; \end{array}$$

Here, *id* is the identity function.

Figure 2.3 shows the CFTA constructed for this DSL and the input-output example $1 \rightarrow 9$. [3] It represents the set of DSL programs with at most two $+$ or $\times$ operators that satisfy the given example. For readability, we use circles to represent states of the form q_n^c, diamonds to represent q_x^c and squares to represent q_t^c, and the number labeling the node shows the value of c.

We now explain how we construct states and transitions in the CFTA for Example 2.2.1. There is a state q_x^1 since the value of x is 1 in the input

[3]We visualize a CFTA as a graph. Nodes in the graph correspond to states in the CFTA and are labeled with concrete values. Edges correspond to transitions and are labeled with the operator (*i.e.*, $+$ or $\times$) followed by the constant operand (*i.e.*., 2 or 3). For example, the three transitions shown in Figure 2.3 (1) are represented graphically in Figure 2.3 (2). Note that, in order to simplify our graphical representation, we do not include transitions that involve nullary functions in the graph. For instance, the transition $q_2^2 \rightarrow q_t^2$ in (1) is not included in (2). A transition of the form $f(q_n^{c_1}, q_t^{c_2}) \rightarrow q_n^{c_3}$ is represented by an edge from a node labeled c_1 to another node labeled c_3, and the connecting edge is labeled by f followed by c_2. For instance, the transition $+(q_n^1, q_t^2) \rightarrow q_n^3$ in (1) is represented by an edge from 1 to 3 with label $+2$ in (2).



(1)
Transitions
$id(q_x^1) \rightarrow q_n^1$
$+(q_n^1, q_t^2) \rightarrow q_n^3$
$q_2^2 \rightarrow q_t^2$
(2)
Representation we use in examples
id
CFTA

Figure 2.3: A CFTA example.

example (Var rule). The transitions are constructed using the DSL's concrete semantics (Prod rule). For instance, there is a transition $id(q_x^1) \rightarrow q_n^1$ because $id(1)$ yields value 1 for symbol n. Similarly, there is a transition $+(q_n^1, q_t^2) \rightarrow q_n^3$

since the result of adding 1 and 2 is 3. The only accepting state is q_n^9 since the start symbol in the grammar is n and the output example is 9. This CFTA accepts two programs, namely, $(id(x) + 2) \times 3$ and $(id(x) \times 3) \times 3$. Observe that these are the only two programs with at most two $+$ or $\times$ operators in the DSL that are consistent with the given input-output example $1 \rightarrow 9$.

2.3 Implementation

We have implemented our FTA-based synthesis algorithm in a framework called Blaze, written in Java. Blaze is parametrized over a DSL and its operational semantics. It consists of two main modules, namely, an FTA construction procedure and a ranking algorithm. Since our implementation of the FTA construction procedure follows our technical presentation, we only focus on the implementation of the ranking algorithm, which is used to find a "best" program that is accepted by the FTA.

Our heuristic ranking algorithm returns a *minimum-cost* AST accepted by the FTA, where the cost of an AST is defined as follows:

$$\begin{array}{lcl} Cost(Leaf(t)) & = & Cost(t) \\ Cost(Node(f, \vec{\Pi})) & = & Cost(f) + \sum_i Cost(\Pi_i) \end{array}$$

In the above definition, $Leaf(t)$ represents a leaf node of the AST labeled with terminal t, and $Node(f, \vec{\Pi})$ represents a non-leaf node labeled with DSL operator f and sub-trees $\vec{\Pi}$. Observe that the cost of an AST is essentially calculated using the costs of the DSL operators and terminals, which are provided by the domain expert.

In our implementation, we identify a minimum-cost AST accepted by an FTA using the algorithm presented by [19] for finding a *minimum weight* B-path in a weighted hypergraph. In the context of the ranking algorithm, we view an FTA as a hypergraph where states correspond to nodes and a transition $f(q_1, \cdots, q_n) \to q$ represents a B-arc $(\{q_1, \cdots, q_n\}, \{q\})$ where the weight of the arc is given by the cost of the DSL operator f. We also add a dummy node r in the hypergraph and an edge with weight $cost(s)$ from r to every node labeled q_s^c where s is a terminal symbol in the grammar. Given such a hypergraph representation of the FTA, the minimum-cost AST accepted by the FTA corresponds to a minimum-weight B-path from the dummy node r to a node representing a final state in the FTA.

2.4 Application

We instantiate Blaze in an application domain called data completion. In what follows, we briefly describe what data completion tasks look like as well as how we instantiate our Blaze framework in this domain.

Data Completion. Many applications store data in a tabular format. For example, Excel spreadsheets, R dataframes, and relational databases all view the underlying data as a 2-dimensional table consisting of *cells*. In this context, a common task is to fill the values of some cells based on values stored in other cells. For instance, consider the following common data completion tasks:

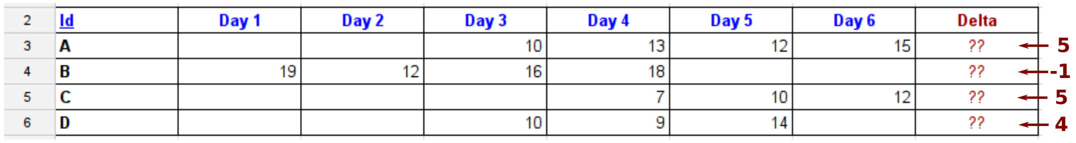
- ***Data imputation:*** In statistics, *imputation* means replacing missing data with substituted values. Since missing values can hinder data analytics tasks, users often need to fill missing values using other related entries in the table. For instance, data imputation arises frequently in statistical computing frameworks, such as R and *pandas*.

- ***Spreadsheet computation:*** In spreadsheets, users need to calculate the value of a cell based on values from other cells. For instance, a common task is to introduce new columns, where each value in the new column is derived from values in existing columns.

- ***Virtual columns in databases:*** In relational databases, users sometimes create *views* that store the result of some database query. In this context, a common task is to add *virtual columns* whose values are computed using existing entries in the view.

As illustrated by these examples, users often need to complete missing values in tabular data. While some of these data completion tasks are fairly straightforward, many others require non-trivial programming knowledge that is beyond the expertise of end-users and data scientists.

To illustrate a typical data completion task, consider an example shown in Figure 2.4. Here, the table stores measurements for different people during a certain time period, where each row represents a person and each column



2	Id	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Delta	
3	A			10	13	12	15	??	5
4	B	19	12	16	18			??	-1
5	C				7	10	12	??	5
6	D			10	9	14		??	4

Figure 2.4: An example data completion task.

corresponds to a day. As explained in a StackOverflow post [4], a data scientist analyzing this data wants to compute the difference of the measurements between the first and last days for each person and record this information in the *Delta* column. Since the table contains a large number of rows (of which only a small subset is shown in Figure 2.4), manually computing this data is prohibitively cumbersome. Furthermore, since each person's start and end date is different, automating this task requires non-trivial programming logic.

We have applied program synthesis techniques in this domain in order to allow non-expert users to automate data completion tasks in tabular data sources, such as dataframes, spreadsheets, and relational databases. Our synthesis methodology is based on two key insights that we gained by analyzing dozens of posts on online forums: First, it is often easy for end-users to specify *which operators* should be used in the data completion task and provide a specific instantiation of the operands for a few example cells. However, it is typically very difficult for end-users to express the *general operand extraction logic*. For instance, for the example from Figure 2.4, the user knows that the

[4]http://stackoverflow.com/questions/30952426/substract-last-cell-in-row-from-first-cell-with-number

missing value can be computed as $C_1 - C_2$, but he is not sure how to implement the logic for extracting C_1, C_2 in the general case.

Based on this observation, our synthesis methodology for data completion combines *program sketching* and *programming-by-example.* Specifically, given a *formula sketch* (*e.g.*, `SUM(`$?_1$`,AVG(`$?_2$`,`$?_3$`)))` and a few input-output examples for each hole, our technique automatically synthesizes a program that can be used to fill *all* missing values in the table. For instance, in our running example, the user provides the sketch `MINUS(`$?_1$`,`$?_2$`)` and the following input-output examples for the two holes:

$?_1$	$?_2$
(A, Delta) $\mapsto$ (A, Day 6)	(A, Delta) $\mapsto$ (A, Day 3)
(B, Delta) $\mapsto$ (B, Day 4)	(B, Delta) $\mapsto$ (B, Day 1)

Given these examples, our technique automatically synthesizes a program that can be used to fill all values in the *Delta* column in Figure 2.4.

In what follows, we first describe our specification language that combines sketching and examples. Then, we present a PBE technique that generates programs from the given examples to replace holes in the sketch. In particular, our PBE technique is based on an instantiation of the BLAZE framework.

Specifications. A specification in our synthesis methodology is a pair $(\mathcal{S}, \mathcal{E})$, where $\mathcal{S}$ is a formula sketch and $\mathcal{E}$ is a set of input-output examples. Specifically, formula sketches are defined by the following grammar:

$$\begin{array}{rcl} \text{Sketch } \mathcal{S} & := & t \mid F(\mathcal{S}_1, \cdots, \mathcal{S}_n), \quad F \in \Lambda \\ \text{Term } t & := & const \mid ?_{id} \end{array}$$

Here, Λ denotes a family of pre-defined functions, such as `SUM`, `MAX`, etc. Holes in the sketch represent unknown *cell extraction programs* to be synthesized. Observe that formula sketches can contain multiple functions. For instance, $\texttt{SUM}(\texttt{MAX}(\texttt{?}_1, \texttt{?}_2), 1)$ is a valid sketch and indicates that a missing value in the table should be filled by adding 1 to the maximum of two unknown cells.

In many cases, the data completion task involves copying values from an existing cell. In this case, the user can express the intent using the identity sketch $\texttt{ID}(\texttt{?}_1)$. Since this sketch is quite common, we abbreviate it using the notation $\texttt{?}_1$.

In addition to the sketch, users are also expected to provide one or more input-output examples $\mathcal{E}$ for each hole. Specifically, examples $\mathcal{E}$ map each hole $\texttt{?}_{id}$ in the sketch to a set of pairs of the form $i \mapsto [o_1, \cdots, o_n]$, where i is an input cell and $[o_1, \cdots, o_n]$ is the desired list of output cells. Hence, examples have the following shape:

$$\text{Examples } \mathcal{E} := \Big\{ \texttt{?}_{id} \hookrightarrow \{ i \mapsto [o_1, \cdots, o_n] \} \Big\}$$

Here, each cell in the table is represented as a pair (x, y), where x and y denote the row and column of the cell respectively.

Given a specification $(\mathcal{S}, \mathcal{E})$, the key learning task is to synthesize a program P_{id} for each hole $\texttt{?}_{id}$ such that P_{id} satisfies all examples $\mathcal{E}[\texttt{?}_{id}]$. [5] For a list of programs $\mathcal{P} = [P_1, \cdots, P_n]$, we write $\mathcal{S}[\mathcal{P}]$ to denote the resulting

[5] Since expressions in the holes of the sketch formula only depend on input cells (and not on other holes), running the synthesis algorithm once per hole is sufficient.

program that is obtained by replacing each hole $\texttt{?}_{id}$ in sketch $\mathcal{S}$ with P_{id}. Once a cell extraction program P_{id} is synthesized for each hole, it computes missing values in table T using $\mathcal{S}[\mathcal{P}](\mathsf{T}, c)$ where c denotes a cell with missing value. In the rest of this section, we assume that missing values in the table are identified using the special symbol ?. For instance, the analog of ? is the symbol `NA` in R and blank cell in Excel.

Domain-specific language. The syntax of the DSL is shown in Figure 2.5, and its denotational semantics is presented in Figure 2.6. We now review the key constructs in the DSL together with their semantics.

A cell extraction program π takes as input a table T and a cell x, and returns a list of cells $[c_1, \cdots, c_n]$ or the special value $\perp$. Here, $\perp$ can be viewed as an "exception" and indicates that π fails to extract any cell for input x. A cell extraction program π is either a *simple program* ρ without branches or a conditional of the form $\mathsf{Seq}(\rho, \pi)$. As shown in Figure 2.6, the semantics of $\mathsf{Seq}(\rho, \pi)$ is that the argument π is evaluated only if ρ fails (*i.e.*, returns $\perp$). Our DSL includes conditionals in the form of a Seq construct rather than a full-fledged conditional statement (e.g., *if*(C) *then* $\cdots$ *else* $\cdots$) because we have found it to be sufficiently expressive to capture most real-world data completion scenarios. This design choice also simplifies the synthesis task because the learning algorithm does not need to infer predicates for each branch.

Let us now consider the syntax and semantics of simple programs ρ. A simple program is either a list of cell extraction programs (*i.e.*., $\mathsf{List}(\tau_1, \cdots, \tau_n)$),

$$
\begin{array}{rcl}
\text{Extractor } \pi & := & \lambda \mathsf{T}.\lambda x.\rho \mid \lambda \mathsf{T}.\lambda x.\mathsf{Seq}(\rho, \pi) \\
\text{Simple prog. } \rho & := & \mathsf{List}(\tau_1, \cdots, \tau_n) \mid \mathsf{Filter}(\tau_1, \tau_2, \tau_3, \lambda y.\lambda z.p) \\
\text{Cell prog. } \tau & := & x \mid \mathsf{GetCell}(\tau, \mathsf{dir}, k, \lambda y.\lambda z.p) \\
\text{Predicate } p & := & \mathsf{True} \mid \mathsf{Val}(\chi(z)) = s \mid \mathsf{Val}(\chi(z)) \neq s \mid \mathsf{Val}(\chi(y)) = \mathsf{Val}(\chi(z)) \mid p_1 \wedge p_2 \\
\text{Cell mapper } \chi & := & \lambda c.c \mid \lambda c.(k, \mathsf{col}(c)) \mid \lambda c.(\mathsf{row}(c), k) \\
\text{Direction } \mathsf{dir} & := & \mathsf{u} \mid \mathsf{d} \mid \mathsf{l} \mid \mathsf{r}
\end{array}
$$

Figure 2.5: Data completion DSL syntax.

or a filter construct of the form $\mathsf{Filter}(\tau_1, \tau_2, \tau_3, \lambda y.\lambda z.p)$. Here, τ denotes a *cell program* for extracting a *single* cell. Filter returns all cells between τ_2 and τ_3 that satisfy the predicate ϕ. Here, ϕ takes two arguments y and z, where y is bound to the result of τ_1 and z is bound to each of the cells between τ_2 and τ_3. List and Filter constructs are necessary because many data completion tasks require extracting a *range* of values rather than a single value.

The key building block of cell extraction programs is the GetCell construct. In the simplest case, it has the form $\mathsf{GetCell}(x, \mathsf{dir}, k, \lambda y.\lambda z.p)$ where x is a cell, dir is a direction (up u, down d, left l, right r), k is an integer constant drawn from the range $[-3, 3]$, and p is a predicate. The semantics of this construct is that it finds the k'th cell satisfying predicate ϕ in direction dir from the starting cell x. For instance, the expression $\mathsf{GetCell}(x, \mathsf{r}, 0, \lambda y.\lambda z.\mathsf{True})$ refers to x itself, while $\mathsf{GetCell}(x, \mathsf{r}, 1, \lambda y.\lambda z.\mathsf{True})$ extracts the neighboring cell to the right of cell x. An interesting point about the GetCell construct is that it is recursive: For instance, if x is bound to cell (r, c), then the expression

$$\mathsf{GetCell}(\mathsf{GetCell}(x, \mathsf{u}, 1, \lambda y.\lambda z.\mathsf{True}), \mathsf{r}, 1, \lambda y.\lambda z.\mathsf{True})$$

$$
\begin{aligned}
[\![\mathsf{Val}(\chi(z)) = s]\!]_{\mathsf{T},c_1,c_2} &= \mathrm{Eval}(\mathsf{T}(\chi(c_2)), =, s) \\
[\![\mathsf{Val}(\chi(z)) \neq s]\!]_{\mathsf{T},c_1,c_2} &= \mathrm{Eval}(\mathsf{T}(\chi(c_2)), \neq, s) \\
[\![\mathsf{Val}(\chi(y)) = \mathsf{Val}(\chi(z))]\!]_{\mathsf{T},c_1,c_2} &= \mathrm{Eval}(\mathsf{T}(\chi(c_1)), =, \mathsf{T}(\chi(c_2))) \\
[\![p_1 \wedge p_2]\!]_{\mathsf{T},c_1,c_2} &= [\![p_1]\!]_{\mathsf{T},c_1,c_2} \wedge [\![p_2]\!]_{\mathsf{T},c_1,c_2}
\end{aligned}
$$

$$
\mathrm{Eval}(s_1, \lhd, s_2) = \begin{cases} \text{false} & \text{if } s_1 = \text{?} \text{ or } s_2 = \text{?} \\ s_1 \lhd s_2 & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
[\![x]\!]_{\mathsf{T},c} &= c \\
[\![\mathsf{GetCell}(\tau, \mathsf{dir}, k, \lambda y.\lambda z.p)]\!]_{\mathsf{T},c} &= \begin{cases} \bot & \text{if } [\![\tau]\!]_{\mathsf{T},c} = \bot \ \text{ or } |k| \geq \mathit{len}(L) \\ L.\mathit{get}(k) & \text{if } k \geq 0 \\ L.\mathit{get}\big(\mathit{len}(L) - |k|\big) & \text{if } k < 0 \end{cases} \\
&\quad \text{where } L = \mathtt{filter}\big(\mathtt{range}([\![\tau]\!]_{\mathsf{T},c}, \mathsf{dir}), (\lambda y.\lambda z.\ p)[\![\tau]\!]_{\mathsf{T},c}\big)
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{List}(\tau_1, \cdots, \tau_n)]\!]_{\mathsf{T},c} &= \overline{[\![\tau_1]\!]_{\mathsf{T},c}} \uplus \cdots \uplus \overline{[\![\tau_n]\!]_{\mathsf{T},c}} \\
[\![\mathsf{Filter}(\tau_1, \tau_2, \tau_3, \lambda y.\lambda z.p)]\!]_{\mathsf{T},c} &= \begin{cases} \bot & \text{if } [\![\tau_1]\!]_{\mathsf{T},c} = \bot \ \text{ or } [\![\tau_2]\!]_{\mathsf{T},c} = \bot \ \text{ or } [\![\tau_3]\!]_{\mathsf{T},c} = \bot \\ \mathtt{filter}\big(\mathtt{range}([\![\tau_2]\!]_{\mathsf{T},c}, [\![\tau_3]\!]_{\mathsf{T},c}), (\lambda y \lambda z.\ p)[\![\tau_1]\!]_{\mathsf{T},c}\big) & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\overline{c} &= \begin{cases} \bot & \text{if } c = \bot \\ [c] & \text{otherwise} \end{cases} \\
\overline{c_1} \uplus \overline{c_2} &= \begin{cases} \bot & \text{if } \overline{c_1} = \bot \text{ or } \overline{c_2} = \bot \\ \overline{c_1} :: \overline{c_2} & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
[\![\mathsf{Seq}(\rho, \pi)]\!]_{\mathsf{T},c} = \begin{cases} [\![\rho]\!]_{\mathsf{T},c} & \text{if } [\![\rho]\!]_{\mathsf{T},c} \neq \bot \\ [\![\pi]\!]_{\mathsf{T},c} & \text{otherwise} \end{cases}
$$

Figure 2.6: Data completion DSL semantics.

retrieves the cell at row $r-1$ and column $c+1$. Effectively, the recursive $\mathsf{GetCell}$ construct allows the program to "make turns" when searching for the target cell.

Another important point about the $\mathsf{GetCell}$ construct is that it returns $\perp$ if the k'th entry from the starting cell falls outside the range of the table. For instance, if the input table has 3 rows and variable x is bound to the cell in the third row and first column of the table, then $\mathsf{GetCell}(x, \mathsf{d}, 1, \lambda y.\lambda z.\mathsf{True})$ yields $\perp$. Finally, another subtlety about $\mathsf{GetCell}$ is that the k value can be negative. For instance, $\mathsf{GetCell}(x, \mathsf{u}, -1,\ \lambda y.\lambda z.\mathsf{True})$ returns the uppermost cell in x's column.

So far, we have seen how the $\mathsf{GetCell}$ construct allows us to express geometrical relationships by specifying a direction and a distance. However, many real-world data extraction tasks require combining geometrical and relational reasoning. For this purpose, predicates in our DSL can be constructed using conjunctions of relations from an expressive family. For example, unary predicates $\mathsf{Val}(\chi(z)) = s$ and $\mathsf{Val}(\chi(z)) \neq s$ in our DSL check whether or not the value of a cell $\chi(z)$ is equal to a string constant s. Similarly, binary predicates $\mathsf{Val}(\chi(y)) = \mathsf{Val}(\chi(z))$ check whether two cells contain the same value. Observe that the mapper function χ used in the predicate yields a new cell that shares some property with its input cell z. For instance, the cell mapper $\lambda c.(\mathsf{row}(c), 1)$ yields a cell that has the same row as c but whose column is 1. The use of mapper functions in predicates allows us to further combine geometric and relational reasoning.

2.5 Evaluation

Now we present our experimental results on 84 data completion benchmarks that are collected from online forums.

Benchmark information. To evaluate our FTA-based synthesis technique, we collected a total of 84 data completion problems from StackOverflow using the following methodology: First, we collected all those posts that contain relevant keywords such as *"data imputation", "missing value", "missing data", "spreadsheet formula"*, and so on. Then, we inspected each of these posts and retained exactly those that satisfy the following criteria:

- The question in the post should involve a data completion task.
- The post should contain at least one example.
- The post should include either the desired program or its English description.

Among the 84 benchmarks collected using this methodology, 46 involve data imputation in languages such as R and Python, 32 perform spreadsheet computation in Excel and Google Sheets, and 6 involve data completion in relational databases. More detailed statistics can be found in Figure 2.7.

Recall that an input to our synthesis algorithm consists of (a) a small example table, (b) a sketch formula, and (c) a mapping from each hole in the sketch to a set of examples of the form $i \mapsto [o_1, \cdots, o_n]$. As it turns out, most

	Benchmark category description	Formula sketch	Count	Avg. table size	Avg. # examples per hole
1	Fill missing value by previous/next non-missing value with/without same keys.	$?_1$	24	24.4	5.3
2	Fill missing value by previous (next) non-missing value with/without same keys if one exists, otherwise use next (previous) non-missing value	$?_1$	9	25.6	5.7
3	Replace missing value by the average of previous and next non-missing values.	$\texttt{AVG}(?_1, ?_2)$	3	12.7	2.3
4	Fill missing value by the average of previous and next non-missing values, but if either one does not exist, fill by the other one.	$\texttt{AVG}(?_1)$	2	21.5	4
5	Replace missing value by the sum of previous non-missing value (with or without the same key) and a constant.	$\texttt{SUM}(?_1, c)$	3	31.3	5.7
6	Replace missing value by the average of all non-missing values in the same row/column (with or without same keys).	$\texttt{AVG}(?_1)$	7	21.7	3.1
7	Replace missing value by the max/min of all non-missing values in the column with the same key.	$\texttt{MAX}(?_1),\ \texttt{MIN}(?_1)$	2	28.0	3
8	Fill missing value by linear interpolation of previous/next non-missing values.	$\texttt{INTERPOLATE}(?_1, ?_2)$	2	28.0	7.5
9	Fill cells by copying values from other cells in various non-trivial ways, such as by copying the first/last entered entry in the same/previous/next row, and etc.	$?_1$	13	44.5	10.2
10	Fill value by the sum of a range of cells in various ways, such as by summing all values to the left with the same keys.	$\texttt{SUM}(?_1)$	4	47.8	10.3
11	Fill cells with the count of non-empty cells in a range.	$\texttt{COUNT}(?_1)$	1	32.0	3
12	Fill cells in a column by the sum of values from two other cells.	$\texttt{SUM}(?_1, ?_2)$	2	38.3	6.5
13	Fill each value in a column by the difference of values in two other cells in different columns found in various ways.	$\texttt{MINUS}(?_1, ?_2)$	4	39.0	3.5
14	Replace missing value by the average of two non-missing values to the left.	$\texttt{AVG}(?_1, ?_2)$	1	32.0	5
15	Complete a column so that each value is the difference of the sum of a range of cells and another fixed cell.	$\texttt{MINUS}(\texttt{SUM}(?_1), ?_2)$	1	27.0	8
16	Fill each value in a column by the difference of a cell and sum of a range of cells.	$\texttt{MINUS}(?_1, \texttt{SUM}(?_2))$	1	10.0	3
17	Create column where each value is the max of previous five cells in sibling column.	$\texttt{MAX}(?_1)$	1	60.0	15
18	Fill blank cell in a column by concatenating two values to its right.	$\texttt{CONCAT}(?_1, ?_2)$	1	12.0	2
19	Fill missing value by the linear extrapolation of the next two non-missing values to the right, but if there is only one or zero such entries, fill by the linear extrapolation of the previous two non-missing values to the left.	$\texttt{EXTRAPOLATE}(?_1)$	1	121.0	16
20	Replace missing values by applying an equation (provided by the user) to the previous and next non-missing values.	$\texttt{SUM}(?_1, \frac{\texttt{MINUS}(?_1, ?_2)}{\texttt{ROW}(?_2) - \texttt{ROW}(?_1)})$	1	60.0	9
21	Fill missing value using the highest value or linear interpolation of two values before and after it, based on two different criteria.	—	1	60.0	10
	Summary		84	32.0	6.3

Figure 2.7: Data completion benchmark statistics.

posts contain exactly the type of information: Most questions related to data completion already come with a small example table, a simple formula (or a short description in English), and a few examples that show how to instantiate the formula for concrete cells in the table.

Experimental setup. Since Blaze is meant to be used in an interactive mode where the user iteratively provides more examples, we simulated a realistic usage scenario in the following way: First, for each benchmark, we collected the set $\mathcal{S}$ of all examples provided by the user in the original StackOverflow post. We then randomly picked a single example e from $\mathcal{S}$ and used Blaze to synthesize a program P satisfying e. If P failed any of the examples in $\mathcal{S}$, we then randomly sampled a failing test case e' from $\mathcal{S}$ and used Blaze to synthesize a program that satisfies both e and e'. We repeated this process of randomly sampling examples from $\mathcal{S}$ until either (a) the synthesized program P satisfies all examples in $\mathcal{S}$, or (b) we exhaust all examples in $\mathcal{S}$, or (c) we reach a time-out of 30 seconds per synthesis task. At the end of this process, we manually inspected the program P synthesized by Blaze and checked whether P conforms to the description provided by the user.

Results. We present the main results of our evaluation of Blaze in Figure 2.8. The column *"# Solved"* shows the number of benchmarks that can be successfully solved by Blaze for each benchmark category. Overall, Blaze can successfully solve over 92% of the benchmarks. Among the six benchmarks

Category	Count	Blaze					Prose				
		# Solved	*Running time per benchmark (sec)*		*# Examples used per hole*		# Solved	*Running time per benchmark (sec)*		*# Examples used per hole*	
			Avg.	Med.	Avg.	Med.		Avg.	Med.	Avg.	Med.
1	24	24	0.41	0.04	1.1	1.0	24	1.32	0.73	1.1	1.0
2	9	9	0.50	0.13	2.7	3.0	7	4.88	1.13	2.4	2.0
3	3	3	0.05	0.04	1.0	1.0	3	5.16	5.89	1.0	1.0
4	2	2	0.19	0.19	2.0	2.0	1	2.11	2.11	2.0	2.0
5	3	3	0.18	0.14	1.3	1.0	3	0.90	0.99	1.7	1.0
6	7	6	0.09	0.07	1.8	2.0	5	15.86	8.31	1.8	2.0
7	2	2	0.66	0.66	2.0	2.0	1	296.17	296.17	3.0	3.0
8	2	2	0.15	0.15	1.0	1.0	1	19.72	19.72	1.0	1.0
9	13	10	1.55	0.31	2.8	2.0	5	6.02	1.52	1.4	1.0
10	4	3	0.42	0.30	1.7	2.0	1	2.27	2.27	2.0	2.0
11	1	1	0.59	0.59	1.0	1.0	0	—	—	—	—
12	2	2	0.51	0.51	1.0	1.0	1	66.95	66.95	2.0	2.0
13	4	4	0.51	0.46	2.0	2.0	2	1.52	1.52	2.0	2.0
14	1	1	0.16	0.16	3.0	3.0	0	—	—	—	—
15	1	1	0.11	0.11	2.0	2.0	1	148.95	148.95	3.0	3.0
16	1	1	0.03	0.03	2.0	2.0	0	—	—	—	—
17	1	1	1.96	1.96	4.0	4.0	1	183.19	183.19	2.0	2.0
18	1	1	0.01	0.01	1.0	1.0	1	1.44	1.44	1.0	1.0
19	1	1	13.66	13.66	5.0	5.0	0	—	—	—	—
20	1	1	1.92	1.92	1.0	1.0	0	—	—	—	—
21	1	0	—	—	—	—	0	—	—	—	—
All	84	78	0.70	0.19	1.8	2.0	57	16.09	1.18	1.5	1.0

Figure 2.8: Blaze vs. Prose in data completion domain.

that cannot be solved by Blaze, one benchmark (Category 21) cannot be expressed using our specification language. For the remaining 5 benchmarks, Blaze fails to synthesize the correct program due to limitations of our DSL, mainly caused by the restricted vocabulary of predicates. For instance, two benchmarks require capturing the concept "nearest", which is not expressible by our current predicate language.

Next, let us consider the running time of Blaze, which is shown in the column labeled *"Running time per benchmark"*. We see that Blaze is quite fast in general and takes an average of 0.7 seconds to solve a benchmark. The median time to solve these benchmarks is 0.19 seconds. In cases where the sketch contains multiple holes, the reported running times include the time to synthesize *all* holes in the sketch. In more detail, Blaze can synthesize 75% of the benchmarks in under one second and 87% of the benchmarks in under three seconds. There is one benchmark (Category 19) where Blaze's running time exceeds 10 seconds. This is because (a) the size of the example table provided by the user is large in comparison to other example tables, and (b) the table contains over 100 irrelevant strings that form the universe of constants used in predicates. These irrelevant entries cause Blaze to consider over $30,000$ predicates to be used in the GetCell and Filter programs.

Finally, let us look at the number of examples used by Blaze, as shown in the column labeled *"# Examples used per hole"*. As we can see, the number of examples used by Blaze is much smaller than the total number of examples provided in the benchmark (as shown in Figure 2.7). Specifically, while Stack-Overflow users provide about 6 examples on average, Blaze requires only about 2 examples to synthesize the correct program. This statistic highlights that Blaze can effectively learn general programs from very few input-output examples.

Comparison with Prose. Since our FTA-based synthesis technique can be viewed as a new version space learning algorithm, we also empirically compare our approach against Prose [50], which is the state-of-the-art version space learning framework that has been deployed in Microsoft products. Prose propagates example-based constraints on subexpressions using the inverse semantics of DSL operators and then represents all programs that are consistent with the examples using the VSA data structure [31].

The Prose results are presented under the Prose column in Figure 2.8. Overall, Prose can successfully solve 68% of the benchmarks in an average of 15 seconds, whereas BLAZE can solve 92% of the benchmarks in an average of 0.7 seconds. These results indicate that BLAZE is superior to Prose, both in terms of its running time and the number of benchmarks that it can solve. Upon further inspection, we found that the tasks that can be automated using Prose tend to be relatively simple ones, where the input table size is very small or the desired program is relatively simple. For benchmarks that have larger tables or involve more complex synthesis tasks (*e.g.*, require the use of Filter operator), Prose does not scale well – it might take much longer time than BLAZE, time out in 10 minutes, or run out of memory. On the other hand, BLAZE achieves better performance than Prose because the FTA representation used in BLAZE is more compact than the VSA data structure used in Prose. In particular, in our experiments, the average FTA size is 2k, whereas the average VSA volume is 70k. Among the benchmarks that both techniques can solve, the reduction ratio of the data structure size ranges from 2x to 100x.

The careful reader may have observed in Figure 2.8 that Prose requires fewer examples on average than Blaze (1.5 vs. 1.8). However, this is quite misleading, as the benchmarks that can be solved using Prose are relatively simple and therefore require fewer examples on average.

Comparison with Sketch. Since our synthesis methodology involves a sketching component in addition to examples, we also compare Blaze against Sketch, which is the state-of-the-art tool for program sketching, and the results are shown in Figure 2.9. To compare Blaze against Sketch, we define the DSL operators using nested and recursive structures in Sketch. For each struct, we define two corresponding functions, namely RunOp and LearnOp. The RunOp function defines the semantics of the operator whereas LearnOp encodes a Sketch generator that defines the bounded space of all possible expressions in the DSL. The specification is encoded as a sequence of assert statements of the form assert RunExtractor(LearnExtractor(), i_k) $==$ L_k, where (i_k, L_k) denotes the input-output examples. To optimize the sketch encoding further, we use the input-output examples inside the LearnOp functions, and we also manually unroll and limit the recursion in predicates and cell programs to 3 and 4 respectively.

When we use the complete DSL encoding, Sketch was able to solve only 1 benchmark out of 84 within a time limit of 10 minutes per benchmark. We then simplified the Sketch encoding by removing the Seq operator, which allows us to synthesize only conditional-free programs. As shown in Figure 2.8,

Category	Count	**Blaze**					**Sketch**					
		# Solved	*Running time per benchmark (sec)*		*# Examples used per hole*		# Solved (2 exs)	*Running time per benchmark (sec)*		# Solved (3 exs)	*Running time per benchmark (sec)*	
			Avg.	Med.	Avg.	Med.		Avg.	Med.		Avg.	Med.
1	24	24	0.41	0.04	1.1	1.0	6	230	224	6	314	281
2	9	9	0.50	0.13	2.7	3.0	2	182	182	0	—	—
3	3	3	0.05	0.04	1.0	1.0	0	—	—	0	—	—
4	2	2	0.19	0.19	2.0	2.0	0	—	—	0	—	—
5	3	3	0.18	0.14	1.3	1.0	0	—	—	0	—	—
6	7	6	0.09	0.07	1.8	2.0	5	353	352	4	399	400
7	2	2	0.66	0.66	2.0	2.0	0	—	—	0	—	—
8	2	2	0.15	0.15	1.0	1.0	1	501	501	0	—	—
9	13	10	1.55	0.31	2.8	2.0	2	507	507	0	—	—
10	4	3	0.42	0.30	1.7	2.0	0	—	—	0	—	—
11	1	1	0.59	0.59	1.0	1.0	3	223	182	3	353	298
12	2	2	0.51	0.51	1.0	1.0	0	—	—	0	—	—
13	4	4	0.51	0.46	2.0	2.0	0	—	—	0	—	—
14	1	1	0.16	0.16	3.0	3.0	0	—	—	0	—	—
15	1	1	0.11	0.11	2.0	2.0	0	—	—	0	—	—
16	1	1	0.03	0.03	2.0	2.0	0	—	—	0	—	—
17	1	1	1.96	1.96	4.0	4.0	1	78	78	1	81	81
18	1	1	0.01	0.01	1.0	1.0	0	—	—	0	—	—
19	1	1	13.66	13.66	5.0	5.0	0	—	—	0	—	—
20	1	1	1.92	1.92	1.0	1.0	0	—	—	0	—	—
21	1	0	—	—	—	—	0	—	—	0	—	—
All	84	78	0.70	0.19	1.8	2.0	20	289	226	14	330	314

Figure 2.9: Blaze vs. Sketch in data completion domain.

Sketch terminated on 20 benchmarks within 10 minutes using 2 input-output examples. The average time to solve each benchmark was 289 seconds. However, on manual inspection, we found that most of the synthesized programs were not the desired ones. When we increase the number of input-output examples to 3, 14 benchmarks terminated with an average of 330 seconds, but only 5 of these 14 programs were the desired ones. We believe that Sketch performs poorly due to two reasons: First, the constraint-based encoding in Sketch does not scale for complex synthesis tasks that arise in the data completion domain. Second, since it is difficult to encode our domain-specific ranking heuristics using primitive cost operations supported by Sketch, it often generates undesired programs. In summary, this experiment confirms that a general-purpose program sketching tool is not adequate for automating the kinds of data completion tasks that arise in practice.

Chapter 3

Improving Efficiency using Abstract Interpretation

In the previous chapter, we saw a novel synthesis approach that is based on CFTAs. Essentially, a CFTA associates each grammar symbol with *concrete values* by executing the DSL constructs on the provided input examples. While this CFTA-based approach is quite generic, it suffers from scalability issues as the number of states in the automaton grows exponentially.

In this chapter, we present a more scalable synthesis algorithm, which is based on *abstract interpretation* [12]. We first describe an abstraction-based synthesis algorithm that is able to synthesize programs consistent with the specification according to the abstract semantics. However, the synthesized program might not satisfy the specification according to the concrete semantics. We then describe a technique that iteratively refines the abstraction and ensures we always synthesize programs that satisfy the specification according to the concrete semantics.

3.1 Program Synthesis using Abstract Finite Tree Automata

In this section, we present an efficient programming-by-example technique based on *abstract finite tree automata* (AFTAs). Given a DSL with its *abstract semantics* as well as a set of input-output examples, our AFTA-based algorithm synthesizes a program that satisfies the given examples with respect to the DSL's abstract semantics. This abstraction-based approach essentially performs *predicate abstraction* over the concrete values of each grammar symbol. Therefore, we first start by reviewing some requirements on the underlying abstract domain before we describe our AFTA-based synthesis algorithm.

3.1.1 Abstractions

As mentioned earlier, CFTAs associate each grammar symbol with *concrete values* by executing the DSL's *concrete semantics* on the user-provided inputs. To construct AFTAs, we will associate each grammar symbol with *abstract values* by executing the DSL's *abstract semantics* on the user-provided inputs. In the rest of this dissertation, we assume abstract values are represented as *conjunctions* of predicates of the form $f(s)$ *op* c, where s is a symbol in the DSL's grammar, f is a function, and c is a constant. For instance, if symbol s corresponds to an array, then predicate $len(s) > 0$ may indicate that the array is non-empty. Similarly, if s represents a matrix, then $rows(s) = 4$ could indicate that s contains exactly 4 rows.

Our AFTA-based synthesis algorithm is parametrized with a DSL that is provided by a *domain expert*. In particular, a domain expert provides the DSL's syntax (written in a context-free grammar) and its *abstract semantics*. We assume the abstract semantics is specified by a universe of predicates and a set of abstract transformers.

Universe of predicates. A domain expert provides a suitable universe $\mathcal{U}$ of predicates that may appear in the abstract domain used in our algorithm [1]. In particular, given a family of functions $\mathcal{F}$, a set of operators $\mathcal{O}$, and a set of constants $\mathcal{C}$ specified by the domain expert, the universe $\mathcal{U}$ includes any predicate of the form $f(s)$ *op* c where $f \in \mathcal{F}$, $op \in \mathcal{O}$, $c \in \mathcal{C}$, and s is symbol in the DSL's grammar. To ensure the completeness of our synthesis approach, we require that $\mathcal{F}$ always contains the identity function, $\mathcal{O}$ includes equality, and $\mathcal{C}$ includes all concrete values that grammar symbols can take. As we will see, this requirement ensures that every CFTA can be expressed as an AFTA over our predicate abstraction. We also assume that $\mathcal{U}$ always includes *true*, again in order to ensure the completeness of our synthesis algorithm.

Notations. Given two abstract values $\varphi_1 \in \mathcal{U}$ and $\varphi_2 \in \mathcal{U}$, we write $\varphi_1 \sqsubseteq \varphi_2$ *iff* the formula $\varphi_1 \Rightarrow \varphi_2$ is logically valid. As standard in abstract interpretation [12], we write $\gamma(\varphi)$ to denote the set of concrete values represented by abstract value φ. Given a set of predicates $\mathcal{P} = \{p_1, \cdots, p_n\} \subseteq \mathcal{U}$ and a predicate $\varphi \in \mathcal{U}$, we write $\alpha^{\mathcal{P}}(\varphi)$ to denote the *strongest* conjunction of predicates

[1]An abstract domain is always a subset of the universe $\mathcal{U}$.

in $\mathcal{P}$ that is logically implied by φ. Finally, given a vector of abstract values $\vec{\varphi} = [\varphi_1, \cdots, \varphi_n]$, we write $\alpha^{\mathcal{P}}(\vec{\varphi})$ to mean $\vec{\varphi}'$ where each $\varphi'_i = \alpha^{\mathcal{P}}(\varphi_i)$.

Abstract transformers. In addition to specifying a universe of predicates, we assume that the domain expert also specifies the DSL's abstract semantics by providing abstract transformers over predicates in $\mathcal{U}$ for each DSL construct. For a production $s \to f(s_1, \cdots, s_n)$ in the grammar with DSL construct f, we represent its abstract transformer using the notation $[\![f(\varphi_1, \cdots, \varphi_n)]\!]^\sharp$. That is, given abstract values $\varphi_1, \cdots, \varphi_n$ for the arguments $s_1, \cdots, s_n$, the transformer $[\![f(\varphi_1, \cdots, \varphi_n)]\!]^\sharp$ returns an abstract value φ for s. We require that the abstract transformers are *sound*, *i.e.*:

$$\begin{aligned}&\text{If } [\![f(\varphi_1, \cdots, \varphi_n)]\!]^\sharp = \varphi \text{ and } c_1 \in \gamma(\varphi_1), \cdots, c_n \in \gamma(\varphi_n),\\ &\text{then } [\![f(c_1, \cdots, c_n)]\!] \in \gamma(\varphi)\end{aligned}$$

However, in general, we do not require the abstract transformers to be *precise*. That is, if we have $[\![f(\varphi_1, \cdots, \varphi_n)]\!]^\sharp = \varphi$, it is possible that $\varphi \sqsupseteq \alpha^{\mathcal{U}}(S)$ where S is the set of concrete values that contains exactly $[\![f(c_1, \cdots, c_n)]\!]$ for every $c_i \in \gamma(\varphi_i)$. In other words, we allow an abstract transformer to produce an abstract value that is not the strongest over the universe $\mathcal{U}$. We do not require precision because it may be cumbersome to define the most precise abstract transformer for some DSL constructs. However, we do require an abstract transformer $[\![f(\varphi_1, \cdots, \varphi_n)]\!]^\sharp$ where each φ_i is of the form $s_i = c_i$ to be precise. Note that this can be realized using the DSL's concrete semantics:

$$[\![f(s_1 = c_1, \cdots, s_n = c_n)]\!]^\sharp = (s = [\![f(c_1, \cdots, c_n)]\!])$$

Example 3.1.1. Consider the same DSL that we used in Example 2.2.1 and suppose the universe $\mathcal{U}$ includes *true*, all predicates of the form $x = c$, $t = c$, and $n = c$ where c is an integer, and predicates $0 < n \leq 4, 0 < n \leq 8$. Then, the abstract semantics can be defined by the following abstract transformers:

$$\llbracket id(x = c) \rrbracket^\sharp := (n = c)$$

$$\llbracket (n = c_1) + (t = c_2) \rrbracket^\sharp := (n = (c_1 + c_2))$$

$$\llbracket (n = c_1) \times (t = c_2) \rrbracket^\sharp := (n = c_1 c_2)$$

$$\llbracket (0 < n \leq 4) + (t = c) \rrbracket^\sharp := \begin{cases} 0 < n \leq 4 & c = 0 \\ 0 < n \leq 8 & 0 < c \leq 4 \\ \mathit{true} & \text{otherwise} \end{cases}$$

$$\llbracket (0 < n \leq 4) \times (t = c) \rrbracket^\sharp := \begin{cases} 0 < n \leq 4 & c = 1 \\ 0 < n \leq 8 & c = 2 \\ \mathit{true} & \text{otherwise} \end{cases}$$

$$\llbracket (0 < n \leq 8) + (t = c) \rrbracket^\sharp := \begin{cases} 0 < n \leq 8 & c = 0 \\ \mathit{true} & \text{otherwise} \end{cases}$$

$$\llbracket \left(\textstyle\bigwedge_i p_i\right) \diamond \left(\textstyle\bigwedge_j p_j\right) \rrbracket^\sharp := \textstyle\bigsqcap_i \bigsqcap_j \llbracket p_i \diamond p_j \rrbracket^\sharp \qquad \diamond \in \{+, \times\}$$

In addition, an abstract transformer returns *true* if any argument is *true*.

3.1.2 Abstract Finite Tree Automata

Now we are ready to explain our synthesis algorithm based on abstract finite tree automata (AFTAs).

As mentioned earlier, AFTAs generalize CFTAs by associating abstract – rather than concrete – values with each symbol in the grammar. Because an abstract value can represent *many* different concrete values, multiple states in a CFTA might correspond to a *single* state in the AFTA. Therefore, AFTAs typically have far fewer states than their corresponding CFTAs, allowing us to construct and analyze them much more efficiently than CFTAs.

States in an AFTA are of the form $q_s^{\vec{\varphi}}$ where s is a grammar symbol and $\vec{\varphi}$ is a vector of abstract values. A transition $f(q_{s_1}^{\vec{\varphi_1}}, \cdots, q_{s_n}^{\vec{\varphi_n}}) \rightarrow q_s^{\vec{\varphi}}$ in the AFTA indicates that we have $[\![f(\varphi_{1j}, \cdots, \varphi_{nj})]\!]^\sharp \sqsubseteq \varphi_j$. Because our abstract transformers are sound, this means that formula φ_j over-approximates the result of running f on the concrete values represented by $\varphi_{1j}, \cdots, \varphi_{nj}$.

Let us now consider the AFTA construction rules shown in Figure 3.1. Similar to CFTAs, the AFTA construction requires a set of input-output examples $\vec{e}$ and the DSL's grammar $G = (T, N, P, s_0)$. In addition, the construction requires the abstract transformers for all DSL constructs (*i.e.*., $[\![f(\cdots)]\!]^\sharp$) as well as a set of predicates $\mathcal{P} \subseteq \mathcal{U}$ over which we construct our abstraction (*i.e.*, $\mathcal{P}$ defines an abstract domain).

The first two rules from Figure 3.1 are very similar to their counterparts from the CFTA construction rules in Figure 2.2. According to the Var rule, the states Q of the AFTA include a state $q_x^{\vec{\varphi}}$ where x is the input variable and $\vec{\varphi}$ is the abstraction of the input examples $\vec{e}_{in}$ with respect to the set of predicates $\mathcal{P}$. Similarly, the Const rule states that we have $q_t^{\vec{\varphi}} \in Q$ whenever t is a constant in the grammar and $\vec{\varphi}$ is the abstraction of $[t = [\![t]\!], \cdots, t = [\![t]\!]]$ with respect to $\mathcal{P}$. The next rule, labeled Final, defines the final states of the AFTA. Assuming the start symbol in the grammar is s_0, then $q_{s_0}^{\vec{\varphi}}$ is a final

$$\frac{\vec{\varphi} = \alpha^{\mathcal{P}}\big([x = \vec{e}_{in,1}, \cdots, x = \vec{e}_{in,|\vec{e}|}]\big)}{q_x^{\vec{\varphi}} \in Q} \quad (\textsc{Var})$$

$$\frac{t \in T_C \quad \vec{\varphi} = \alpha^{\mathcal{P}}\big([t = [\![t]\!], \cdots, t = [\![t]\!]]\big) \quad |\vec{\varphi}| = |\vec{e}|}{q_t^{\vec{\varphi}} \in Q} \quad (\textsc{Const})$$

$$\frac{q_{s_0}^{\vec{\varphi}} \in Q \quad \forall j \in [1, |\vec{e}_{out}|].\ (s_0 = e_{out,j}) \sqsubseteq \varphi_j}{q_{s_0}^{\vec{\varphi}} \in Q_f} \quad (\textsc{Final})$$

$$\frac{\begin{array}{c}(s \to f(s_1, \cdots, s_n)) \in P \quad q_{s_1}^{\vec{\varphi}_1} \in Q, \cdots, q_{s_n}^{\vec{\varphi}_n} \in Q \\ \varphi_j = \alpha^{\mathcal{P}}\big([\![f(\varphi_{1j}, \cdots, \varphi_{nj})]\!]^\sharp\big) \quad \vec{\varphi} = [\varphi_1, \cdots, \varphi_{|\vec{e}|}]\end{array}}{q_s^{\vec{\varphi}} \in Q, \quad \big(f(q_{s_1}^{\vec{\varphi}_1}, \cdots, q_{s_n}^{\vec{\varphi}_n}) \to q_s^{\vec{\varphi}}\big) \in \Delta} \quad (\textsc{Prod})$$

Figure 3.1: AFTA construction rules.

state whenever the concretization of $\vec{\varphi}$ includes the output examples.

The last rule, labeled Prod, deals with grammar productions of the form $s \to f(s_1, \cdots, s_n)$. Suppose that the AFTA contains states $q_{s_1}^{\vec{\varphi}_1}, \cdots, q_{s_n}^{\vec{\varphi}_n}$, which, intuitively, means that grammar symbols $s_1, \cdots, s_n$ can take abstract values $\vec{\varphi}_1, \cdots, \vec{\varphi}_n$. In this rule, we first "run" the abstract transformer for f on abstract values $\varphi_{1j}, \cdots, \varphi_{nj}$ to obtain an abstract value $[\![f(\varphi_{1j}, \cdots, \varphi_{nj})]\!]^\sharp$ over the universe $\mathcal{U}$. Then, we compute its abstraction with respect to $\mathcal{P}$ by applying the abstraction function $\alpha^{\mathcal{P}}$ to $[\![f(\varphi_{1j}, \cdots, \varphi_{nj})]\!]^\sharp$ to find the strongest conjunction φ_j of predicates over $\mathcal{P}$ that overapproximates $[\![f(\varphi_{1j}, \cdots, \varphi_{nj})]\!]^\sharp$. We add the state $q_s^{\vec{\varphi}}$ to the AFTA and the transition $f(q_{s_1}^{\vec{\varphi}_1}, \cdots, q_{s_n}^{\vec{\varphi}_n}) \to q_s^{\vec{\varphi}}$, since symbol s can take abstract value $\vec{\varphi}$.

Example 3.1.2. Consider the same DSL that we used in Example 2.2.1 as well as the universe and abstract transformers given in Example 3.1.1. Now, let us consider the set of predicates $\mathcal{P} = \{true, t = 2, t = 3, x = c\}$ where c stands for any integer value. Figure 3.2 shows the AFTA constructed for the input-output example $1 \rightarrow 9$ over predicates $\mathcal{P}$. Since the abstraction of $x = 1$ over $\mathcal{P}$ is $x = 1$, the AFTA includes a state $q_x^{x=1}$, shown simply as $x = 1$. Since $\mathcal{P}$ only has *true* for symbol n, the AFTA contains a transition $id(q_x^{x=1}) \rightarrow q_n^{true}$, where q_n^{true} is abbreviated as *true* in Figure 3.2. The AFTA also includes transitions $+(q_n^{true}, t = c) \rightarrow q_n^{true}$ and $\times(q_n^{true}, t = c) \rightarrow q_n^{true}$ for $c \in \{2, 3\}$. Observe that q_n^{true} is the only final state since n is the start symbol and the concretization of *true* includes 9 (the output example). As we can see, the language of this AFTA includes all programs that start with $id(x)$.

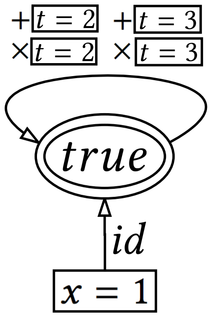


Figure 3.2: An AFTA example.

Theorem 3.1.1. **(Soundness of AFTA)** Let $\mathcal{A}$ be the AFTA constructed for a DSL (with abstract semantics), examples $\vec{e}$ and predicates $\mathcal{P}$. If Π is a program that is accepted by $\mathcal{A}$, then Π is consistent with examples $\vec{e}$ with respect to the DSL's abstract semantics under the abstract domain defined by $\mathcal{P}$.

Theorem 3.1.2. **(Completeness of AFTA)** Let $\mathcal{A}$ be the AFTA constructed for a DSL (with abstract semantics), examples $\vec{e}$ and predicates $\mathcal{P}$. If Π is a program that is consistent with examples $\vec{e}$ with respect to the DSL's abstract semantics under the abstract domain defined by $\mathcal{P}$, then Π is accepted by $\mathcal{A}$.

3.2 Program Synthesis using Abstraction Refinement

So far, we have seen an abstraction-based synthesis algorithm that is able to synthesize programs consistent with the specification according to abstract semantics. While this approach is quite efficient, one obvious implication is that the synthesized programs might be *spurious*. That is, the synthesized program may not actually satisfy the specification in terms of the concrete semantics. In other words, the synthesized programs might be wrong.

This section presents a technique that ensures we always synthesize programs that satisfy the specification according to the concrete semantics. The key idea underlying our technique is to perform abstraction refinement. That is, we first generate a candidate program using our AFTA-based method with a coarse initial abstraction. In case the candidate program is spurious, we iteratively refine the abstraction and its corresponding AFTA until we either find a program that is consistent with the given specification (in terms of the concrete semantics) or prove that there exists no such programs.

3.2.1 Algorithm Architecture

The high-level structure of our refinement-based synthesis algorithm is shown in Figure 3.3. The Learn procedure takes as input a set of examples $\vec{e}$, a context-free grammar G of the DSL, a set of predicates $\mathcal{P}$ that defines an initial abstract domain, and a universe of predicates $\mathcal{U}$. We implicitly assume that we have access to the concrete and abstract semantics of the DSL. Also, it is worth noting that the initial abstraction $\mathcal{P}$ is optional. In cases where the domain expert does not specify $\mathcal{P}$, it is set to include only the predicate *true*.

Our synthesis algorithm consists of a refinement loop (lines 2–9), in which we alternate between AFTA construction, counterexample generation, and predicate learning. In each iteration, it first constructs an AFTA $\mathcal{A}$ using the current set of predicates $\mathcal{P}$ (line 3). If the language of $\mathcal{A}$ is empty, we have a proof that there is no DSL program that satisfies the input-output examples; therefore, the algorithm returns *null* in this case (line 4). Otherwise, we use a heuristic *ranking algorithm* to choose a "best" program Π from the language defined by $\mathcal{A}$ (line 5).[2] In the remainder of this dissertation, we assume that programs are represented as abstract syntax trees (ASTs).

Once we find a program Π accepted by the current AFTA, we run it on the input examples $\vec{e}_{in}$. If the result matches the expected outputs $\vec{e}_{out}$, we return Π as a solution (line 6). Otherwise, we refine the current abstraction $\mathcal{P}$

[2]Here, we note that we do not fix a particular algorithm for Rank, so the synthesizer is free to choose any ranking heuristic as long as Rank returns a program that has the lowest cost with respect to a deterministic cost metric.

```
1: procedure LEARN($\vec{e}, G, \mathcal{P}, \mathcal{U}$)
   input: a set of input-output examples $\vec{e}$, a context-free grammar $G$ of the
   DSL, an initial abstract domain $\mathcal{P}$, and a universe $\mathcal{U}$.
   output: a DSL program consistent with input-output examples $\vec{e}$.
2:     while true do                                          ▷ Refinement loop.
3:         $\mathcal{A}$ := CONSTRUCTAFTA($\vec{e}, G, \mathcal{P}$);
4:         if $\mathcal{L}(\mathcal{A}) = \emptyset$ then return null;
5:         $\Pi$ := RANK($\mathcal{A}$);
6:         if $[\![\Pi]\!]\vec{e}_{in} = \vec{e}_{out}$ then return $\Pi$;
7:         $e$ := FINDCOUNTEREXAMPLE($\Pi, \vec{e}$);      ▷ $e \in \vec{e}$ and $[\![\Pi]\!]e_{in} \neq e_{out}$.
8:         $\mathcal{I}$ := CONSTRUCTPROOF($\Pi, e, \mathcal{P}, \mathcal{U}$);
9:         $\mathcal{P} := \mathcal{P} \cup$ EXTRACTPREDICATES($\mathcal{I}$);
```

Figure 3.3: Top-level structure of our synthesis algorithm.

so that the spurious program Π will no longer be accepted by the refined AFTA. Towards this goal, we find a single input-output example e that is inconsistent with program Π (line 7), *i.e.*., a *counterexample*, and then we construct a so-called *incorrectness proof* $\mathcal{I}$ of Π with respect to the counterexample e (line 8). In particular, $\mathcal{I}$ is a mapping from the nodes in Π's AST to abstract values in universe $\mathcal{U}$ and serves as a proof that program Π is inconsistent with example e. More formally, an incorrectness proof $\mathcal{I}$ is defined as follows.

Definition 3.2.1. **(Incorrectness Proof)** Let Π be the AST of a program that does not satisfy example e according to the concrete semantics. Then, an incorrectness proof of Π with respect to e has the following properties:

1. If v is a leaf node of Π labeled with constant t, then $(t = [\![t]\!]e_{in}) \sqsubseteq \mathcal{I}(v)$.

2. If v is an internal node labeled with function f and has children $v_1, \cdots, v_n$, then $[\![f(\mathcal{I}(v_1), \cdots, \mathcal{I}(v_n))]\!]^\sharp \sqsubseteq \mathcal{I}(v)$.

3. If $\mathcal{I}$ maps the root node of Π to abstract value φ, then $e_{out} \notin \gamma(\varphi)$.

Here, the first two properties collectively state that $\mathcal{I}$ constitutes a valid proof that executing Π (in terms of the abstract semantics) on input e_{in} yields an abstract output $\mathcal{I}(\mathit{root}(\Pi))$. The third property further shows that $\mathcal{I}$ proves Π is spurious, since Π's abstract output is not consistent with e_{out}. The following theorem states that such a proof always exists.

Theorem 3.2.1. **(Existence of Incorrectness Proofs)** Given a spurious program Π that does not satisfy example e according to concrete semantics, an incorrectness proof of Π satisfying properties in Definition 3.2.1. always exists.

Our synthesis algorithm uses such a proof $\mathcal{I}$ to refine the abstraction. In particular, the abstraction that we use in the next iteration includes all predicates that appear in $\mathcal{I}$ in addition to those in the current abstract domain defined by $\mathcal{P}$. This ensures that the AFTA constructed in the next iteration does *not accept* the spurious program Π from the current iteration.

Theorem 3.2.2. **(Progress)** Let $\mathcal{A}_i$ be the AFTA constructed in the i'th iteration of the LEARN procedure from Figure 3.3, and let Π_i be a spurious program returned by RANK. Then, we have $\Pi_i \notin \mathcal{L}(\mathcal{A}_{i+1})$ and $\mathcal{L}(\mathcal{A}_{i+1}) \subset \mathcal{L}(\mathcal{A}_i)$.

Example 3.2.1. Consider the AFTA shown in Figure 3.2 and suppose the program returned by RANK is $id(x)$. Since this program is not consistent with the input-output example $1 \to 9$, our algorithm constructs an incorrectness proof for it shown in Figure 3.4. In particular, this proof labels the root node of the AST with a new abstract value $0 < n \leq 8$, establishing that $id(x)$ is spurious because $9 \not\in [0, 8]$. In the next iteration, we add $0 < n \leq 8$ in the abstract domain $\mathcal{P}$ and construct the *refined AFTA* shown in Figure 3.4. Observe that the spurious program $id(x)$ is no longer accepted by this refined AFTA.

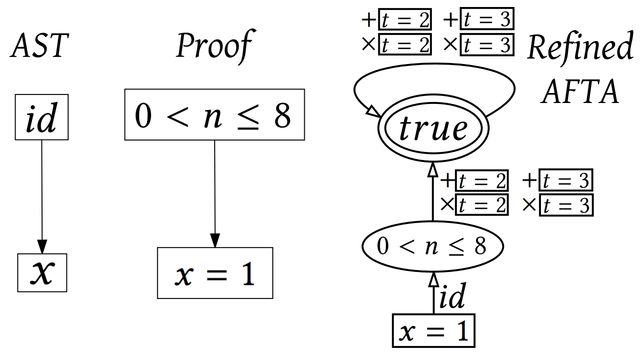


Figure 3.4: An incorrectness proof example.

Theorem 3.2.3. **(Soundness of Algorithm in Figure 3.3)** If the LEARN procedure from Figure 3.3 returns a program Π for examples $\vec{e}$, then Π satisfies $\vec{e}$, namely, $[\![\Pi]\!]\vec{e}_{in} = \vec{e}_{out}$.

Theorem 3.2.4. **(Completeness of Algorithm in Figure 3.3)** If there exists a program in the DSL that satisfies the input-output examples $\vec{e}$, then given the DSL and examples $\vec{e}$, the LEARN procedure from Figure 3.3 will return a DSL program Π such that $[\![\Pi]\!]\vec{e}_{in} = \vec{e}_{out}$.

3.2.2 Constructing Incorrectness Proofs

So far, we have seen how to incorrectness proofs are used to eliminate spurious programs from the search space. Now we discuss how to automatically construct such proofs given a spurious program.

Our proof construction algorithm ConstructProof is shown in Figure 3.5. The algorithm takes as input a spurious program Π represented as an AST with vertices V and an input-output example e such that $[\![\Pi]\!]e_{in} \neq e_{out}$. The procedure also requires the current abstraction $\mathcal{P}$ as well as the universe of predicates $\mathcal{U}$. The output is a valid incorrectness proof that maps from the verices V of Π to new abstract values proving that Π is inconsistent with e.

At a high level, the ConstructProof procedure processes the AST top-down, starting at the root node r. Specifically, we first find an annotation $\mathcal{I}(r)$ for r such that we have $e_{out} \notin \gamma(\mathcal{I}(r))$. In other words, the annotation $\mathcal{I}(r)$ is sufficient for showing that Π is spurious (property (3) from Definition 3.2.1). After we find an annotation for the root node r (lines 2–4), we add r to *worklist* and find suitable annotations for the children of all nodes in the worklist. In particular, the loop in lines 6–15 ensures that $\mathcal{I}$ also satisfies properties (1) and (2) from Definition 3.2.1.

Let us now consider the ConstructProof procedure in more detail. To find the annotation for the root node r, we first compute r's abstract value in the abstract domain $\mathcal{P}$. Towards this goal, we use a procedure called Eval-Abstract, shown in Figure 3.6, which symbolically executes Π on e_{in} using

```
1: procedure ConstructProof($\Pi, e, \mathcal{P}, \mathcal{U}$)
   input: a spurious program $\Pi$ represented as an AST with vertices $V$.
   input: a counterexample $e$ such that $[\![\Pi]\!]e_{in} \neq e_{out}$.
   input: current abstract domain $\mathcal{P}$ and the universe of predicates $\mathcal{U}$.
   output: an incorrectness proof $\mathcal{I}$ mapping from $V$ to abstract values over $\mathcal{U}$.
                          ▷ Find annotation $\mathcal{I}(r)$ for root $r$ such that $e_{out} \notin \gamma(\mathcal{I}(r))$.
2:     $\varphi :=$ EvalAbstract($\Pi, e_{in}, \mathcal{P}$);
3:     $\psi :=$ StrengthenRoot$\big(s_0 = [\![\Pi]\!]e_{in}, \varphi, s_0 \neq e_{out}, \mathcal{U}\big)$;
4:     $\mathcal{I}(\mathsf{root}(\Pi)) := \varphi \wedge \psi$;
                                                      ▷ Process all nodes other than root.
5:     $worklist := \big\{\mathsf{root}(\Pi)\big\}$;
6:     while $worklist \neq \emptyset$ do
              ▷ Find annotation $\mathcal{I}(v_i)$ for each $v_i$ s.t $[\![f(\mathcal{I}(v_1), \cdots, \mathcal{I}(v_n))]\!]^\sharp \sqsubseteq \mathcal{I}(cur)$.
7:         $cur := worklist.\mathsf{remove}()$;
8:         $\vec{\Pi} := \mathsf{ChildrenASTs}(cur)$;
9:         $\vec{\phi} := \big[\ s_i = c_i \ \big|\ c_i = [\![\Pi_i]\!]e_{in}, i \in [1, |\vec{\Pi}|], s_i = \mathsf{Symbol}(\Pi_i)\ \big]$;
10:        $\vec{\varphi} := \big[\ \varphi_i \ \big|\ \varphi_i = $ EvalAbstract$(\Pi_i, e_{in}, \mathcal{P}), i \in [1, |\vec{\Pi}|]\ \big]$;
11:        $\vec{\psi} :=$ StrengthenChildren$\big(\vec{\phi}, \vec{\varphi}, \mathcal{I}(cur), \mathcal{U}, \mathsf{label}(cur)\big)$;
12:        for $i = 1, \cdots, |\vec{\Pi}|$ do
13:            $\mathcal{I}(\mathsf{root}(\Pi_i)) := \varphi_i \wedge \psi_i$;
14:            if $\neg\mathsf{IsLeaf}(\mathsf{root}(\Pi_i))$ then
15:                $worklist.\mathsf{add}(\mathsf{root}(\Pi_i))$;
16:    return $\mathcal{I}$;
```

Figure 3.5: Incorrectness proof construction algorithm.

$$
\begin{aligned}
&\mathsf{EvalAbstract}(\mathsf{Leaf(x)}, e_{in}, \mathcal{P}) && = \alpha^{\mathcal{P}}(x = e_{in}) \\
&\mathsf{EvalAbstract}(\mathsf{Leaf(t)}, e_{in}, \mathcal{P}) && = \alpha^{\mathcal{P}}\big(t = [\![t]\!]\big) \\
&\mathsf{EvalAbstract}(\mathsf{Node}(f, \vec{\Pi}), e_{in}, \mathcal{P}) && = \\
&\qquad \alpha^{\mathcal{P}}\Big([\![f\big(\mathsf{EvalAbstract}(\Pi_1, e_{in}, \mathcal{P}), \cdots, \mathsf{EvalAbstract}\big(\Pi_{|\vec{\Pi}|}, e_{in}, \mathcal{P})\big)]\!]^\sharp\Big)
\end{aligned}
$$

Figure 3.6: Auxiliary EvalAbstract procedure used in Figure 3.5.

the abstract transformers (over $\mathcal{P}$). The return value φ of EvalAbstract at line 2 has the property that $e_{out} \in \gamma(\varphi)$, since the AFTA constructed using predicates $\mathcal{P}$ yields the spurious program Π. We then try to *strengthen* φ using a new formula ψ over predicates $\mathcal{U}$ such that the following properties hold:

1. $(s_0 = \llbracket\Pi\rrbracket e_{in}) \Rightarrow \psi$ where s_0 is the start symbol of the grammar,
2. $\varphi \wedge \psi \Rightarrow (s_0 \neq e_{out})$.

Here, the first property says that the output of Π on input e_{in} should satisfy ψ; otherwise ψ would not be a correct strengthening. The second property says that ψ, together with the previous abstract value φ, should be strong enough to show that Π is *inconsistent* with the input-output example e.

While any strengthening ψ that satisfies these two properties will be sufficient to prove that Π is spurious, we would ideally want our strengthening to rule out many other spurious programs. For this reason, we want ψ to be as general (*i.e.*, logically weak) as possible. Intuitively, the more general the proof, the more spurious programs it can likely prove incorrect. For example, while a predicate such as $s_0 = \llbracket\Pi\rrbracket e_{in}$ can prove that Π is incorrect, it only proves the spuriousness of programs that produce the same concrete output as Π on e_{in}. On the other hand, a more general predicate that is logically weaker than $s_0 = \llbracket\Pi\rrbracket e_{in}$ can potentially prove the spuriousness of other programs that may not necessarily return the same concrete output as Π on e_{in}.

To find such a suitable strengthening ψ, our algorithm makes use of a procedure called StrengthenRoot, described in Figure 3.7. In a nutshell,

```
1: procedure StrengthenRoot($p_+, p_-, \varphi, \mathcal{U}$)
   input: predicates $p_+$ and $p_-$, formula $\varphi$, and universe $\mathcal{U}$.
   output: formula $\psi^*$ such that $p_+ \Rightarrow (\varphi \wedge \psi^*) \Rightarrow p_-$.
2:     $\Phi := \{p \in \mathcal{U} \mid p_+ \Rightarrow p\}$;   $\Psi := \Phi$;   ▷ Construct universe of relevant predicates.
3:     for $i = 1, \cdots, k$ do   ▷ Generate all possible conjunctions up to length $k$.
4:         $\Psi := \Psi \bigcup \{\psi \wedge p \mid \psi \in \Psi, p \in \Phi\}$;
5:     $\psi^* := p_+$;   ▷ Find most general formula with desired property.
6:     for $\psi \in \Psi$ do
7:         if $\psi^* \Rightarrow \psi$ and $(\varphi \wedge \psi) \Rightarrow p_-$ then $\psi^* := \psi$;
8:     return $\psi^*$;
```

Figure 3.7: Algorithm for finding a strengthening for the root.

```
1: procedure StrengthenChildren($\vec{\phi}, \vec{\varphi}, \varphi_p, \mathcal{U}, f$)
   input: predicates $\vec{\phi}$, formulas $\vec{\varphi}$, formula $\varphi_p$, and universe $\mathcal{U}$.
   output: $\vec{\psi}^*$ such that $\forall i \in [1, |\vec{\psi}^*|].\ \phi_i \Rightarrow \psi_i^*$ and $[\![f(\varphi_1 \wedge \psi_1^* \cdots, \varphi_n \wedge \psi_n^*)]\!]^\sharp \Rightarrow \varphi_p$.
2:     $\vec{\Phi} := [\Phi_i \mid \Phi_i = \{p \in \mathcal{U} \mid \phi_i \Rightarrow p\}]$;   $\vec{\Psi} := \vec{\Phi}$   ▷ Construct universe of relevant
   predicates.
3:     for $i = 1, \cdots, k$ do   ▷ Generate all possible conjunctions up to length $k$.
4:         for $j = 1, \cdots, |\vec{\Psi}|$ do
5:             $\Psi_j := \Psi_j \bigcup \{\psi \wedge p \mid \psi \in \Psi_j, p \in \Phi_j\}$
6:     $\vec{\psi}^* := \vec{\phi}$;   ▷ Find most general formula with desired property.
7:     for all $\vec{\psi}$ where $\psi_i \in \Psi_i$ do
8:         if $\forall i \in [1, |\vec{\phi}|].\ \psi_i^* \Rightarrow \psi_i$ and $[\![f(\varphi_1 \wedge \psi_1, \cdots, \varphi_n \wedge \psi_n)]\!]^\sharp \Rightarrow \varphi_p$ then $\vec{\psi}^* := \vec{\psi}$;
9:     return $\vec{\psi}^*$;
```

Figure 3.8: Algorithm for finding a strengthening for nodes other than root.

this procedure returns the most general conjunctive formula ψ using at most k predicates in $\mathcal{U}$ such that the above two properties are satisfied. Since formula ψ, together with the old abstract value φ, proves the spuriousness of Π, our proof $\mathcal{I}$ maps the root node to the new strengthened abstract value $\varphi \wedge \psi$ (line 4 of ConstructProof).

The loop in lines 5–15 of ConstructProof finds annotations for all nodes other than the root node. Any AST node *cur* that has been removed from the worklist at line 7 must be in the domain of $\mathcal{I}$ (*i.e.*, we have already found an annotation for *cur*). Now, our goal is to find a suitable annotation for *cur*'s children such that $\mathcal{I}$ satisfies properties (1) and (2) from Definition 3.2.1. To find the annotation for each child v_i of node *cur*, we first compute the concrete value ϕ_i and abstract value φ_i for v_i (lines 9–10). We then invoke the StrengthenChildren procedure, shown in Figure 3.8, to find a strengthening $\vec{\psi}$ such that:

1. $\forall i \in [1, |\vec{\psi}|].\ \phi_i \Rightarrow \psi_i$
2. $[\![f(\varphi_1 \wedge \psi_1, \cdots, \varphi_n \wedge \psi_n)]\!]^\sharp \Rightarrow \mathcal{I}(cur)$

Here, the first property ensures that $\mathcal{I}$ satisfies property (1) from Definition 3.2.1. In other words, the first condition says that our strengthening over-approximates the concrete output of sub-program Π_i rooted at v_i on input e_{in}. The second condition enforces property (2) from Definition 3.2.1. In particular, it says that the annotation for the parent node is provable from the annotations of the children using the DSL's abstract semantics.

In addition to satisfying these afore-mentioned properties, the strengthening $\vec{\psi}$ returned by StrengthenChildren has some useful generality guarantees. In particular, $\vec{\psi}$ is pareto-optimal in the sense that we cannot obtain a valid strengthening $\vec{\psi}'$ (with a fixed number of conjuncts) by weakening any of the ψ_i's in $\vec{\psi}$. As mentioned earlier, finding such *maximally general* annotations is useful because it allows our synthesis procedure to rule out many spurious programs in addition to the specific one returned by RANK.

Example 3.2.2. To better understand how we construct incorrectness proofs, consider the AFTA shown in Figure 3.9 (1). Suppose that the ranking algorithm returns the program $id(x) + 2$, which is clearly spurious with respect to the input-output example $1 \rightarrow 9$. Figure 3.9 (2)-(4) show the AST for the program $id(x) + 2$ as well as the old abstract and concrete values for each AST node. Note that the abstract values in Figure 3.9 (3) correspond to the results of EVALABSTRACT in the CONSTRUCTPROOF algorithm from Figure 3.5. Our proof construction algorithm starts by strengthening the root node v_1 of the AST. Since $[\![\Pi]\!]e_{in}$ is 3, the first argument of the STRENGTHENROOT procedure is provided as $n = 3$. Since the output example is 9, the second argument is $n \neq 9$. Now, we invoke the STRENGTHENROOT procedure to find a formula ψ such that $n = 3 \Rightarrow (true \wedge \psi) \Rightarrow n \neq 9$ holds. The most general conjunctive formula over $\mathcal{U}$ that has this property is $0 < n \leq 8$; hence, we obtain the annotation $\mathcal{I}(v_1) = 0 < n \leq 8$ for the root node of the AST. The CONSTRUCTPROOF algorithm now "recurses down" to the children of v_1 to find suitable annotations for v_2 and v_3. When processing v_1 inside the while loop in Figure 3.5, we have $\vec{\phi} = [n = 1, t = 2]$ since $1, 2$ correspond to the concrete values for v_2, v_3. Similarly, we have $\vec{\varphi} = [0 < n \leq 8, t = 2]$ for the abstract values for v_2 and v_3. We now invoke STRENTHENCHILDREN to find

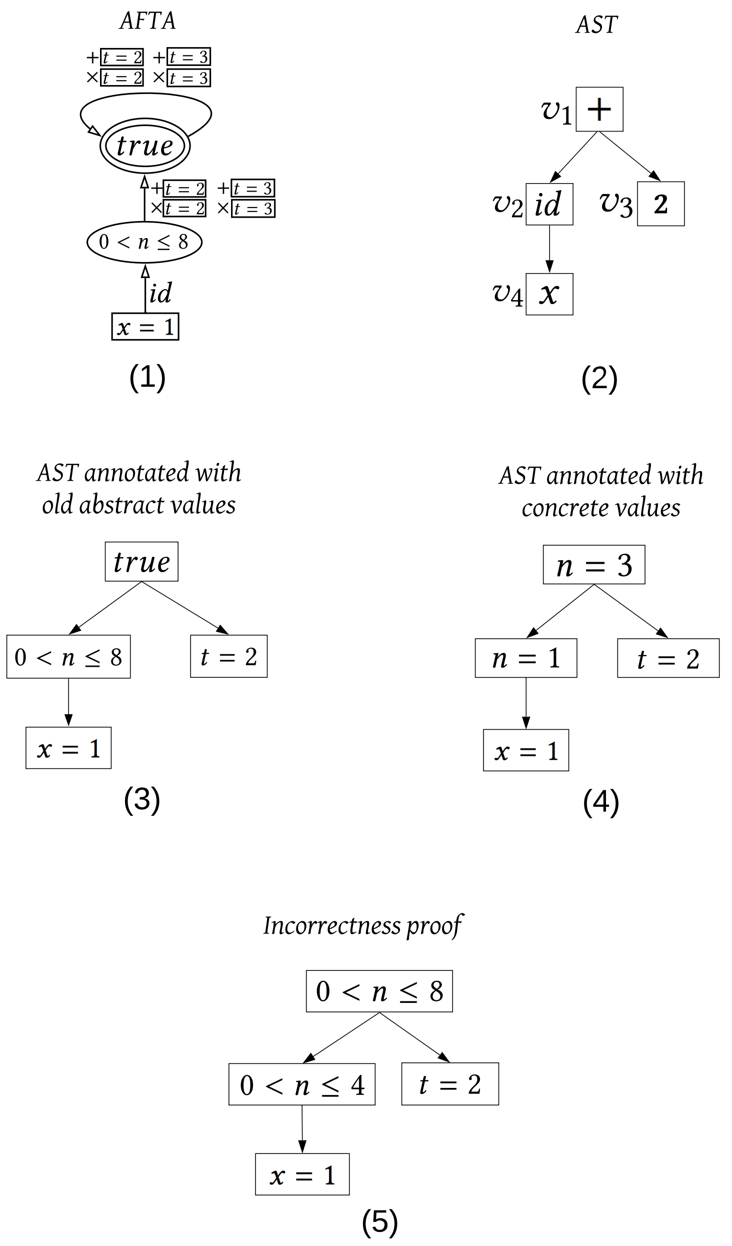


Figure 3.9: A proof construction example.

a $\vec{\psi} = [\psi_1, \psi_2]$ such that:

$$n = 1 \Rightarrow \psi_1 \qquad t = 2 \Rightarrow \psi_2$$
$$[\![+(0 < n \leq 8 \wedge \psi_1,\ t = 2 \wedge \psi_2)]\!]^\sharp \Rightarrow 0 < n \leq 8$$

In this case, StrengthenChildren yields the solution $\psi_1 = 0 < n \leq 4$ and $\psi_2 = \mathit{true}$. Therefore, we have $\mathcal{I}(v_2) = 0 < n \leq 4$ and $\mathcal{I}(v_3) = (t = 2)$. The final proof of incorrectness for this example is shown in Figure 3.9 (5).

Theorem 3.2.5. **(Correctness of Algorithm in Figure 3.5)** The mapping $\mathcal{I}$ returned by the ConstructProof procedure from Figure 3.5 satisfies the properties from Definition 3.2.1.

Complexity analysis. The complexity of our synthesis algorithm shown in Figure 3.3 is mainly determined by the number of iterations, and the complexity of FTA construction, ranking and proof construction. In particular, an FTA with size m [3] can be constructed in time $\mathcal{O}(m)$ (without any pruning). The complexity of ranking over an FTA depends on the particular ranking heuristic. For the one used in our implementation (see Chapter 3.4), the time complexity is $\mathcal{O}(m \cdot \log d)$ where m is the FTA size and d is the number of states in the FTA. The complexity of proof construction for an AST is $\mathcal{O}(l \cdot p)$ where l is the number of nodes in the AST and p is the number of conjunctions under consideration. Therefore, the overall complexity of our synthesis algorithm is $\mathcal{O}(t \cdot (l \cdot p + m \cdot \log d))$ where t is the number of refinement steps.

[3]FTA size is defined as $\sum_{\delta \in \Delta} |\delta|$ where $|\delta| = n + 1$ for a transition $\delta : f(q_1, \cdots, q_n) \rightarrow q$.

3.3 A Working Example

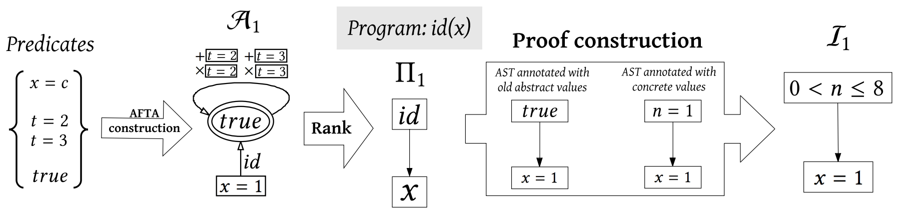
In the previous sections, we illustrated various aspects of our synthesis algorithm using the DSL from Example 2.2.1 on input-output example $1 \mapsto 9$. We now walk through the entire algorithm and show how it synthesizes the desired program $(id(x) + 2) \times 3$. We use the abstract semantics and universe of predicates $\mathcal{U}$ given in Example 3.1.1, and we use the initial abstract domain defined by $\mathcal{P}$ given in Example 3.1.2. Furthermore, we assume that Rank always favors smaller programs over larger ones. In the case of a tie, it favors programs that use + and those that use smaller constants.

Figure 3.10 illustrates all iterations of the synthesis algorithm shown in Figure 3.3. Let us now consider Figure 3.10 in more detail.

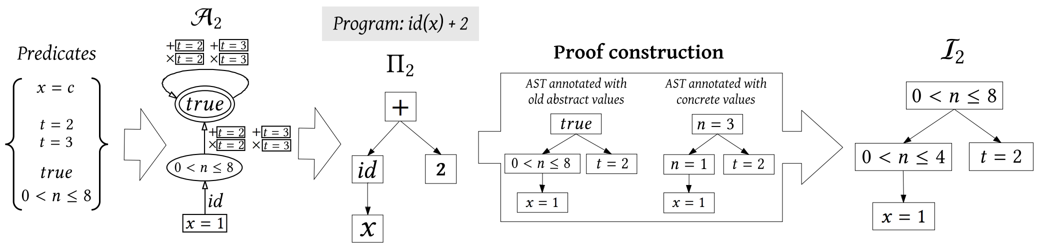
Iteration 1. As explained in Example 3.1.2, the initial AFTA $\mathcal{A}_1$ constructed by our algorithm accepts all DSL programs starting with $id(x)$. Hence, in the first iteration, we obtain the program $\Pi_1 = id(x)$ as a candidate solution. Since Π_1 does not satisfy the example $1 \mapsto 9$, we construct an incorrectness proof $\mathcal{I}_1$, which introduces a new abstract value $0 < n \leq 8$ in our set of predicates.

Iteration 2. During the second iteration, we construct the AFTA labeled as $\mathcal{A}_2$ in Figure 3.10, which contains a new state $0 < n \leq 8$. While $\mathcal{A}_2$ no longer accepts the program $id(x)$, it does accept the spurious program $\Pi_2 = id(x)+2$, which is returned by the ranking algorithm. Then we construct the incorrectness proof for Π_2, and we obtain a new predicate $0 < n \leq 4$.

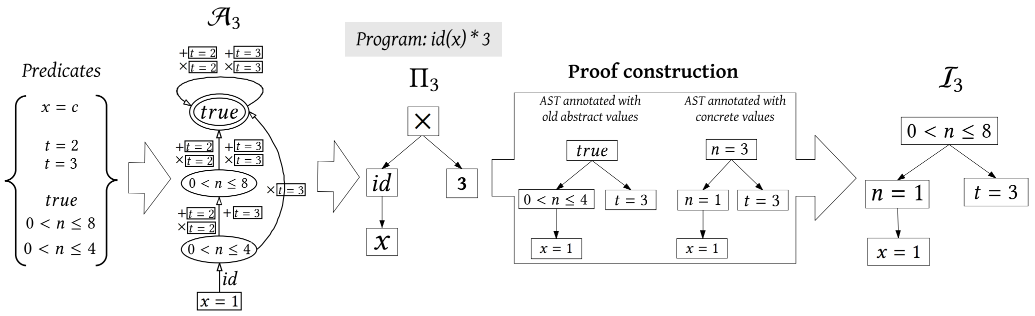
Iteration 3. In the next iteration, we construct the AFTA labeled as $\mathcal{A}_3$.



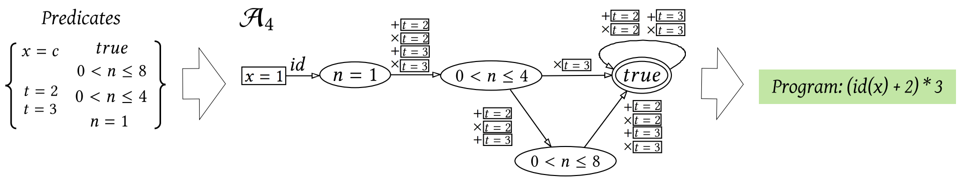
Iteration 1: Incorrectness proof $\mathcal{I}_1$ for spurious program Π_1.



Iteration 2: Incorrectness proof $\mathcal{I}_2$ for spurious program Π_2.



Iteration 3: Incorrectness proof $\mathcal{I}_3$ for spurious program Π_3.



Iteration 4: Rank returns a desired program.

Figure 3.10: An end-to-end working example.

Observe that $\mathcal{A}_3$ no longer accepts the spurious program Π_2 and also rules out two other programs, namely $id(x) + 3$ and $id(x) \times 2$. RANK now returns the program $\Pi_3 = id(x) \times 3$, which is again spurious. After constructing the incorrectness proof of Π_3, we now obtain a new predicate $n = 1$.

Iteration 4. In the final iteration, we construct the AFTA labeled as $\mathcal{A}_4$, which rules out all programs containg a single operator ($+$ or $\times$) as well as 12 programs that use two operators. When we run the ranking algorithm on $\mathcal{A}_4$, we obtain the candidate program $(id(x) + 2) \times 3$, which is indeed consistent with the example $1 \mapsto 9$. Thus, the synthesis algorithm terminates with this solution.

Discussion. As this example illustrates, our approach explores far fewer programs compared to enumeration-based techniques. For instance, our algorithm only tested *four* candidate programs against the input-output examples, whereas an enumeration-based approach would need to explore 24 programs. However, since each candidate program is generated using abstract finite tree automata, each iteration has a higher overhead. In contrast, the CFTA-based approach discussed in Chapter 2.2 *always* explores a single program, but the corresponding finite tree automaton may be *very* large. Thus, our technique can be seen as providing a useful tuning knob between enumeration-based synthesis algorithms and representation-based techniques (*e.g.*, CFTAs and *version space algebras*) that construct a data structure representing all programs consistent with the input-output examples.

3.4 Implementation

The Blaze implementation now is further parametrized with the DSL's abstract semantics (in the form of a universe of predicates and a set of abstract transformers). Blaze now takes as input a DSL with its syntax and abstract semantics as well as a set of input-output examples. Its implementation consists of three main modules: AFTA construction, ranking, and incorrectness proof generation. The AFTA construction implementation reuses the CFTA construction procedure except that now we use the DSL's abstract semantics. The ranking algorithm completely follows the implementation in Chapter 2.3. Finally, our implementation of the incorrectness proof generation follows our technical presentation. Therefore, we do not discuss more details here.

3.5 Applications

Now, we describe two instantiations of the Blaze framework in two different application domains, namely, string processing and tensor reshaping. In particular, to instantiate the Blaze framework for a specific domain, the domain expert needs to provide a (cost-annotated) domain-specific language, a universe of possible predicates to be used in the abstraction, the abstract semantics of each DSL construct, and optionally an initial abstraction to use when constructing the initial AFTA.

$$
\begin{array}{rrcl}
\text{String expr} & e & := & \texttt{Str}(f) \mid \texttt{Concat}(f, e); \\
\text{Substring expr} & f & := & \texttt{ConstStr}(s) \mid \texttt{SubStr}(x, p_1, p_2); \\
\text{Position} & p & := & \texttt{Pos}(x, \tau, k, d) \mid \texttt{ConstPos}(k); \\
\text{Direction} & d & := & \texttt{Start} \mid \texttt{End};
\end{array}
$$

Figure 3.11: String processing DSL.

3.5.1 String Processing

We now describe our instantiation of the Blaze framework for synthesizing string processing programs.

Domain-specific language. Since there is significant prior work on automating string processing using PBE [21, 56, 50], we directly adopt the DSL presented by [56] as shown in Figure 3.11. This DSL essentially allows concatenating substrings of the input string x, where each substring is extracted using a start position p_1 and an end position p_2. A position can either be a constant index ($\texttt{ConstPos}(k)$) or the (start or end) index of the k'th occurrence of the match of token τ in the input string ($\texttt{Pos}(x, \tau, k, d)$).

Universe of predicates. A natural abstraction when reasoning about strings is to consider their length; hence, our universe of predicates in this domain includes predicates of the form $len(s) = i$, where s is a symbol of type string and i represents any integer. We also consider predicates of the form $s[i] = c$ indicating that the i'th character in string s is c. Finally, recall from Chapter 3.1.1 that our universe must include (1) predicates of the form $s = c$, where

$$\begin{array}{rcl}
[\![f(s_1 = c_1, \cdots, s_n = c_n)]\!]^\sharp & := & \big(s = [\![f(c_1, \cdots, c_n)]\!]\big) \\
[\![\texttt{Concat}(\mathit{len}(f) = i_1, \mathit{len}(e) = i_2)]\!]^\sharp & := & \big(\mathit{len}(e) = (i_1 + i_2)\big) \\
[\![\texttt{Concat}(\mathit{len}(f) = i_1, e[i_2] = c)]\!]^\sharp & := & \big(e[i_1 + i_2] = c\big) \\
[\![\texttt{Concat}(\mathit{len}(f) = i, e = c)]\!]^\sharp & := & \big(\mathit{len}(e) = (i + \mathit{len}(c)) \land \bigwedge_{j=0,\cdots,\mathit{len}(c)-1} e[i+j] = c[j]\big) \\
[\![\texttt{Concat}(f[i] = c, p)]\!]^\sharp & := & \big(e[i] = c\big) \\
[\![\texttt{Concat}(f = c, \mathit{len}(e) = i)]\!]^\sharp & := & \big(\mathit{len}(e) = (\mathit{len}(c) + i) \land \bigwedge_{j=0,\cdots,\mathit{len}(c)-1} e[j] = c[j]\big) \\
[\![\texttt{Concat}(f = c_1, e[i] = c_2)]\!]^\sharp & := & \big(e[\mathit{len}(c_1) + i] = c_2 \land \bigwedge_{j=0,\cdots,\mathit{len}(c_1)-1} e[j] = c_1[j]\big) \\
[\![\texttt{Str}(p)]\!]^\sharp & := & p
\end{array}$$

Figure 3.12: Abstract transformers for string processing DSL.

c is a concrete value that symbol s can take, and (2) the predicate *true*. Hence, our universe of predicates for the string domain is given by:

$$\begin{array}{rl}
\mathcal{U} = & \big\{\mathit{len}(s) = i \mid i \in \mathbb{N}\big\} \cup \big\{s[i] = c \mid i \in \mathbb{N}, c \in \texttt{Char}]\big\} \\
& \cup\big\{s = c \mid c \in \texttt{Type}(s)\big\} \cup \ \big\{\mathit{true}\big\}
\end{array}$$

Abstract transformers. The domain expert must also provide an abstract transformer $[\![f(\varphi_1, \cdots, \varphi_n)]\!]^\sharp$ for each grammar production $s \to f(s_1, \cdots, s_n)$ and abstract values $\varphi_1, \cdots, \varphi_n$. Since our universe of predicates can be viewed as the union of three different abstract domains for reasoning string length, character position, and string equality, our abstract transformers effectively define the reduced product of these abstract domains. In particular, we define a generic transformer for conjunctions of predicates as follows:

$$f\big((\bigwedge_{i_1} p_{i_1}), \cdots, (\bigwedge_{i_n} p_{i_n})\big) := \prod_{i_1} \cdots \prod_{i_n} f(p_{i_1}, \cdots, p_{i_n})$$

Therefore, instead of defining a transformer for every possible abstract value (with arbitrarily many conjuncts), it suffices to define an abstract transformer for every combination of *atomic* predicates (shown in Figure 3.12).

Initial abstraction. Our initial abstraction includes predicates of the form $len(s) = i$, where s is a symbol of type string and i is an integer, as well as the predicate *true*.

3.5.2 Tensor Reshaping

Motivated by the abundance of questions on how to perform various matrix and tensor transformations in MATLAB, we also use the BLAZE framework to synthesize tensor manipulation programs.[4] We believe this application domain is a good stress test for the BLAZE framework because (1) tensors are complex data structures which make the search space larger, and (2) the input-output examples in this domain are typically much larger in size. Finally, we wish to show that the BLAZE framework can be immediately used to generate a practical synthesis tool for a new unexplored domain.

Domain-specific language. Our DSL for the tensor reshaping is inspired by existing MATLAB functions and is shown in Figure 3.13. In this DSL, tensor operators include `Reshape`, `Permute`, `Fliplr`, and `Flipud` and correspond to their namesakes in MATLAB[5]. For example, `Reshape`(t, v) takes a tensor t and a size vector v and reshapes t so that its dimension becomes v. Similarly, `Permute`(t, v) rearranges the dimensions of tensor t so that they are in the order specified by vector v. Next, `Fliplr(`t`)` returns tensor t with its columns

[4]Tensors generalize matrices from 2 dimensions to an arbitrary number of dimensions.

[5]See the MATLAB documentation https://www.mathworks.com/help/matlab/ref/x.html where x refers to the name of the corresponding function.

$$\begin{array}{llcl} \text{Tensor expr} & t & := & \texttt{id}(x) \mid \texttt{Reshape}(t, v) \mid \texttt{Permute}(t, v) \mid \texttt{Fliplr}(t) \mid \texttt{Flipud}(t); \\ \text{Vector expr} & v & := & [k_1, k_2] \mid \texttt{Cons}(k, v); \end{array}$$

Figure 3.13: Tensor reshaping DSL.

flipped in the left-right direction, and $\texttt{Flipud}(t)$ returns tensor t with its rows flipped in the up-down direction. Vector expressions are constructed recursively using the $\texttt{Cons}(k, v)$ construct, which yields a vector with first element k (an integer), followed by elements in vector v.

Example 3.5.1. Suppose that we have a vector v and we would like to reshape it in a row-wise manner so that it yields a matrix with 2 rows and 3 columns[6]. For example, if the input vector is $[1, 2, 3, 4, 5, 6]$, then we should obtain the matrix $[1, 2, 3; 4, 5, 6]$ where the semi-colon indicates a new row. This transformation can be expressed by the DSL program $\texttt{Permute}(\texttt{Reshape}(v, [3, 2]), [2, 1])$.

Universe of predicates. Similar to the strings, a natural abstraction for vectors is to consider their length. Therefore, our universe includes predicates of the form $\mathit{len}(v) = i$, indicating that vector v has length i. In the case of tensors, our abstraction keeps track of the number of elements and number of dimensions of the tensors. In particular, the predicate $\mathit{numDims}(t) = i$ indicates that t is an i-dimensional tensor. Similarly, the predicate $\mathit{numElems}(t) = i$ indicates that tensor t contains a total of i entries. Therefore, the universe of

[6]StackOverflow link: https://stackoverflow.com/questions/16592386/reshape-matlab-vector-in-row-wise-manner.

$$
\begin{array}{rcl}
[\![f(s_1 = c_1, \cdots, s_n = c_n)]\!]^\sharp & := & \big(s = [\![f(c_1, \cdots, c_n)]\!]\big) \\
[\![\texttt{Cons}(k = i_1, \mathit{len}(v) = i_2)]\!]^\sharp & := & \big(\mathit{len}(v) = (i_2 + 1)\big) \\
[\![\texttt{Permute}(\mathit{numDims}(t) = i, p)]\!]^\sharp & := & \big(\mathit{numDims}(t) = i\big) \\
[\![\texttt{Permute}(\mathit{numElems}(t) = i, p)]\!]^\sharp & := & \big(\mathit{numElems}(t) = i\big) \\
[\![\texttt{Reshape}(\mathit{numDims}(t) = i_1, \mathit{len}(v) = i_2)]\!]^\sharp & := & \big(\mathit{numDims}(t) = i_2\big) \\
[\![\texttt{Reshape}(\mathit{numDims}(t) = i, v = c)]\!]^\sharp & := & \big(\mathit{numDims}(t) = \mathit{len}(c)\big) \\
[\![\texttt{Reshape}(\mathit{numElems}(t) = i, p)]\!]^\sharp & := & \big(\mathit{numElems}(t) = i\big) \\
[\![\texttt{Reshape}(t = c, \mathit{len}(v) = i)]\!]^\sharp & := & \big(\mathit{numElems}(t) = \mathit{numElems}(c)\big) \\
[\![\texttt{Flipud}(p)]\!]^\sharp & := & p \\
[\![\texttt{Fliplr}(p)]\!]^\sharp & := & p
\end{array}
$$

Figure 3.14: Abstract transformers for tensor reshaping DSL.

predicates is given by:

$$
\mathcal{U} = \begin{array}{l} \big\{\mathit{numDims}(t) = i \mid i \in \mathbb{N}\big\} \cup \big\{\mathit{numElems}(t) = i \mid i \in \mathbb{N}\big\} \\ \cup\big\{\mathit{len}(v) = i \mid i \in \mathbb{N}\big\} \cup \big\{s = c \mid c \in \texttt{Type}(s)\big\} \ \cup \ \big\{\mathit{true}\big\} \end{array}
$$

Abstract transformers. The abstract transformers for all combinations of atomic predicates for each DSL construct are given in Figure 3.14. As in the string domain, we define a generic transformer for conjunctions of predicates as follows:

$$
f\big((\bigwedge_{i_1} p_{i_1}), \cdots, (\bigwedge_{i_n} p_{i_n})\big) := \bigsqcap_{i_1} \cdots \bigsqcap_{i_n} f(p_{i_1}, \cdots, p_{i_n})
$$

Initial abstraction. Our initial abstraction includes only *true*.

3.6 Evaluation

We evaluate Blaze by using it to automate string and tensor manipulation tasks collected from online forums and standard data sets. The goal of our evaluation is to answer the following questions:

- **Q1:** How does Blaze perform on synthesis tasks across domains?
- **Q2:** How does Blaze compare with existing synthesis techniques?
- **Q3:** How many refinement steps does Blaze take to converge?
- **Q4:** What is the benefit of abstraction refinement in practice?

3.6.1 String Processing

We evaluate Blaze on *all* 108 string processing benchmarks from the PBE track of the SyGuS competition [3]. We believe string precessing is a good testbed because of the existence of mature tools like FlashFill [21] and the presence of a SyGuS benchmark suite for string transformations.

Benchmark information. Among the 108 string processing benchmarks, the number of examples range from 4 to 400, with an average of 78.2 and a median of 14. The average input example string length is 13.6 and the median is 13.0. The maximum (resp. minimum) string length is 54 (resp. 8).

Experimental setup. We instantiate Blaze using the string processing DSL shown in Figure 3.11 and the predicates and abstract transformers from Chapter 3.5.1. For each benchmark, we provide Blaze with all input-output examples at the same time.[7] We also compare Blaze with the following existing synthesis techniques:

- **FlashFill:** This tool is the state-of-the-art synthesizer for automating string manipulation tasks and is shipped in Microsoft PowerShell as the “convert-string” commandlet. It propagates examples backwards using the inverse semantics of DSL operators, and adopts the version space algebra data structure to compactly represent the search space.
- **ENUM-EQ:** This technique is based on enumerative search and has been adopted to solve different kinds of synthesis problems [2, 68, 11, 3]. It enumerates programs according to their size, groups them into equivalence classes based on their (concrete) input-output behavior, and returns the first program that is consistent with the examples.
- **CFTA:** This is an implementation of the synthesis algorithm presented in Chapter 2. It uses the concrete semantics of the DSL operators to construct a CFTA whose language is exactly the set of programs that are consistent with the specification according to the concrete semantics.

To allow a fair comparison, we evaluate ENUM-EQ and CFTA using the same DSL and ranking heuristics that we use to evaluate Blaze. For FlashFill, we use the “convert-string” commandlet from Microsoft Powershell.

[7]Blaze typically uses a fraction of these examples during abstraction refinement.

Because the baseline techniques mentioned above perform *much better* when the examples are provided in an interactive fashion [8], we evaluate them in the following way: Given a set of examples E for each benchmark, we first sample an example e in E, use each technique to synthesize a program P that satisfies e, and check if P satisfies all examples in E. If not, we sample another example e' in E for which P does not produce the desired output, and repeat the synthesis process using both e and e'. The synthesizer terminates when it either successfully learns a program that satisfies all examples, proves that no program in the DSL satisfies the examples, or times out in 10 minutes.

Blaze *results* Figure 3.15 summarizes the results of our evaluation of Blaze for string processing. [9] Because it is not feasible to give statistics for all 108 SyGuS benchmarks, we only show the detailed results for one benchmark from each of the 27 categories. Note that the four benchmarks within a category are very similar and only differ in the number of provided examples. The main take-away message from our evaluation is that Blaze can successfully solve 70% of the benchmarks in under a second, and 85% of the benchmarks in un-

[8]Because Blaze is not very sensitive to the number of examples, we used Blaze in a non-interactive mode by providing all examples at once. Since the baseline tools do not scale as well in the number of examples, we used them in an interactive mode, with the goal of casting them in the best light possible.

[9]In Figure 3.15, $|\vec{e}|$ denotes the total number of examples available in each benchmark and T_{syn} denotes the synthesis time (in seconds). The next columns labeled T_x denote the times for FTA construction, ranking, proof construction, and all remaining parts (*e.g.*, FTA minimization). *#Iters* gives the number of refinement steps, and $|Q_{\mathit{final}}|$ and $|\Delta_{\mathit{final}}|$ denotes the number of states and transitions in the AFTA in the last iteration. The last column $|\Pi_{syn}|$ shows the size of the synthesized program (measured by the number of AST nodes).

Benchmark	$\lvert\vec{e}\rvert$	T_{syn} (sec)	$T_{\mathcal{A}}$	T_{rank}	$T_{\mathcal{I}}$	T_{other}	*#Iters*	$\lvert Q_{final}\rvert$	$\lvert\Delta_{final}\rvert$	$\lvert\Pi_{syn}\rvert$
bikes	6	0.05	0.05	0.00	0.00	0.00	1	52	135	13
dr-name	4	0.16	0.09	0.02	0.01	0.04	17	95	513	19
firstname	4	0.08	0.08	0.00	0.00	0.00	1	71	350	13
initials	4	0.11	0.09	0.00	0.01	0.01	14	68	209	32
lastname	4	0.10	0.10	0.00	0.00	0.00	3	79	450	13
name-combine-2	4	0.20	0.12	0.02	0.01	0.05	45	101	549	32
name-combine-3	6	0.16	0.10	0.01	0.02	0.03	26	80	305	32
name-combine-4	5	0.30	0.14	0.03	0.05	0.08	62	114	725	35
name-combine	6	0.16	0.10	0.02	0.02	0.02	20	87	427	29
phone-1	6	0.07	0.07	0.00	0.00	0.00	2	43	79	13
phone-10	7	1.99	0.69	0.34	0.30	0.66	539	471	4754	48
phone-2	6	0.06	0.06	0.00	0.00	0.00	3	43	77	13
phone-3	7	0.25	0.12	0.03	0.05	0.05	59	88	355	35
phone-4	6	0.23	0.10	0.03	0.04	0.06	63	155	1256	45
phone-5	7	0.08	0.08	0.00	0.00	0.00	1	53	114	13
phone-6	7	0.10	0.10	0.00	0.00	0.00	2	53	112	13
phone-7	7	0.08	0.08	0.00	0.00	0.00	3	53	108	13
phone-8	7	0.11	0.11	0.00	0.00	0.00	4	53	106	13
phone-9	7	1.09	0.34	0.19	0.15	0.41	269	454	7355	61
phone	6	0.07	0.07	0.00	0.00	0.00	1	43	80	13
reverse-name	6	0.14	0.08	0.01	0.02	0.03	20	83	414	29
univ_1	6	1.34	0.61	0.21	0.12	0.40	149	348	9618	32
univ_2	6	T/O	—	—	—	—	—	—	—	—
univ_3	6	3.69	1.63	0.57	0.15	1.34	405	467	18960	22
univ_4	8	T/O	—	—	—	—	—	—	—	—
univ_5	8	T/O	—	—	—	—	—	—	—	—
univ_6	8	T/O	—	—	—	—	—	—	—	—
Median	6	0.14	0.10	0.01	0.01	0.02	17	80	355	22
Average	6.1	0.46	0.22	0.06	0.04	0.14	74.3	137.1	2045.7	25.3

Figure 3.15: Blaze results for string processing domain.

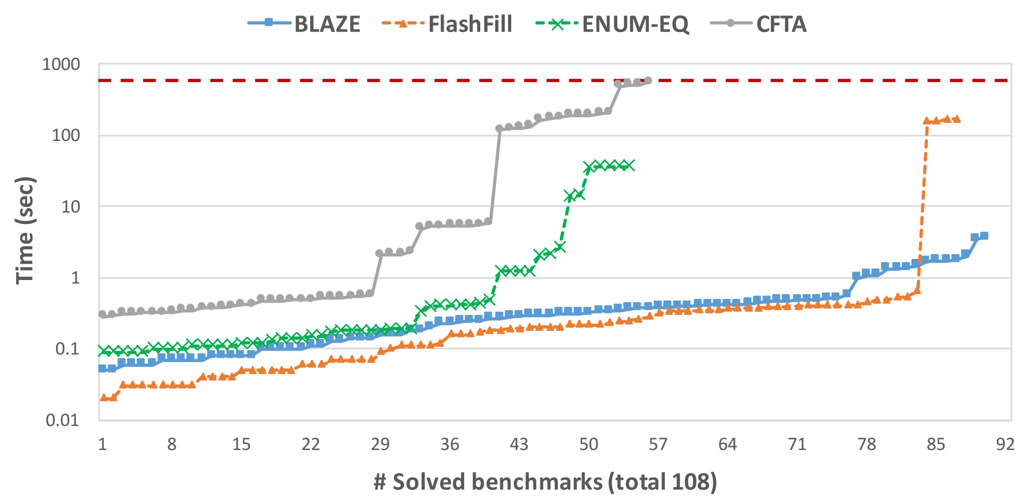
der 4 seconds, with a median running time of 0.14 seconds. In comparison, the best solver, *i.e.*., EUSolver [5], in the SyGuS'16 competition is able to solve in total 45 benchmarks within the timeout of 60 minutes [4].

For most benchmarks, Blaze spends the majority of its running time on FTA construction, whereas the time on proof construction is typically negligible. This is because the number of predicates that are considered in the proof construction phase is usually quite small. It takes Blaze an average of 74 refinement steps before it finds the correct program. However, the median number of refinement steps is much smaller (17). Furthermore, as expected, there is a clear correlation between the number of iterations and total running time. Finally, we can observe that the synthesized programs are non-trival, with an average size of 25 in terms of the number of AST nodes.

Comparison. Figure 3.16 compares the running times of Blaze with FlashFill, ENUM-EQ, and CFTA on *all* 108 SyGuS benchmarks. Overall, Blaze solves the most number of benchmarks (90), with an average running time of 0.49 seconds. Furthermore, any benchmark that can be solved using FlashFill, ENUM-EQ, or CFTA can also be solved by Blaze.

Compared to CFTA, Blaze solves 60% more benchmarks (90 vs. 56) and outperforms CFTA by 363x (in terms of running time) on the 56 benchmarks that can be solved by both techniques. This result demonstrates that abstraction refinement helps scale up the CFTA-based synthesis technique to solve more benchmarks in much less time.

Compared to ENUM-EQ, the improvement is moderate for relatively simple benchmarks. In particular, for the 40 benchmarks that ENUM-EQ can solve in under 1 second, Blaze (only) shows a 1.5x improvement in running time. However, for more complex synthesis tasks, the performance of Blaze is significantly better than ENUM-EQ. For the 54 benchmarks that can be solved by both techniques, we observe a 16x improvement in running time. Furthermore, Blaze can solve 36 benchmarks on which ENUM-EQ times out. We believe this result demonstrates the advantage of using abstract values for search space reduction.



	# Solved	Average time (sec)
Blaze	90	0.49
FlashFill	87	7.66
ENUM-EQ	54	4.25
CFTA	56	73.91

Figure 3.16: Blaze vs. existing techniques for string processing domain.

Finally, Blaze compares favorably with FlashFill, a state-of-the-art technique for automating string processing tasks. In particular, Blaze achieves competitive performance for the benchmarks that both techniques can solve. Furthermore, Blaze solves 3 benchmarks on which FlashFill times out. Since FlashFill is a domain-specific synthesizer that has been crafted specifically for automating string manipulation tasks, we believe these results demonstrate that Blaze can compete with domain-specific state-of-the-art synthesizers.

Outlier analysis. All techniques, including Blaze, time out on 18 benchmarks for the univ_x category. We investigated the cause of failure for these benchmarks and found that the desired program for most of these benchmarks cannot be expressed in the underlying DSL.

3.6.2 Tensor Reshaping

In our second experiment, we evaluate Blaze on tensor reshaping benchmarks collected from online forums. Because tensors are more complicated data structures than strings, the search space in this application tends to be larger on average compared to the string processing application. Furthermore, since automating tensor reshaping is a useful (yet unexplored) application of programming-by-example, we believe this application domain is an interesting target for Blaze.

To perform our evaluation, we collected 39 benchmarks from two online forums, namely StackOverflow and MathWorks.[10] Our benchmarks were collected using the following methodology: We searched for the keyword *"matlab matrix reshape"* and then sorted the results according to their relevance. We then looked at the first 100 posts from each forum and retained posts that contain at least one example and the target program is in one of the responses.

Benchmark information. Since the overwhelming majority of forum entries contain a single example, we only provide one input-output example for each benchmark. The number of entries in the input tensor ranges from 6 to

[10]MathWorks (`https://www.mathworks.com/matlabcentral/answers/`) is a help forum for MATLAB users.

640, with an average of 73.5 and a median of 36. Among all benchmarks, 29 involve reshaping the input example into tensors of dimension great than 2.

Experimental setup. We instantiate Blaze with the DSL shown in Figure 3.13 and the abstract semantics presented in Chapter 3.5.2. Similar to the string processing domain, we also compare Blaze with ENUM-EQ and CFTA. However, since there is no existing domain-specific synthesizer for automating tensor reshaping tasks, we implemented a specialized VSA-based synthesizer for our matrix domain by instantiating the Prose framework [50]. In particular, to instantiate Prose, we provide precise witness functions (inverse semantics) for all the operators in our DSL. To allow a fair comparison, we use the same DSL for all the synthesizers, as well as the same ranking heuristics. We also experiment with all baseline synthesizers in the interactive setting, as we did for the string processing domain. The timeout is set to be 10 minutes.

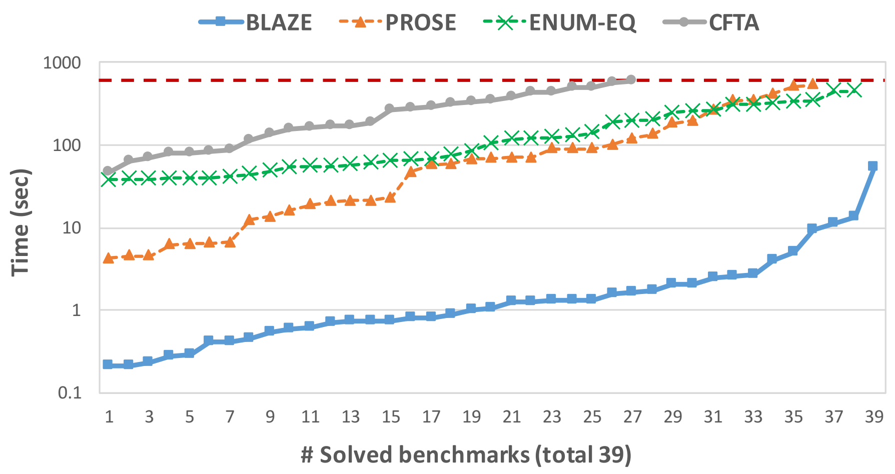
Blaze *results.* The results of our evaluation on Blaze are summarized in Figure 3.17. As we can see, Blaze can successfully solve all benchmarks with an average (resp. median) synthesis time of 3.35 (resp. 1.07) seconds. Furthermore, Blaze can solve 46% of the benchmarks in under 1 second, and 87% of the benchmarks in under 5 seconds. These results demonstrate that Blaze is also practical for automating tensor reshaping tasks.

Looking at Figure 3.17 in more detail, Blaze takes an average of 165 refinement steps to synthesize a correct program. Unlike in the string processing domain where Blaze spends most of its time in FTA construction, proof

Benchmark	T_{syn} (sec)	$T_{\mathcal{A}}$	T_{rank}	$T_{\mathcal{I}}$	T_{other}	*#Iters*	$\lvert Q_{final}\rvert$	$\lvert \Delta_{final}\rvert$	$\lvert \Pi_{syn}\rvert$
stackoverflow-1	0.29	0.14	0.02	0.08	0.05	39	125	993	10
stackoverflow-2	2.74	0.86	0.10	1.52	0.26	319	279	4483	22
stackoverflow-3	0.72	0.20	0.03	0.43	0.06	57	143	1334	14
stackoverflow-4	13.32	0.31	0.04	12.89	0.08	166	165	959	22
stackoverflow-5	1.34	0.57	0.08	0.48	0.21	222	236	2595	18
stackoverflow-6	0.42	0.17	0.02	0.17	0.06	48	129	1012	10
stackoverflow-7	2.04	0.59	0.07	1.20	0.18	217	244	2607	18
stackoverflow-8	2.04	0.83	0.08	0.90	0.23	288	280	3447	18
stackoverflow-9	1.67	0.90	0.08	0.44	0.25	114	374	5389	16
stackoverflow-10	0.23	0.12	0.01	0.06	0.04	28	114	715	10
stackoverflow-11	0.74	0.34	0.05	0.24	0.11	106	155	1004	18
stackoverflow-12	0.82	0.12	0.02	0.63	0.05	38	124	929	10
stackoverflow-13	0.59	0.17	0.02	0.34	0.06	49	143	1227	12
stackoverflow-14	52.94	1.36	0.11	51.24	0.23	385	324	4321	22
stackoverflow-15	0.41	0.12	0.01	0.24	0.04	31	121	611	14
stackoverflow-16	5.02	0.38	0.06	4.45	0.13	228	172	1083	22
stackoverflow-17	2.54	0.79	0.09	1.42	0.24	319	279	4483	22
stackoverflow-18	0.54	0.25	0.03	0.18	0.08	65	144	1201	14
stackoverflow-19	0.73	0.36	0.06	0.17	0.14	142	162	1180	18
stackoverflow-20	1.31	0.36	0.05	0.78	0.12	165	160	786	18
stackoverflow-21	1.01	0.52	0.06	0.27	0.16	180	195	1566	18
stackoverflow-22	0.21	0.10	0.01	0.07	0.03	19	106	526	10
stackoverflow-23	1.24	0.26	0.04	0.85	0.09	108	181	2493	14
stackoverflow-24	0.62	0.14	0.02	0.41	0.05	52	138	1183	12
stackoverflow-25	0.81	0.20	0.03	0.51	0.07	72	170	2201	14
mathworks-1	0.71	0.15	0.02	0.48	0.06	55	137	1103	12
mathworks-2	0.88	0.11	0.02	0.71	0.04	34	126	848	14
mathworks-3	1.07	0.58	0.06	0.27	0.16	180	195	1566	18
mathworks-4	3.94	0.22	0.03	3.62	0.07	89	195	2589	14
mathworks-5	0.45	0.15	0.02	0.22	0.06	45	134	963	12
mathworks-6	1.30	0.42	0.07	0.63	0.18	195	222	2100	18
mathworks-7	0.21	0.10	0.01	0.06	0.04	28	116	717	10
mathworks-8	0.27	0.13	0.02	0.07	0.05	39	125	993	10
mathworks-9	1.73	0.23	0.03	1.39	0.08	104	160	955	10
mathworks-10	1.57	0.30	0.05	1.10	0.12	145	172	1176	14
mathworks-11	9.40	5.72	0.50	1.83	1.35	613	583	25924	22
mathworks-12	1.25	0.36	0.07	0.66	0.16	187	203	1799	18
mathworks-13	2.49	1.45	0.17	0.41	0.46	462	295	2574	15
mathworks-14	11.10	6.18	1.19	0.60	3.13	827	678	34176	22
Median	1.07	0.30	0.04	0.48	0.09	108	165	1201	14
Average	3.35	0.67	0.09	2.36	0.23	165.6	205.2	3225.9	15.5

Figure 3.17: Blaze results for tensor reshaping domain.

construction now seems to also take significant time. We conjecture this is because for tensor reshaping tasks Blaze needs to search for predicates in a large space during proof construction. The final AFTA contains an average of 205 states, and the synthesized program has 16 AST nodes on average.



	# Solved	Average time (sec)
Blaze	39	3.35
Prose	36	113.13
ENUM-EQ	38	147.88
CFTA	27	252.80

Figure 3.18: Blaze vs. existing techniques for tensor reshaping domain.

Comparison. As shown in Figure 3.18, Blaze significantly outperforms all existing techniques, both in terms of the number of solved benchmarks as well as the running time. In particular, we observe a 262x improvement over CFTA, 115x improvement over ENUM-EQ, and 90x improvement over Prose in terms of the running time. This experiment also demonstrates the advantage of using abstractions and abstraction refinement for synthesizing tensor reshaping programs.

Outlier analysis. The benchmark "stackoverflow-14" takes 53 seconds because the input example tensor is the largest one we have in our benchmark suite (with 640 entries). As a result, in the proof construction phase Blaze needs to search for the desired formula in a space with over 10^5 formulas. This makes the overall synthesis process computationally expensive.

3.6.3 Discussion

The reader may wonder why Blaze performs much better in the tensor reshaping domain compared to VSA-based techniques (FlashFill and Prose) than in the string processing domain. We conjecture that this discrepancy can be explained by considering the size of the search space measured in terms of the number of (intermediate) concrete values produced by the DSL programs. For the string domain, the search space size is dominated by the number of substrings, and FlashFill constructs n^2 nodes for substrings in the VSA data structure, where n is the length of the output example. For the tensor domain, the search space size is mostly determined by the number of intermediate tensors; in the worst case Prose would have to explore $\mathcal{O}(n!)$ nodes, where n is the number of entries in the example tensors. Hence, the size of the search space in the tensor domain is potentially much larger for VSA-based techniques than that in the string domain. In contrast, Blaze performs quite well in both application domains, since it uses abstract values (instead of concrete values) to represent equivalence classes.

Chapter 4

Learning Abstractions for Program Synthesis

So far, we have seen an abstraction-guided synthesis paradigm. While this paradigm has proven to be quite powerful, a down-side of such techniques is that they require a domain expert to manually come up with a suitable abstraction. For instance, the Blaze synthesis framework expects a domain expert to manually specify an abstraction in the form of a universe of predicate templates together with sound abstract transformers for every DSL function. Unfortunately, this process is not only time-consuming but also requires significant insight about the application domain *as well as* the internal workings of the synthesizer.

In this chapter, we present a novel technique for automatically learning domain-specific abstractions that are useful for instantiating an example-guided synthesis framework in a new domain.

4.1 Overview

Given a DSL and a training set of synthesis problems (*i.e.*, input-output examples), our method learns an abstraction in the form of predicate templates and infers sound abstract transformers for each DSL construct. In addition to

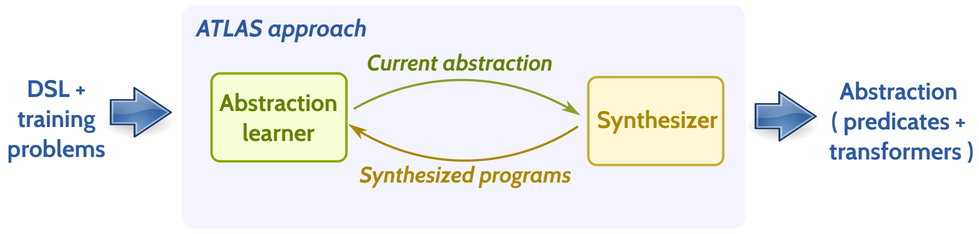


Figure 4.1: Schematic overview of our abstraction learning technique.

eliminating the significant manual effort required from a domain expert, the abstractions learned by our method often outperform manually-crafted ones in terms of their benefit to synthesizer performance.

The workflow of our approach, henceforth called ATLAS [1], is shown schematically in Figure 4.1. Since ATLAS is meant to be used as an *off-line* training step for a general-purpose PBE system, it takes as input a DSL as well as a set of synthesis problems $\vec{\mathcal{E}}$ that can be used for training purposes. Given these inputs, our method enters a refinement loop where an *Abstraction Learner* component discovers a sequence of increasingly precise abstract domains $\mathcal{A}_1, \cdots, \mathcal{A}_n$, and their corresponding abstract transformers $\mathcal{T}_1, \cdots, \mathcal{T}_n$, in order to help the *Abstraction-Guided Synthesizer* (AGS) solve all training problems. While the AGS can reject many incorrect solutions using an abstract domain $\mathcal{A}_i$, it might still return an incorrect solution due to the insufficiency of $\mathcal{A}_i$. Thus, whenever the AGS returns an incorrect solution to any training problem, the Abstraction Learner discovers a more precise abstract domain and automatically synthesizes the corresponding abstract transformers. Upon

[1]ATLAS stands for AuTomated Learning of AbStractions.

termination of our algorithm, the final abstract domain $\mathcal{A}_n$ and transformers $\mathcal{T}_n$ are sufficient for the AGS to correctly solve *all* training problems. Furthermore, because our method learns *general* abstractions in the form of predicate *templates*, the learned abstractions are expected to be useful for solving many *other* synthesis problems beyond those in the training set.

From a technical perspective, the Abstraction Learner is based on two key ideas, namely *tree interpolation* and *data-driven constraint solving*, for learning useful abstract domains and transformers respectively. Specifically, given an incorrect program Π that cannot be refuted by the AGS using the current abstract domain $\mathcal{A}_i$, the Abstraction Learner generates a tree interpolant $\mathcal{I}_i$ that serves as a proof of Π's incorrectness and constructs a new abstract domain $\mathcal{A}_{i+1}$ by extracting templates from the predicates used in $\mathcal{I}_i$. The Abstraction Learner also synthesizes the corresponding abstract transformers for $\mathcal{A}_{i+1}$ by setting up a *second-order constraint solving* problem where the goal is to find the unknown relationship between symbolic constants used in the predicate templates. Our method solves this problem in a data-driven way by sampling input-output examples for DSL operators and ultimately reduces the transformer learning problem to solving a system of linear equations.

4.2 An Illustrative Example

Suppose that we wish to use the Blaze framework to automate the class of string processing tasks considered by FlashFill [21] and BlinkFill [56]. In the Blaze framework, a domain expert needs to come up with a universe of suitable predicate templates as well as abstract transformers for each DSL construct. We will now illustrate how Atlas automates this process, given a suitable DSL and its semantics (*e.g.*, the one used in [56]).

In order to use Atlas, one needs to provide a set of synthesis problems $\vec{\mathcal{E}}$ (*i.e.*, input-output examples) that will be used in the training process. Specifically, let us consider the three synthesis problems given below:

$$\vec{\mathcal{E}} = \left\{ \begin{array}{lcl} \mathcal{E}_1 & : & \{ \text{“}\texttt{CAV}\text{”} \mapsto \text{“}\texttt{CAV2018}\text{”}, \text{“}\texttt{SAS}\text{”} \mapsto \text{“}\texttt{SAS2018}\text{”}, \text{“}\texttt{FSE}\text{”} \mapsto \text{“}\texttt{FSE2018}\text{”} \}, \\ \mathcal{E}_2 & : & \{ \text{“}\texttt{510.220.5586}\text{”} \mapsto \text{“}\texttt{510-220-5586}\text{”} \}, \\ \mathcal{E}_3 & : & \left\{ \begin{array}{l} \text{“}\texttt{\textbackslash Company\textbackslash Code\textbackslash index.html}\text{”} \mapsto \text{“}\texttt{\textbackslash Company\textbackslash Code\textbackslash}\text{”}, \\ \text{“}\texttt{\textbackslash Company\textbackslash Docs\textbackslash Spec\textbackslash specs.html}\text{”} \mapsto \text{“}\texttt{\textbackslash Company\textbackslash Docs\textbackslash Spec\textbackslash}\text{”} \end{array} \right\} \end{array} \right\}.$$

In order to construct the abstract domain $\mathcal{A}$ and transformers $\mathcal{T}$, Atlas starts with the trivial abstract domain $\mathcal{A}_0 = \{\top\}$ and transformers $\mathcal{T}_0$, defined as $[\![F(\top, \cdots, \top)]\!]^\sharp := \top$ for every DSL construct F. Using this abstraction, Atlas invokes Blaze to find a program Π_0 that satisfies specification $\mathcal{E}_1$ under the current abstraction $(\mathcal{A}_0, \mathcal{T}_0)$. However, since the program Π_0 returned by Blaze is incorrect with respect to the concrete semantics, Atlas tries to find a more precise abstraction that allows Blaze to succeed.

Towards this goal, Atlas enters a refinement loop that culminates in the discovery of the abstract domain $\mathcal{A}_1 = \{\top, \mathit{len}(\boxed{\alpha}) = \mathtt{c}, \mathit{len}(\boxed{\alpha}) \neq \mathtt{c}\}$, where $\boxed{\alpha}$ denotes a variable and $\mathtt{c}$ is an integer constant. In other words, $\mathcal{A}_1$ tracks equality and inequality constraints on the length of strings. After learning these predicate templates, Atlas also synthesizes the corresponding abstract transformers $\mathcal{T}_1$. In particular, for each DSL construct, Atlas learns one abstract transformer for each combination of predicate templates in $\mathcal{A}_1$.

For instance, for the `Concat` operator which returns the concatenation y of two strings x_1, x_2, Atlas synthesizes the following abstract transformers, where $\star$ denotes any predicate:

$$\mathcal{T}_1 = \left\{ \begin{array}{rcl} [\![\texttt{Concat}(\top, \star)]\!]^\sharp & := & \top \\ [\![\texttt{Concat}(\star, \top)]\!]^\sharp & := & \top \\ [\![\texttt{Concat}\big(\mathit{len}(x_1) \neq \texttt{c}_1, \mathit{len}(x_2) \neq \texttt{c}_2\big)]\!]^\sharp & := & \top \\ [\![\texttt{Concat}\big(\mathit{len}(x_1) = \texttt{c}_1, \mathit{len}(x_2) = \texttt{c}_2\big)]\!]^\sharp & := & \big(\mathit{len}(y) = \texttt{c}_1 + \texttt{c}_2\big) \\ [\![\texttt{Concat}\big(\mathit{len}(x_1) = \texttt{c}_1, \mathit{len}(x_2) \neq \texttt{c}_2\big)]\!]^\sharp & := & \big(\mathit{len}(y) \neq \texttt{c}_1 + \texttt{c}_2\big) \\ [\![\texttt{Concat}\big(\mathit{len}(x_1) \neq \texttt{c}_1, \mathit{len}(x_2) = \texttt{c}_2\big)]\!]^\sharp & := & \big(\mathit{len}(y) \neq \texttt{c}_1 + \texttt{c}_2\big) \end{array} \right\}.$$

Since the AGS can successfully solve $\mathcal{E}_1$ using $(\mathcal{A}_1, \mathcal{T}_1)$, Atlas now moves on to the next training problem.

For synthesis problem $\mathcal{E}_2$, the current abstraction $(\mathcal{A}_1, \mathcal{T}_1)$ is *not* sufficient for Blaze to discover the correct program. After processing $\mathcal{E}_2$, Atlas refines the abstract domain to the following set of predicate templates:

$$\mathcal{A}_2 = \left\{ \top, \mathit{len}(\boxed{\alpha}) = \texttt{c}, \mathit{len}(\boxed{\alpha}) \neq \texttt{c}, \mathit{charAt}(\boxed{\alpha}, \texttt{i}) = \texttt{c}, \mathit{charAt}(\boxed{\alpha}, \texttt{i}) \neq \texttt{c} \right\}.$$

Observe that Atlas has discovered two additional predicate templates that track positions of characters in the string. Atlas also learns the corresponding abstract transformers $\mathcal{T}_2$ for $\mathcal{A}_2$.

Moving on to the final training problem $\mathcal{E}_3$, Blaze can already successfully solve it using $(\mathcal{A}_2, \mathcal{T}_2)$; thus, Atlas terminates with this abstraction.

4.3 Overall Abstraction Learning Algorithm

Our top-level algorithm for learning abstractions, called LearnAbstractions, is shown in Figure 4.2. The algorithm takes two inputs, namely

```
1: procedure $\textsc{LearnAbstractions}(\mathcal{L}, \vec{\mathcal{E}})$
   input: Domain-specific language $\mathcal{L}$ and a set of training problems $\vec{\mathcal{E}}$.
   output: Abstract domain $\mathcal{A}$ and transformers $\mathcal{T}$.
2:     $\mathcal{A} := \{ \top \};$                                                  ▷ Initialization.
3:     $\mathcal{T} := \{ [\![F(\top, \cdots, \top)]\!]^\sharp := \top \mid F \in \mathsf{Constructs}(\mathcal{L}) \};$
4:     for $i := 1, \cdots, |\vec{\mathcal{E}}|$ do
5:         while true do                                                  ▷ Refinement loop.
6:             $\Pi := \mathsf{Synthesize}(\mathcal{L}, \mathcal{E}_i, \mathcal{A}, \mathcal{T});$                  ▷ Invoke AGS.
7:             if $\Pi = \mathit{null}$ then break;
8:             if $\mathsf{IsCorrect}(\Pi, \mathcal{E}_i)$ then break;
9:             $\mathcal{A} := \mathcal{A} \cup \textsc{LearnAbstractDomain}(\Pi, \mathcal{E}_i);$
10:            $\mathcal{T} := \textsc{LearnTransformers}(\mathcal{L}, \mathcal{A});$
11:    return $(\mathcal{A}, \mathcal{T});$
```

Figure 4.2: Overall learning algorithm.

a domain-specific language $\mathcal{L}$ (both syntax and semantics) as well as a set of training problems $\vec{\mathcal{E}}$, where each problem is specified as a *set* of input-output examples $\mathcal{E}_i$. The output of our algorithm is a pair $(\mathcal{A}, \mathcal{T})$, where $\mathcal{A}$ is an abstraction represented by a set of predicate templates and $\mathcal{T}$ is the corresponding abstract transformers.

At a high-level, the LearnAbstractions procedure starts with the most imprecise abstraction $\mathcal{A}$ (just consisting of $\top$) and incrementally improves the its precision whenever the AGS fails to synthesize a correct program using $\mathcal{A}$. Specifically, the outer loop (lines 4–10) considers each training instance $\mathcal{E}_i$ and performs a fixed-point computation (lines 5–10) that terminates when the current abstract domain $\mathcal{A}$ is good enough to solve problem $\mathcal{E}_i$. Thus, upon termination, the learned abstract domain $\mathcal{A}$ is sufficiently precise for the AGS to solve all training problems $\vec{\mathcal{E}}$.

Specifically, in order to find an abstraction that is sufficient for solving $\mathcal{E}_i$, our algorithm invokes the AGS with the current abstract domain $\mathcal{A}$ and corresponding transformers $\mathcal{T}$ (line 6). We assume $\mathsf{Synthesize}$ returns a program Π that is consistent with $\mathcal{E}_i$ under abstraction ($\mathcal{A}$, $\mathcal{T}$). That is, symbolically executing Π (according to $\mathcal{T}$) on inputs $\mathcal{E}_i^{in}$ yields abstract values $\vec{\varphi}$ that are consistent with the outputs $\mathcal{E}_i^{out}$ (*i.e.*, $\forall j.\ \mathcal{E}_{ij}^{out} \in \gamma(\varphi_j)$). However, while Π is guaranteed to be consistent with $\mathcal{E}_i$ under the abstract semantics, it may not satisfy $\mathcal{E}_i$ under the concrete semantics. In other words, Π is *spurious*.

Thus, whenever the call to $\mathsf{IsCorrect}$ fails at line 8, we invoke the LearnAbstractDomain procedure (line 9) to learn additional predicate templates that are later added to $\mathcal{A}$. Since the refinement of $\mathcal{A}$ necessitates the synthesis of new transformers, we then call LearnTransformers (line 10) to learn a new $\mathcal{T}$. The new abstraction is guaranteed to rule out the spurious program Π as long as there is a unique best transformer of each DSL construct for $\mathcal{A}$.

4.4 Synthesis of Predicate Templates

In this section, we present the LearnAbstractDomain procedure: Given a spurious program Π and a synthesis problem $\mathcal{E}$ that Π does not solve, our goal is to find new predicate templates $\mathcal{A}'$ to add to the abstract domain $\mathcal{A}$ such that the Abstraction-Guided Synthesizer no longer returns Π as a valid solution to the synthesis problem $\mathcal{E}$. Our key insight is that we can mine for such useful predicate templates by constructing a *tree interpolation* problem. In what follows, we first review tree interpolants (based on [10]) and then explain how we use this concept to find useful predicate templates.

Definition 4.4.1 (Tree interpolation problem). A tree interpolation problem $T = (V, r, P, L)$ is a directed labeled tree, where V is a finite set of nodes, $r \in V$ is the root, $P : (V \backslash \{r\}) \mapsto V$ is a function that maps children nodes to their parents, and $L : V \mapsto \mathbb{F}$ is a labeling function that maps nodes to formulas from a set $\mathbb{F}$ of first-order formulas such that $\bigwedge_{v \in V} L(v)$ is unsatisfiable.

In other words, a tree interpolation problem is defined by a tree T where each node is labeled with a formula and the conjunction of these formulas is unsatisfiable. In what follows, we write $\mathit{Desc}(v)$ to denote the set of all descendants of node v, including v itself, and we write $\mathit{NonDesc}(v)$ to denote all nodes other than those in $\mathit{Desc}(v)$ (*i.e.*, $V \backslash \mathit{Desc}(v)$). Also, given a set of nodes V', we write $L(V')$ to denote the set of formulas labeling nodes in V'.

Given a tree interpolation problem T, a *tree interpolant* $\mathcal{I}$ is an annotation from every node in V to a formula such that the label of the root node is *false* and the label of an internal node v is entailed by the conjunction of annotations of its children nodes. More formally, a tree interpolant is defined as follows:

Definition 4.4.2 (Tree interpolant). Given a tree interpolation problem $T = (V, r, P, L)$, a tree interpolant for T is a function $\mathcal{I} : V \mapsto \mathbb{F}$ that satisfies the following conditions:

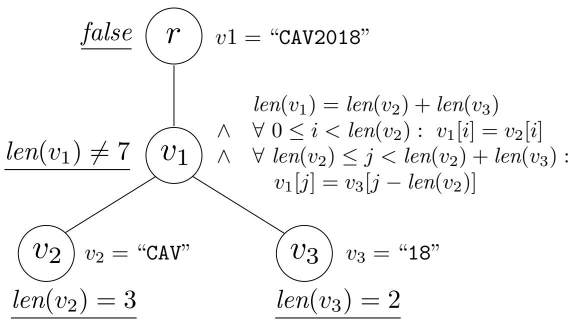

false
r
$v1 = $ "CAV2018"
$len(v_1) = len(v_2) + len(v_3)$
$\wedge \ \forall\, 0 \le i < len(v_2) : \ v_1[i] = v_2[i]$
$\wedge \ \forall\, len(v_2) \le j < len(v_2) + len(v_3) :$
$v_1[j] = v_3[j - len(v_2)]$
$len(v_1) \neq 7$
v_1
v_2
$v_2 = $ "CAV"
v_3
$v_3 = $ "18"
$len(v_2) = 3$
$len(v_3) = 2$

Figure 4.3: A tree interpolation problem and a tree interpolant (underlined).

1. $\mathcal{I}(r) = \mathit{false}$;

2. For each $v \in V$: $\Big(\big(\bigwedge_{P(c_i)=v} \mathcal{I}(c_i) \big) \wedge L(v) \Big) \Rightarrow \mathcal{I}(v)$;

3. For each $v \in V$: $\mathit{Vars}\big(\mathcal{I}(v)\big) \subseteq \mathit{Vars}\big(L(\mathit{Desc}(v))\big) \bigcap \mathit{Vars}\big(L(\mathit{NonDesc}(v))\big)$.

Intuitively, the first condition ensures that $\mathcal{I}$ establishes the unsatisfiability of formulas in T, and the second condition states that $\mathcal{I}$ is a valid annotation. As standard in Craig interpolation [41, 40], the third condition stipulates a "shared vocabulary" condition by ensuring that the annotation at each node v refers to the common variables between the descendants and non-descendants of v.

Example 4.4.1. Consider the tree interpolation problem $T = (V, r, P, L)$ in Figure 4.3, where $L(v)$ is shown to the right of each node v. A tree interpolant $\mathcal{I}$ for this problem maps each node to the corresponding underlined formula. For instance, we have $\mathcal{I}(v_1) = (len(v_1) \neq 7)$. It is easy to confirm that $\mathcal{I}$ is a valid interpolant according to Definition 4.4.2.

```
1: procedure LearnAbstractDomain($\Pi, \mathcal{E}$)
   input: Program $\Pi$ that does not solve problem $\mathcal{E}$ (set of examples).
   output: Set of predicate templates $\mathcal{A}'$.
2:     $\mathcal{A}' := \emptyset$;
3:     for each $(e_{in}, e_{out}) \in \mathcal{E}$ do
4:         if $[\![\Pi]\!]e_{in} \neq e_{out}$ then
5:             $T := \textsc{ConstructTree}(\Pi, e_{in}, e_{out})$;
6:             $\mathcal{I} := \mathsf{FindTreeItp}(T)$;
7:             for each $v \in \mathsf{Nodes}(T)\backslash\{r\}$ do
8:                 $\mathcal{A}' := \mathcal{A}' \cup \big\{\mathsf{MakeSymbolic}\big(\mathcal{I}(v)\big)\big\}$;
9:     return $\mathcal{A}'$;
```

Figure 4.4: Algorithm for learning abstract domain using tree interpolation.

To see how tree interpolation is useful for learning predicates, suppose that the spurious program Π is represented as an abstract syntax tree (AST), where each non-leaf node is labeled with the axiomatic semantics of the corresponding DSL construct. Now, since Π does not satisfy the given input-output example (e_{in}, e_{out}), we are able to use this information to construct a labeled tree where the conjunction of labels is unsatisfiable. Our key idea is to mine useful predicate templates from the formulas in the resulting tree interpolant.

With this intuition in mind, let us consider the LearnAbstractDomain procedure in Figure 4.4: It uses a procedure called ConstructTree to generate a tree interpolation problem T for each example (e_{in}, e_{out}) [2] that program Π does not satisfy (line 5). Specifically, letting Π denote the AST

[2]Without loss of generality, we assume that programs take a single input x, as we can always represent multiple inputs as a list.

representation of Π, we construct $T = (V, r, P, L)$ as follows:

- V consists of all AST nodes in Π as well as a "dummy" node d.
- The root r of T is the dummy node d.
- P is a function that maps children AST nodes to their parents and maps the root AST node to the dummy node d.
- L maps each node $v \in V$ to a formula as follows:

$$L(v) = \begin{cases} v' = e_{out} & v \text{ is the dummy root node with child } v'. \\ v = e_{in} & v \text{ is a leaf representing program input } e_{in}. \\ v = c & v \text{ is a leaf representing constant } c. \\ \phi_F[\vec{v}'/\vec{x}, v/y] & \begin{array}{l} v \text{ represents DSL operator } F \text{ with axiomatic semantics} \\ \phi_F(\vec{x}, y) \text{ and } \vec{v}' \text{ represents children of } v. \end{array} \end{cases}$$

Essentially, the ConstructTree procedure labels any leaf node representing the program input with the input example e_{in} and the root node with the output example e_{out}. All other internal nodes are labeled with the axiomatic semantics of the corresponding DSL operator (modulo renaming).[3] Observe that the formula $\bigwedge_{v \in V} L(v)$ is guaranteed to be unsatisfiable since Π does not satisfy the I/O example (e_{in}, e_{out}); thus, we can obtain a tree interpolant for T.

[3]Here, we assume access to the DSL's axiomatic semantics. If this is not the case (*i.e.*, we are only given the DSL's operational semantics), we can still annotate each node as $v = c$ where c denotes the output of the partial program rooted at node v when executed on e_{in}. However, this may affect the quality of the resulting interpolant.

Example 4.4.2. Consider a program Π : `Concat`$(x, $"`18`"$)$ which concatenates constant string "`18`" to input x. Figure 4.3 shows the result of invoking ConstructTree for Π and input-output example ("`CAV`", "`CAV2018`"). As mentioned in Example 4.4.1, the tree interpolant $\mathcal{I}$ for this problem is indicated with the underlined formulas.

Since the tree interpolant $\mathcal{I}$ effectively establishes the incorrectness of program Π, the predicates used in $\mathcal{I}$ serve as useful abstract values that the synthesizer (AGS) should consider during the synthesis task. Towards this goal, the LearnAbstractDomain algorithm iterates over each predicate used in $\mathcal{I}$ (lines 7–8 in Figure 4.4) and converts it to a suitable template by replacing the constants and variables used in $\mathcal{I}(v)$ with symbolic names (or "holes"). Because the original predicates used in $\mathcal{I}$ may be too specific for the current input-output example, extracting templates from the interpolant allows our method to learn reusable abstract domains.

Example 4.4.3. Given the tree interpolant $\mathcal{I}$ from Example 4.4.1, LearnAbstractDomain extracts two predicate templates, namely, $\mathit{len}(\boxed{\alpha}) = \mathsf{c}$ and $\mathit{len}(\boxed{\alpha}) \neq \mathsf{c}$.

4.5 Synthesis of Abstract Transformers

Now, we turn our attention to the LearnTransformers procedure that synthesizes abstract transformers $\mathcal{T}$ for a given abstract domain $\mathcal{A}$. Following our presentation in Chapter 3, we consider abstract transformers that are described using equations of the following form:

$$[\![F\big(\chi_1(x_1,\vec{c_1}),\cdots,\chi_n(x_n,\vec{c_n})\big)]\!]^\sharp = \bigwedge_{1\le j\le m} \chi'_j\big(y,\vec{f_j}(\vec{c})\big) \tag{4.1}$$

Here, F is a DSL construct, χ_i, χ'_j are predicate templates [4], x_i is the i'th input of F, y is F's output, $\vec{c_1},\cdots,\vec{c_n}$ are vectors of *symbolic* constants, and $\vec{f_j}$ denotes a vector of *affine functions* over $\vec{c}=\vec{c_1},\cdots,\vec{c_n}$. Intuitively, given concrete predicates describing the inputs to F, the transformer returns concrete predicates describing the output. Given such a transformer τ, let $\mathsf{Outputs}(\tau)$ be the set of pairs $(\chi'_j,\vec{f_j})$ in Eqn. 4.1.

We define the soundness of a transformer τ for DSL operator F with respect to F's axiomatic semantics ϕ_F. In particular, we say that the abstract transformer from Eqn. 4.1 is *sound* if the following implication is valid:

$$\Big(\phi_F(\vec{x},y) \land \bigwedge_{1\le i\le n} \chi_i(x_i,\vec{c_i})\Big) \Rightarrow \bigwedge_{1\le j\le m} \chi'_j\big(y,\vec{f_j}(\vec{c})\big) \tag{4.2}$$

That is, the transformer for F is sound if the (symbolic) output predicate is indeed implied by the (symbolic) input predicates according to F's semantics.

Our key observation is that the problem of learning sound transformers can be reduced to solving the following *second-order constraint*:

$$\exists\vec{f}.\ \forall\vec{V}.\Big(\big(\phi_F(\vec{x},y) \land \bigwedge_{1\le i\le n} \chi_i(x_i,\vec{c_i})\big) \Rightarrow \bigwedge_{1\le j\le m} \chi'_j\big(y,\vec{f_j}(\vec{c})\big)\Big) \tag{4.3}$$

where $\vec{f}=\vec{f_1},\cdots,\vec{f_m}$ and $\vec{V}$ includes all variables and functions from Eqn. 4.2 other than $\vec{f}$. In other words, the goal of this constraint solving problem is

[4]We assume that $\chi'_1,\cdots,\chi'_m$ are distinct.

to find interpretations of the unknown functions $\vec{f}$ that make Eqn. 4.2 valid. Our key insight is to solve this problem in a *data-driven* way by exploiting the fact that each unknown function $f_{j,k}$ is affine.

Towards this goal, we first express each affine function $f_{j,k}(\vec{c})$ as follows:

$$f_{j,k}(\vec{c}) = p_{j,k,1} \cdot c_1 + \cdots + p_{j,k,|\vec{c}|} \cdot c_{|\vec{c}|} + p_{j,k,|\vec{c}|+1}$$

where each $p_{j,k,l}$ corresponds to an unknown integer constant that we would like to learn. Now, arranging the coefficients of functions $f_{j,1}, \cdots, f_{j,|\vec{f_j}|}$ in $\vec{f_j}$ into a $|\vec{f_j}| \times (|\vec{c}| + 1)$ matrix P_j, we can represent $\vec{f_j}(\vec{c})$ in the following way:

$$\vec{f_j}(\vec{c})^\intercal = \underbrace{\begin{bmatrix} f_{j,1}(\vec{c}) \\ \cdots \\ f_{j,|\vec{f_j}|}(\vec{c}) \end{bmatrix}}_{\vec{c'}_j^\intercal} = \underbrace{\begin{bmatrix} p_{j,1,1} & \cdots & p_{j,1,|\vec{c}|+1} \\ \cdots & & \cdots \\ p_{j,|\vec{f_j}|,1} & \cdots & p_{j,|\vec{f_j}|,|\vec{c}|+1} \end{bmatrix}}_{P_j} \underbrace{\begin{bmatrix} c_1 \\ \cdots \\ c_{|\vec{c}|} \\ 1 \end{bmatrix}}_{\vec{c}^\dagger} \tag{4.4}$$

where $\vec{c}^\dagger$ is $\vec{c}^\intercal$ appended with the constant 1.

Given this representation, it is easy to see that the problem of synthesizing the unknown functions $\vec{f_1}, \cdots, \vec{f_m}$ from Eqn. 4.2 boils down to finding the unknown matrices $P_1, \cdots, P_m$ such that each P_j makes the following implication valid:

$$\Lambda \equiv \Big(\Big((\vec{c'}_j^\intercal = P_j\vec{c}^\dagger) \land \phi_F(\vec{x}, y) \land \bigwedge_{1\leq i\leq n} \chi_i(x_i, \vec{c}_i)\Big) \Rightarrow \chi'_j(y, \vec{c'}_j)\Big) \tag{4.5}$$

Our key idea is to infer these unknown matrices $P_1, \cdots, P_m$ in a data-driven way by generating input-output examples of the form $[i_1, \cdots, i_{|\vec{c}|}] \mapsto [o_1, \cdots, o_{|\vec{f_j}|}]$ for each $\vec{f_j}$. In other words, $\vec{i}$ and $\vec{o}$ correspond to instantiations

```
1: procedure $\textsc{LearnTransformers}(\mathcal{L}, \mathcal{A})$
   input: DSL $\mathcal{L}$ and abstract domain $\mathcal{A}$.
   output: A set of transformers $\mathcal{T}$ for constructs in $\mathcal{L}$ and abstract domain $\mathcal{A}$.
2:     for each $F \in \mathsf{Constructs}(\mathcal{L})$ do
3:         for $(\chi_1, \cdots, \chi_n) \in \mathcal{A}^n$ do
4:             $\varphi := \top$;                              ▷ $\varphi$ is output of transformer.
5:             for $\chi'_j \in \mathcal{A}$ do
6:                 $E := \textsc{GenerateExamples}(\phi_F, \chi'_j, \chi_1, \cdots, \chi_n)$;
7:                 $\vec{f_j} := \mathsf{Solve}(E)$;
8:                 if $\vec{f_j} \neq null \wedge \mathsf{Valid}(\Lambda[\vec{f_j}])$ then $\varphi := (\varphi \wedge \chi'_j(y, \vec{f_j}(\vec{c}_1, \cdots, \vec{c}_n)))$
9:             $\mathcal{T} := \mathcal{T} \cup \{[\![F(\chi_1(x_1, \vec{c}_1), \cdots, \chi_n(x_n, \vec{c}_n))]\!]^\sharp = \varphi\}$;
10:    return $\mathcal{T}$;
```

Figure 4.5: Algorithm for synthesizing abstract transformers.

of $\vec{c}$ and $\vec{f_j}(\vec{c})$ respectively. Given sufficiently many such examples for every $\vec{f_j}$, we can then reduce the problem of learning each unknown matrix P_j to the problem of solving a system of linear equations.

Based on this intuition, the LearnTransformers procedure from Figure 4.5 describes our algorithm for learning abstract transformers $\mathcal{T}$ for a given abstract domain $\mathcal{A}$. At a high-level, our algorithm synthesizes one abstract transformer for each DSL construct F and n argument predicate templates $\chi_1, \cdots, \chi_n$. In particular, given F and $\chi_1, \cdots, \chi_n$, the algorithm constructs the "return value" of the transformer as:

$$\varphi = \bigwedge_{1 \leq j \leq m} \chi'_j(y, \vec{f_j}(\vec{c}))$$

where $\vec{f_j}$ is the inferred affine function for each predicate template χ'_j.

The key part of our LearnTransformers procedure is the inner loop (lines 5–8) for inferring each of these $\vec{f_j}$'s. Specifically, given an output predicate template χ'_j, our algorithm first generates a set of input-output examples E of the form $[p_1, \cdots, p_n] \mapsto p_0$ such that $[\![F(p_1, \cdots, p_n)]\!]^\sharp = p_0$ is a sound (albeit overly specific) transformer. Essentially, each p_i is a concrete instantiation of a predicate template, so the examples E generated at line 6 of the algorithm can be viewed as sound input-output examples for the symbolic transformer shown in Eqn. 4.1. We will describe the GenerateExamples procedure in Chapter 4.5.1.

Once we generate these examples E, the next step of the algorithm is to learn the unknown coefficients of matrix P_j from Eqn. 4.5 by solving a system of linear equations (line 7). Specifically, observe that we can use each input-output example $[p_1, \cdots, p_n] \mapsto p_0$ in E to construct one row of Eqn. 4.4. In particular, we can directly extract $\vec{c} = \vec{c}_1, \cdots, \vec{c}_n$ from $p_1, \cdots, p_n$ and the corresponding value of $\vec{f_j}(\vec{c})$ from p_0. Since we have one instantiation of Eqn. 4.4 for each of the input-output examples in E, the problem of inferring matrix P_j now reduces to solving a system of linear equations of the form $AP_j^T = B$ where A is a $|E| \times (|\vec{c}| + 1)$ (input) matrix and B is a $|E| \times |\vec{f_j}|$ (output) matrix. Thus, a solution to the equation $AP_j^T = B$ generated from E corresponds to a candidate solution for matrix P_j, which in turn uniquely defines $\vec{f_j}$.

Observe that the call to Solve at line 7 may return *null* if no affine function exists. Furthermore, any *non-null* $\vec{f_j}$ returned by Solve is just a *candidate* solution and may not satisfy Eqn. 4.5. For example, this situation can arise if we do not have sufficiently many examples in E and end up discovering an affine function that is "over-fitted" to the examples. Thus, the validity check at line 8 of the algorithm ensures the learned transformers are actually sound.

4.5.1 Example Generation

In our discussion so far, we assumed an oracle that is capable of generating valid input-output examples for a given transformer. We now explain our GenerateExamples procedure from Figure 4.6 that essentially implements this oracle. In a nutshell, the goal of GenerateExamples is to synthesize input-output examples of the form $[p_1, \cdots, p_n] \mapsto p_0$ such that $[\![F(p_1, \cdots, p_n)]\!]^\sharp = p_0$ is sound where each p_i is a concrete predicate (rather than symbolic).

Going into more detail, GenerateExamples takes as input the semantics ϕ_F of DSL construct F for which we want to learn a transformer for as well as the input predicate templates $\chi_1, \cdots, \chi_n$ and output predicate template χ_0 that are supposed to be used in the transformer. For any example $[p_1, \cdots, p_n] \mapsto p_0$ synthesized by GenerateExamples, each concrete predicate p_i is an instantiation of the predicate template χ_i where the symbolic constants used in χ_i are substituted with *concrete* values.

Conceptually, the GenerateExamples algorithm proceeds as follows: First, it generates *concrete* input-output examples $[s_1, \cdots, s_n] \mapsto s_0$ by evaluating F on randomly-generated inputs $s_1, \cdots, s_n$ (lines 4–5). Now, for each

```
1: procedure GenerateExamples($\phi_F, \chi_0, \cdots, \chi_n$)
   input: axiomatic semantics $\phi_F$ of DSL operator $F$ and predicate templates $\chi_0, \cdots, \chi_n$ for the output and inputs.
   output: a set of valid input-output examples $E$ for DSL construct $F$.
2:     $E := \emptyset$;
3:     while $\neg\mathsf{FullRank}(E)$ do
4:         Draw $(s_1, \cdots, s_n)$ randomly from distribution $R_F$ over $\mathsf{Domain}(F)$;
5:         $s_0 := [\![F(s_1, \cdots, s_n)]\!]$;
6:         $(A_0, \cdots, A_n) := \mathsf{Abstract}(s_0, \chi_0, \cdots, s_n, \chi_n)$;
7:         for each $(p_0, \cdots, p_n) \in A_0 \times \cdots \times A_n$ do
8:             if $\mathsf{Valid}\big(\bigwedge_{1 \le i \le n} p_i \wedge \phi_F \Rightarrow p_0\big)$ then $E := E \cup \big\{[p_1, \cdot\cdot, p_n] \mapsto p_0\big\}$;
9:     return $E$;
```

Figure 4.6: Example generation for learning abstract transformers.

concrete I/O example $[s_1, \cdots, s_n] \mapsto s_0$, we generate a set of *abstract* I/O examples of the form $[p_1, \cdots, p_n] \mapsto p_0$ (line 6). Specifically, we assume that the return value $(A_0, \cdots, A_n)$ of $\mathsf{Abstract}$ at line 6 satisfies the following properties for every $p_i \in A_i$:

- p_i is an instantiation of template χ_i.
- p_i is a sound over-approximation of s_i (i.e., $s_i \in \gamma(p_i)$).
- For any other p'_i satisfying the above two conditions, p'_i is not logically stronger than p_i.

In other words, we assume that $\mathsf{Abstract}$ returns a set of "best" sound abstractions of $(s_0, \cdots, s_n)$ under predicate templates $(\chi_0, \cdots, \chi_n)$.

Next, given abstractions $(A_0, \cdots, A_n)$ for $(s_0, \cdots, s_n)$, we consider each candidate abstract example of the form $[p_1, \cdots, p_n] \mapsto p_0$ where $p_i \in A_i$. Even though each p_i is a sound abstraction of s_i, the example $[p_1, \cdots, p_n] \mapsto p_0$ may not be valid according to the semantics of operator F. Thus, the validity check at line 8 ensures that each example added to E is in fact valid.

Example 4.5.1. Given abstract domain $\mathcal{A} = \{\mathit{len}(\boxed{\alpha}) = \mathsf{c}\}$, suppose we want to learn an abstract transformer τ for `Concat` of the following form:

$$[\![\texttt{Concat}\big(\mathit{len}(x_1) = \mathsf{c}_1, \mathit{len}(x_2) = \mathsf{c}_2\big)]\!]^\sharp = \big(\mathit{len}(y) = f([\mathsf{c}_1, \mathsf{c}_2])\big)$$

We learn the affine function f used in the transformer by first generating a set E of input-output examples for f (line 6 in LearnTransformers). In particular, GenerateExamples generates concrete input values for Concat at random and obtains the corresponding output values by executing Concat on the input values. For instance, it may generate $s_1 =$ "*abc*" and $s_2 =$ "*de*" as inputs, and obtain $s_0 =$ "*abcde*" as output. Then, it abstracts these values under the given templates. In this case, we have an abstract example with $p_1 = \big(\mathit{len}(x_1) = 3\big)$, $p_2 = \big(\mathit{len}(x_2) = 2\big)$ and $p_0 = \big(\mathit{len}(y) = 5\big)$. Since $[p_1, p_2] \mapsto p_0$ is a valid example, it is added in E (line 8 in GenerateExamples). At this point, E is not yet full rank, so the algorithm keeps generating more examples. Suppose it generates two more valid examples $\big(\mathit{len}(x_1) = 1, \mathit{len}(x_2) = 4\big) \mapsto \big(\mathit{len}(y) = 5\big)$ and $\big(\mathit{len}(x_1) = 6, \mathit{len}(x_2) = 4\big) \mapsto \big(\mathit{len}(y) = 10\big)$. Now E is full rank, so LearnTransformers computes f by solving the following system of linear equations:

$$\begin{bmatrix} 3 & 2 & 1 \\ 1 & 4 & 1 \\ 6 & 4 & 1 \end{bmatrix} P^T = \begin{bmatrix} 5 \\ 5 \\ 10 \end{bmatrix} \xRightarrow{\textsf{Solve}} P = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}$$

Here, P corresponds to the function $f([\texttt{c}_1, \texttt{c}_2]) = \texttt{c}_1 + \texttt{c}_2$, and this function defines the sound transformer: $[\![\texttt{Concat}\big(len(x_1) = \texttt{c}_1, len(x_2) = \texttt{c}_2\big)]\!]^\sharp = \big(len(y) = \texttt{c}_1 + \texttt{c}_2\big)$ which is added to $\mathcal{T}$ at line 9 in LearnTransformers.

4.6 Evaluation

We have implemented the proposed method as a new tool called Atlas, which is written in Java. Atlas takes as input a set of training problems, an Abstraction-Guided Synthesizer (AGS), and a DSL and returns an abstract domain (in the form of predicate templates) and the corresponding transformers. Internally, Atlas uses the Z3 theorem prover [73] to compute tree interpolants and the JLinAlg linear algebra library [28] to solve linear equations.

To assess the usefulness of Atlas, we conduct an experimental evaluation in which our goal is to answer the following two questions:

1. How does Atlas perform during training? That is, how many training problems does it require and how long does training take?

2. How useful are the abstractions learned by Atlas in the context of synthesis?

4.6.1 Abstraction Learning

To answer our first question, we use Atlas to automatically learn abstractions for two application domains: (i) string manipulations and (ii) tensor transformations. We provide Atlas with the DSLs used in Chapter 3.5 and employ Blaze as the underlying Abstraction-Guided Synthesizer. Axiomatic semantics for each DSL construct were given in the theory of equality with uninterpreted functions.

Training set information. For the string domain, our training set consists of exactly the four problems used as motivating examples in BlinkFill [56]. Specifically, each training problem consists of 4-6 examples that demonstrate the desired string transformation. For the tensor domain, our training set consists of four (randomly selected) synthesis problems taken from online forums. Since almost all online posts contain one single input-output example, each training problem includes one example illustrating the desired reshaping task.

Main results. Our main results are summarized in Figure 4.7 [5]. The main take-away message is that Atlas can learn abstractions quite efficiently and does not require a large training set. For example, Atlas learns 5 predicate templates and 30 abstract transformers for the string domain in a total of 10.2

[5]Here, $|\mathcal{A}|$, $|\mathcal{T}|$, Iters denote the number of predicate templates, abstract transformers, and iterations taken per training instance (lines 5-10 from Figure 4.2), respectively. $T_{\mathsf{AGS}}, T_{\mathcal{A}}, T_{\mathcal{T}}$ denote the times for invoking the AGS, learning the abstract domain, and learning the abstract transformers, respectively. T_{total} shows the total training time in seconds

	$\lvert\mathcal{A}\rvert$	$\lvert\mathcal{T}\rvert$	Iters.	Running time (sec)			
				T_{AGS}	$T_{\mathcal{A}}$	$T_{\mathcal{T}}$	T_{total}
$\mathcal{E}_1$	5	30	4	0.6	0.2	0.2	1.0
$\mathcal{E}_2$	5	30	1	4.9	0	0	4.9
$\mathcal{E}_3$	5	30	1	0.2	0	0	0.2
$\mathcal{E}_4$	5	30	1	4.1	0	0	4.1
Total	5	30	7	9.8	0.2	0.2	**10.2**

String domain

	$\lvert\mathcal{A}\rvert$	$\lvert\mathcal{T}\rvert$	Iters.	Running time (sec)			
				T_{AGS}	$T_{\mathcal{A}}$	$T_{\mathcal{T}}$	T_{total}
$\mathcal{E}_1$	8	45	3	2.9	0.7	0.5	4.1
$\mathcal{E}_2$	8	45	1	2.8	0	0	2.8
$\mathcal{E}_3$	10	59	2	0.5	0.3	0.2	1.0
$\mathcal{E}_4$	10	59	1	14.6	0	0	14.6
Total	10	59	7	20.8	1.0	0.7	**22.5**

Tensor domain

Figure 4.7: Training results of Atlas.

seconds. Interestingly, Atlas does not need all the training problems to infer these four predicates and converges to the final abstraction after just processing the first training instance. Furthermore, for the first training instance, it takes Atlas 4 iterations in the learning loop (lines 5-10 from Figure 4.2) before it converges to the final abstraction. Since this abstraction is sufficient, it takes just one iteration for each following training problem to synthesize a correct program.

For the tensor domain in Figure 4.7, we also observe similar results. In particular, ATLAS learns 10 predicate templates and 59 abstract transformers in a total of 22.5 seconds. Furthermore, ATLAS converges to the final abstract domain after processing the first three problems, and the number of iterations for each training instance is also quite small (ranging from 1 to 3).

4.6.2 Evaluating the Usefulness of Learned Abstractions

To answer our second question, we integrated the abstractions synthesized by ATLAS into the BLAZE framework. In the remainder of this chapter, we refer to all instantiations of BLAZE using the ATLAS-generated abstractions as BLAZE*. To assess how useful the automatically generated abstractions are, we compare BLAZE* against BLAZE†, which refers to the manually-constructed instantiations of BLAZE described in Chapter 3.

Benchmark information. For the string domain, our benchmark suite consists of (1) *all* 108 string transformation benchmarks that were used to evaluate BLAZE† and (2) 40 additional challenging problems that are collected from online forums involving manipulating file paths, URLs, etc. The number of examples for each benchmark ranges from 1 to 400, with a median of 7. For the tensor domain, our benchmark set includes (1) *all* 39 tensor transformation benchmarks in the BLAZE† benchmark suite and (2) 20 additional challenging problems collected from online forums. *We emphasize that the set of benchmarks used for evaluating* BLAZE* *are completely* disjoint *from the set*

	Original Blaze† benchmarks				Additional benchmarks				All benchmarks		
	#Solved		Running time improvement		#Solved		Running time improvement		Time (sec)	Running time improvement	
	Blaze*	Blaze†	max.	avg.	Blaze*	Blaze†	max.	avg.	avg.	max.	avg.
String	**93**	91	15.7×	1.5×	**40**	40	56×	18.8×	**2.8**	**56**×	**3.2**×
Tensor	**39**	39	6.1×	2.9×	**20**	19	83×	15.6×	**5.0**	**83**×	**5.0**×

Figure 4.8: Improvement of Blaze* over Blaze†.

of synthesis problems used for training Atlas.

Experimental setup. We evaluate Blaze* and Blaze† using the same DSLs presented in Chapter 3.5. For each benchmark, we provide the same set of input-output examples to Blaze* and Blaze†, and use a time limit of 20 minutes per synthesis task.

Main results. Our main evaluation results are summarized in Figure 4.8. The key observation is that Blaze* consistently improves upon Blaze† for both string and tensor transformations. In particular, Blaze* not only solves more benchmarks than Blaze† for both domains, but also achieves about an order of magnitude speed-up on average for the common benchmarks that both tools can solve. Specifically, for the string domain, Blaze* solves 133 (out of 148) benchmarks within an average of 2.8 seconds and achieves an average 3.2× speed-up over Blaze†. For the tensor domain, we also observe a very similar result where Blaze* leads to an overall speed-up of 5.0× on average.

In summary, this experiment confirms that the abstractions discovered by Atlas are indeed useful and that they outperform manually-crafted abstractions despite eliminating human effort.

Chapter 5

Related Work

In this chapter, we compare our techniques against related approaches in the synthesis and verification literature.

CEGAR in model checking. Our approach is inspired by the use of counterexample-guided abstraction refinement (CEGAR) in software model checking [9, 24, 23, 6]. The idea is to start with a coarse abstraction of the program and then perform model checking over this abstraction. Since any errors encountered using this approach may be spurious, the model checker then looks for a counterexample trace and refines the abstraction if the error is indeed spurious. While there are many ways to perform refinement, a popular approach is to refine the abstraction using *interpolation*, which provides a proof of unsatisfiability of a trace [23]. Our synthesis approach is very similar to CEGAR-based model checkers in the overall workflow, however, we perform abstraction refinement whenever we find a *spurious program* as opposed to a spurious error trace. In addition, the incorrectness proofs that we utilize in our synthesis technique can be viewed as a form of *tree interpolant* [42, 53].

Abstraction in program synthesis. The only prior work that uses abstraction refinement in the context of synthesis is the *abstraction-guided synthesis* (AGS) technique by Vechev *et al.* for learning efficient synchronization for concurrent programs [69]. Unlike BLAZE which aims to learn an *entire* program from input-output examples, AGS requires an input concurrent program and only performs small modifications to the program by adding synchronization primitives. Specifically, AGS first abstracts the program and then checks whether there are any counterexample (abstract) interleavings that violate the given safety constraint. If there is no violation, it returns the current program. Otherwise, it non-deterministically chooses to either refine the abstraction or modify the program by adding synchronization primitives such that the violating interleaving is removed. AGS can be viewed as a program repair technique and cannot be used for synthesizing programs from input-output examples.

Other synthesizers that bear similarities to the approach proposed in this dissertation include SYNQUID [49] and MORPHEUS [15]. In particular, both of these techniques use specifications of DSL constructs in the form of refinement types and first-order formulas respectively, and use these specifications to refute programs that do not satisfy the specification. Similarly, BLAZE uses abstract semantics of the DSL, which can also be viewed as specifications. However, unlike SYNQUID, the specifications in BLAZE and MORPHEUS over-approximate the behavior of the DSL constructs. Furthermore, BLAZE differs from both techniques in that it performs abstraction refinement and learns programs using finite tree automata.

There is a line of work that uses abstractions in the context of component-based program synthesis [20, 67]. These techniques annotate each component with a "decoration" that serves as an abstraction of the semantics of that component. The use of such abstractions simplifies the synthesis task by reducing a complex $\exists\forall$ problem to a simpler $\exists\exists$ constraint solving problem, albeit at the cost of sacrificing the completeness. In contrast to these techniques, our method uses abstractions to construct a compact finite tree automaton and performs abstraction refinement to rule out spurious programs.

The use of abstraction refinement has also been explored in the context of superoptimizing compilers [47]. In particular, Phothilimthana *et al.* use test cases to construct an (over-approximate) abstraction of the program behavior and "refine" this abstraction by iteratively including more test cases. However, since this abstraction is heuristically applied only to the "promising" parts of the candidate space, this method may not be able to find the desired equivalent program. This technique differs significantly from our method in that they use an orthogonal definition of abstraction and perform abstraction refinement in a different and heuristic-guided manner.

Another related technique is Storyboard Programming [59] for learning data structure manipulation programs from examples by combining abstract interpretation and shape analysis. However, it differs from Blaze in that the user needs to manually provide precise abstractions for input-output examples as well as abstract transformers for data structure operations. Furthermore, there is no automated refinement phase.

Programming-by-example (PBE). The problem of automatically learning programs that are consistent with a set of input-output examples has been the subject of research for the last four decades [55]. Recent advances in algorithmic and logical reasoning techniques have led to the development of PBE systems in several domains including regular expression based string transformations [21, 56], data structure manipulations [16, 71], file manipulations [22], interactive parser synthesis [34], and synthesizing map-reduce distributed programs [60]. It has also been studied from different perspectives, such as type-theoretic interpretation [54, 45, 17], version space learning [50, 21], and deep learning [46, 13].

Our method presents a new approach to example-guided program synthesis using abstraction refinement. Unlike most of the earlier PBE approaches that prune the search space using the concrete semantics of DSL operators [2, 68], Blaze uses the DSL's abstract semantics and iteratively refines the abstraction until it finds a program that satisfies the input-output examples. We instantiate Blaze in three application domains, namely data completion, string processing and tensor reshaping, and we believe that Blaze can be used to complement many previous PBE systems to make synthesis more efficient.

Counterexample-guided inductive synthesis. Counterexample-guided inductive synthesis (CEGIS) [64, 62] is a popular algorithm for solving synthesis problems of the form $\exists P \, \forall i : \phi(P, i)$ where the goal is to find a program P such that the specification ϕ holds for all inputs i. The key idea in CEGIS is to reduce the solving of the second-order formula to two first-order formulas: (1) $\exists P : \phi(P, i_1) \wedge \cdots \wedge \phi(P, i_k)$ (synthesis) and (2) $\exists i : \neg\phi(P, i)$ (verification). In the first phase, we synthesize a program P that is consistent with a *finite* set of inputs $(i_1, \cdots, i_k)$, whereas in the second phase we perform verification on P to find a counterexample input i that violates the specification ϕ. If such an input i exists, it is added to the set of current inputs and the synthesis phase is repeated. This iterative process continues until either the verification check succeeds (*i.e.*, the synthesized program satisfies the specification) or if the synthesis check fails (*i.e.*, there is no program that satisfies the specification).

CEGIS bears similarities to BLAZE in that both approaches are guided by counterexamples (*i.e.*, incorrect programs). However, they are very different in that CEGIS abstracts the specification, whereas BLAZE abstracts the program. In particular, the synthesis phase in CEGIS uses a finite set of examples to *under-approximate* the specification, whereas BLAZE uses program abstractions to *over-approximate* the program behavior in programming-by-example. Because BLAZE is intended for example-guided synthesis, we believe that it can be used to complement the synthesis phase in CEGIS.

Tree automata. Tree automata, which generalize word (string) automata, were originally used for proving the existence of a decision procedure for weak monadic second-order logic [66]. Since then, tree automata have found applications in analyzing XML documents [38, 25], software verification [1, 27, 18, 44] and natural language processing [29, 39]. Recent work by Kafle and Gallagher is particularly related in that they use counterexample-guided abstraction refinement to solve a system of constrained Horn Clauses and perform refinement using finite tree automata [18]. In contrast to their approach, we use finite tree automata for synthesis rather than for refinement.

Tree automata have also found interesting applications in the context of program synthesis. For example, Parthasarathy uses tree automata as a theoretical basis for reactive synthesis [37]. Specifically, given an ω-specification of the reactive system, their technique constructs a tree automaton that accepts all programs that meet the specification. In this dissertation, we use finite tree automata for programming-by-example in a general setting. We also introduce introduce the concept of abstract finite tree automata (AFTAs) and describe a method for counterexample-guided synthesis using AFTAs.

Abstract transformers. Many verification techniques use logical abstract domains [35, 36, 48, 30, 52]. Some of these work, following Yorsh *et al.* [51] use sampling with a decision procedure to evaluate the abstract transformer [65]. Interpolation has also been used to compile efficient symbolic abstract transformers [26]. However, these techniques are restricted to only finite domains or domains of finite height in order to allow convergence. Here, BLAZE uses infinite parameterized domains to obtain better generalization; hence, the abstract transformer computation in our context is more challenging. Nonetheless, the approach might also be applicable in verification.

Chapter 6

Conclusion

This dissertation describes a programming-by-example framework that is both generic and efficient. The underpinning idea is a novel program synthesis paradigm that consists of two main components: an abstraction-based synthesis component that synthesizes programs with respect to an abstraction and an abstraction refinement component that refines the abstraction whenever it is not precise. We present a particular development of this idea based on finite tree automata and the notion of incorrectness proofs. We have implemented this framework in a tool, called BLAZE, that can be instantiated to different application domains by providing a domain-specific language with its syntax and abstract semantics. Our evaluation demonstrates that BLAZE can successfully synthesize non-trivial programs across three different application domains and achieves orders of magnitude improvement in terms of the synthesis speed compared to existing state-of-the-art synthesis techniques.

Appendices

Appendix A

Proofs of Theorems

Theorem 2.2.1 **(Soundness of CFTA)** Let $\mathcal{A}$ be the CFTA constructed for a DSL (with concrete semantics) and examples $\vec{e}$. If Π is a program that is accepted by $\mathcal{A}$, then Π is consistent with examples $\vec{e}$ with respect to the DSL's concrete semantics.

Proof. □

Theorem 2.2.2 **(Completeness of CFTA)**

Proof. □

Theorem 3.1.1 **(Soundness of AFTA)** Let $\mathcal{A}$ be the AFTA constructed for a DSL (with abstract semantics), examples $\vec{e}$ and predicates $\mathcal{P}$. If Π is a program that is accepted by $\mathcal{A}$, then Π is consistent with examples $\vec{e}$ with respect to the DSL's abstract semantics under the abstract domain defined by $\mathcal{P}$.

Proof. □

???

Theorem A.0.1. **(Completeness of AFTA)** Let $\mathcal{A}$ be the AFTA constructed for a DSL (with abstract semantics), examples $\vec{e}$ and predicates $\mathcal{P}$. If Π is a program that is consistent with examples $\vec{e}$ with respect to the DSL's abstract semantics under the abstract domain defined by $\mathcal{P}$, then Π is accepted by $\mathcal{A}$.

Theorem A.0.2. **(Existence of Incorrectness Proofs)** Given a spurious program Π that does not satisfy example e according to concrete semantics, an incorrectness proof of Π satisfying properties in Definition 3.2.1. always exists.

Theorem A.0.3. **(Progress)** Let $\mathcal{A}_i$ be the AFTA constructed in the i'th iteration of the Learn procedure from Figure 3.3, and let Π_i be a spurious program returned by Rank. Then, we have $\Pi_i \notin \mathcal{L}(\mathcal{A}_{i+1})$ and $\mathcal{L}(\mathcal{A}_{i+1}) \subset \mathcal{L}(\mathcal{A}_i)$.

Theorem A.0.4. **(Soundness of Algorithm in Figure 3.3)** If the Learn procedure from Figure 3.3 returns a program Π for examples $\vec{e}$, then Π satisfies $\vec{e}$, namely, $[\![\Pi]\!]\vec{e}_{in} = \vec{e}_{out}$.

Theorem A.0.5. **(Completeness of Algorithm in Figure 3.3)** If there exists a program in the DSL that satisfies the input-output examples $\vec{e}$, then given the DSL and examples $\vec{e}$, the Learn procedure from Figure 3.3 will return a DSL program Π such that $[\![\Pi]\!]\vec{e}_{in} = \vec{e}_{out}$.

Theorem A.0.6. **(Correctness of Algorithm in Figure 3.5)** The mapping $\mathcal{I}$ returned by the ConstructProof procedure from Figure 3.5 satisfies the properties from Definition 3.2.1.

Bibliography

[1] Parosh A Abdulla, Ahmed Bouajjani, Lukáš Holík, Lisa Kaati, and Tomáš Vojnar. Composed bisimulation for tree automata. In *International Conference on Implementation and Application of Automata*, pages 212–222. Springer, 2008.

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

[3] 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. *Dependable Software Systems Engineering*, 40:1–25, 2015.

[4] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. SyGuS-Comp 2016: Results and Analysis. In *SYNT*, pages 178–202, 2016.

[5] 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*, TACAS, pages 319–336. Springer, 2017.

[6] Thomas Ball, Vladimir Levin, and Sriram K Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76, 2011.

[7] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

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

[9] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[10] Régis Blanc, Ashutosh Gupta, Laura Kovács, and Bernhard Kragl. Tree Interpolation in Vampire. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 173–181. Springer, 2013.

[11] 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*, pages 1732–1736. ACM, 2012.

[12] 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*, POPL, pages 238–252, 1977.

[13] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdelrahman Mohamed, and Pushmeet Kohli. RobustFill: Neural program learning under noisy I/O. *arXiv preprint arXiv:1703.07469*, 2017.

[14] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. *arXiv preprint arXiv:1711.08029*, 2017.

[15] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 422–436. ACM, 2017.

[16] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 229–239. ACM, 2015.

[17] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic.

Example-directed synthesis: A type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 802–815. ACM, 2016.

[18] John Gallagher and German Puebla. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. *Practical Aspects of Declarative Languages*, pages 243–261, 2002.

[19] Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. Directed hypergraphs and applications. *Discrete Appl. Math.*, 42(2-3):177–201, 1993.

[20] Adrià Gascón, Ashish Tiwari, Brent Carmer, and Umang Mathur. Look for the proof to find the program: Decorated-component-based program synthesis. In *International Conference on Computer Aided Verification*, CAV, pages 86–103. Springer, 2017.

[21] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 317–330. ACM, 2011.

[22] Sumit Gulwani, Mikaël Mayer, Filip Niksic, and Ruzica Piskac. StriSynth: Synthesis for live programming. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE, pages 701–704. IEEE, 2015.

[23] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 232–244. ACM, 2004.

[24] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *International SPIN Workshop on Model Checking of Software*, pages 235–239. Springer, 2003.

[25] Haruo Hosoya and Benjamin C Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

[26] Ranjit Jhala and Kenneth L. McMillan. Interpolant-based transition relation approximation. *Logical Methods in Computer Science*, 3(4), 2007.

[27] Bishoksan Kafle and John P Gallagher. Tree automata-based refinement with application to horn clause verification. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, VMCAI, pages 209–226. Springer, 2015.

[28] A Keilhauer, SD Levy, A Lochbihler, S Ökmen, GL Thimm, and C Würzebesser. JLinAlg: A Java-library for Linear Algebra without Rounding Errors. Technical report, Technical report (2003-2010), http://jlinalg.sourceforge.net/.

[29] Kevin Knight and Jonathan May. Applications of weighted automata in natural language processing. In *Handbook of Weighted Automata*, pages

571–596. Springer, 2009.

[30] Shuvendu K. Lahiri and Randal E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI*, pages 267–281, 2004.

[31] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

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

[33] Vu Le and Sumit Gulwani. FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 542–553. ACM, 2014.

[34] Alan Leung, John Sarracino, and Sorin Lerner. Interactive parser synthesis by example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 565–574. ACM, 2015.

[35] Tal Lev-Ami, Roman Manevich, and Mooly Sagiv. TVLA: A System for Generating Abstract Interpreters. *Building the Information Society*, pages 367–375, 2004.

[36] Tal Lev-Ami and Mooly Sagiv. TVLA: A System for Implementing Static Analyses. *Static Analysis*, pages 105–110, 2000.

[37] Parthasarathy Madhusudan. Synthesizing reactive programs. In *Lipics-Leibniz International Proceedings in Informatics*, volume 12, 2011.

[38] Wim Martens and Joachim Niehren. Minimizing tree automata for unranked trees. In *International Workshop on Database Programming Languages*, pages 232–246. Springer, 2005.

[39] Jonathan May and Kevin Knight. A primer on tree automata software for natural language processing, 2008.

[40] Kenneth L McMillan. Interpolation and SAT-based Model Checking. In *International Conference on Computer Aided Verification*, CAV, pages 1–13. Springer, 2003.

[41] Kenneth L McMillan. Applications of Craig Interpolants in Model Checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–12. Springer, 2005.

[42] Kenneth L McMillan and Andrey Rybalchenko. Solving constrained horn clauses using interpolation. *Tech. Rep. MSR-TR-2013-6*, 2013.

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

[44] David Monniaux. Abstracting cryptographic protocols with tree automata. In *International Static Analysis Symposium*, pages 149–163. Springer, 1999.

[45] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 619–630. ACM, 2015.

[46] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[47] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 297–310. ACM, 2016.

[48] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, pages 82–97, 2001.

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

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

[51] Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic Implementation of the Best Transformer. In *VMCAI*, volume 2937, pages 252–266. Springer, 2004.

[52] Thomas Reps and Aditya Thakur. Automating Abstract Interpretation. In *VMCAI*, pages 3–40. Springer, 2016.

[53] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Classifying and solving horn clauses for verification. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 1–21. Springer, 2013.

[54] Gabriel Scherer and Didier Rémy. Which simple types have a unique inhabitant? In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP, pages 243–255. ACM, 2015.

[55] David E. Shaw, William R. Swartout, and C. Cordell Green. Inferring LISP programs from examples. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, IJCAI, pages 260–267, 1975.

[56] Rishabh Singh. BlinkFill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment*, 9(10):816–827, 2016.

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

[58] Rishabh Singh and Sumit Gulwani. Transforming spreadsheet data types using examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 343–356. ACM, 2016.

[59] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE, pages 289–299, 2011.

[60] Calvin Smith and Aws Albarghouthi. MapReduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 326–340. ACM, 2016.

[61] Sunbeom So and Hakjoo Oh. Synthesizing imperative programs from examples guided by static analysis. In *Static Analysis Symposium*, pages 364–381. Springer International Publishing, 2017.

[62] Armando Solar-Lezama. *Program synthesis by sketching.* PhD thesis, 2008.

[63] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 281–294. ACM, 2005.

[64] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 404–415. ACM, 2006.

[65] Aditya V Thakur and Thomas W Reps. A Method for Symbolic Computation of Abstract Operations. In *International Conference on Computer Aided Verification*, volume 12, pages 174–192. Springer, 2012.

[66] James W Thatcher and Jesse B Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Theory of Computing Systems*, 2(1):57–81, 1968.

[67] Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. Program synthesis using dual interpretation. In *International Conference on Automated Deduction*, pages 482–497. Springer, 2015.

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

[69] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 327–338, 2010.

[70] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 452–466. ACM, 2017.

[71] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 508–521. ACM, 2016.

[72] Yifei Yuan, Rajeev Alur, and Boon Thau Loo. NetEgg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 20. ACM, 2014.

[73] Z3. https://github.com/Z3Prover/z3.

Vita

Xinyu Wang was born in Weifang, Shandong, China. He graduate from Shanghai Jiao Tong University in 2013 with a B.E. degree in Information Engineering. In August 2013, he entered the doctoral program in the Department of Computer Science at the University of Texas at Austin. He obtained a M.S. degree in Computer Science in May 2019.

Email address: xinyuwangsd@gmail.com

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.