Program Synthesis using Abstraction Refinement

XINYU WANG, University of Texas at Austin, USA
ISIL DILLIG, University of Texas at Austin, USA
RISHABH SINGH, Microsoft Research, USA

We present a new approach to example-guided program synthesis based on *counterexample-guided abstraction refinement*. Our method uses the abstract semantics of the underlying DSL to find a program P whose *abstract* behavior satisfies the examples. However, since program P may be spurious with respect to the concrete semantics, our approach iteratively refines the abstraction until we either find a program that satisfies the examples or prove that no such DSL program exists. Because many programs have the same input-output behavior in terms of their *abstract semantics*, this synthesis methodology significantly reduces the search space compared to existing techniques that use purely concrete semantics.

While *synthesis using abstraction refinement (SYNGAR)* could be implemented in different settings, we propose a refinement-based synthesis algorithm that uses *abstract finite tree automata (AFTA)*. Our technique uses a coarse initial program abstraction to construct an initial AFTA, which is iteratively refined by constructing a *proof of incorrectness* of any spurious program. In addition to ruling out the spurious program accepted by the previous AFTA, proofs of incorrectness are also useful for ruling out many other spurious programs.

We implement these ideas in a framework called Blaze, which can be instantiated in different domains by providing a suitable DSL and its corresponding concrete and abstract semantics. We have used the Blaze framework to build synthesizers for string and matrix transformations, and we compare Blaze with existing techniques. Our results for the string domain show that Blaze compares favorably with FlashFill, a domain-specific synthesizer that is now deployed in Microsoft PowerShell. In the context of matrix manipulations, we compare Blaze against Prose, a state-of-the-art general-purpose VSA-based synthesizer, and show that Blaze results in a 90x speed-up over Prose. In both application domains, Blaze also consistently improves upon the performance of two other existing techniques by at least an order of magnitude.

CCS Concepts: • **Software and its engineering → Programming by example**; **Formal software verification**; • **Theory of computation → Abstraction**;

Additional Key Words and Phrases: Program Synthesis, Abstract Interpretation, Counterexample Guided Abstraction Refinement, Tree Automata

ACM Reference Format:
Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program Synthesis using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63 (January 2018), 29 pages. https://doi.org/10.1145/3158151

1 INTRODUCTION

In recent years, there has been significant interest in automatically synthesizing programs from input-output examples. Such programming-by-example (PBE) techniques have been successfully used to synthesize string and format transformations [Gulwani 2011; Singh and Gulwani 2016] ,

Authors' addresses: Xinyu Wang, Computer Science, University of Texas at Austin, Austin, TX, USA, xwang@cs.utexas.edu; Isil Dillig, Computer Science, University of Texas at Austin, Austin, TX, USA, isil@cs.utexas.edu; Rishabh Singh, Microsoft Research, Redmond, WA, USA, risin@microsoft.com.

2475-1421/2018/1-ART63
https://doi.org/10.1145/3158151

automate data wrangling tasks [Feng et al. 2017], and synthesize programs that manipulate data structures [Feser et al. 2015; Osera and Zdancewic 2015; Yaghmazadeh et al. 2016]. Due to its potential to automate many tasks encountered by end-users, programming-by-example has now become a burgeoning research area.

Because program synthesis is effectively a very difficult search problem, a key challenge in this area is how to deal with the enormous size of the underlying search space. Even if we restrict ourselves to short programs of fixed length 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 that yield the same *concrete* output on the same input are indistinguishable with respect to the user-provided specification. Based on this observation, many techniques use a canonical representation of a large set of programs that have the same input-output behavior. For instance, enumeration-based techniques, such as Escher [Albarghouthi et al. 2013] and Transit [Udupa et al. 2013], discard programs that yield the same output as a previously explored program. Similarly, synthesis algorithms in the Flash* family [Gulwani 2011; Polozov and Gulwani 2015], 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 concrete output values produced by the DSL programs on the given inputs.

In this paper, we aim to develop a more scalable general-purpose synthesis algorithm by using the *abstract* semantics of DSL constructs rather than their concrete semantics. Building on the insight that we can reduce the size of the search space by exploiting commonalities in the 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. Starting from the input example, our algorithm symbolically executes programs in the DSL using their *abstract semantics* and merges any programs that have 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 size in a more dramatic way.

Of course, one obvious implication of such an abstraction-based approach is that the synthesized programs may now be *spurious*: That is, a program that is consistent with the provided examples based on its abstract semantics may not actually satisfy the examples. Our synthesis algorithm iteratively eliminates such spurious programs by performing a form of *counterexample-guided abstraction refinement*: Starting with a coarse initial abstraction, we first find a program P that is consistent with the input-output examples with respect to its abstract semantics. If P is also consistent with the examples using 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. As shown in Figure 1, this refinement process continues until we either find a program that satisfies the input-output examples, or prove that no such DSL program exists.

While the general idea of program synthesis using abstractions can be realized in different ways, we develop this idea by generalizing a recently-proposed synthesis algorithm that uses *finite tree automata* (FTA) [Wang et al. 2017b]. The key idea underlying this technique is to use the *concrete* semantics of the DSL to construct an FTA whose language is exactly the set of programs that are consistent with the input-output examples. While this approach can, in principle, be used to synthesize programs over any DSL, it suffers from the same scalability problems as other techniques that use concrete program semantics.

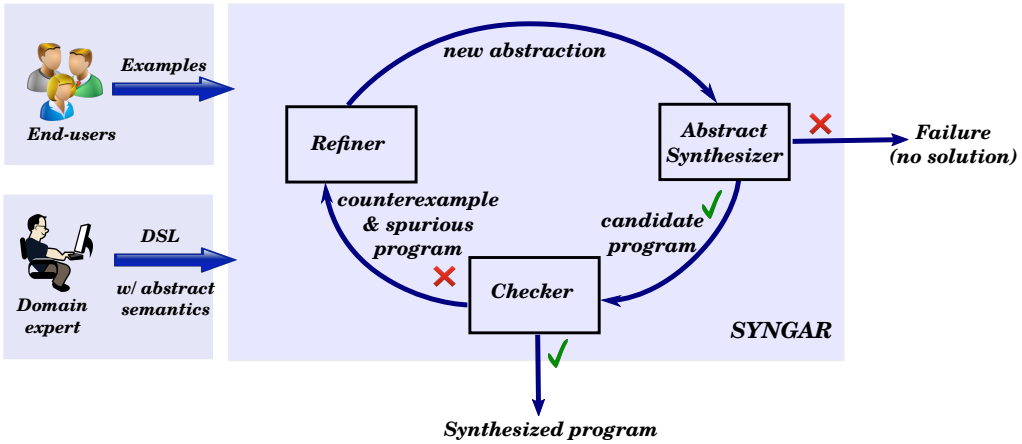


Fig. 1. Workflow illustrating *Synthesis using abstraction refinement (SYNGAR)*. Since our approach is domain-agnostic, it is parametrized over a domain-specific language with both concrete and abstract semantics. From a user's perspective, the only input to the algorithm is a set of input-output examples.

In this paper, we introduce the notion of *abstract finite tree automata (AFTA)*, which can be used to synthesize programs over the DSL's abstract semantics. Specifically, states in an AFTA correspond to *abstract* values and transitions are constructed using the DSL's *abstract* semantics. Any program accepted by the AFTA is consistent with the specification in the DSL's abstract semantics, but not necessarily in its concrete semantics. Given a spurious program P accepted by the AFTA, our technique automatically refines the current abstraction by constructing a so-called *proof of incorrectness*. Such a proof annotates the nodes of the abstract syntax tree representing P with predicates that should be used in the new abstraction. The AFTA constructed in the next iteration is guaranteed to reject P, alongside many other spurious programs accepted by the previous AFTA.

We have implemented our proposed idea in a synthesis framework called Blaze, which can be instantiated in different domains by providing a suitable DSL with its corresponding concrete and abstract semantics. As one application, we use Blaze to automate string transformations from the SyGuS benchmarks [Alur et al. 2015] and empirically compare Blaze against FlashFill, a synthesizer shipped with Microsoft PowerShell and that specifically targets string transformations [Gulwani 2011]. In another application, we have used Blaze to automatically synthesize non-trivial matrix and tensor manipulations in MATLAB and compare Blaze with Prose, a state-of-the-art synthesis tool based on version space algebra [Polozov and Gulwani 2015]. Our evaluation shows that Blaze compares favorably with FlashFill in the string domain and that it outperforms Prose by 90x when synthesizing matrix transformations. We also compare Blaze against enumerative search techniques in the style of Escher and Transit [Albarghouthi et al. 2013; Udupa et al. 2013] and show that Blaze results in at least an order of magnitude speedup for both application domains. Finally, we demonstrate the advantages of abstraction refinement by comparing Blaze against a baseline synthesizer that constructs finite tree automata using the DSL's concrete semantics.

Contributions. To summarize, this paper makes the following key contributions:

- We propose a new synthesis methodology based on abstraction refinement. Our methodology reduces the size of the search space by using the abstract semantics of DSL constructs and automatically refines the abstraction whenever the synthesized program is spurious with respect to the input-output examples.

- We introduce *abstract finite tree automata* and show how they can be used in program synthesis.
- We describe a technique for automatically constructing a *proof of incorrectness* of a spurious program and discuss how to use such proofs for abstraction refinement.
- We develop a general synthesis framework called Blaze, which can be instantiated in different domains by providing a suitable DSL with concrete and abstract semantics.
- We instantiate the Blaze framework in two different domains involving string and matrix transformations. Our evaluation shows that Blaze can synthesize non-trivial programs and that it results in significant improvement over existing techniques. Our evaluation also demonstrates the benefits of performing abstraction refinement.

Organization. We first provide some background on finite tree automata (FTA) and review a synthesis algorithm based on FTAs (Section 2). We then introduce *abstract finite tree automata* (Section 3), describe our refinement-based synthesis algorithm (Section 4), and then illustarate the technique using a concrete example (Section 5). The next section explains how to instantiate Blaze in different domains and provides implementaion details. Finally, Section 7 presents our experimental evaluation and Section 8 discusses related work.

2 PRELIMINARIES

In this section, we give background on finite tree automata (FTA) and briefly review (a generalization of) an FTA-based synthesis algorithm proposed in previous work [Wang et al. 2017b].

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 automata by accepting trees rather than strings.

Definition 2.1. **(FTA)** A (bottom-up) finite tree automaton (FTA) over alphabet F is a tuple $\mathcal{A} = (Q, F, Q_f, \Delta)$ where Q is a set of states, $Q_f \subseteq Q$ 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 $q, q_1, \cdots, q_n \in Q$ and $f \in F$.

We assume that every symbol f in alphabet F has an arity (rank) associated with it, 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 some 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 accepted by $\mathcal{A}$.

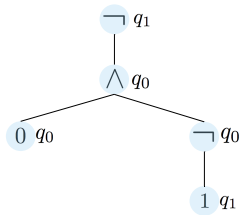


Fig. 2. Tree for ¬(0 ∧ ¬1), annotated with states.

Example 2.2. **(FTA)** Consider the tree automaton $\mathcal{A}$ defined by states $Q = \{q_0, q_1\}$, $F_0 = \{0, 1\}$, $F_1 = \{\neg\}$, $F_2 = \{\wedge\}$, final states $Q_f = \{q_0\}$, and the following transitions Δ:

$$1 \rightarrow q_1 \qquad 0 \rightarrow q_0 \qquad \wedge(q_0, q_0) \rightarrow q_0 \qquad \wedge(q_0, q_1) \rightarrow q_0$$
$$\neg(q_0) \rightarrow q_1 \qquad \neg(q_1) \rightarrow q_0 \qquad \wedge(q_1, q_0) \rightarrow q_0 \qquad \wedge(q_1, q_1) \rightarrow q_1$$

This tree automaton accepts those propositional logic formulas (without variables) that evaluate to *false*. As an example, Fig. 2 shows the tree for formula ¬(0 ∧ ¬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 input to state q_1, which is not a final state.

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

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

Fig. 3. Rules for constructing CFTA $\mathcal{A} = (Q, F, Q_f, \Delta)$ given examples $\vec{e}$ and grammar $G = (T, N, P, s_0)$.

2.2 Synthesis using Concrete Finite Tree Automata

Since our approach builds on a prior synthesis technique that uses finite tree automata, we first review the key ideas underlying the work of Wang et al. [2017b]. However, since that work uses finite tree automata in the specific context of synthesizing data completion scripts, our formulation generalizes their approach to synthesis tasks over a broad class of DSLs.

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

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

- T is a set of terminal symbols, including input variable x. We refer to terminals other than x as *constants*, and use the notation T_C to denote these constants.
- N is a finite set of non-terminal symbols that represent sub-expressions in the DSL.
- P is a set of productions 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 (start symbol) in the grammar.

We can construct the CFTA for examples $\vec{e}$ and grammar G using the rules shown in Fig. 3. 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 and $\vec{c}$ is a vector of concrete values. Intuitively, the existence of a state $q_s^{\vec{c}}$ indicates that symbol s can take concrete 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 concrete values $c_{1j}, \cdots, c_{nj}$ yields c_j. Hence, as mentioned earlier, transitions of the CFTA are constructed using the concrete semantics of the DSL constructs.

We now briefly explain the rules from Fig. 3 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 second rule, labeled Const, adds a state $q_t^{[[\![t]\!], \cdots, [\![t]\!]]}$ for each constant t in the grammar. The next rule, called Final, indicates that $q_{s_0}^{\vec{c}}$ is a final state whenever s_0 is the start symbol in the grammar and $\vec{c}$ is the output examples. The last rule, labeled Prod, generates new CFTA states and transitions for each production $s \to f(s_1, \cdots, s_n)$. 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 $c_{1j}, \cdots, c_{nj}$ yields value c_j, then we also have a state $q_s^{\vec{c}}$ in the CFTA and a transition $f(q_{s_1}^{\vec{c_1}} \cdots, q_{s_n}^{\vec{c_n}}) \to q_s^{\vec{c}}$.

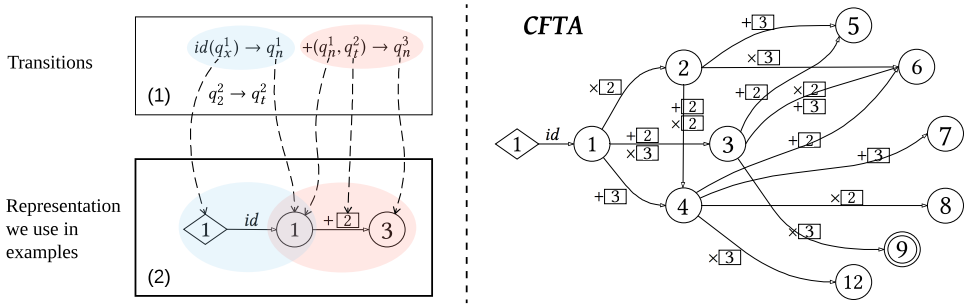


Fig. 4. The CFTA constructed for Example 2.3. We visualize the CFTA as a graph where nodes are labeled with concrete values for symbols. Edges correspond to transitions and are labeled with the operator (i.e., + or ×) followed by the constant operand (i.e., 2 or 3). For example, for the upper two transitions shown in (1) on the left, the graphical representation is shown in (2). Moreover, to make our representation easier to view, we do not include transitions that involve nullary functions in the graph. For instance, the transition $q_2^2 \to q_t^2$ in (1) is not included in (2). A transition of the form $f(q_n^{c_1}, q_t^{c_2}) \to q_n^{c_3}$ is represented by an edge from a node labeled c_1 to another node labeled c_3, and the edge is labeled by f followed by c_2. For instance, the transition $+(q_n^1, q_t^2) \to q_n^3$ in (1) is represented by an edge from 1 to 3 with label +2 in (2).

It can be shown that the language of the CFTA constructed from Fig. 3 is exactly the set of abstract syntax trees (ASTs) of DSL programs that are consistent with the input-output examples. [1] 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 ASTs accepted by the CFTA, one can use heuristics to identify the "best" program that satisfies the input-output examples.

Remark. In general, the tree automata constructed using the rules from Fig. 3 may have infinitely many states. As standard in synthesis literature [Polozov and Gulwani 2015; Solar-Lezama 2008], we therefore assume that the size of programs under consideration should be less than a given bound. In terms of the CFTA construction, this means we only add a state $q_s^{\vec{c}}$ if the size of the smallest tree accepted by the automaton $(Q, F, \{q_s^{\vec{c}}\}, \Delta)$ is lower than the threshold.

Example 2.3. To see how to construct CFTAs, let us consider the following very simple toy DSL, which only contains two constants and allows addition and multiplication by constants:

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

Here, *id* is just the identity function. The CFTA representing the set of all DSL programs with at most two + or × operators for the input-output example $1 \to 9$ is shown in Fig. 4. 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. There is a state q_x^1 since the value of x is 1 in the provided example (Var rule). We construct transitions using the concrete semantics of the DSL constructs (Prod rule). For instance, there is a transition $id(q_x^1) \to q_n^1$ because $id(1)$ yields value 1 for symbol n. Similarly, there is a transition $+(q_n^1, q_t^2) \to q_n^3$ 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 has value 9 for the given example. 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 × operators in the DSL that are consistent with the example $1 \to 9$.

[1] The proof can be found in the extended version of this paper [Wang et al. 2017a].

3 ABSTRACT FINITE TREE AUTOMATA

In this section, we introduce *abstract finite tree automata (AFTA)*, which form the basis of the synthesis algorithm that we will present in Section 4. However, since our approach performs *predicate abstraction* over the concrete values of grammar symbols, we first start by reviewing some requirements on the underlying abstract domain.

3.1 Abstractions

In the previous section, we saw that CFTAs associate a *concrete value* for each grammar symbol by executing the concrete semantics of the DSL on the user-provided inputs. To construct abstract FTAs, we will instead associate an *abstract value* with each symbol. In the rest of the paper, we assume that abstract values are represented as *conjunctions* of predicates of the form $f(s)\ op\ c$, where s is a symbol in the grammar defining the DSL, f is a function, and c is a constant. For example, if symbol s represents an array, then predicate $len(s) > 0$ may indicate that the array is non-empty. Similarly, if s is a matrix, then $rows(s) = 4$ could indicate that s contains exactly 4 rows.

Universe of predicates. As mentioned earlier, our approach is parametrized over a DSL constructed by a *domain expert*. We will assume that the domain expert also specifies a suitable universe $\mathcal{U}$ of predicates that may appear in the abstractions used in our synthesis algorithm. In particular, given a family of functions $\mathcal{F}$, a set of operators O, and a set of constants C specified by the domain expert, the universe $\mathcal{U}$ includes any predicate $f(s)\ op\ c$ where $f \in \mathcal{F}$, $op \in O$, $c \in C$, and s is a grammar symbol. To ensure the completeness of our approach, we require that $\mathcal{F}$ always contains the identity function, O includes equality, and C includes all concrete values that symbols in the grammar 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 the universe of predicates includes *true* and *false*. In the remainder of this paper, we use the notation $\mathcal{U}$ to denote the universe of all possible predicates that can be used in our algorithm.

Notation. Given two abstract values φ_1 and φ_2, we write $\varphi_1 \sqsubseteq \varphi_2$ iff the formula $\varphi_1 \Rightarrow \varphi_2$ is logically valid. As standard in abstract interpretation [Cousot and Cousot 1977], we write $\gamma(\varphi)$ to denote the set of concrete values represented by abstract value φ. Given predicates $\mathcal{P} = \{p_1, \cdots, p_n\} \subseteq \mathcal{U}$ and a formula (abstract value) φ over universe $\mathcal{U}$, we write $\alpha^{\mathcal{P}}(\varphi)$ to denote the strongest conjunction of predicates $p_i \in \mathcal{P}$ 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 $\varphi'_i = \alpha^{\mathcal{P}}(\varphi_i)$.

Abstract semantics. In addition to specifying the universe of predicates, we assume that the domain expert also specifies the abstract semantics of each DSL construct by providing symbolic post-conditions over the universe of predicates $\mathcal{U}$. We represent the abstract semantics for a production $s \rightarrow f(s_1, \cdots, s_n)$ using the notation $[\![f(\varphi_1, \cdots, \varphi_n)]\!]^\sharp$. That is, given abstract values $\varphi_1, \cdots, \varphi_n$ for the argument symbols $s_1, \cdots, s_n$, the abstract 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.:

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

However, in general, we do not require the abstract transformers to be *precise*. That is, if we have $[\![f(\varphi_1, \cdots, \varphi_n)]\!]^\sharp = \varphi$ and S is the set containing $[\![f(c_1, \cdots, c_n)]\!]$ for every $c_i \in \gamma(\varphi_i)$, then it is possible that $\varphi \sqsupseteq \alpha^{\mathcal{U}}(S)$. In other words, we allow each abstract transformer to produce an abstract value that is weaker (coarser) than the value produced by the most precise transformer over the given abstract domain. We do not require the abstract semantics to be precise because it may be cumbersome to define the most precise abstract transformer for some DSL constructs. On

$$\frac{\vec{\varphi} = \alpha^{\mathcal{P}}(\lfloor x = \vec{e}_{in,1}, \cdots, x = \vec{e}_{in,|\vec{e}|}\rfloor)}{q_x^{\vec{\varphi}} \in Q} \text{ (Var)} \qquad \frac{t \in T_C \quad \vec{\varphi} = \alpha^{\mathcal{P}}(\lfloor t = [\![t]\!], \cdots, t = [\![t]\!]\rfloor) \quad |\vec{\varphi}| = |\vec{e}|}{q_t^{\vec{\varphi}} \in Q} \text{ (Const)}$$

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

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

Fig. 5. Rules for constructing AFTA $\mathcal{A} = (Q, F, Q_f, \Delta)$ given examples $\vec{e}$, grammar $G = (T, N, P, s_0)$ and a set of predicates $\mathcal{P} \subseteq \mathcal{U}$.

the other hand, we 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 easily implemented using the concrete semantics:

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

Example 3.1. Consider the same DSL that we used in Example 2.3 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 as follows:

$[\![id(x = c)]\!]^\sharp := (n = c)$

$[\![(n = c_1) + (t = c_2)]\!]^\sharp := (n = (c_1 + c_2))$

$[\![(n = c_1) \times (t = c_2)]\!]^\sharp := (n = c_1 c_2)$

$$[\![(0 < n \leq 4) + (t = c)]\!]^\sharp := \begin{cases} 0 < n \leq 4 & c = 0 \\ 0 < n \leq 8 & 0 < c \leq 4 \\ \textit{true} & \text{otherwise} \end{cases}$$

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

$$[\![(0 < n \leq 8) + (t = c)]\!]^\sharp := \begin{cases} 0 < n \leq 8 & c = 0 \\ \textit{true} & \text{otherwise} \end{cases}$$

$$[\![(0 < n \leq 8) \times (t = c)]\!]^\sharp := \begin{cases} 0 < n \leq 8 & c = 1 \\ \textit{true} & \text{otherwise} \end{cases}$$

$[\![(\bigwedge_i p_i) \diamond (\bigwedge_j p_j)]\!]^\sharp := \bigsqcap_i \bigsqcap_j [\![p_i \diamond p_j]\!]^\sharp$ $\quad \diamond \in \{+, \times\}$

In addition, the abstract transformer returns *true* if any of its arguments is *true*.

3.2 Abstract Finite Tree Automata

As mentioned earlier, abstract finite tree automata (AFTA) generalize concrete FTAs 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 symbol in the grammar and $\vec{\varphi}$ is a vector of abstract values. If there is a transition $f(q_{s_1}^{\vec{\varphi}_1}, \cdots, q_{s_n}^{\vec{\varphi}_n}) \to q_s^{\vec{\varphi}}$ in the AFTA, it is always the case that $[\![f(\varphi_{1j}, \cdots, \varphi_{nj})]\!]^\sharp \sqsubseteq \varphi_j$. Since our abstract transformers are sound, this means that φ_j overapproximates 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 Fig. 5. Similar to CFTAs, our construction requires the set of input-output examples $\vec{e}$ as well as the grammar $G = (T, N, P, s_0)$ defining the DSL. In addition, the AFTA construction requires the abstract semantics of the 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.

The first two rules from Fig. 5 are very similar to their counterparts from the CFTA construction rules: 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 predicates $\mathcal{P}$. Similarly, the Const rules states that $q_t^{\vec{\varphi}} \in Q$ whenever t is a constant (terminal) in the grammar and $\vec{\varphi}$ is the abstraction of $[t = [\![t]\!], \cdots, t = [\![t]\!]]$ with respect to predicates $\mathcal{P}$. The next rule, labeled Final in Fig. 5, 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 state whenever the concretization of $\vec{\varphi}$ includes the output examples.

The last rule, labeled Prod, deals with grammar productions of the form $s \longrightarrow f(s_1, \cdots, s_n)$. Suppose that the AFTA contains states $q_{s_1}^{\vec{\varphi}_1}, \cdots, q_{s_n}^{\vec{\varphi}_n}$, which, intuitively, means that symbols $s_1, \cdots, s_n$ can take abstract values $\vec{\varphi}_1, \cdots, \vec{\varphi}_n$. In the Prod 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 $\mathcal{U}$. However, since the set of predicates $\mathcal{P}$ may be a strict subset of the universe $\mathcal{U}$, $[\![f(\varphi_{1j}, \cdots, \varphi_{nj})]\!]^\sharp$ may not be a valid abstract value with respect to predicates $\mathcal{P}$. Hence, we apply 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 $[\![f(\varphi_{1j}, \cdots, \varphi_{nj})]\!]^\sharp$. Since symbol s in the grammar can take abstract value $\vec{\varphi}$, we add the state $q_s^{\vec{\varphi}}$ to the AFTA, as well as the transition $f(q_{s_1}^{\vec{\varphi}_1}, \cdots, q_{s_n}^{\vec{\varphi}_n}) \longrightarrow q_s^{\vec{\varphi}}$.

Example 3.2. Consider the same DSL that we used in Example 2.3 as well as the universe and abstract transformers given in Example 3.1. Now, let us consider the set of predicates $\mathcal{P} = \{true, t = 2, t = 3, x = c\}$ where c stands for any integer value. Fig. 6 shows the AFTA constructed for the input-output example $1 \longrightarrow 9$ 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 $id(q_x^{x=1}) \longrightarrow q_n^{true}$, where q_n^{true} is abbreviated as *true* in Fig. 6. The AFTA also includes transitions $+(q_n^{true}, t = c) \longrightarrow q_n^{true}$ and $\times(q_n^{true}, t = c) \longrightarrow q_n^{true}$ 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). Thus, the language of this AFTA includes all programs that start with $id(x)$.

Theorem 3.3. **(Soundness of AFTA)** *Let $\mathcal{A}$ be the AFTA constructed for examples $\vec{e}$ and grammar G using the abstraction defined by finite set of predicates $\mathcal{P}$ (including* true*). If Π is a program that is consistent with examples $\vec{e}$, then Π is accepted by $\mathcal{A}$.*

Proof. The proof can be found in the extended version of this paper [Wang et al. 2017a]. □

4 SYNTHESIS USING ABSTRACTION REFINEMENT

We now turn our attention to the top-level synthesis algorithm using abstraction refinement. The key idea underlying our technique is to construct an abstract FTA using a coarse initial abstraction. We then iteratively refine this abstraction and its corresponding AFTA until we either find a program that is consistent with the input-output examples or prove that there is no DSL program that satisfies them. In the remainder of this section, we first explain the top-level synthesis algorithm and then describe the auxiliary procedures in later subsections.

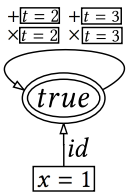


Fig. 6. AFTA in Example 3.2.

4.1 Top-level Synthesis Algorithm

The high-level structure of our refinement-based synthesis algorithm is shown in Fig. 7. The Learn procedure from Fig. 7 takes as input a set of examples $\vec{e}$, a grammar G defining the DSL, an initial set of predicates $\mathcal{P}$, and the universe of all possible predicates $\mathcal{U}$. We implicitly assume that we also have access to the concrete and abstract semantics of the DSL. Also, it is worth noting that the initial set of predicates $\mathcal{P}$ is optional. In cases where the domain expert does not specify $\mathcal{P}$,

the initial abstraction includes *true*, predicates of the form $x = c$ where c is any value that has the same type as x, and predicates of the form $t = [\![t]\!]$ where t is a constant (terminal) in the grammar.

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 of the refinement loop, we first construct an AFTA using the current set of predicates $\mathcal{P}$ (line 3). If the language of the AFTA is empty, we have a proof that there is no DSL program that satisfies the input-output examples; hence, the algorithm returns *null* in this case (line 4). Otherwise, we use a heuristic *ranking algorithm* to choose a "best" program Π that is accepted by the current AFTA $\mathcal{A}$ (line 5). In the remainder of this section, we assume that programs are represented as abstract syntax trees where each node is labeled with the corresponding DSL construct. We do not fix a particular ranking algorithm for Rank, so the synthesizer is free to choose between any number of different ranking heuristics as long as Rank returns a program that has the lowest cost with respect to a deterministic cost metric.

Once we find a program Π accepted by the current AFTA, we run it on the input examples $\vec{e}_{in}$ (line 6). If the result matches the expected outputs $\vec{e}_{out}$, we return Π as a solution of the synthesis algorithm. Otherwise, we refine the current abstraction so that the spurious program Π is no longer 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 construct a *proof of incorrectness* $\mathcal{I}$ of Π with respect to the counterexample e (line 8). In particular, $\mathcal{I}$ is a mapping from the AST nodes in Π to abstract values over universe $\mathcal{U}$ and provides a proof (over the abstract semantics) that program Π is inconsistent with example e. More formally, a proof of incorrectness $\mathcal{I}$ must satisfy the following definition:

Definition 4.1. **(Proof of Incorrectness)** Let Π be the AST of a program that does not satisfy example e. Then, a *proof of incorrectness* of Π with respect to e has the following properties:

(1) If v is a leaf node of Π with label t, then $(t = [\![t]\!]e_{in}) \sqsubseteq \mathcal{I}(v)$.
(2) If v is an internal node with label f and 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 φ, then $e_{out} \notin \gamma(\varphi)$.

Here, the first two properties state that $\mathcal{I}$ constitutes a proof (with respect to the abstract semantics) that executing Π on input e_{in} yields an output that satisfies $\mathcal{I}(root(\Pi))$. The third property states that $\mathcal{I}$ proves that Π is spurious, since e_{out} does not satisfy $\mathcal{I}(root(\Pi))$. The following theorem states that a proof of incorrectness of a spurious program always exists.

Theorem 4.2. **(Existence of Proof)** *Given a spurious program Π that does not satisfy example e, we can always find a proof of incorrectness of Π satisfying the properties from Definition 4.1.*

Proof. The proof can be found in the extended version of this paper [Wang et al. 2017a]. □

Our synthesis algorithm uses such a proof of incorrectness $\mathcal{I}$ to refine the current abstraction. In particular, the predicates that we use in the next iteration include all predicates that appear in $\mathcal{I}$ in addition to the old set of predicates $\mathcal{P}$. Furthermore, as stated by the following theorem, the AFTA constructed in the next iteration is guaranteed to *not accept* the spurious program Π from the current iteration.

Theorem 4.3. **(Progress)** *Let $\mathcal{A}_i$ be the AFTA constructed during the i'th iteration of the* Learn *algorithm from Fig. 7, and let Π_i be a spurious program returned by* Rank*, i.e., Π_i is accepted by $\mathcal{A}_i$ and does not satisfy input-output examples e. Then, we have $\Pi_i \notin \mathcal{L}(\mathcal{A}_{i+1})$ and $\mathcal{L}(\mathcal{A}_{i+1}) \subset \mathcal{L}(\mathcal{A}_i)$.*

Proof. The proof can be found in the extended version of this paper [Wang et al. 2017a]. □

1: **procedure** LEARN($\vec{e}, G, \mathcal{P}, \mathcal{U}$)
input: Input-output examples $\vec{e}$, context-free grammar G, initial predicates $\mathcal{P}$, and universe $\mathcal{U}$.
output: A program consistent with the examples.
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: Π := RANK($\mathcal{A}$);
6: **if** $[\![\Pi]\!]\vec{e}_{in} = \vec{e}_{out}$ **then return** Π;
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: $\mathcal{P} := \mathcal{P} \bigcup$ EXTRACTPREDICATES($\mathcal{I}$);

Fig. 7. The top-level structure of our synthesis algorithm using abstraction refinement.

Example 4.4. Consider the AFTA constructed in Example 3.2, and suppose the program returned by RANK is $id(x)$. Since this program is inconsistent with the input-output example $1 \longrightarrow 9$, our algorithm constructs the proof of incorrectness shown in Fig. 8. In particular, the proof labels the root node of the AST with the new abstract value $0 < n \leq 8$, which establishes that $id(x)$ is spurious because $9 \notin \gamma(0 < n \leq 8)$. In the next iteration, we add $0 < n \leq 8$ to our set of predicates $\mathcal{P}$ and construct the new AFTA shown in Fig. 8. Observe that the spurious program $id(x)$ is no longer accepted by the refined AFTA.

THEOREM 4.5. **(Soundness and Completeness)** *If there exists a DSL program that satisfies the input-output examples $\vec{e}$, then the* LEARN *procedure from Fig. 7 will return a program Π such that $[\![\Pi]\!]\vec{e}_{\text{in}} = \vec{e}_{\text{out}}$.*

PROOF. The proof can be found in the extended version of this paper [Wang et al. 2017a]. □

4.2 Constructing Proofs of Incorrectness

In the previous subsection, we saw how proofs of incorrectness are used to rule out spurious programs from the search space (i.e., language of the AFTA). We now discuss how to automatically construct such proofs given a spurious program.

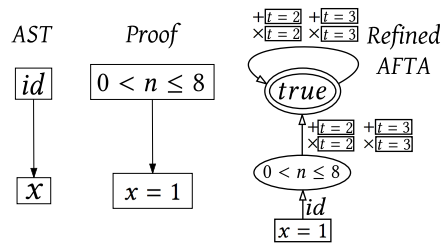


Fig. 8. Proof of incorrectness for Example 3.2.

Our algorithm for constructing a proof of incorrectness is shown in Fig. 9. The CONSTRUCTPROOF procedure 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 defined by predicates $\mathcal{P}$ as well as the universe of all predicates $\mathcal{U}$. The output of this procedure is a mapping from the vertices 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 the root node such that $e_{out} \notin \gamma(\mathcal{I}(r))$. In other words, the annotation $\mathcal{I}(r)$ is sufficient for showing that Π is spurious (property (3) from Definition 4.1). 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 $\mathcal{I}$ satisfies properties (1) and (2) from Definition 4.1.

1: **procedure** ConstructProof($\Pi, e, \mathcal{P}, \mathcal{U}$)

input: A spurious program Π represented as an AST with vertices V.
input: A counterexample e such that $[\![\Pi]\!]e_{in} \neq e_{out}$.
input: Current set of predicates $\mathcal{P}$ and the universe of predicates $\mathcal{U}$.
output: A proof $\mathcal{I}$ of incorrectness of Π represented as 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: φ :=EvalAbstract($\Pi, e_{in}, \mathcal{P}$);
3: ψ :=StrengthenRoot$\big(s_0 = [\![\Pi]\!]e_{in}, \varphi, s_0 \neq e_{out}, \mathcal{U}\big)$;
4: $\mathcal{I}(\text{root}(\Pi)) := \varphi \wedge \psi$;

▷ Process all nodes other than root.

5: *worklist* := $\{\text{root}(\Pi)\}$;
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*.remove();
8: $\vec{\Pi}$:= ChildrenASTs(*cur*);
9: $\vec{\phi} := \big[s_i = c_i \;\big|\; c_i = [\![\Pi_i]\!]e_{in}, i \in [1, |\vec{\Pi}|], s_i = \text{Symbol}(\Pi_i) \big]$;
10: $\vec{\varphi} := \big[\varphi_i \;\big|\; \varphi_i = \text{EvalAbstract}(\Pi_i, e_{in}, \mathcal{P}), i \in [1, |\vec{\Pi}|] \big]$;
11: $\vec{\psi}$:= StrengthenChildren$\big(\vec{\phi}, \vec{\varphi}, \mathcal{I}(cur), \mathcal{U}, \text{label}(cur)\big)$;
12: **for** $i = 1, \cdots, |\vec{\Pi}|$ **do**
13: $\mathcal{I}(\text{root}(\Pi_i)) := \varphi_i \wedge \psi_i$;
14: **if** $\neg$IsLeaf(root(Π_i)) **then**
15: *worklist*.add(root(Π_i));
16: **return** $\mathcal{I}$;

Fig. 9. Algorithm for constructing proof of incorrectness of Π with respect to example e. In the algorithm, ChildrenASTs(v) returns the sub-ASTs rooted at the children of v. The function Symbol(Π) yields the grammar symbol for the root node of Π.

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

Fig. 10. Definition of auxiliary EvalAbstract procedure used in ComputeProof algorithm from Fig. 9. Node($f, \vec{\Pi}$) represents an internal node with label f and subtrees $\vec{\Pi}$.

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 domain defined by predicates $\mathcal{P}$. Towards this goal, we use a procedure called EvalAbstract, shown in Fig. 10, which symbolically executes Π on e_{in} using 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) $(s_0 = [\![\Pi]\!]e_{in}) \Rightarrow \psi$ 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.

1: **procedure** StrengthenRoot(p_+, p_-, φ, $\mathcal{U}$)

input: Predicates p_+ and p_-, formula φ, and universe $\mathcal{U}$.
output: Formula ψ^* 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** ψ^*;

Fig. 11. Algorithm for finding a strengthening for the root.

1: **procedure** StrengthenChildren($\vec{\phi}$, $\vec{\varphi}$, φ_p, $\mathcal{U}$, f)

input: Predicates $\vec{\phi}$, formulas $\vec{\varphi}$, formula φ_p, and universe $\mathcal{U}$.
output: Formulas $\vec{\psi}^*$ 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: $\vec{\Phi} := \left[\Phi_i \mid \Phi_i = \{p \in \mathcal{U} \mid \phi_i \Rightarrow p\}\right]$; $\vec{\Psi} := \vec{\Phi}$ ▷ 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** $\vec{\psi}^*$;

Fig. 12. Algorithm for finding a strengthening for nodes other than the root.

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 $s_0 = [\![\Pi]\!]e_{in}$ 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 $s_0 = [\![\Pi]\!]e_{in}$ 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 Fig. 11. In a nutshell, 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 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 has the property that *cur* is 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 4.1. To find the annotation for each child v_i of *cur*, we first compute the concrete and abstract values (ϕ_i and φ_i from lines 9–10) associated with each v_i. We then invoke the StrengthenChildren procedure, shown in Fig. 12, to find a strengthening $\vec{\psi}$ such that:

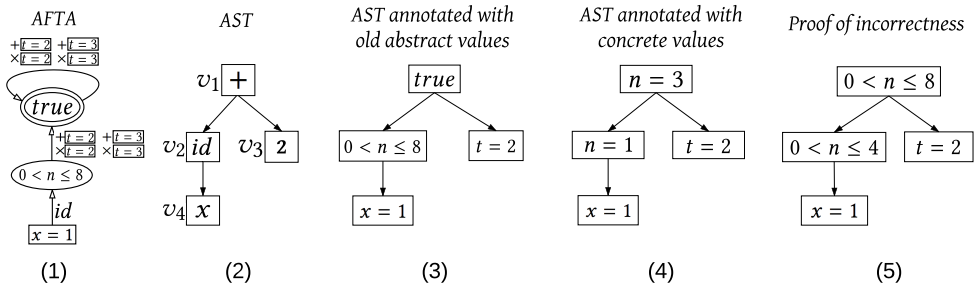


Fig. 13. Illustration of the proof construction process for Example 4.6.

(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 4.1. In other words, the first condition says that our strengthening overapproximates the concrete output of subprogram Π_i rooted at v_i on input e_{in}. The second condition enforces property (2) from Definition 4.1. In particular, it says that the annotation for the parent node is provable from the annotations of the children using the abstract semantics of the DSL constructs.

In addition to satisfying these afore-mentioned properties, the strengthening $\vec{\psi}$ returned by StrengthenChildren has some useful generality guarantees. In particular, the return value of the function 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 the ranking algorithm.

Example 4.6. To better understand how we construct proofs of incorrectness, consider the AFTA shown in Fig. 13(1). Suppose that the ranking algorithm returns the program $id(x) + 2$, which is clearly spurious with respect to the input-output example $1 \rightarrow 9$. Fig. 13(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 from Fig. 13(3) correspond to the results of EvalAbstract in the ConstructProof algorithm from Fig. 9. 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 value in the 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 Fig. 9, 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 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 $\psi_2 = true$. Thus, 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 Fig. 13(5).

Theorem 4.7. **(Correctness of Proof)** *The mapping $\mathcal{I}$ returned by the* ConstructProof *procedure satisfies the properties from Definition 4.1.*

Proof. The proof can be found in the extended version of this paper [Wang et al. 2017a]. □

Complexity analysis. The complexity of our synthesis algorithm is mainly determined by the number of iterations, and the complexity of FTA construction, ranking and proof construction. In particular, the FTA can be constructed in time $O(m)$ where m is the size of the resulting FTA[2] (before any pruning). The complexity of performing ranking over an FTA depends on the ranking heuristic. For the one used in our implementation (see Section 6.1), the time complexity is $O(m \cdot \log d)$ 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 $O(l \cdot p)$ where l is the number of nodes in the AST and p is the number of conjunctions under consideration. Therefore, the complexity of our synthesis algorithm is given by $O(t \cdot (l \cdot p + m \cdot \log d))$ where t is the number of iterations of the abstraction refinement process.

5 A WORKING EXAMPLE

In the previous sections, we illustrated various aspects of our synthesis algorithm using the DSL from Example 2.3 on the 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, and we use the initial set of predicates $\mathcal{P}$ given in Example 3.2. We will assume that the ranking algorithm always favors smaller programs over larger ones. In the case of a tie, the ranking algorithm favors programs that use + and those that use smaller constants.

Fig. 14 illustrates all iterations of the synthesis algorithm until we find the desired program. Let us now consider Fig. 14 in more detail.

Iteration 1. As explained in Example 3.2, 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 this program does not satisfy the example $1 \mapsto 9$, we construct a proof of incorrectness $\mathcal{I}_1$, which introduces a new abstract value $0 < n \leq 8$ in our set of predicates.

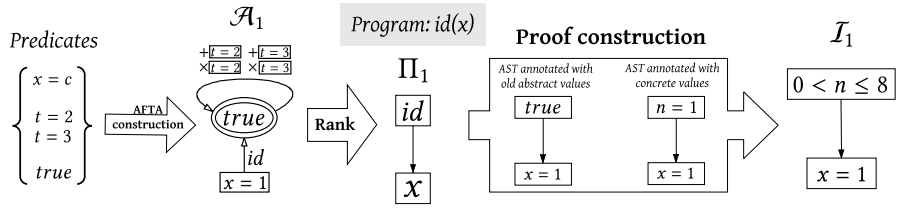
Iteration 2. During the second iteration, we construct the AFTA labeled as $\mathcal{A}_2$ in Fig. 14, which contains a new state $0 < n \leq 8$. While $\mathcal{A}_2$ no longer accepts the program $id(x)$, it does accept the spurious program $\Pi_2 = id(x) + 2$, which is returned by the ranking algorithm. Then we construct the proof of incorrectness 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 $\mathcal{A}_3$. 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 proof of incorrectness of Π_3, we now obtain a new predicate $n = 1$.

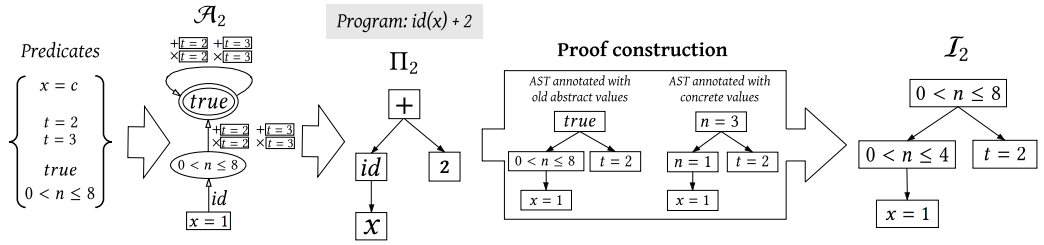
Iteration 4. In the final iteration, we construct the AFTA labeled as $\mathcal{A}_4$, which rules out all programs containg a single operator (+ or ×) as well as 12 programs that use two operators. When we run the ranking algorithm on $\mathcal{A}_4$, we obtain the candidate program $(id(x) + 2) \times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

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 Section 2.2 *always* explores a single program, but the corresponding finite tree automaton may be *very* large.

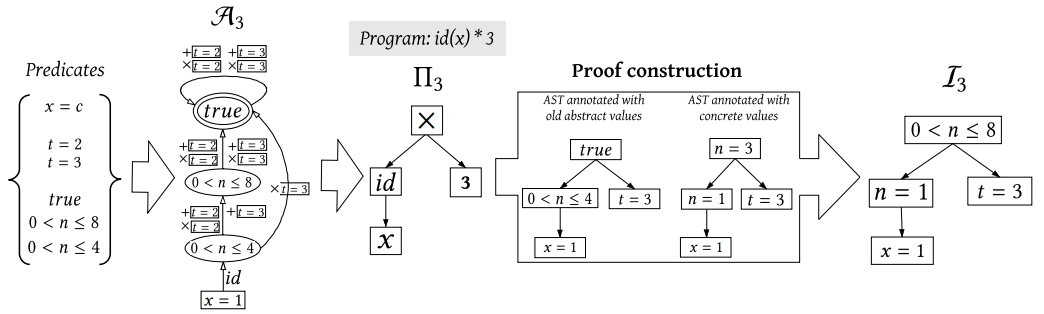
[2]FTA size is defined to be $\sum_{\delta \in \Delta} |\delta|$ where $|\delta| = n + 1$ for a transition δ of the form $f(q_1, \cdots, q_n) \to q$.



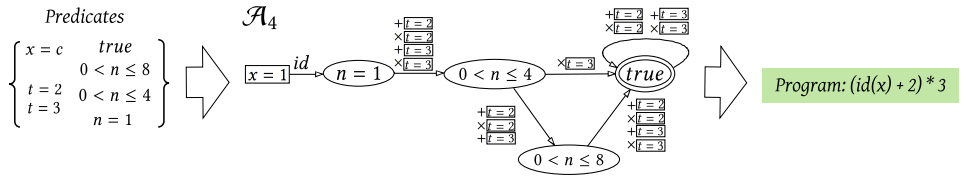
Iteration 1: The constructed AFTA is $\mathcal{A}_1$, Rank returns Π_1, Π_1 is spurious, and the proof of incorrectness is $\mathcal{I}_1$.



Iteration 2: The constructed AFTA is $\mathcal{A}_2$, Rank returns Π_2, Π_2 is spurious, and the proof of incorrectness is $\mathcal{I}_2$.



Iteration 3: The constructed AFTA is $\mathcal{A}_3$, Rank returns Π_3, Π_3 is spurious, and the proof of incorrectness is $\mathcal{I}_3$.



Iteration 4: The constructed AFTA is $\mathcal{A}_4$, and Rank returns the desired program.

Fig. 14. Illustration of the synthesis algorithm.

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.

6 IMPLEMENTATION AND INSTANTIATIONS

We have implemented the synthesis algorithm proposed in this paper in a framework called Blaze, written in Java. Blaze is parametrized over a DSL and its abstract semantics. We have also

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

Fig. 15. DSL for string transformations where τ represents a token, k is an integer, and s is a string constant.

instantiated Blaze for two different domains, string transformation and matrix reshaping. In what follows, we describe our implementation of the Blaze framework and its instantiations.

6.1 Implementation of Blaze Framework

Our implementation of the Blaze framework consists of three main modules, namely FTA construction, ranking algorithm, and proof generation. Since our implementation of FTA construction and proof generation mostly 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:

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

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

In our implementation, we identify a minimum-cost AST accepted by a finite tree automaton using the algorithm presented by Gallo et al. [1993] 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 $f(q_1, \cdots, q_n) \rightarrow q$ represents a B-arc $(\{q_1, \cdots, q_n\}, \{q\})$ where the weight of the arc is given by the cost of DSL operator f. We also add a dummy node r to the hypergraph and an edge with weight $\mathit{cost}(s)$ 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.

6.2 Instantiating Blaze for String Transformations

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

Domain-specific language. Since there is significant prior work on automating string transformations using PBE [Gulwani 2011; Polozov and Gulwani 2015; Singh 2016], we directly adopt the DSL presented by Singh [2016] as shown in Fig. 15. 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. 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 $\mathit{len}(s) = i$, where s is a

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

Fig. 16. Abstract semantics for the DSL shown in Fig. 15.

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

Fig. 17. DSL for matrix transformations where k is an integer.

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 Section 3 that our universe must include predicates of the form $s = c$, where c is a concrete value that symbol s can take. Hence, our universe of predicates for the string domain is given by:

$$\mathcal{U} = \{len(s) = i \mid i \in \mathbb{N}\} \cup \{s[i] = c \mid i \in \mathbb{N}, c \in \texttt{Char}\} \cup \{s = c \mid c \in \texttt{Type}(s)\} \cup \{true, false\}$$

Abstract semantics. Recall from Section 3 that the DSL designer must provide an abstract transformer $[\![f(\varphi_1, \cdots, \varphi_n)]\!]^\sharp$ for each grammar production $s \rightarrow f(s_1, \cdots, s_n)$ 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:

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

Hence, instead of defining a transformer for every possible abstract value (which may have arbitrarily many conjuncts), it suffices to define an abstract transformer for every combination of atomic predicates. We show the abstract transformers for all possible combinations of atomic predicates in Fig. 16.

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 predicates in the default initial abstraction (see Section 4.1 for the definition).

6.3 Instantiating Blaze for Matrix and Tensor Transformations

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.[3] We believe this application domain is a good stress test for the Blaze framework because (a) tensors are complex data structures which makes the search space larger, and (b) 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 by providing a suitable DSL and its abstract semantics.

[3]Tensors are generalization of matrices from 2 dimensions to an arbitrary number of dimensions.

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

Fig. 18. Abstract semantics the DSL shown in Fig. 17.

Domain-specific language. Our DSL for the tensor domain is inspired by existing MATLAB functions and is shown in Fig. 17. In this DSL, tensor operators include `Reshape`, `Permute`, `Fliplr`, and `Flipud` and correspond to their namesakes in MATLAB[4]. For example, $\texttt{Reshape}(t, v)$ takes a tensor t and a size vector v and reshapes t so that its dimension becomes v. Similarly, $\texttt{Permute}(t, v)$ rearranges the dimensions of tensor t so that they are in the order specified by vector v. Next, $\texttt{fliplr}(t)$ returns tensor t with its columns 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 6.1. 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[5]. 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 string domain, 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. Thus, the universe of predicates is given by:

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

Abstract semantics. The abstract transformers for all possible combinations of atomic predicates for the DSL constructs are given in Fig. 18. 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. We use the default initial abstraction (see Section 4.1 for the definition).

[4]See the MATLAB documentation https://www.mathworks.com/help/matlab/ref/x.html where x refers to the name of the corresponding function.
[5]StackOverflow post link: https://stackoverflow.com/questions/16592386/reshape-matlab-vector-in-row-wise-manner.

7 EVALUATION

We evaluate Blaze by using it to automate string and matrix manipulation tasks collected from on-line forums and existing PBE benchmarks. The goal of our evaluation is to answer the following questions:

- **Q1:** How does Blaze perform on different synthesis tasks from the string and matrix domains?
- **Q2:** How many refinement steps does Blaze take to find the correct program?
- **Q3:** What percentage of its running time does Blaze spend in FTA vs. proof construction?
- **Q4:** How does Blaze compare with existing synthesis techniques?
- **Q5:** What is the benefit of performing abstraction refinement in practice?

7.1 Results for the String Domain

In our first experiment, we evaluate Blaze on *all* 108 string manipulation benchmarks from the PBE track of the SyGuS competition [Alur et al. 2015]. We believe that the string domain is a good testbed for evaluating Blaze because of the existence of mature tools like FlashFill [Gulwani 2011] and the presence of a SyGuS benchmark suite for string transformations.

Benchmark information. Among the 108 SyGuS benchmarks related to string transformations, 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 evaluate Blaze using the string manipulation DSL shown in Fig. 15 and the predicates and abstract semantics from Section 6.2. For each benchmark, we provide Blaze with all input-output examples at the same time.[6] 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 VSA data structure to compactly represent the search space.
- **ENUM-EQ:** This technique based on enumerative search has been adopted to solve different kinds of synthesis problems [Albarghouthi et al. 2013; Alur et al. 2015; Cheung et al. 2012; Udupa et al. 2013]. It enumerates programs according to their size, groups them into equivalence classes based on their (concrete) input-output behavior to compress the search space, and returns the first program that is consistent with the examples.
- **CFTA:** This is an implementation of the synthesis algorithm presented in Section 2. It uses the concrete semantics of the DSL operators to construct an FTA whose language is exactly the set of programs that are consistent with the input-output examples.

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 that uses the same DSL.

Because the baseline techniques mentioned above perform *much better* when the examples are provided in an interactive fashion [7], 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

[6] However, Blaze typically uses a fraction of these examples when performing abstraction refinement.

[7] 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.

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

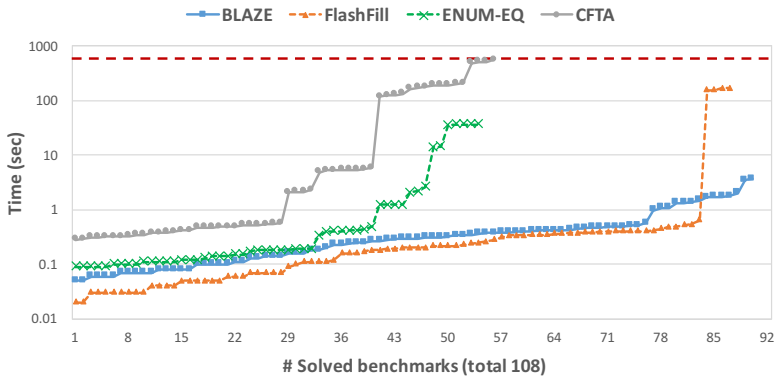
Fig. 19. Blaze results for the string domain, where $\lvert\vec{e}\rvert$ shows the number of examples and T_{syn} gives synthesis time in seconds. The next columns labeled T_x show the time for FTA construction, ranking, proof construction, and all remaining parts (e.g. FTA minimization). *#Iters* shows the number of refinement steps, and $\lvert Q_{final}\rvert$ and $\lvert\Delta_{final}\rvert$ show the number of states and transitions in the final AFTA. The last column labeled $\lvert\Pi_{syn}\rvert$ shows the size of the synthesized program (measured by number of AST nodes). The timeout is set to be 10 minutes.

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. Fig. 19 summarizes the results of our evaluation of Blaze in the string domain. 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 under 4 seconds, with a median running time of 0.14 seconds. In comparison, the best solver, i.e., EUSolver [Alur et al. 2017], in the SyGuS'16 competition is able to solve in total 45 benchmarks within the timeout of 60 minutes [Alur et al. 2016].

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. Fig. 20 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



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

Fig. 20. Comparison with existing techniques. A data point (X, Y) means that X benchmarks are solved within a maximum running time of Y seconds (per benchmark). The timeout is set to be 10 minutes.

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 of Blaze 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.

Finally, Blaze also compares favorably with FlashFill, the state-of-the-art technique for automating string transformation tasks. In particular, Blaze achieves very competitive performance for the benchmarks that both techniques can solve. Furthermore, Blaze can solve 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.

7.2 Results for the Matrix Domain

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

To perform our evaluation, we collected 39 benchmarks from two on-line forums, namely StackOverflow and MathWorks.[8] Our benchmarks were collected using the following methodology:

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

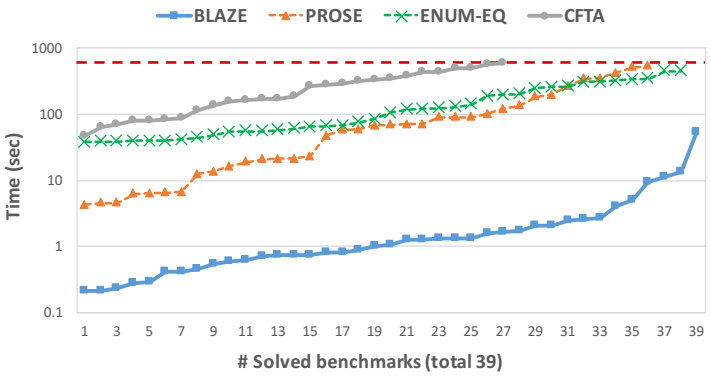
Benchmark	T_{syn} (sec)	$T_{\mathcal{A}}$	T_{rank}	$T_{\mathcal{I}}$	T_{other}	*#Iters*	$\lvert Q_{final}\rvert$	$\lvert \Delta_{final}\rvert$	$\lvert \Pi_{syn}\rvert$
stackoverflow-1	0.29	0.14	0.02	0.08	0.05	39	125	993	10
stackoverflow-2	2.74	0.86	0.10	1.52	0.26	319	279	4483	22
stackoverflow-3	0.72	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

Fig. 21. Blaze results for matrix domain. We use the same notation explained in the caption of Fig. 19.

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 as well as 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 640. The average number is 73.5, and the median is 36. Among all benchmarks, 29 involve transforming the input example into tensors of dimension great than 2.

Experimental setup. We evaluate Blaze using the DSL shown in Fig. 17 and the abstract semantics presented in Section 6.3. Similar to the string domain, we also compare Blaze with ENUM-EQ and CFTA. However, since there is no existing domain-specific synthesizer for automating matrix transformation tasks, we implemented a specialized VSA-based synthesizer for our matrix domain by instantiating the Prose framework [Polozov and Gulwani 2015]. In particular, we provide precise witness functions for all the operators in our DSL, which allows Prose to effectively decompose the synthesis task. 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 domain. The timeout is set to be 10 minutes.



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

Fig. 22. Comparison with existing techniques.

Blaze results. The results of our evaluation on Blaze are summarized in Fig. 21. As shown in the figure, 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 matrix/tensor reshaping tasks.

Looking at Fig. 21 in more detail, Blaze takes an average of 165 refinement steps to find a correct program. Unlike the string domain where Blaze spends most of its time in FTA construction, proof construction also seems to take significant time in the matrix domain. We conjecture this is because Blaze needs to search for predicates in a large space. The final AFTA constructed by Blaze contains an average of 205 states, and the average AST size of synthesized programs is 16.

Comparison. As shown in Fig. 22, 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, a 115x improvement over ENUM-EQ, and a 90x improvement over Prose in terms of the running time. Therefore, this experiment also demonstrates the advantage of using abstract values and abstraction refinement in the matrix domain.

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

7.3 Discussion

The reader may wonder why Blaze performs much better in the matrix domain compared to VSA-based techniques (FlashFill and Prose) than in the string 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 matrix domain, the search space size is mostly determined by the number of intermediate matrices; in the worst case Prose would have to explore $O(n!)$ nodes, where n is the number of entries in the example matrix. Hence, the size of the search space in the matrix 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.

8 RELATED WORK

In this section, we compare our technique 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 in software model checking [Ball et al. 2011; Beyer et al. 2007; Henzinger et al. 2004, 2003]. The idea here 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 [Henzinger et al. 2004]. Our approach is very similar to CEGAR-based model checkers in that we perform abstraction refinement whenever we find a *spurious program* as opposed to a spurious error trace. In addition, the proofs of incorrectness that we utilize in this paper can be viewed as a form of *tree interpolant* [McMillan and Rybalchenko 2013; Rümmer et al. 2013].

Abstraction in Program Synthesis. The only prior work that uses abstraction refinement in the context of synthesis is the *abstraction-guided synthesis* (AGS) technique of Vechev et al. for learning efficient synchronization in concurrent programs [Vechev et al. 2010]. Unlike Blaze which aims to learn an entire program from scratch using input-output examples, AGS requires an input concurrent program and only performs small modifications to the program by adding synchronization primitives. In more detail, AGS first performs an abstraction of the program and 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 abstract interleaving is removed. AGS can be viewed as a program repair technique for concurrent programs and cannot be used for synthesizing programs from input-output examples.

Other synthesizers that bear similarities to the approach proposed in this paper include Synquid [Polikarpova et al. 2016] and Morpheus [Feng et al. 2017]. 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 DSL constructs, which can be viewed as specifications. However, unlike Synquid, the specifications in Blaze and Morpheus overapproximate the behavior of the DSL constructs. Blaze further differs from both of these 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 [Gascón et al. 2017; Tiwari et al. 2015]. 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 the completeness. In contrast to these techniques, our method uses abstractions to construct a more compact abstract FTA and performs abstraction refinement to rule out spurious programs.

The use of abstraction refinement has also been explored in the context of superoptimizing compilers [Phothilimthana et al. 2016]. In particular, Phothilimthana et al. use test cases to construct an (underapproximate) abstraction of program behavior and "refine" this abstraction by iteratively including more test cases. However, since this abstraction is heuristically applied to "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 heuristic-guided manner.

Another related technique is Storyboard Programming [Singh and Solar-Lezama 2011] 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 [Shaw et al. 1975]. 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 [Gulwani 2011; Singh 2016], data structure manipulations [Feser et al. 2015; Yaghmazadeh et al. 2016], network policies [Yuan et al. 2014], data filtering [Wang et al. 2016], file manipulations [Gulwani et al. 2015], interactive parser synthesis [Leung et al. 2015], and synthesizing map-reduce distributed programs [Smith and Albarghouthi 2016]. It has also been studied from different perspectives, such as type-theoretic interpretation [Frankle et al. 2016; Osera and Zdancewic 2015; Scherer and Rémy 2015], version space learning [Gulwani 2011; Polozov and Gulwani 2015], and deep learning [Devlin et al. 2017; Parisotto et al. 2016].

Our method, SYNGAR, 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 [Albarghouthi et al. 2013; Udupa et al. 2013], SYNGAR, instead, uses their abstract semantics and iteratively refines the abstraction until it finds a program that satisfies the input-output examples. Although we instantiate SYNGAR in two domains, namely string and matrix transformations, we believe the SYNGAR approach can be used to complement many previous PBE systems to make synthesis more efficient.

Counterexample-guided Inductive Synthesis (CEGIS). Counterexample guided inductive synthesis (CEGIS) [Solar-Lezama 2008; Solar-Lezama et al. 2006] is a popular algorithm used for solving synthesis problems of the form $\exists P\ \forall i : \phi(P, i)$ where the goal is to synthesize a program P such that the specification ϕ holds for all inputs i. The main idea of the algorithm is to reduce the solving of the second-order formula to two first-order formulas: i) $\exists P : \phi(P, I_1, \cdots, I_k)$ (synthesis), and ii) $\exists i : \neg\phi(P, i)$ (verification). The first phase learns a program P that is consistent with a finite set of inputs $(I_1, \cdots, I_k)$, whereas the second phase performs verification on the learnt candidate program P to find a counterexample input i that violates the specification. If such an input i exists, the input 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 learnt program P satisfies the specification) or if the synthesis check fails (i.e., there is no program that satisfies the specification).

CEGIS bears similarities to SYNGAR in that both approaches are guided by counterexamples (i.e., incorrect programs). However, the two approaches are very different in that CEGIS performs abstraction over the specification, whereas SYNGAR performs abstraction over the DSL constructs. In particular, the input-output examples used in the synthesis phase of CEGIS *under-approximate* the specification, whereas the abstract finite tree automata in SYNGAR *over-approximate* the set of programs that are consistent with the specification. Since SYNGAR is intended for example-guided synthesis, we believe that it can be used to complement the synthesis phase in CEGIS.

Finite Tree Automata (FTA). Tree automata, which generalize finite word automata, date back to 1968 and were originally used for proving the existence of a decision procedure for weak monadic second-order logic [Thatcher and Wright 1968]. Since then, tree automata have been found applications in the analysis of XML documents [Hosoya and Pierce 2003; Martens and Niehren 2005], software verification [Abdulla et al. 2008; Gallagher and Puebla 2002; Kafle and

Gallagher 2015; Monniaux 1999] and natural language processing [Knight and May 2009; May and Knight 2008]. 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 [Gallagher and Puebla 2002]. In contrast to their approach, we use finite tree automata for synthesis rather than for refinement.

Finite tree automata have also found interesting applications in the context of program synthesis. For example, Parthasarathy uses finite tree automata as a theoretical basis for reactive synthesis [Madhusudan 2011]. Specifically, given an ω-specification of the reactive system, their technique constructs a tree automaton that accepts all programs that meet the specification. Recent work by Wang et al. also uses finite tree automata for synthesizing data completion scripts from input-output examples [Wang et al. 2017b]. In this work, we generalize the technique of Wang et al. by showing how it can be used to synthesize programs over any arbitrary DSL described in a context-free grammar. We also introduce the concept of abstract finite tree automata (AFTA) and describe a method for counterexample-guided synthesis using AFTAs.

9 CONCLUSION

We proposed a new synthesis methodology, called SYNGAR, for synthesizing programs that are consistent with a given set of input-output examples. The key idea is to find a program that satisfies the examples according to the abstract semantics of the DSL constructs and then check the correctness of the learnt program with respect to the concrete semantics. Our approach returns the synthesized program if it satisfies the examples, and automatically refines the abstraction otherwise. Our method performs synthesis using abstract finite tree automata and refines the abstraction by constructing a proof of incorrectness of the learnt program for a given input-output example.

We have implemented the proposed methodology in a synthesis framework called Blaze and instantiated it for two different domains, namely string and matrix transformations. Our evaluation shows that Blaze is competitive with FlashFill in the string domain and that it outperforms Prose by 90x in the matrix domain. Our evaluation also shows the advantages of using abstract semantics and performing abstraction refinement.

10 LIMITATIONS AND FUTURE WORK

While our proposed SYNGAR framework can be realized using different synthesis algorithms, the FTA-based method described in this paper has three key limitations: First, our current synthesis method does not support let bindings, thus, it cannot be used to synthesize programs over DSLs that allow variable introduction. Second, our method treats λ-abstractions as constants; hence, it may not perform very well for DSLs that encourage heavy use of higher-order combinators. Third, our method requires abstract values to be drawn from a decidable logic – i.e., we assume that it is possible to over-approximate values of intermediate DSL expressions using formulas from a decidable logic. In future work, we plan to develop new abstract synthesis algorithms that support DSLs with richer language features, including let bindings. We also plan to explore more efficient techniques for synthesizing programs over DSLs that make heavy use of higher-order combinators.

ACKNOWLEDGMENTS

We would like to thank members in the UToPiA group for their insightful comments, as well as the anonymous reviewers for their constructive feedback. This work was supported in part by NSF Award #1712060, NSF Award #1453386 and AFRL Award #8750-14-2-0270. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

REFERENCES

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

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *International Conference on Computer Aided Verification (CAV)*. Springer, 934–950.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-guided synthesis. *Dependable Software Systems Engineering* 40 (2015), 1–25.

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

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 319–336.

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

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

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

Patrick Cousot and Radhia Cousot. 1977. 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)*. 238–252.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. *arXiv preprint arXiv:1703.07469* (2017).

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 422–436.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 229–239.

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed Synthesis: A Type-theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 802–815.

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

Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. 1993. Directed Hypergraphs and Applications. *Discrete Appl. Math.* 42, 2-3 (1993), 177–201.

Adrià Gascón, Ashish Tiwari, Brent Carmer, and Umang Mathur. 2017. Look for the Proof to Find the Program: Decorated-Component-Based Program Synthesis. In *International Conference on Computer Aided Verification (CAV)*. Springer, 86–103.

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 317–330.

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

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

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

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

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

Kevin Knight and Jonathan May. 2009. Applications of weighted automata in natural language processing. In *Handbook of Weighted Automata*. Springer, 571–596.

Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive Parser Synthesis by Example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 565–574.

Parthasarathy Madhusudan. 2011. Synthesizing reactive programs. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 12. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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

Jonathan May and Kevin Knight. 2008. A Primer on Tree Automata Software for Natural Language Processing. (2008).

Kenneth L McMillan and Andrey Rybalchenko. 2013. Solving constrained Horn clauses using interpolation. *Tech. Rep. MSR-TR-2013-6* (2013).

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

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 619–630.

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

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling Up Superoptimization. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 297–310.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 522–538.

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 107–126.

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

Gabriel Scherer and Didier Rémy. 2015. Which Simple Types Have a Unique Inhabitant?. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 243–255.

David E. Shaw, William R. Swartout, and C. Cordell Green. 1975. Inferring LISP Programs from Examples. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence (IJCAI)*. 260–267.

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

Rishabh Singh and Sumit Gulwani. 2016. Transforming Spreadsheet Data Types Using Examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 343–356.

Rishabh Singh and Armando Solar-Lezama. 2011. 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)*. 289–299.

Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 326–340.

Armando Solar-Lezama. 2008. *Program synthesis by sketching*. Ph.D. Dissertation.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 404–415.

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

Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. 2015. Program Synthesis Using Dual Interpretation. In *International Conference on Automated Deduction*. Springer, 482–497.

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

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

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017a. Program Synthesis using Abstraction Refinement. *arXiv preprint arXiv:1710.07740* (2017).

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Synthesis of Data Completion Scripts Using Finite Tree Automata. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017), 62:1–62:26.

Xinyu Wang, Sumit Gulwani, and Rishabh Singh. 2016. FIDEX: Filtering Spreadsheet Data using Examples. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 195–213.

Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing Transformations on Hierarchically Structured Data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 508–521.

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