Learning Abstractions for Program Synthesis

Xinyu Wang[1], Greg Anderson[1], Isil Dillig[1] and K. L. McMillan[2]

[1] University of Texas, Austin
[2] Microsoft Research, Redmond

Abstract. Many example-guided program synthesis techniques use *abstractions* to prune the search space. While abstraction-based synthesis has proven to be very powerful, a domain expert needs to provide a suitable abstract domain, together with the abstract transformers of each DSL construct. However, coming up with useful abstractions can be non-trivial, as it requires both domain expertise and knowledge about the synthesizer. In this paper, we propose a new technique for learning abstractions that are useful for instantiating a general synthesis framework in a new domain. Given a DSL and a small set of training problems, our method uses *tree interpolation* to infer reusable predicate templates that speed up synthesis in a given domain. Our method also learns suitable abstract transformers by solving a certain kind of second-order constraint solving problem in a data-driven way. We have implemented the proposed method in a tool called Atlas and evaluate it in the context of the Blaze meta-synthesizer. Our evaluation shows that (a) Atlas can learn useful abstract domains and transformers from few training problems, and (b) the abstractions learned by Atlas allow Blaze to achieve significantly better results compared to manually-crafted abstractions.

1 Introduction

Program synthesis is a powerful technique for automatically generating programs from high-level specifications, such as input-output examples. Due to its myriad use cases across a wide range of application domains (e.g., spreadsheet automation [1,2,3], data science [4,5,6], cryptography [7,8], improving programming productivity [9,10,11]), program synthesis has received widespread attention from the research community in recent years.

Because program synthesis is, in essence, a very difficult search problem, many recent solutions prune the search space by utilizing *program abstractions* [12,4,13,14,15,16]. For example, state-of-the-art synthesis tools, such as Blaze [14], Morpheus [4] and Scythe [16], symbolically execute (partial) programs over some abstract domain and reject those programs whose abstract behavior is inconsistent with the given specification. Because many programs share the same behavior in terms of their abstract semantics, the use of abstractions allows these synthesis tools to significantly reduce the search space.

While the abstraction-guided synthesis paradigm has proven to be quite powerful, a down-side of such techniques is that they require a domain expert to manually come up with a suitable abstract domain and write abstract transformers

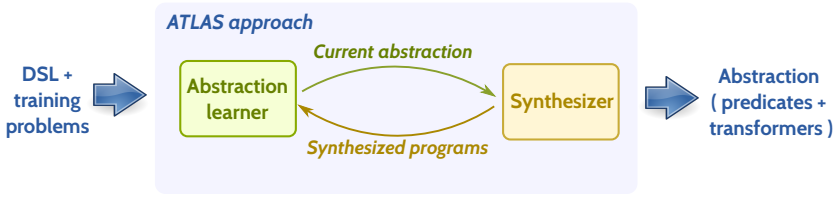

Fig. 1: Schematic overview of our approach.

for each DSL construct. For instance, the BLAZE synthesis framework [14] expects a domain expert to manually specify a universe of predicate templates, together with sound abstract transformers for every DSL construct. Unfortunately, this process is not only time-consuming but also requires significant insight about the application domain as well as the internal workings of the synthesizer.

In this paper, we propose a novel technique for automatically learning domain-specific abstractions that are useful for instantiating an example-guided synthesis framework in a new domain. Given a DSL and a training set of synthesis problems (i.e., input-output examples), our method learns a useful abstract domain in the form of predicate templates and infers sound abstract transformers for each DSL construct. In addition to eliminating the significant manual effort required from a domain expert, the abstractions learned by our method often outperform manually-crafted ones in terms of their benefit to synthesizer performance.

The workflow of our approach, henceforth called ATLAS [3], is shown schematically in Fig. 1. Since ATLAS is meant to be used as an *off-line* training step for a general-purpose programming-by-example (PBE) system, it takes as input a DSL as well as a set of synthesis problems $\boldsymbol{\mathcal{E}}$ that can be used for training purposes. Given these inputs, our method enters a refinement loop where an *Abstraction Learner* component discovers a sequence of increasingly precise abstract domains $\mathcal{A}_1, \cdot\cdot, \mathcal{A}_n$, and their corresponding abstract transformers $\mathcal{T}_1, \cdot\cdot, \mathcal{T}_n$, in order to help the *Abstraction-Guided Synthesizer* (AGS) solve all training problems. While the AGS can reject many incorrect solutions using an abstract domain $\mathcal{A}_i$, it might still return some incorrect solutions due to the insufficiency of $\mathcal{A}_i$. Thus, whenever the AGS returns an incorrect solution to any training problem, the Abstraction Learner discovers a more precise abstract domain and automatically synthesizes the corresponding abstract transformers. Upon termination of the algorithm, the final abstract domain $\mathcal{A}_n$ and transformers $\mathcal{T}_n$ are sufficient for the AGS to correctly solve *all* training problems. Furthermore, because our method learns *general* abstractions in the form of predicate templates, the learned abstractions are expected to be useful for solving many *other* synthesis problems beyond those in the training set.

From a technical perspective, the Abstraction Learner uses two key ideas, namely *tree interpolation* and *data-driven constraint solving*, for learning useful abstract domains and transformers respectively. Specifically, given an incorrect program $\mathcal{P}$ that cannot be refuted by the AGS using the current abstract domain $\mathcal{A}_i$, the Abstraction Learner generates a tree interpolant $\mathcal{I}_i$ that serves as a proof of $\mathcal{P}$'s incorrectness and constructs a new abstract domain $\mathcal{A}_{i+1}$ by

[3] ATLAS stands for AuTomated Learning of AbStractions.

extracting templates from the predicates used in $\mathcal{I}_i$. The Abstraction Learner also synthesizes the corresponding abstract transformers for $\mathcal{A}_{i+1}$ by setting up a *second-order constraint solving* problem where the goal is to find the unknown relationship between symbolic constants used in the predicate templates. Our method solves this problem in a data-driven way by sampling input-output examples for DSL operators and ultimately reduces the transformer learning problem to solving a system of linear equations.

We have implemented these ideas in a tool called Atlas and evaluate it in the context of the Blaze program synthesis framework [14]. Our evaluation shows that the proposed technique eliminates the manual effort involved in designing useful abstractions. More surprisingly, our evaluation also shows that the abstractions generated by Atlas outperform manually-crafted ones in terms of the performance of the Blaze synthesizer in two different application domains.

To summarize, this paper makes the following key contributions:

- We describe a method for learning abstractions (domains/transformers) that are useful for instantiating program synthesis frameworks in new domains.
- We show how tree interpolation can be used for learning abstract domains (i.e., predicate templates) from a few training problems.
- We describe a method for automatically synthesizing transformers for a given abstract domain under certain assumptions. Our method is guaranteed to find the unique best transformer if one exists.
- We implement our method in a tool called Atlas and experimentally evaluate it in the context of the Blaze synthesis framework. Our results demonstrate that the abstractions discovered by Atlas outperform manually-written ones used for evaluating Blaze in two application domains.

2 Illustrative Example

Suppose that we wish to use the Blaze meta-synthesizer to automate the class of string transformations considered by FlashFill [1] and BlinkFill [17]. In the original version of the Blaze framework [14], a domain expert needs to come up with a universe of suitable predicate templates as well as abstract transformers for each DSL construct. We will now illustrate how Atlas automates this process, given a suitable DSL and its semantics (e.g., the one used in [17]).

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

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

In order to construct the abstract domain $\mathcal{A}$ and transformers $\mathcal{T}$, Atlas starts with the trivial abstract domain $\mathcal{A}_0 = \{\top\}$ and transformers $\mathcal{T}_0$, defined as $[\![F(\top, \cdot\cdot, \top)]\!]^\sharp = \top$ for each DSL construct F. Using this abstraction, Atlas invokes Blaze to find a program $\mathcal{P}_0$ that satisfies specification $\mathcal{E}_1$ under the

current abstraction $(\mathcal{A}_0, \mathcal{T}_0)$. However, since the program $\mathcal{P}_0$ returned by Blaze is incorrect with respect to the concrete semantics, Atlas tries to find a more precise abstraction that allows Blaze to succeed.

Towards this goal, Atlas enters a refinement loop that culminates in the discovery of the abstract domain $\mathcal{A}_1 = \{\top, \mathit{len}(\boxed{\alpha}) = \mathtt{c}, \mathit{len}(\boxed{\alpha}) \neq \mathtt{c}\}$, where α denotes a variable and $\mathtt{c}$ is an integer constant. In other words, $\mathcal{A}_1$ tracks equality and inequality constraints on the length of strings. After learning these predicate templates, Atlas also synthesizes the corresponding abstract transformers $\mathcal{T}_1$. In particular, for each DSL construct, Atlas learns one abstract transformer for each combination of predicate templates used in $\mathcal{A}_1$. For instance, for the `Concat` operator which returns the concatenation y of two strings x_1, x_2, Atlas synthesizes the following abstract transformers, where $\star$ denotes any predicate:

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

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

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

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

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

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

3 Overall Abstraction Learning Algorithm

Our top-level algorithm for learning abstractions, called LearnAbstractions, is shown in Fig. 2. The algorithm takes two inputs, namely a domain-specific language $\mathcal{L}$ (both syntax and semantics) as well as a set of training problems $\boldsymbol{\mathcal{E}}$, where each problem is specified as a *set* of input-output examples $\mathcal{E}_i$. The output of our algorithm is a pair $(\mathcal{A}, \mathcal{T})$, where $\mathcal{A}$ is an abstract domain represented by a set of predicate templates and $\mathcal{T}$ is the corresponding abstract transformers.

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

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

Fig. 2: Overall learning algorithm. Constructs gives the DSL constructs in $\mathcal{L}$.

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

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

4 Learning Abstract Domain using Tree Interpolation

In this section, we present the LearnAbstractDomain procedure: Given a spurious program $\mathcal{P}$ and a synthesis problem $\mathcal{E}$ that $\mathcal{P}$ does not solve, our goal is to find new predicate templates $\mathcal{A}'$ to add to the abstract domain $\mathcal{A}$ such that the Abstraction-Guided Synthesizer no longer returns $\mathcal{P}$ as a valid solution to the synthesis problem $\mathcal{E}$. Our key insight is that we can mine for such useful predicate templates by constructing a *tree interpolation* problem. In what follows, we first review tree interpolants (based on [18]) and then explain how we use this concept to find useful predicate templates.

Definition 1 (Tree interpolation problem). *A tree interpolation problem $T = (V, r, P, L)$ is a directed labeled tree, where V is a finite set of nodes, $r \in V$ is the root, $P : (V \backslash \{r\}) \mapsto V$ is a function that maps children nodes to their parents, and $L : V \mapsto \mathbb{F}$ is a labeling function that maps nodes to formulas from a set $\mathbb{F}$ of first-order formulas such that $\bigwedge_{v \in V} L(v)$ is unsatisfiable.*

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

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

Definition 2 (Tree interpolant). *Given a tree interpolation problem $T = (V, r, P, L)$, a tree interpolant for T is a function $\mathcal{I} : V \mapsto \mathbb{F}$ that satisfies the following conditions:*

1. $\mathcal{I}(r) = \text{false}$;
2. *For each* $v \in V$: $\Big(\big(\bigwedge_{P(c_i)=v} \mathcal{I}(c_i) \big) \wedge L(v) \Big) \Rightarrow \mathcal{I}(v)$;
3. *For each* $v \in V$: $Vars\big(\mathcal{I}(v)\big) \subseteq Vars\big(L(Desc(v))\big) \bigcap Vars\big(L(NonDesc(v))\big)$.

Intuitively, the first condition ensures that $\mathcal{I}$ establishes the unsatisfiability of formulas in T, and the second condition states that $\mathcal{I}$ is a valid annotation. As standard in Craig interpolation [19,20], the third condition stipulates a "shared vocabulary" condition by ensuring that the annotation at each node v refers to the common variables between the descendants and non-descendants of v.

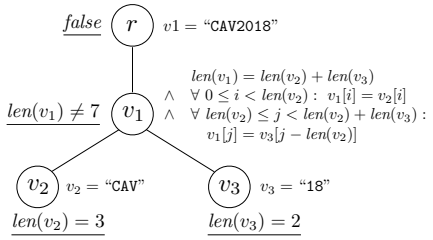

Fig. 3: A tree interpolation problem and a tree interpolant (underlined).

Example 1. Consider the tree interpolation problem $T = (V, r, P, L)$ in Fig. 3, where $L(v)$ is shown to the right of each node v. A tree interpolant $\mathcal{I}$ for this problem maps each node to the corresponding underlined formula. For instance, we have $\mathcal{I}(v_1) = (len(v_1) \neq 7)$. It is easy to confirm that $\mathcal{I}$ is a valid interpolant according to Definition 2.

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

With this intuition in mind, let us consider the LearnAbstractDomain procedure shown in Fig. 4: The algorithm uses a procedure called ConstructTree to generate a tree interpolation problem T for each input-output example

1: **procedure** LearnAbstractDomain($\mathcal{P}, \mathcal{E}$)

input: Program $\mathcal{P}$ that does not solve problem $\mathcal{E}$ (set of examples).
output: Set of predicate templates $\mathcal{A}'$.

2: $\mathcal{A}' \leftarrow \emptyset$;
3: **for each** $(e_{in}, e_{out}) \in \mathcal{E}$ **do**
4: **if** $[\![\mathcal{P}]\!]e_{in} \neq e_{out}$ **then**
5: $T \leftarrow$ ConstructTree($\mathcal{P}, e_{in}, e_{out}$);
6: $\mathcal{I} \leftarrow \mathsf{FindTreeItp}(T)$;
7: **for each** $v \in \mathsf{Nodes}(T) \backslash \{r\}$ **do**
8: $\mathcal{A}' \leftarrow \mathcal{A}' \cup \{\mathsf{MakeSymbolic}(\mathcal{I}(v))\}$;
9: **return** $\mathcal{A}'$;

Fig. 4: Algorithm for learning abstract domain using tree interpolation.

(e_{in}, e_{out})[4] that program $\mathcal{P}$ does not satisfy (line 5). Specifically, letting Π denote the AST representation of $\mathcal{P}$, we construct $T = (V, r, P, L)$ as follows:

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

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

Essentially, the ConstructTree procedure labels any leaf node representing the program input with the input example e_{in} and the root node with the output example e_{out}. All other internal nodes are labeled with the axiomatic semantics of the corresponding DSL operator (modulo renaming).[5] Observe that the formula $\bigwedge_{v \in V} L(v)$ is guaranteed to be unsatisfiable since $\mathcal{P}$ does not satisfy the I/O example (e_{in}, e_{out}); thus, we can obtain a tree interpolant for T.

Example 2. Consider program $\mathcal{P}$: `Concat`(x, "`18`") which concatenates constant string "`18`" to input x. Fig. 3 shows the result of invoking ConstructTree for $\mathcal{P}$ and input-output example ("`CAV`", "`CAV2018`"). As mentioned in Example 1, the tree interpolant $\mathcal{I}$ for this problem is indicated with the underlined formulas.

Since the tree interpolant $\mathcal{I}$ effectively establishes the incorrectness of program $\mathcal{P}$, the predicates used in $\mathcal{I}$ serve as useful abstract values that the synthesizer (AGS) should consider during the synthesis task. Towards this goal,

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

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

the LearnAbstractDomain algorithm iterates over each predicate used in $\mathcal{I}$ (lines 7–8 in Fig. 4) and converts it to a suitable template by replacing the constants and variables used in $\mathcal{I}(v)$ with symbolic names (or "holes"). Because the original predicates used in $\mathcal{I}$ may be too specific for the current input-output example, extracting templates from the interpolant allows our method to learn reusable abstract domains.

Example 3. Given the tree interpolant $\mathcal{I}$ from Example 1, LearnAbstractDomain extracts two predicate templates, namely, $\mathit{len}(\boxed{\alpha}) = \mathsf{c}$ and $\mathit{len}(\boxed{\alpha}) \neq \mathsf{c}$.

5 Synthesis of Abstract Transformers

In this section, we turn our attention to the LearnTransformers procedure for synthesizing abstract transformers $\mathcal{T}$ for a given abstract domain $\mathcal{A}$. Following presentation in prior work [14], we consider abstract transformers that are described using equations of the following form:

$$[\![F\big(\chi_1(x_1, \boldsymbol{c}_1), \cdot\cdot, \chi_n(x_n, \boldsymbol{c}_n)\big)]\!]^\sharp = \bigwedge_{1 \leq j \leq m} \chi'_j\big(y, \boldsymbol{f}_j(\boldsymbol{c})\big) \tag{1}$$

Here, F is a DSL construct, χ_i, χ'_j are predicate templates [6], x_i is the i'th input of F, y is F's output, $\boldsymbol{c_1}, \cdot\cdot, \boldsymbol{c_n}$ are vectors of *symbolic* constants, and $\boldsymbol{f}_j$ denotes a vector of *affine functions* over $\boldsymbol{c} = \boldsymbol{c_1}, \cdot\cdot, \boldsymbol{c_n}$. Intuitively, given concrete predicates describing the inputs to F, the transformer returns concrete predicates describing the output. Given such a transformer τ, let $\mathsf{Outputs}(\tau)$ be the set of pairs $(\chi'_j, \boldsymbol{f}_j)$ in Eqn. 1.

We define the soundness of a transformer τ for DSL operator F with respect to F's axiomatic semantics ϕ_F. In particular, we say that the abstract transformer from Eqn. 1 is *sound* if the following implication is valid:

$$\Big(\phi_F(\boldsymbol{x}, y) \wedge \bigwedge_{1 \leq i \leq n} \chi_i(x_i, \boldsymbol{c}_i)\Big) \Rightarrow \bigwedge_{1 \leq j \leq m} \chi'_j\big(y, \boldsymbol{f}_j(\boldsymbol{c})\big) \tag{2}$$

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

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

$$\exists \boldsymbol{f}.\ \forall \boldsymbol{V}.\Big(\big(\phi_F(\boldsymbol{x}, y) \wedge \bigwedge_{1 \leq i \leq n} \chi_i(x_i, \boldsymbol{c}_i)\big) \Rightarrow \bigwedge_{1 \leq j \leq m} \chi'_j\big(y, \boldsymbol{f}_j(\boldsymbol{c})\big)\Big) \tag{3}$$

where $\boldsymbol{f} = \boldsymbol{f}_1, \cdot\cdot, \boldsymbol{f}_m$ and $\boldsymbol{V}$ includes all variables and functions from Eqn. 2 other than $\boldsymbol{f}$. In other words, the goal of this constraint solving problem is to find interpretations of the unknown functions $\boldsymbol{f}$ that make Eqn. 2 valid. Our key insight is to solve this problem in a *data-driven* way by exploiting the fact that each unknown function $f_{j,k}$ is affine.

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

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

[6] We assume that $\chi'_1, \cdot\cdot, \chi'_m$ are distinct.

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

Fig. 5: Algorithm for synthesizing abstract transformers. ϕ_F at line 6 denotes the axiomatic semantics of DSL construct F. Formula Λ at line 8 refers to Eqn. 5.

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

$$\boldsymbol{f}_j(\boldsymbol{c})^\intercal = \underbrace{\begin{bmatrix} f_{j,1}(\boldsymbol{c}) \\ \cdot\cdot \\ f_{j,|\boldsymbol{f}_j|}(\boldsymbol{c}) \end{bmatrix}}_{\boldsymbol{c}'^\intercal_j} = \underbrace{\begin{bmatrix} p_{j,1,1} & \cdot\cdot & p_{j,1,|\boldsymbol{c}|+1} \\ \cdot\cdot & & \cdot\cdot \\ p_{j,|\boldsymbol{f}_j|,1} & \cdot\cdot & p_{j,|\boldsymbol{f}_j|,|\boldsymbol{c}|+1} \end{bmatrix}}_{P_j} \underbrace{\begin{bmatrix} c_1 \\ \cdot\cdot \\ c_{|\boldsymbol{c}|} \\ 1 \end{bmatrix}}_{\boldsymbol{c}^\dagger} \tag{4}$$

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

Given this representation, it is easy to see that the problem of synthesizing the unknown functions $\boldsymbol{f}_1, \cdot\cdot, \boldsymbol{f}_m$ from Eqn. 2 boils down to finding the unknown matrices $P_1, \cdot\cdot, P_m$ such that each P_j makes the following implication valid:

$$\Lambda \;\equiv\; \Big(\Big((\boldsymbol{c}'^\intercal_j = P_j \boldsymbol{c}^\dagger) \wedge \phi_F(\boldsymbol{x}, y) \wedge \bigwedge_{1 \leq i \leq n} \chi_i(x_i, \boldsymbol{c}_i)\Big) \Rightarrow \chi'_j(y, \boldsymbol{c}'_j)\Big) \tag{5}$$

Our key idea is to infer these unknown matrices $P_1, \cdot\cdot, P_m$ in a data-driven way by generating input-output examples of the form $[i_1, \cdot\cdot, i_{|\boldsymbol{c}|}] \mapsto [o_1, \cdot\cdot, o_{|\boldsymbol{f}_j|}]$ for each $\boldsymbol{f}_j$. In other words, $\boldsymbol{i}$ and $\boldsymbol{o}$ correspond to instantiations of $\boldsymbol{c}$ and $\boldsymbol{f}_j(\boldsymbol{c})$ respectively. Given sufficiently many such examples for every $\boldsymbol{f}_j$, we can then reduce the problem of learning each unknown matrix P_j to the problem of solving a system of linear equations.

Based on this intuition, the LearnTransformers procedure from Fig. 5 describes our algorithm for learning abstract transformers $\mathcal{T}$ for a given abstract domain $\mathcal{A}$. At a high-level, our algorithm synthesizes one abstract transformer for each DSL construct F and n argument predicate templates $\chi_1, \cdot\cdot, \chi_n$. In particular, given F and $\chi_1, \cdot\cdot, \chi_n$, the algorithm constructs the "return value" of the transformer as:

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

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

The key part of our LearnTransformers procedure is the inner loop (lines 5–8) for inferring each of these $\boldsymbol{f}_j$'s. Specifically, given an output predicate template χ'_j, our algorithm first generates a set of input-output examples E of the form $[p_1, \cdot\cdot, p_n] \mapsto p_0$ such that $[\![F(p_1, \cdot\cdot, p_n)]\!]^\sharp = p_0$ is a sound (albeit overly specific) transformer. Essentially, each p_i is a concrete instantiation of a predicate template, so the examples E generated at line 6 of the algorithm can be viewed as sound input-output examples for the general symbolic transformer given in Eqn. 1. (We will describe the GenerateExamples procedure in Section 5.1).

Once we generate these examples E, the next step of the algorithm is to learn the unknown coefficients of matrix P_j from Eqn. 5 by solving a system of linear equations (line 7). Specifically, observe that we can use each input-output example $[p_1, \cdot\cdot, p_n] \mapsto p_0$ in E to construct one row of Eqn. 4. In particular, we can directly extract $\boldsymbol{c} = \boldsymbol{c}_1, \cdot\cdot, \boldsymbol{c}_n$ from $p_1, \cdot\cdot, p_n$ and the corresponding value of $\boldsymbol{f}_j(\boldsymbol{c})$ from p_0. Since we have one instantiation of Eqn. 4 for each of the input-output examples in E, the problem of inferring matrix P_j now reduces to solving a system of linear equations of the form $AP_j^T = B$ where A is a $|E| \times (|\boldsymbol{c}| + 1)$ (input) matrix and B is a $|E| \times |\boldsymbol{f}_j|$ (output) matrix. Thus, a solution to the equation $AP_j^T = B$ generated from E corresponds to a candidate solution for matrix P_j, which in turn uniquely defines $\boldsymbol{f}_j$.

Observe that the call to Solve at line 7 may return *null* if no affine function exists. Furthermore, any *non-null* $\boldsymbol{f}_j$ returned by Solve is just a *candidate* solution and may not satisfy Eqn. 5. For example, this situation can arise if we do not have sufficiently many examples in E and end up discovering an affine function that is "over-fitted" to the examples. Thus, the validity check at line 8 of the algorithm ensures that the learned transformers are actually sound.

5.1 Example Generation

In our discussion so far, we assumed an oracle that is capable of generating valid input-output examples for a given transformer. We now explain our GenerateExamples procedure from Fig. 6 that essentially implements this oracle. In a nutshell, the goal of GenerateExamples is to synthesize input-output examples of the form $[p_1, \cdot\cdot, p_n] \mapsto p_0$ such that $[\![F(p_1, \cdot\cdot, p_n)]\!]^\sharp = p_0$ is sound where each p_i is a concrete predicate (rather than symbolic).

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

Conceptually, the GenerateExamples algorithm proceeds as follows: First, it generates *concrete* input-output examples $[s_1, \cdot\cdot, s_n] \mapsto s_0$ by evaluating F on randomly-generated inputs $s_1, \cdot\cdot, s_n$ (lines 4–5). Now, for each concrete I/O example $[s_1, \cdot\cdot, s_n] \mapsto s_0$, we generate a set of *abstract* I/O examples of the form $[p_1, \cdot\cdot, p_n] \mapsto p_0$ (line 6). Specifically, we assume that the return value $(A_0, \cdot\cdot, A_n)$ of Abstract at line 6 satisfies the following properties for every $p_i \in A_i$:

1: **procedure** GENERATEEXAMPLES($\phi_F, \chi_0, \cdot\cdot, \chi_n$)

input: Semantics ϕ_F of operator F and templates $\chi_0, \cdot\cdot, \chi_n$ for output and inputs.
output: A set of valid input-output examples E for DSL construct F.

2: $E \leftarrow \emptyset$;
3: **while** $\neg$FullRank(E) **do**
4: Draw $(s_1, \cdot\cdot, s_n)$ randomly from distribution R_F over Domain(F);
5: $s_0 \leftarrow [\![F(s_1, \cdot\cdot, s_n)]\!]$;
6: $(A_0, \cdot\cdot, A_n) \leftarrow$ Abstract($s_0, \chi_0, \cdot\cdot, s_n, \chi_n$);
7: **for each** $(p_0, \cdot\cdot, p_n) \in A_0 \times \cdot\cdot \times A_n$ **do**
8: **if** Valid$\big(\bigwedge_{1 \le i \le n} p_i \wedge \phi_F \Rightarrow p_0\big)$ **then** $E \leftarrow E \cup \big\{[p_1, \cdot\cdot, p_n] \mapsto p_0\big\}$;
9: **return** E;

Fig. 6: Example generation for learning abstract transformers.

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

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

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

Example 4. Given abstract domain $\mathcal{A} = \{len(\boxed{\alpha}) = \mathsf{c}\}$, suppose we want to learn an abstract transformer τ for the `Concat` operator of the following form:

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

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

$$\begin{bmatrix} 3\ 2\ 1 \\ 1\ 4\ 1 \\ 6\ 4\ 1 \end{bmatrix} P^T = \begin{bmatrix} 5 \\ 5 \\ 10 \end{bmatrix} \xrightarrow{\text{Solve}} P = [\,1\ 1\ 0\,]$$

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

6 Soundness and Completeness

In this section we present theorems stating some of the soundness, completeness, and termination guarantees of our approach. All proofs can be found in the extended version of this paper [21].

Theorem 1 (Soundness of LearnTransformers). *Let $\mathcal{T}$ be the set of transformers returned by* LearnTransformers. *Then, every $\tau \in \mathcal{T}$ is sound according to Eqn. 2.*

The remaining theorems are predicated on the assumptions that for each DSL construct F and input predicate templates $\chi_1, \cdot\cdot, \chi_n$ (i) there exists a unique best abstract transformer and (ii) the strongest transformer expressible in Eqn. 2 is logically equivalent to the unique best transformer. Thus, before stating these theorems, we first state what we mean by a *unique best abstract transformer*.

Definition 3 (Unique best function). *Consider a family of transformers of the shape $[\![F(\chi_1(x_1, \boldsymbol{c}_1), \cdot\cdot, \chi_n(x_n, \boldsymbol{c}_n))]\!]^\sharp = \chi'(y, \star)$. We say that $\boldsymbol{f}$ is the unique best function for $(F, \chi_1, \cdot\cdot, \chi_n, \chi')$ if (a) replacing $\star$ with $\boldsymbol{f}$ yields a sound transformer, and (b) replacing $\star$ with any other $\boldsymbol{f'}$ yields a transformer that is either unsound or strictly worse (i.e., $\chi'(y, \boldsymbol{f}) \Rightarrow \chi'(y, \boldsymbol{f'})$ and $\chi'(y, \boldsymbol{f'}) \not\Rightarrow \chi'(y, \boldsymbol{f})$).*

We now define unique best transformer in terms of unique best function:

Definition 4 (Unique best transformer). *Let F be a DSL construct and let $(\chi_1, \cdot\cdot, \chi_n) \in \mathcal{A}^n$ be the input templates for F. We say that the abstract transformer τ is a unique best transformer for $F, \chi_1, \cdot\cdot, \chi_n$ if (a) τ is sound, and (b) for any predicate template $\chi \in \mathcal{A}$, we have $(\chi, \boldsymbol{f}) \in \mathsf{Outputs}(\tau)$ if and only if $\boldsymbol{f}$ is a unique best function for $(F, \chi_1, \cdot\cdot, \chi_n, \chi)$ for some affine $\boldsymbol{f}$.*

Definition 5 (Complete sampling oracle). *Let F be a construct, $\mathcal{A}$ an abstract domain, and R_F a probability distribution over $\textsc{Domain}(F)$ with finite support S. Futher, for any input predicate templates $\chi_1, \cdot\cdot, \chi_n$ and output predicate template χ_0 in $\mathcal{A}$ admitting a unique best function $\boldsymbol{f}$, let $C(\chi_0, \cdot\cdot, \chi_n)$ be the set of tuples $(c_0, \cdot\cdot, c_n)$ such that $(\chi_0(y, c_0), \chi_1(x_1, c_1), \cdot\cdot, \chi_n(x_n, c_n)) \in A_0 \times \cdot\cdot \times A_n$ and $c_0 = \boldsymbol{f}(c_1, \cdot\cdot, c_n)$, where $A_0 \times \cdot\cdot \times A_n = \textsc{Abstract}(s_0, \chi_0, \cdot\cdot, s_n, \chi_n)$ and $(s_1, \cdot\cdot, s_n) \in S$ and $s_0 = [\![F(s_1, \cdot\cdot, s_n)]\!]$. The distribution R_F is a* complete sampling oracle *if $C(\chi_0, \cdot\cdot, \chi_n)$ has full rank for all $\chi_0, \cdot\cdot, \chi_n$.*

The following theorem states that LearnTransformers is guaranteed to synthesize the best transformer if a unique one exists:

Theorem 2 (Completeness of LearnTransformers). *Given an abstract domain $\mathcal{A}$ and a complete sampling oracle R_F for $\mathcal{A}$,* LearnTransformers *terminates. Further, let $\mathcal{T}$ be the set of transformers returned and let τ be the unique best transformer for DSL construct F and input predicate templates $\chi_1, \cdot\cdot, \chi_n \in \mathcal{A}^n$. Then we have $\tau \in \mathcal{T}$.*

Using this completeness (modulo unique best transformer) result, we can now state the termination guarantees of our LearnAbstractions algorithm:

Theorem 3 (Termination of LearnAbstractions). *Given a complete sampling oracle R_F for every abstract domain and the unique best transformer assumption, if there exists a solution for every problem $\mathcal{E}_i \in \boldsymbol{\mathcal{E}}$, then* LearnAbstractions *terminates.*

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

String domain

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

Matrix domain

Fig. 7: Training results. $|\mathcal{A}|, |\mathcal{T}|$, Iters denote the number of predicate templates, abstract transformers, and iterations taken per training instance (lines 5-10 from Fig. 2), respectively. $T_{\mathsf{AGS}}, T_{\mathcal{A}}, T_{\mathcal{T}}$ denote the times for invoking the synthesizer (AGS), learning the abstract domain, and learning the abstract transformers, respectively. T_{total} shows the total training time in seconds.

7 Implementation and Evaluation

We have implemented the proposed method as a new tool called Atlas, which is written in Java. Atlas takes as input a set of training problems, an Abstraction-Guided Synthesizer (AGS), and a DSL and returns an abstract domain (in the form of predicate templates) and the corresponding transformers. Internally, Atlas uses the Z3 theorem prover [22] to compute tree interpolants and the JLinAlg linear algebra library [23] to solve linear equations.

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

1. How does Atlas perform during training? That is, how many training problems does it require and how long does training take?
2. How useful are the abstractions learned by Atlas in the context of synthesis?

7.1 Abstraction Learning

To answer our first question, we use Atlas to automatically learn abstractions for two application domains: (i) string manipulations and (ii) matrix transformations. We provide Atlas with the DSLs used in [14] and employ Blaze as the underlying Abstraction-Guided Synthesizer. Axiomatic semantics for each DSL construct were given in the theory of equality with uninterpreted functions.

Training set information. For the string domain, our training set consists of exactly the four problems used as motivating examples in the BlinkFill paper [17]. Specifically, each training problem consists of 4-6 examples that demonstrate the desired string transformation. For the matrix domain, our training set consists of four (randomly selected) synthesis problems taken from online forums. Since almost all online posts contain a single input-output example, each training problem includes one example illustrating the desired matrix transformation.

Main results. Our main results are summarized in Fig. 7. The main take-away message is that Atlas can learn abstractions quite efficiently and does not require a large training set. For example, Atlas learns 5 predicate templates and 30 abstract transformers for the string domain in a total of 10.2 seconds. Interestingly, Atlas does not need all the training problems to infer these four

	Original BLAZE† benchmarks				Additional benchmarks				All benchmarks		
	#Solved		Running time improvement		#Solved		Running time improvement		Time (sec)	Running time improvement	
	BLAZE⋆	BLAZE†	max.	avg.	BLAZE⋆	BLAZE†	max.	avg.	avg.	max.	avg.
String	**93**	91	15.7×	2.1×	**40**	40	56×	22.3×	**2.8**	**56**×	**8.3**×
Matrix	**39**	39	6.1×	3.1×	**20**	19	83×	21.5×	**5.0**	**83**×	**9.2**×

Fig. 8: Improvement of BLAZE⋆ over BLAZE† on string and matrix benchmarks.

predicates and converges to the final abstraction after just processing the first training instance. Furthermore, for the first training instance, it takes ATLAS 4 iterations in the learning loop (lines 5-10 from Fig. 2) before it converges to the final abstraction. Since this abstraction is sufficient, it takes just one iteration for each following training problem to synthesize a correct program.

Looking at the right side of Fig. 7, we also observe similar results for the matrix domain. In particular, ATLAS learns 10 predicate templates and 59 abstract transformers in a total of 22.5 seconds. Furthermore, ATLAS converges to the final abstract domain after processing the first three problems [7] and the number of iterations for each training instance is also quite small (ranging from 1 to 3).

7.2 Evaluating the Usefulness of Learned Abstractions

To answer our second question, we integrated the abstractions synthesized by ATLAS into the BLAZE meta-synthesizer. In the remainder of this section, we refer to all instantiations of BLAZE using the ATLAS-generated abstractions as BLAZE⋆. To assess how useful the automatically generated abstractions are, we compare BLAZE⋆ against BLAZE†, which refers to the manually-constructed instantiations of BLAZE described in [14].

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

Experimental setup. We evaluate BLAZE⋆ and BLAZE† using the same DSLs from the BLAZE paper [14]. For each benchmark, we provide the same set of input-output examples to BLAZE⋆ and BLAZE†, and use a time limit of 20 minutes per synthesis task.

Main results. Our main evaluation results are summarized in Fig. 8. The key observation is that BLAZE⋆ consistently improves upon BLAZE† for both string and matrix transformations. In particular, BLAZE⋆ not only solves more bench-

[7] The learned abstractions can be found in the extended version of this paper [21].

marks than BLAZE† for both domains, but also achieves about an order of magnitude speed-up on average for the common benchmarks that both tools can solve. Specifically, for the string domain, BLAZE⋆ solves 133 (out of 148) benchmarks within an average of 2.8 seconds and achieves an average 8.3× speed-up over BLAZE†. For the matrix domain, we also observe a very similar result where BLAZE⋆ leads to an overall speed-up of 9.2× on average.

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

8 Related Work

To our knowledge, this paper is the first one to automatically learn abstract domains and transformers that are useful for program synthesis. We also believe it is the first to apply interpolation to program synthesis, although interpolation has been used to synthesize other artifacts such as circuits [24] and strategies for infinite games [25]. In what follows, we briefly survey existing work related to program synthesis, abstraction learning, and abstract transformer computations.

Program synthesis. Our work is intended to complement example-guided program synthesis techniques that utilize program abstractions to prune the search space [15,16,4,14]. For example, SIMPL [15] uses abstract interpretation to speed up search-based synthesis and applies this technique to the generation of imperative programs for introductory programming assignments. Similarly, SCYTHE [16] and MORPHEUS [4] perform enumeration over program sketches and use abstractions to reject sketches that do not have any valid completion. Somewhat different from these techniques, BLAZE constructs a finite tree automaton that accepts all programs whose behavior is consistent with the specification according to the DSL's abstract semantics. We believe that the method described in this paper can be useful to all such abstraction-guided synthesizers.

Abstraction refinement. In verification, as opposed to synthesis, there have been many works that use Craig interpolants to refine abstractions [20,26,27]. Typically, these techniques generalize the interpolants to abstract domains by extracting a vocabulary of predicates, but they do not generalize by adding parameters to form templates. In our case, this is essential because interpolants derived from fixed input values are too specific to be directly useful. Moreover, we *reuse* the resulting abstractions for subsequent synthesis problems. In verification, this would be analogous to re-using an abstraction from one property or program to the next. It is conceivable that template-based generalization could be applied in verification to facilitate such reuse.

Abstract transformers. Many verification techniques use logical abstract domains [28,29,30,31,32]. Some of these, following Yorsh, *et al.* [33] use sampling with a decision procedure to evaluate the abstract transformer [34]. Interpolation has also been used to compile efficient symbolic abstract transformers [35]. However, these techniques are restricted to finite domains or domains of finite height to allow convergence. Here, we use infinite parameterized domains to obtain better generalization; hence, the abstract transformer computation is more challenging. Nonetheless, the approach might also be applicable in verification.

9 Limitations

While this paper takes a first step towards automatically inferring useful abstractions for synthesis, our proposed method has the following limitations:

Shapes of transformers. Following prior work [14], our algorithm assumes that abstract transformers have the shape given in Eqn. 1. We additionally assume that constants $\boldsymbol{c}$ used in predicate templates are numeric values and that functions in Eqn. 1 are affine. This assumption holds in several domains considered in prior work [4,14] and allows us to develop an efficient learning algorithm that reduces the problem to solving a system of linear equations.

DSL semantics. Our method requires the DSL designer to provide the DSL's logical semantics. We believe that giving logical semantics is much easier than coming up with useful abstractions, as it does not require insights about the internal workings of the synthesizer. Furthermore, our technique could, in principle, also work without logical specifications although the learned abstract domain may not be as effective (see Footnote 3 in Section 4) and the synthesized transformers would not be provably sound.

UBT assumption. Our completeness and termination theorems are predicated on the *unique best transformer (UBT)* assumption. While this assumption holds in our evaluation, it may not hold in general. However, as mentioned in Section 6, we can always guarantee termination by including the concrete predicates used in the interpolant $\mathcal{I}$ in addition to the symbolic templates extracted from $\mathcal{I}$.

10 Conclusion

We proposed a new technique for automatically instantiating abstraction-guided synthesis frameworks in new domains. Given a DSL and a few training problems, our method automatically discovers a useful abstract domain and the corresponding transformers for each DSL construct. From a technical perspective, our method uses tree interpolation to extract reusable templates from failed synthesis attempts and automatically synthesizes unique best transformers if they exist. We have incorporated the proposed approach into the Blaze meta-synthesizer and show that the abstractions discovered by Atlas are very useful.

While we have applied the proposed technique to program synthesis, we believe that some of the ideas introduced here are more broadly applicable. For instance, the idea of extracting reusable predicate templates from interpolants and synthesizing transformers in a data-driven way could also be useful in the context of program verification.

Acknowledgements

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, NSF Award #1811865, 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

1. Gulwani, S.: 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 (2011) 317–330
2. Singh, R., Gulwani, S.: Transforming Spreadsheet Data Types Using Examples. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL, ACM (2016) 343–356
3. Wang, X., Gulwani, S., Singh, R.: FIDEX: Filtering Spreadsheet Data using Examples. OOPSLA, ACM (2016) 195–213
4. Feng, Y., Martins, R., Van Geffen, J., Dillig, I., Chaudhuri, S.: Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In: PLDI, ACM (2017) 422–436
5. Wang, X., Dillig, I., Singh, R.: Synthesis of Data Completion Scripts Using Finite Tree Automata. Proc. ACM Program. Lang. **1**(OOPSLA) (October 2017) 62:1–62:26
6. Yaghmazadeh, N., Wang, X., Dillig, I.: Automated Migration of Hierarchical Data to Relational Tables using Programming-by-Example. Proceedings of the VLDB Endowment (2018)
7. Gascón, A., Tiwari, A., Carmer, B., Mathur, U.: Look for the Proof to Find the Program: Decorated-Component-Based Program Synthesis. In: International Conference on Computer Aided Verification. CAV, Springer (2017) 86–103
8. Tiwari, A., Gascón, A., Dutertre, B.: Program Synthesis Using Dual Interpretation. In: International Conference on Automated Deduction, Springer (2015) 482–497
9. Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.W.: Component-based synthesis for complex APIs. In: POPL. Volume 52., ACM (2017) 599–612
10. Gvero, T., Kuncak, V., Kuraj, I., Piskac, R.: Complete Completion Using Types and Weights. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI, ACM (2013) 27–38
11. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid Mining: Helping to Navigate the API Jungle. In: Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI, ACM (2005) 48–61
12. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing Data Structure Transformations from Input-output Examples. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI, ACM (2015) 229–239
13. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program Synthesis from Polymorphic Refinement Types. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI, ACM (2016) 522–538
14. Wang, X., Dillig, I., Singh, R.: Program Synthesis using Abstraction Refinement. Volume 2., ACM (2017) 63:1–63:30
15. So, S., Oh, H.: Synthesizing Imperative Programs from Examples Guided by Static Analysis. In: Static Analysis Symposium, Springer International Publishing (2017) 364–381
16. Wang, C., Cheung, A., Bodik, R.: Synthesizing highly expressive sql queries from input-output examples. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI, ACM (2017) 452–466
17. Singh, R.: BlinkFill: Semi-supervised Programming by Example for Syntactic String Transformations. Proceedings of the VLDB Endowment **9**(10) (2016) 816–827

18. Blanc, R., Gupta, A., Kovács, L., Kragl, B.: Tree Interpolation in Vampire. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning, Springer (2013) 173–181
19. McMillan, K.L.: Applications of Craig Interpolants in Model Checking. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer (2005) 1–12
20. McMillan, K.L.: Interpolation and SAT-based Model Checking. In: International Conference on Computer Aided Verification. CAV, Springer (2003) 1–13
21. Wang, X., Dillig, I., Singh, R.: Learning Abstractions for Program Synthesis. arXiv preprint arXiv:1804.04152 (2018)
22. Z3: https://github.com/Z3Prover/z3
23. Keilhauer, A., Levy, S., Lochbihler, A., Ökmen, S., Thimm, G., Würzebesser, C.: JLinAlg: A Java-library for Linear Algebra without Rounding Errors. Technical report, Technical report (2003-2010), http://jlinalg.sourceforge.net/
24. Bloem, R., Egly, U., Klampfl, P., Könighofer, R., Lonsing, F.: Sat-based methods for circuit synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014, IEEE (2014) 31–34
25. Farzan, A., Kincaid, Z.: Strategy Synthesis for Linear Arithmetic Games. Proceedings of the ACM on Programming Languages **2**(POPL) (2017) 61
26. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The Software Model Checker BLAST. International Journal on Software Tools for Technology Transfer **9**(5-6) (2007) 505–525
27. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: A Framework for Abstraction- and Interpolation-based Software Verification. In: International Conference on Computer Aided Verification, Springer (2012) 672–678
28. Lev-Ami, T., Manevich, R., Sagiv, M.: TVLA: A System for Generating Abstract Interpreters. Building the Information Society (2004) 367–375
29. Lev-Ami, T., Sagiv, M.: TVLA: A System for Implementing Static Analyses. Static Analysis (2000) 105–110
30. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In Margaria, T., Yi, W., eds.: Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings. Volume 2031 of Lecture Notes in Computer Science., Springer (2001) 82–97
31. Lahiri, S.K., Bryant, R.E.: Constructing quantified invariants via predicate abstraction. In Steffen, B., Levi, G., eds.: Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings. Volume 2937 of Lecture Notes in Computer Science., Springer (2004) 267–281
32. Reps, T., Thakur, A.: Automating Abstract Interpretation. In: VMCAI, Springer (2016) 3–40
33. Reps, T., Sagiv, M., Yorsh, G.: Symbolic Implementation of the Best Transformer. In: VMCAI. Volume 2937., Springer (2004) 252–266
34. Thakur, A.V., Reps, T.W.: A Method for Symbolic Computation of Abstract Operations. In: International Conference on Computer Aided Verification. Volume 12., Springer (2012) 174–192
35. Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. Logical Methods in Computer Science **3**(4) (2007)