Proceedings of Machine Learning Research 93:139–154, 2018 International Conference on Grammatical Inference

Inferring Program Extensions from Traces

Roman Manevich ROMANM@CS.BGU.AC.IL
Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva 8410501, Israel

Sharon Shoham SHARON.SHOHAM@CS.TAU.AC.IL
School of Computer Science
Tel Aviv University
Tel Aviv 6997801, Israel

Editors: Olgierd Unold, Witold Dyrka, and Wojciech Wieczorek

Abstract

We present an algorithm for learning a non-trivial class of imperative programs. The algorithm accepts positive traces—input stores followed by a sequence of commands—and returns a program that *extends* the target program. That is, it behaves the same as the target program on all valid inputs—inputs for which the target program successfully terminates, and may behave arbitrarily on other inputs. Our algorithm is based on a quotient construction of the control flow graph of the target program. Since not all programs have a quotient in a convenient form, the ability to infer an extension of the target program increases the class of inferred programs. We have implemented our algorithm and applied it successfully to learn a variety of programs that operate over linked data structures and integer arithmetic.

Keywords: Learning, Program Extensions, Traces

1. Introduction

We address the problem of inferring an imperative program from a set of example executions. This problem has many potential applications, including learning robotic tasks—a form of imperative programs—from *experiences* (experiences are essentially an input configuration followed by a sequence of actions) (Mokhtari et al., 2017), generalizing plans generated by automated planners to form procedures that handle similar planning problems (Srivastava et al., 2012), computing models of *opaque code* (Heule et al., 2015)—code which is executable but whose code is unavailable, and programming by demonstration (Biermann et al., 1975; Lau et al., 2003). Inferring programs from traces may also serve as a good starting point for synthesizing programs from input-output examples, by first synthesizing candidate traces for each input-output example and then inferring a program for those traces.

Learning Program Extensions. A major problem in learning programs from examples is that programs are often not defined over all possible inputs. That is, they represent partial functions from inputs to outputs. Executing a program on an invalid input may result in unpredictable (i.e., non-deterministic) behavior, crashing, or getting stuck (possibly in an infinite loop). It is therefore not realistic to expect *negative traces*—example traces

starting from invalid inputs. Under the assumption that negative traces are unavailable, a learning algorithm cannot be expected to learn a program whose functional semantics is equivalent to that of the target program, as that would imply learning an acceptor of all valid inputs from only positive examples. Our observation is that this is typically also not the intent. The intent is to learn a program whose functional representation matches that of the target program only on the valid inputs. In other words, we aim to learn a program whose functional representation *extends* that of the target program. A program extension is not unique, thus our learning criterion is different from the usual criterion, which requires full semantic equivalence.

Control-flow Inference and Guard Inference. We represent programs by an automaton whose transitions are labelled by guards and commands (similar to *control-flow graphs*, used by compilers). There are two components to learning a program from traces: 1) inferring a *control structure*—an automaton that is isomorphic to an extension of the target program, up to the guards labeling its transitions; and 2) inferring correct guards, for which we adapt algorithms for learning decision trees.

The main contributions of this paper are as follows:

- We identify a new learning criterion for computational models whose semantics is a partial function: learning semantic *extensions*—programs that are behaviorally-equivalent to the target program on the domain of the function (i.e., valid inputs). While a computation model may not admit a canonical form sought by a given inference algorithm, one of its extension may. This provides the inference algorithm with greater flexibility, allowing it to increase the class of models it can infer.

- We present an algorithm for inferring extensions of a non-trivial class of programs from traces. Our algorithm is generic in the underlying language and makes very weak assumptions about it, namely an access to an interpreter. The interpreter is used to compute intermediate stores and to evaluate basic Boolean expressions, which is needed to infer the guards. The worst-case running time is exponential due to a search over an exponential space of quotients. Since part of the quotient is known, the effective search space is not large in practice. In our experiments, the search always succeeds without backtracking, leading effectively to a polynomial run time.

- We have implemented our algorithm for a Java-like language with dynamic object allocation and integer arithmetic and empirically evaluated the algorithm on a suite of benchmarks. Our experiments show that, in practice, the number of examples needed to infer the target program is quite small and the running time is low. We have made our tool and experiments publicly available.[1]

Outline. The rest of the paper is organized as follows. Section 2 formalizes programs and states the learning problem. Section 3 defines the class of programs addressed in this paper. Section 4 presents the overall structure of our inference algorithm. Section 5 defines our guard inference algorithm and puts together all the components of our algorithm. Section 6 contains our empirical evaluation. Section 7 discusses related work and concludes the paper.

We defer additional details and proofs to the full version of the paper.

1. https://github.com/rumster/program-extension-synthesis.

2. Programs and Problem Statement

In this section, we formalize programs and the learning problem addressed by this paper.

Definition 1 (Program Automaton) *A* program automaton $\langle \mathit{Grd}, \mathit{Cmd}, Q, \delta, q_0, q_F \rangle$ *consists of a finite non-empty set of* guard symbols Grd*, a finite non-empty set of* command symbols Cmd*, a finite non-empty set of* locations Q*, an* initial location $q_0 \in Q$*; a* final location $q_F \in Q$*; and a* transition function $\delta : (Q \setminus \{q_F\}) \times (\mathit{Grd} \times \mathit{Cmd}) \rightharpoonup Q$*.*

The program automaton can be understood as the control-flow graph of the program.

We define the location transition relation as follows: $q \xrightarrow{a} q' \Leftrightarrow q' \in \delta(q, a)$.

We define the set of *actions* as $\mathit{Action} \stackrel{\text{def}}{=} \mathsf{Grd} \times \mathsf{Cmd}$ and denote an action as g/c ($g \in \mathsf{Grd}, c \in \mathsf{Cmd}$).

Definition 2 (Program Semantics) *A* program semantics $(\mathit{Store}, \mathcal{G}, \mathcal{C})$ *consists of a set of* stores Store*; a* guard evaluation function $\mathcal{G} : \mathit{Grd} \to \mathit{Store} \to \{0, 1\}$*; and a* command evaluation function $\mathcal{C} : \mathit{Cmd} \to \mathit{Store} \to \mathit{Store}$*.*

A *program* $P \stackrel{\text{def}}{=} (M, \mathit{Sem}) \in \mathsf{Prog}$ consists of a program automaton M and a semantics Sem. In the sequel, we will often equate P with M; the semantics will be clear from the context. We will also use the shorthand notations $g(s)$ for $\mathcal{G}(g)(s)$ and $c(s)$ for $\mathcal{C}(c)(s)$ where $g \in \mathsf{Grd}$, $c \in \mathsf{Cmd}$ and $s \in \mathsf{Store}$. An action $a = g/c$ denotes a partial function $\mathsf{Store} \rightharpoonup \mathsf{Store}$ defined as $a(s) = s'$ if $g(s) = 1$ and $c(s) = s'$. Finally, the meaning of a sequence of commands (respectively, actions) is given by sequential composition.

A program is *deterministic* if the guards labeling the outgoing transitions of any location are mutually exclusive. That is, at most one holds for any store. In the sequel, we consider only deterministic programs as our target programs (the programs we aim to learn).

A *state* of P is a pair $(q, s) \in Q \times \mathsf{Store}$. We define labeled transitions between states as follows:

$$(q, s) \xrightarrow{a}_P (q', a(s)) \text{ if } q' = \delta(q, a) \wedge a(s) \neq \bot \ .$$

For deterministic programs, we will also write $(q, s) \xrightarrow{c}_P (q', c(s))$ to denote that $(q, s) \xrightarrow{a}_P (q', a(s))$ for $a = g/c$ where g is the unique guard such that $\delta(q, a) = q'$. These definitions naturally extend to sequences of actions (respectively, commands) by composition. In the sequel, we will drop the subscript P where no confusion is likely.

A *command trace*, or *trace* for short, is a pair $(s_0, \overline{c}) \in \mathsf{Store} \times \mathsf{Cmd}^*$ where s_0 is an input store and $\overline{c} = c_1 \cdot \ldots \cdot c_n$ is a finite sequence of commands. A trace $(s_0, \overline{c})$ belongs to P if there exist $g_1, \ldots, g_n \in \mathsf{Grd}$ such that $(q_0, s_0) \xrightarrow{g_1/c_1 \cdot \ldots \cdot g_n/c_n}_P (q_F, \_)$. When P is deterministic, these guards are unique. We denote the set of traces of a program P by $\mathsf{Traces}(P)$.

The size of a trace $(s_0, c_1 \cdot \ldots \cdot c_n)$, denoted as $|(s_0, c_1 \cdot \ldots \cdot c_n)|$, is n. Note that while δ-paths in the program automaton are oblivious to the store, command traces depend on the store. Consequently, some δ-paths might not correspond to any trace.

We define the *output* of $(s_0, c_1 \cdot \ldots \cdot c_n) \in \mathsf{Traces}(P)$ to be $s \in \mathsf{Store}$ such that $\mathcal{C}(c_1 \cdot \ldots \cdot c_n)(s_0) = s$.

The set of *precondition stores* of P is $\mathit{pre}(P) \stackrel{\text{def}}{=} \{s_0 \in \mathsf{Store} \mid \exists \overline{c}.\ (s_0, \overline{c}) \in \mathsf{Traces}(P)\}$.

Syntax

$x, \ldots \in$ Var		Program variables
$f, \ldots \in$ Field		Class fields
$c, \ldots$		Class names
$n \in \mathbb{Z}$		Integer constant
Cmd	::=	x=Expr | $x.f$=Expr
	|	x=`new` c() | `return` x
Expr	::=	x | n | null | Expr.f | Expr Op Expr
Op	::=	`+` | `-` | `*` | `/`
Grd	::=	true | Expr `==` Expr | Expr `<` Expr
	|	**Grd** `&&` **Grd** | **Grd** `||` **Grd** | `!`**Grd**

Stores

$$\mathsf{Store}^{\mathsf{Jminor}} \stackrel{\text{def}}{=} (\mathit{Env} \times \mathit{Heap}) \cup \{\mathsf{error}\}$$
$$\mathit{Env} \stackrel{\text{def}}{=} \mathsf{Var} \rightharpoonup \mathbb{Z}$$
$$\mathit{Heap} \stackrel{\text{def}}{=} \mathbb{Z} \rightharpoonup_{\mathsf{fin}} \mathsf{Field} \rightharpoonup \mathbb{Z}$$

(Objects are represented by their integer addresses.)

Semantics

$$\mathsf{Jminor} \stackrel{\text{def}}{=} (\mathsf{Store}^{\mathsf{Jminor}}, \mathcal{G}^{\mathsf{Jminor}}, \mathcal{C}^{\mathsf{Jminor}})$$

Figure 1: Elements of Jminor: commands and guards (left) and program semantics (right).

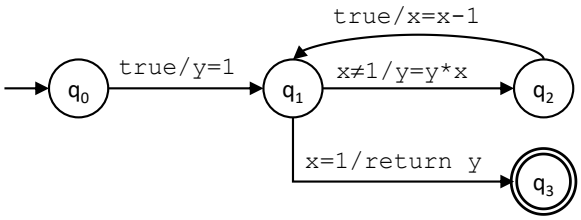

Figure 2: The `factorial` program.

A deterministic program defines a partial function $\mathsf{Store} \rightharpoonup \mathsf{Store}$ that maps every $s_0 \in \mathit{pre}(P)$ to the output of the trace $(s_0, \overline{c}) \in \mathsf{Traces}(P)$. Note that determinism ensures that this trace is unique.

Our algorithm is parametric in the semantics. To exemplify our algorithm, we proceed by defining an instance semantics inspired by the semantics of the Java programming language.

The Jminor Semantics. Fig. 1 defines the sets of guards, commands, and stores for a small Java-like language, which we have used for our experiments. The evaluation functions $\mathcal{G}^{\mathsf{Jminor}}$ and $\mathcal{C}^{\mathsf{Jminor}}$ match the semantics of Java. The command `return` a is treated as `res`=a (the `res` variable holds the return value), followed by dropping all variables except `res` from the store.

Fig. 2 shows the program `factorial`, which computes the factorial of `x` with Jminor.

Example 1 *The following are example traces of* `factorial`[2]*:*

$$E_{\mathit{factorial}} \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} ([x{\mapsto}2] & , \mathtt{y{:=}1 \cdot y{:=}y{*}x \cdot x{:=}x{-}1 \cdot return\ y}) \\ ([x{\mapsto}1] & , \mathtt{y{:=}1 \cdot return\ y}) \end{array} \right\}$$

The precondition of `factorial` *is as follows:*

$$\mathit{pre}(\mathtt{factorial}) \stackrel{\text{def}}{=} \{[x \mapsto n] \mid n > 0\} \ .$$

That is, running `factorial` *on a zero or negative input yields an infinite loop.*

2. Since `factorial` does not access heap objects, we only depict the *Env* component of the store.

2.1. The Learning Problem

We say that a program $P' \stackrel{\text{def}}{=} ((\mathsf{Grd}, \mathsf{Cmd}, Q', \delta', q'_0, q'_F), \mathit{Sem})$ *extends* a program $P \stackrel{\text{def}}{=} ((\mathsf{Grd}, \mathsf{Cmd}, Q, \delta, q_0, q_F), \mathit{Sem})$, denoted $P' \in \mathit{extension}(P)$, if the following holds:

$$\forall s_0, s_f \in \mathsf{Store}.\ \forall \overline{c} \in \mathsf{Cmd}^*.\ (q_0, s_0) \xrightarrow{\overline{c}}_P (q_F, s_f) \implies (q'_0, s_0) \xrightarrow{\overline{c}}_{P'} (q'_F, s_f)\ .$$

This implies that $\mathit{pre}(P) \subseteq \mathit{pre}(P')$, and that P' produces the same set of outputs on $s \in \mathit{pre}(P)$ as P. In fact, it ensures a stronger property, namely that P' follows the same sequences of commands as P on every input store $s \in \mathit{pre}(P)$.

Example 2 *The program inferred from $E_{\texttt{factorial}}$ by our algorithm uses the guards $\boldsymbol{x} \geq 1$ and $\boldsymbol{x} < 1$. It therefore extends* `factorial` *by mapping zero and negative inputs to 1.*

We now formalize the problem addressed by this paper.

Definition 3 (Identifying Program Extensions in the Limit from Traces) *We say that an algorithm $\mathit{Learn} : \wp_{\mathit{fin}}(\mathsf{Store} \times \mathsf{Cmd}^*) \rightharpoonup \mathsf{Prog}$ identifies program extensions in the limit, if for any program P and any enumeration of its traces $\{\tau_i\}_{i=1}^{\infty}$, there exists an index n such that $\mathit{Learn}(\{\tau_i\}_{i=1}^{n}) \in \mathit{extension}(P)$ and $\forall m > n.\ \mathit{Learn}(\{\tau_i\}_{i=1}^{n}) = \mathit{Learn}(\{\tau_i\}_{i=1}^{m})$.*

3. A Class of Programs

In this section, we define the class of programs inferred by our algorithm. For the sequel, we fix a deterministic program $P \stackrel{\text{def}}{=} (M, \mathit{Sem})$ where $M \stackrel{\text{def}}{=} \langle \mathsf{Grd}, \mathsf{Cmd}, Q, \delta, q_0, q_F \rangle$.

In order to uniquely infer locations from traces that only consist of commands, we require a stronger notion of determinism:

Definition 4 *A program is* strongly deterministic *if it is deterministic and the commands labeling its outgoing transitions are unique:*

$$\forall q, q_1, q_2 \in Q.\ g_1, g_2 \in \mathit{Grd}.\ c \in \mathsf{Cmd}.\ \delta(q, g_1/c) = q_1 \wedge \delta(q, g_2/c) = q_2 \implies q_1 = q_2\ .$$

We only consider strongly deterministic programs as our target programs. This is not a serious limitation, since in many cases, if a location q violates strong determinism, i.e., it has outgoing transitions $q \xrightarrow{g_1/c} q_1$ and $q \xrightarrow{g_2/c} q_2$, the two transitions can be merged, thereby deferring the split between q_1 and q_2 to the next step by adapting the guards g_1 and g_2, which recovers strong determinism.

3.1. k-Signature Programs

Let $\mathit{Halt} \notin \mathsf{Cmd}$ be a special command symbol, meaning that the program terminated.

Definition 5 *The* k-signature *of a location q, denoted $\mathit{sig}_k(q)$, is defined as follows:*

$$\begin{aligned} \mathit{sig}_k(q) \stackrel{\text{def}}{=}\ & \{\overline{c_2} \mid \exists (s_0, \overline{c_1} \cdot \overline{c_2} \cdot \overline{c_3}) \in \mathsf{Traces}(P).\ (q_0, s_0) \xrightarrow{\overline{c_1}} (q, \_) \wedge |\overline{c_2}| = k\} \cup \\ & \{\overline{c_2} \cdot \mathit{Halt}^{k-|\overline{c_2}|} \mid \exists (s_0, \overline{c_1} \cdot \overline{c_2}) \in \mathsf{Traces}(P).\ (q_0, s_0) \xrightarrow{\overline{c_1}} (q, \_) \wedge |\overline{c_2}| < k\}\ . \end{aligned}$$

Intuitively, $sig_k(q)$ includes the sequences of commands that can be observed after visiting q in a trace of P. These may be a strict subset of the δ-paths of the program automaton from q, due to the ability of guards to examine the store. Next, we define the class of programs $AllSigs_k$ where program locations can be uniquely identified by their k-signatures.

Let $AllSigs_k(P)$ and $MaxSigs_k(P)$ define the set of all k-signatures appearing in P and their maximal elements (w.r.t the subset relation):

$$\begin{array}{rcl} AllSigs_k(P) & \stackrel{\text{def}}{=} & \{sig_k(q) \mid q \in Q\} \\ MaxSigs_k(P) & \stackrel{\text{def}}{=} & \{sg \in AllSigs_k(P) \mid \forall q \in Q.\ sg \not\subset sig_k(q)\} \ . \end{array}$$

Definition 6 *We say that P is* k-signature*, denoted $P \in Sig_k$, if the following holds:*

$$\forall q_1, q_2 \in Q.\ sig_k(q_1) \subseteq sig_k(q_2) \implies q_1 = q_2 \ .$$

Equivalently: $AllSigs_k(P) = MaxSigs_k(P) \wedge \forall sg \in MaxSigs_k(P).\ \exists! q \in Q. sig_k(q) = sg$.[3]

We say that P is in signature form*, denoted $P \in Sig$, if $P \in \bigcup_{k=1}^{\infty} Sig_k$ holds.*

Example 3 *The following list of signatures shows that* `factorial` $\in Sig_1$*:*

$$\begin{array}{rcl} sig_1(q_0) & = & \{\texttt{y=1}\} \ , \\ sig_1(q_1) & = & \{\texttt{y=y*x}, \ \texttt{return y}\} \ , \\ sig_1(q_2) & = & \{\texttt{x=x-1}\} \ , \\ sig_1(q_3) & = & \{Halt\} \ . \end{array}$$

To enable our algorithm to observe the k-signature of each location in the examples, we need to further constraint the class of programs.

3.2. k-Regular Programs

For a sequence of commands $\overline{c_1}$ and a location $q \in Q$, we define the *sequences of commands generated by* $\overline{c_1}$ *for* q as follows:

$$\begin{array}{rcl} Gen(\overline{c_1}, q) & \stackrel{\text{def}}{=} & \{\overline{c_2} \in \mathsf{Cmd}^* \mid \\ & & \quad \exists s_0 \in \mathsf{Store}.\ \exists \overline{c_3} \in \mathsf{Cmd}^*.\ (s_0, \overline{c_1} \cdot \overline{c_2} \cdot \overline{c_3}) \in \mathsf{Trace}(P) \wedge (q_0, s_0) \xrightarrow{\overline{c_1}}_P (q, \_)\} \ . \end{array}$$

Definition 7 (k-Regular Transition Tree) *We say that a transition $q_1 \xrightarrow{a} q_2$ of P has a* k-regular transition tree *if there exists a prefix of commands $\overline{c} \in \mathsf{Cmd}^*$ such that $sig_k(q_1) \subseteq Gen(\overline{c}, q_1)$ and $sig_k(q_2) \subseteq Gen(\overline{c} \cdot a, q_2)$.*

Intuitively, the command prefix $\overline{c}$ generates both the k-signature of the pre-location of the transition, q_1, as well as the k-signature of its post-location, q_2 (in the sense that appending them to $\overline{c}$ results in a prefix of commands that can be observed on some input store s_0).

Definition 8 *A program P is* k-regular*, denoted $P \in Reg_k$, if there exists a k-regular transition tree for every transition $q_1 \xrightarrow{a} q_2$.*

3. The quantifier $\exists!$ stands for "exists exactly one".

Example 4 *The following prefixes show that* `factorial` $\in Reg_1$*:*

transition	*prefix*
$q_0 \xrightarrow{\texttt{true/y=1}} q_1$	ϵ
$q_1 \xrightarrow{\texttt{x!=1/y=y*x}} q_2$	`y=1`
$q_2 \xrightarrow{\texttt{true/x=x-1}} q_1$	`y=1` $\cdot$ `x=x-1`
$q_1 \xrightarrow{\texttt{x=1/return y}} q_3$	`y=1`

The class of programs satisfying all needed conditions for a value of k is defined as

$$SigReg_k \stackrel{\text{def}}{=} \{P \in Reg_k \text{ strongly deterministic } \mid \exists P' \in Sig_k \cap extension(P).\ AllSigs_k(P') = MaxSigs_k(P)\}\ .$$

Intuitively, $SigReg_k$ is defined such that the extension P' of P is a "quotient" of P.

The class of programs inferred by our algorithm is then

$$SigReg \stackrel{\text{def}}{=} \{P \mid \exists k > 0.\ \exists P' \in SigReg_k.\ \mathsf{Traces}(P) = \mathsf{Traces}(P')\}\ .$$

4. Control-flow Inference

Our algorithm PETI (for Program Extension Trace Inference), shown in Fig. 4, searches for a "semantic program automaton", which, intuitively, is a program automaton that has the same "control structure" as an extension P' of the target program and whose guards are sets of stores (*semantic guards*). For each semantic program automaton, it employs the procedure InfGuards to infer the actual guards from the semantic guards.

4.1. Semantic Program Automata

A *semantic program automaton* $M^\# = \langle \wp_{\mathsf{fin}}(\mathsf{Store}), \mathsf{Cmd}, Q, \delta^\#, q_0, q_F\rangle$ is a program automaton where the set of guards is $\wp(\mathsf{Store})$, i.e., the guards are semantic. We say that a semantic program automaton $M^\#$ represents the program automaton $M' = \langle \mathsf{Grd}, \mathsf{Cmd}, Q, \delta, q_0, q_F\rangle$ of a program $P' = (M', Sem)$ if for every $q_1, q_2 \in Q$ and $c \in \mathsf{Cmd}$ the following holds:

$$\forall g \in \mathsf{Grd}.\ \delta(q_1, g/c) = q_2 \implies \exists S \in \wp(\mathsf{Store}).\ \delta^\#(q_1, S/c) = q_2 \wedge S \subseteq \mathcal{G}(g)^{-1}(1);\ \text{and}$$
$$\forall S \in \wp(\mathsf{Store}).\ \delta^\#(q_1, S/c) = q_2 \implies \exists g \in \mathsf{Grd}.\ \delta(q_1, g/c) = q_2 \wedge S \subseteq \mathcal{G}(g)^{-1}(1).$$

where $\mathcal{G}(g)^{-1}(1) = \{s \in \mathsf{Store} \mid \mathcal{G}(g)(s) = 1\}$ is the set of stores that satisfy the guard g.

A semantic program automaton is *well defined* if whenever $\delta^\#(q, S_1/c_1) = q_1$ and $\delta^\#(q, S_2/c_2) = q_2$ for $q_1 \neq q_2$, it holds that $S_1 \cap S_2 = \emptyset$. If the automaton is not well defined, denoted as $M^\# = \bot$, it is impossible to assign guards to the transitions such that the obtained program automaton M' will be deterministic. As such, throughout the algorithm, we make sure to produce semantic automata that are well defined.

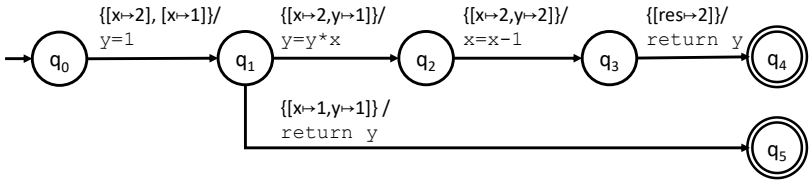

Figure 3: Prefix tree for $E_{\texttt{factorial}}$.

4.2. From Prefix Trees to Semantic Quotients

The goal of control-flow inference is to infer a semantic program automaton that represents an extension M' of the target program. To that end, we start by constructing a *prefix tree automaton* from the given set of traces E.

Definition 9 (Prefix Tree) *Let $E \subseteq \mathsf{Traces}$ be a finite set of traces. The* prefix tree *of E is the semantic program automaton $T^\sharp$ (except that it has a* set *of final locations), defined as follows:*

$$\begin{array}{rcl} T^\sharp & \stackrel{\text{def}}{=} & \langle \wp(\mathsf{Store}), \mathsf{Cmd}, Q^T, \delta^T, q_\epsilon, F \rangle \\ Q^T & \stackrel{\text{def}}{=} & \{q_{\overline{c_1}} \mid (s_0, \overline{c_1} \cdot \overline{c_2} \in E)\} \\ \delta^T(q_{\overline{c_1}}, S/c) & \stackrel{\text{def}}{=} & q_{\overline{c_1} \cdot c} \text{ if } S = \{\overline{c_1}(s_0) \mid (s_0, \overline{c_1} \cdot c \cdot \overline{c_2}) \in E\} \neq \emptyset \\ F & \stackrel{\text{def}}{=} & \{q_{\overline{c}} \mid (s_0, \overline{c}) \in E)\} \ . \end{array}$$

That is, the locations of the prefix tree automaton correspond to the prefixes of command sequences in E and transitions connect consecutive prefixes. The semantic guards accumulate the stores observed along trace prefixes in each location.

We assume that all the locations in F are sink locations, i.e., have no outgoing transitions. This holds if there cannot be $(s, \overline{c}), (s', \overline{c} \cdot \overline{c'}) \in E$ where $\overline{c'} \in \mathsf{Cmd}^+$. In programs this assumption can be obtained, e.g., by appending a "return" command to each trace.

Example 5 *The prefix tree for Example 1 appears in Fig. 3.*

Next, we merge locations in $T^\sharp = \langle \wp(\mathsf{Store}), \mathsf{Cmd}, Q^T, \delta^T, q_\epsilon, F \rangle$ in order to obtain a semantic program automaton. Merging is performed based on a partition of the locations, which is parameterized by k (where k starts from 1 and increases when necessary).

In the sequel, given a semantic program automaton (e.g., the prefix tree automaton $T^\sharp$) with locations $Q^\sharp$, we write $\pi = B_{1..n}$ to denote a partition of the set of locations $Q^\sharp$ into a set of blocks such that the set of final locations comprises one of the blocks. We denote the block containing the location q by $[q] \in \pi$. We say that a partition π' *coarsens* a partition π (alternatively π *refines* π'), denoted $\pi \sqsubseteq \pi'$, if $\forall B \in \pi.\ \exists B' \in \pi'.\ B \subseteq B'$.

Definition 10 (Semantic Quotient) *Let $M^\# = \langle \wp(\mathsf{Store}), \mathsf{Cmd}, Q^\sharp, \delta^\#, q_0, F \rangle$ be a semantic program automaton (with a set of sink final locations) and let π be a partition of its locations where $F \in \pi$. The quotient of $M^\#$ with respect to π, denoted $M^\#/\pi$, is the (possibly non-deterministic) semantic program automaton defined as follows:*

$$\begin{array}{rcl} M^\#/\pi & \stackrel{\text{def}}{=} & \langle \wp(\mathsf{Store}), \mathsf{Cmd}, \pi, \delta^Q, [q_0], F \rangle \\ \delta^Q(B, S'/c) & \stackrel{\text{def}}{=} & \{B' \mid S' = \bigcup \{S \mid \exists q_1 \in B.\ q_2 \in B'.\ \delta^\#(q_1, S/c) = q_2\} \neq \emptyset\} \ . \end{array}$$

That is, $M^\#/\pi$ is obtained from $M^\#$ by the standard definition of quotients, possibly introducing non-determinism, where the semantic guards aggregate guards from the individual locations in each block. Since F comprises a separate block in π, and since all the locations in F are sink locations, the quotient maintains the property that its final location is a sink location.

Computing Signatures for Semantic Automata. Since a prefix tree $T^\sharp$ is constructed out of examples, the k-signature of each location can be directly computed by taking the k-length δ^T-paths for each location. Since a semantic automaton $M^\#$ is constructed from a quotient of a prefix tree $T^\sharp$, the k-signature for each location q is the union of the k-signatures of all the locations in $[q]$. Once the k-signatures are available, we can compute $\mathit{AllSigs}_k$ and $\mathit{MaxSigs}_k$ for $T^\sharp$ and $M^\#$.

Computing a Partition. We define $\mathsf{MaxSigPartition}_k(M^\#)$ to be the partition induced by the following equivalence relation:

$$q \approx q' \stackrel{\text{def}}{=} \mathit{sig}_k(q) = \mathit{sig}_k(q') \in \mathit{MaxSigs}_k(M^\#) \ .$$

The procedure MergeBySig starts with the partition $\pi = \mathsf{MaxSigPartition}_k(T^\sharp)$, which groups locations with the same maximal k-signatures into a single block and leaves all other locations in their own block. If $P \in \mathit{Sig}_k$ then the locations with the same maximal k-signatures represent the same locations in its (quotient) extension.

To complete the partition, the algorithm must match each location q with a non-maximal k-signature with a location q' such that $\mathit{sig}_k(q) \subseteq \mathit{sig}_k(q')$. The admissibility criteria for each such completed partition π' are: (1) it does not change the set of maximal k-signatures (since these are the only signatures that exist in the extension program), (2) the induced automaton is strongly-deterministic, and (3) that appropriate guards can be inferred. To achieve this, the algorithm searches the space of partitions $\{\pi' \mid \pi' \sqsupseteq \pi\}$ and, for each candidate partition, computes the quotient automaton $T^\sharp/\pi'$ and applies determinization (i.e., $\mathit{det}(T^\sharp/\pi')$ merges locations violating strong determinism). The search space is exponential in the worst-case. We therefore apply heuristics to accelerate the search. In our experiments, the search always succeeds without having to backtrack.

Definition 11 (Control-flow Complete Sets) *Let $P = (M, \mathit{Sem})$ be a program in SigReg_k. We say that a finite set of traces $E \subseteq \mathsf{Traces}(P)$ is* control-flow complete *for P if it contains a k-regular transition tree for each transition of M.*

The size of a control-flow complete set E for $P \in \mathit{SigReg}_k$ is in $O(|Q|^2 \times |\mathsf{Cmd}|^k)$.[4]

Theorem 12 *For every program $P = (M, \mathit{Sem}) \in \mathit{SigReg}_k$, if E is control-flow complete for P, then the following properties hold: (1) $M' = \mathsf{MergeBySig}(\mathsf{PrefixTree}(E), k) \in \mathit{Sig}_k$, and (2) M' represents some $(M'', \mathit{Sem}) \in \mathit{extension}(P)$.*

Example 6 *In Fig. 3, $\mathit{sig}_1(q_i)$ is maximal for $i = 0, 1, 2, 4, 5$, but not for $i = 3$. The algorithm starts with $\mathsf{MaxSigPartition}_1(T^\sharp) = \{\{q_0\}, \{q_1\}, \{q_2\}, \{q_3\}, \{q_4, q_5\}\}$. It then merges q_3 into q_1, which results in a semantic automaton that represents an extension of* `factorial`.

4. The number of bits in E is unbounded, as the shortest trace to any given location is generally unbounded.

```
1  PETI(E : ℘(Trace)) {
2    T♯ ← PrefixTree(E)
3    if T♯ = ⊥ {
4      return ⊥
5    }
6    for k = 0 to max_{t∈E} |t| {
7      M ← MergeBySig(T♯, k)
8      if M ≠ ⊥ {
9        return M
10     } // Try greater k
11   }
12 }
```

```
1  MergeBySig(T♯, k : ℕ) {
2    π ← MaxSigPartition_k(T♯)
3    foreach π' ⊒ π {
4      M# ← det(T♯/π')
5      if MaxSigs_k(M#) = MaxSigs_k(T♯) {
6        M ← InfGuards(M#)
7        if M ≠ ⊥ {
8          return M
9        }
10     }
11   }
12   return ⊥
13 }
```

Figure 4: Pseudo-code for PETI.

5. Guard Inference

In this section, we describe how to convert a semantic program automaton into a program automaton represented by it, by inferring the guards for each transition.

The idea is to: (1) extract from the outgoing transitions of each location a *store clustering* function $SC : \mathsf{Cmd} \to \wp(\mathsf{Store})$, which maps each command (outgoing transition) to a set of stores on which the command should be executed, and (2) assign guards that respect the sets of stores in each cluster.

Definition 13 (Valid Guard Assignment) *We say that a function $\Gamma_{Sem} : \mathit{Cmd} \to \mathit{Grd}$ is a* valid guard assignment *for a store clustering $SC : \mathit{Cmd} \to \wp(\mathit{Store})$, relative to the semantics Sem, if the following condition holds:*

$$\begin{aligned} \forall c \in \mathit{Cmd}.\ \forall s \in SC(c). \quad & \Gamma_{Sem}(c)(s) = 1 \\ \forall c_1, c_2 \in \mathit{Cmd}.\ \forall s \in \mathit{Store}. \quad & (\Gamma_{Sem}(c_1)(s) = 1 \wedge \Gamma_{Sem}(c_2)(s) = 1) \implies c_1 = c_2 \ . \end{aligned}$$

Intuitively, Γ_{Sem} assigns a guard to each command such that all the stores mapped to the command satisfy the guard assigned to it and every pair of guards is mutually exclusive. In the sequel, we will drop the semantics subscript when no confusion is likely.

A store clustering $SC : \mathsf{Cmd} \to \wp(\mathsf{Store})$ is *deterministic* if the following holds:

$$\forall c_1, c_2 \in \mathsf{Cmd}.\ SC(c_1) \cap SC(c_2) \neq \emptyset \implies c_1 = c_2.$$

Note that a valid guard assignment can only exist when SC is deterministic.

We say that $\mathsf{InfGuards} : (\mathsf{Cmd} \to \wp(\mathsf{Store})) \rightharpoonup (\mathsf{Cmd} \to \mathsf{Grd})$ is a *guard inference procedure* if it returns a valid guard assignment when its input is deterministic, and $\bot$ otherwise.

Definition 14 (Inferring Guards in the Limit) *We say that a guard inference procedure $\mathit{InfGuards} : (\mathit{Cmd} \to \wp(\mathit{Store})) \rightharpoonup (\mathit{Cmd} \to \mathit{Grd})$ infers guards in the limit if for every store clustering $SC : (\mathit{Cmd} \to \wp(\mathit{Store}))$ and any enumeration of its sub-functions $\{SC_i : (\mathit{Cmd} \to \wp(\mathit{Store})) \mid \forall c.\ SC_i(c) \subseteq SC(c)\}_{i=1}^{\infty}$, there exists an index $n > 0$ such that $\forall m > n.\mathit{InfGuards}(SC_m)$ is a valid guard assignment for SC.*

Note that this definition does not ensure that $\forall m > n.\ \mathsf{InfGuards}(SC_m) = \mathsf{InfGuards}(SC_n)$, i.e., the guard assignment may not stabilize. We explain how to handle this in the sequel.

Guard Inference for Jminor. To infer guards for Jminor, we incrementally grow a set of basic Boolean expressions and apply a decision tree algorithm to infer a Boolean combination of these expressions for each abstract command (action).

Let $\mathrm{Expr}_m \stackrel{\text{def}}{=} \{e \in \mathrm{Expr} \mid |e| \leq m\}$ be the set of terms of size m or less,[5] and let $\mathsf{Grd}_m = \mathit{Bool}(\mathcal{F}_m)$ consist of all Boolean expressions over the following set of propositions:

$$\begin{aligned}\mathcal{F}_m = &\{e_1 = e_2 \mid e_1, e_2 \in \mathrm{Expr}_m \wedge \mathrm{type}(e_1) = \mathrm{type}(e_2)\} \cup \\ &\{a_1 < a_2 \mid a_1, a_2 \in \mathrm{Expr}_m \wedge \mathrm{type}(a_1) = \mathrm{type}(a_2) = \texttt{int}\}\ .\end{aligned}$$

Let $\mathrm{dtree} : (D \to \wp(\mathcal{F})) \to ((L \to \wp(D)) \to (L \to \mathit{Bool}(\mathcal{F})))$ be a decision tree procedure such that given a mapping of data points D (stores in our case) to sets of Boolean attributes $\mathcal{F}$ ($\mathcal{F}_m$ in our case) that hold in them, converts a mapping of labels L (commands in our case) to sets of data points assigned to them into a mapping of each label (command) to a Boolean formula over the feature expressions such that all the data points assigned to $l \in L$ satisfy the corresponding formula, and every pair of formulas is disjoint. Let $\mathit{s2f}_m : \mathsf{Store} \to \wp(\mathcal{F}_m)$ map stores to the set of propositions from $\mathcal{F}_m$ they satisfy. Then:

$$\mathsf{InfGuards}^{\mathsf{Jminor}}(SC) \stackrel{\text{def}}{=} \min_m\{\mathrm{dtree}(\mathit{s2f}_m)(SC) \neq \bot\}\ .$$

Program Automaton Construction. Given the semantic program automaton $M^\# = \langle \wp(\mathsf{Store}), \mathsf{Cmd}, Q^\sharp, \delta^\#, q_0, q_F\rangle$ inferred in the previous step, we define the program automaton $M' = \langle \mathsf{Grd}, \mathsf{Cmd}, Q^\sharp, \delta, q_0, q_F\rangle$ returned by $\mathsf{PETI}(E)$ as follows.

We define the store clustering of location $q \in Q$ as $SC_q : \mathsf{Cmd} \to \wp(\mathsf{Store})$, where for every $c \in \mathsf{Cmd}$, $SC_q(c) = S$ if $\delta^\#(q, S/c)$ is defined. Recall that $M^\#$ is strongly deterministic, hence c uniquely identifies an outgoing transition of q (i.e., no two outgoing transitions are labeled by the same command). Furthermore, recall that $M^\#$ is well defined, hence SC_q is deterministic (and a valid guard assignment may exist).

We define the transition relation δ of the program automaton M' as follows:

$$\delta(q_1, g/c) \stackrel{\text{def}}{=} q_2 \text{ if } \delta^\#(q_1, \_/c) = q_2 \wedge \mathsf{InfGuards}(SC_{q_1})(c) = g.$$

Stabilization. Recall that extensions are not unique, which means that our algorithm may alternate between different extensions. To handle this, we define StablePETI as follows. $\mathsf{StablePETI}(\{e_{1..n}\})$ runs $\mathsf{PETI}(\{e_{1..m}\})$ for $m = 1..n$ and returns the first automaton that is consistent with $\{e_{m+1..n}\}$ (i.e., $\{e_{m+1..n}\}$ belong to its set of traces).

Theorem 15 *For every $P \in \mathit{SigReg}$, StablePETI identifies an extension of P in the limit.*

Proof [sketch] Assume that $P = (M, \mathit{Sem}) \in \mathit{SigReg}_m$ and let $\{\tau_i\}_{i=1}^\infty$ be an enumeration of $\mathsf{Traces}(P)$. For a large enough index n, $E_n \stackrel{\text{def}}{=} \{\tau_i\}_{i=1}^n$ is control-flow complete. Define $T_n^\sharp = \mathsf{PrefixTree}(E_n)$ and $M_{n,k} \stackrel{\text{def}}{=} \mathsf{MergeBySig}(T_n^\sharp, k)$. There are two cases to consider for the loop in line 6 of PETI.

5. For efficiency, we only consider the (finite set of) integer constants appearing in commands.

Case 1: Assume $k = m$. Then, by Theorem 12, $M_{n,m} \in Sig_m$ represents an extension of P. Therefore, k never increases. Since $|Sig_k|$ is finite, for a large enough $n' > n$, $\mathsf{MergeBySig}(T^{\sharp}_{n'}, m)$ belongs to a set of automata that, up to the guard assignments, is finite. Since InfGuards infers guards in the limit, there exists an index $n'' > n$ such that for all $n''' > n''$, we have that $M_{n'',m}$ belongs to a finite set of automata that extend P.

Case 2: Assume the loop stabilizes on a value $k < m$. Recall that $|Sig_k|$ is finite. Repeating the same arguments as above, we have that for a large enough index $n' > n$, $M_{n'',k}$ belongs to a finite set of automata. Additionally, $\mathsf{Traces}(M_{n'',k}, Sem) \supseteq \mathsf{Traces}(P)$ (otherwise, k would increase), meaning that $(M_{n'',k}, Sem) \in extension(P)$. ■

6. Empirical Evaluation

We have implemented our algorithm in Java. Our tool accepts a specification, which includes Jminor classes (akin to C structs), a method signature, and a set of traces. The tool then infers a program automaton and a generates a Java implementation from the automaton.[6]

Validation. The system allows specifying *test traces*, which are not used during the inference phase, and are used to check for membership in the inferred program. Since manual construction of traces is tedious and error-prone, the user may optionally write the target program (in an extension of Jminor with sequencing, if-then-else statements, and while-loops) and specify input stores. The system then generates traces by executing the program on the input stores.

We experimented[7] on the benchmark programs shown in Table 1. The number locations in the inferred automata range from 3 to 12. While the benchmarks are small, their stores and guards can be quite complex For example, the guard for `sll_reverse_merge` has the form `first==null || first!=null && second!=null && second.d<first.d`. Given this, we are encouraged that the number of examples needed to successfully infer program extensions is quite small.

The algorithm can sometimes efficiently infer correct program extensions from few examples, even when they are not control-flow complete (e.g., all examples where $|E| = 1$). If a sufficient subset of the k-signatures appears in the prefix tree, merging them induces further merges due to strong determinization. This sometimes reveals the desired maximal signatures, which has the effect of reducing the space of partitions that need to be searched as well as leading to a cascade of merges towards a program extension.

7. Related Work

In this section, we address the most relevant works from the following research fields.

6. The implementation maintains the current location and executes a loop that, on every step, evaluates the guards labeling the outgoing transitions from the current location, executes the command for the found transition, and updates the current location to the next location. The loop terminates if the final location is reached. We also implemented transformations to "compress" sub-automata to if-then-else statements and while loops whenever possible.
7. On a Yoga 1 64bit laptop with a Core i7 processor, 8GB of RAM, running Java 10 on Windows 10.

Program	Description	$\lvert E\rvert$	$\max_{t\in E}\lvert t\rvert$	Time	k	Ext.
`factorial`	The running example	2	7	<0.1	1	yes
`gcd`	Euclid's algorithm	2	10	<1	1	no
`sqrt_slow`	Integer square root via linear search	3	6	<0.1	1	no
`sll_fill`	Setting a value to all cells	1	9	<0.1	1	no
`sll_find`	Searching for a key	1	6	<0.1	1	no
`sll_max`	Finding the maximal key	4	9	<0.1	1	no
`sll_reverse_merge`	Reverse-merging two sorted lists	3	27	2	1	no
`sll_reverse`	List reversal	1	11	<0.1	1	no
`sll_find_root`	Find the root of a cycling list	1	14	<0.1	2	no
`sll_bubble_sort`	Bubble sort	5	54	3	1	no
`bst_find`	Searching for a key	5	8	<0.1	1	no
`bfs`	BFS over graph nodes with two fields	4	36	<0.1	1	no

Table 1: Benchmark programs. $\lvert E\rvert$ is the number of examples, $\max_{t\in E}\lvert t\rvert$ is the maximal example trace length, time is measured in seconds, k is the value for which MergeBySig succeeded, **Ext.** indicates whether the inferred program extends the target program. `sll` stands for singly-linked lists and `bst` stands for binary search tree.

Automata Learning. Biermann (1972) presents a heuristic algorithm for synthesizing Turing machines from traces including both commands and conditions.

A Mealy machine can be learned from input/output examples by adapting the RPNI algorithm (Oncina and Garcia, 1992). Giantamidis and Tripakis (2016) present an algorithm for identifying Moore machines in the limit from input-output traces. Oncina et al. (1993) present an algorithm for identifying a class of regular transducers in the limit. A major difference between these works and ours is that in the case of transducers, the transition guards are obvious from the examples. We cope with the missing guards by requiring strong determinism and separately inferring them.

Cassel et al. (2016) developed an active learning algorithm to infer register automata. The examples in register automata are abstract relative to the commands in our setting, since they expose the reaction of the environment but not the register assignments. On the other hand the set of guards in our setting is much more expressive.

One important difference between our setting and that of automata learning is that a program may contain two locations that cannot be distinguished by any trace, which is an assumption made by many automata inference algorithms.

Program Synthesis. Lau et al. (2003) present a framework for learning programs from traces containing varying amounts of detail. For our definition of traces their approach is restricted to single-loop programs where the loop body is a fixed-size sequence of commands.

Heule et al. (2015) present an algorithm for learning from traces where the store is partially observable and variables names are missing from commands. Their algorithm is an adaptation of a Metropolis-style search for a program that admits the example traces.

The programming languages community has been intensively working on synthesizing programs from input/output examples. See Gulwani (2010); Gulwani et al. (2015) for recent

surveys. These approaches either assume that the program is loop-free or take as input a control structure template (Solar-Lezama, 2008).

Inductive Programming. Inductive programming (IP) is the problem of inferring a recursive program (either functional or logical) from examples given by input/output pairs or traces. The objects manipulated by the program belong to a language of terms (Kitzelmann and Schmid, 2006; Kitzelmann, 2008). See Kitzelmann (2009) for a survey of IP and Hofmann et al. (2009) for a comparison of several prominent IP systems. IP algorithms differ by the restrictions on the class of learned programs and learning bias. The restrictions sometimes allow to efficiently generate likely traces from input/output examples and can have the advantage of ensuring pleasing properties of the inferred programs, e.g., termination and "simplicity". Our algorithm assumes restrictions in order to narrow the space of programs that need to be considered and to ensure identification in the limit, which as far as we know is not guaranteed by existing algorithms. Another difference is that our algorithm does not assume any structure on the stores. This allows us to infer, e.g., graph programs (BFS), but makes generating likely traces and ensuring termination very hard.

Generalized Planning. In automated planning, learning from input/goal examples is called "generalized planning". Aguas et al. (2016) reduce the problem of generalized planning to (non-generalized) planning. Srivastava et al. (2012) generalizes plans over an unbounded number of objects into programs that ensure loop termination. Mokhtari et al. (2017) employs heuristics to search for a likely (global) loop. Schmid and Kitzelmann (2011) provide evidence that inductive programming can be useful for generalized planning.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments. We thank Alexander Shkatov for implementing the first version of the guard inference and Dana Fisman for helpful discussions. This publication is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). The research was partially supported by Len Blavatnik and the Blavatnik Family foundation, and by the Blavatnik Interdisciplinary Cyber Research Center, Tel Aviv University.

References

J. S. Aguas, S. J. Celorrio, and A. Jonsson. Generalized planning with procedural domain control knowledge. In *ICAPS 2016, London, UK, June 12-17*, pages 285–293, 2016.

A. W. Biermann. On the inference of turing machines from sample computations. *Art. Int.*, 3:181–198, 1972.

A. W. Biermann, R. I. Baum, and F. E. Petry. Speeding up the synthesis of programs from traces. *IEEE Trans. Computers*, 24(2):122–136, 1975.

S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Asp. Comput.*, 28(2):233–263, 2016.

G. Giantamidis and S. Tripakis. Learning Moore machines from input-output traces. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, pages 291–309, 2016.

S. Gulwani. Dimensions in program synthesis. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, page 1, 2010.

S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. G. Zorn. Inductive programming meets the real world. *Commun. ACM*, 58(11):90–99, 2015.

S. Heule, M. Sridharan, and S. Chandra. Mimic: computing models for opaque code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy*, pages 710–720, 2015.

M. Hofmann, E. Kitzelmann, and U. Schmid. A unifying framework for analysis and evaluation of inductive programming systems. pages 55–60, Mar 2009.

E. Kitzelmann. Analytical inductive functional programming. In *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Valencia, Spain, July 17-18, 2008, Revised Selected Papers*, pages 87–102, 2008.

E. Kitzelmann. Inductive programming: A survey of program synthesis techniques. In *Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009, Edinburgh, UK, September 4, 2009. Revised Papers*, pages 50–73, 2009.

E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7:429–454, 2006.

T. A. Lau, P. M. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *Proceedings of the 2nd International Conference on Knowledge Capture (K-CAP 2003), October 23-25, 2003, Sanibel Island, FL, USA*, pages 36–43, 2003.

V. Mokhtari, L. Seabra Lopes, and A. J. Pinho. Learning robot tasks with loops from experiences to enhance robot adaptability. *Pattern Recognition Letters*, 99:57–66, 2017.

J. Oncina and P. Garcia. Identifying regular languages in polynomial time. In *Advanced in Structural and Syntactic Pattern Recognition, Volume 5 of Series in Machine Perception and Artificial Intelligence*, pages 99–108. World Scientific, 1992.

J. Oncina, P. García, and E. Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5):448–458, 1993.

U. Schmid and E. Kitzelmann. Inductive rule learning on the knowledge level. *Cognitive Systems Research*, 12(3-4):237–248, 2011.

A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

S. Srivastava, N. Immerman, and S. Zilberstein. Applicability conditions for plans with loops: Computability results and algorithms. *Artif. Intell.*, 191-192:1–19, 2012.