# Learning a Static Analyzer from Data

Pavol Bielik, Veselin Raychev, and Martin Vechev

Department of Computer Science, ETH Zürich, Switzerland
{pavol.bielik, veselin.raychev, martin.vechev}@inf.ethz.ch

**Abstract.** To be practically useful, modern static analyzers must precisely model the effect of both, statements in the programming language as well as frameworks used by the program under analysis. While important, manually addressing these challenges is difficult for at least two reasons: (i) the effects on the overall analysis can be non-trivial, and (ii) as the size and complexity of modern libraries increase, so is the number of cases the analysis must handle.

In this paper we present a new, automated approach for creating static analyzers: instead of manually providing the various inference rules of the analyzer, the key idea is to learn these rules from a dataset of programs. Our method consists of two ingredients: (i) a synthesis algorithm capable of learning a candidate analyzer from a given dataset, and (ii) a counter-example guided learning procedure which generates new programs beyond those in the initial dataset, critical for discovering corner cases and ensuring the learned analysis generalizes to unseen programs.

We implemented and instantiated our approach to the task of learning a points-to analysis for JavaScript, a challenging yet important problem that has received significant research attention. We show that our approach is effective: our system automatically discovered practical and useful inference rules for many corner cases that are tricky to manually identify and are missed by state-of-the-art, manually tuned solutions.

## 1 Introduction

Static analysis is a fundamental method for automating program reasoning with a myriad of applications in verification, optimization and bug finding. While the theory of static analysis is well understood, building an analyzer for a practical language is a highly non-trivial task, even for experts. This is because one has to address several conflicting goals, including: (i) the analysis must be scalable enough to handle realistic programs, (ii) be precise enough to not report too many false positives, (iii) handle tricky corner cases and specifics of the particular language (e.g., JavaScript), (iv) decide how to precisely model the effect of the environment (e.g., built-in and third party functions), and other concerns. Addressing all of these manually, by-hand, is difficult and can easily result in suboptimal static analyzers, hindering their adoption in practice.

*Problem statement* The goal of our work is to help experts design more robust static analyzers, faster. Towards that, we present a new approach which automatically learns the static analyzer from data.

We state our learning problem as follows: given a domain-specific language $\mathcal{L}$ for expressing the abstract transformers (i.e., transfer functions) of the analyzer, a dataset $\mathcal{D}$ of programs in some programming language (e.g., JavaScript), and an abstraction function $\alpha$ that defines how concrete behaviors are abstracted, the goal is to learn an analyzer $pa \in \mathcal{L}$ (i.e., the transformers) such that programs in $\mathcal{D}$ are analyzed as precisely as possible, subject to $\alpha$.

*Key challenges* There are two main challenges we address in learning a static analyzer. First, static analyzers are typically described via production rules (e.g., type inference rules) or abstract transformers, designed by experts. In contrast, most popular machine learning approaches such as support vector machines and neural networks only produce weights over feature functions as output. When applying these techniques to program analysis [30,25], the result is simply a (linear) combination of existing rules – these approaches do not discover new rules. We address this challenge by introducing domain-specific languages for describing the analyzer (i.e., transformers, rules), and we learn the best analyzer over these languages. That is, our approach discovers new analysis rules.

The second and more challenging problem we address is how to avoid learning a static analyzer that works well on the training data $\mathcal{D}$, but fails to generalize well to programs outside of $\mathcal{D}$ – a problem known in machine learning as *overfitting*. Standard techniques from statistical learning theory [23] such as *regularization*, however, are insufficient for our purposes. The idea of regularization is that simpler models minimize the expected error rate on unseen data, but simpler models also contradict an important desired property of static analyzers to correctly handle tricky corner cases. We address this challenge via a counter-example guided learning procedure that leverages program semantics to generate new data (i.e., programs) for which the learned analysis produces wrong results and which are then used to further refine it.

We implemented our method and instantiated it for the task of learning a realistic points-to analysis for JavaScript, a very challenging problem. Static analysis for JavaScript is an active research area with several recent analyzers, both from industry (e.g., Facebook's Flow [5]) and academia (e.g., [16]). We show that our approach is effective: it discovered inference rules that correctly handle cases missed by these state-of-the-art, hand crafted solutions.

*Main contributions* Our main contributions are:

- A new approach for learning static analyzers from a dataset of programs. To ensure that the analysis *generalizes* beyond the training data we carefully generate counter-examples to the currently learned analyzer using an oracle.
- An algorithm that learns an analyzer from a dataset of examples which learns to over-approximate when it cannot precisely match a training example.
- An instantiation of our approach to the challenging problem of learning points-to analysis for JavaScript. We show that our method produces analyzers which generalize well to new data. It also discovered inference rules that cover many cases missed by existing, manually crafted solutions including Facebook's Flow. Not handling these cases leads to analysis unsoundness.

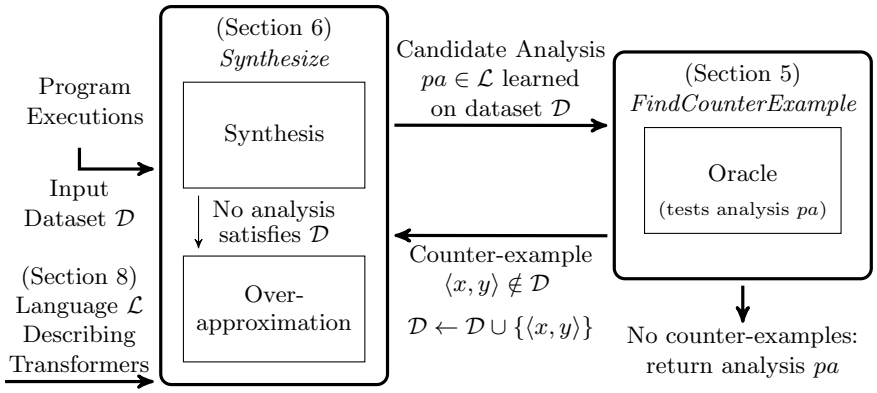



**Fig. 1.** Overview of our approach to learning a static analyzer from data consisting of three components – a language $\mathcal{L}$ for describing transformers, a learning algorithm and an oracle – that interact in a counter-example based refinement loop.

## 2 Our Approach

Our approach to learning static analyzers is shown in Fig. 1 and consists of several ingredients. To use our method one needs to instantiate these ingredients. We next discuss these components as well as our instantiation for each. The instantiations are already fairly general and as we demonstrate, allow for learning interesting and practical static analyzers.

*Training data* $\mathcal{D}$ We rely on a dataset of examples $\mathcal{D}$ for which our learned analyzer should behave correctly. Thus, we need a way for obtaining correct results for the examples in $\mathcal{D}$ so that we can compare those results to the output of the candidate analyzer. To obtain $\mathcal{D}$, we leverage our setting of static analysis where there is a well understood notion of correctness, namely, the analyzer must *approximate* (in the sense of lattice ordering) the concrete program behaviors. That is, we can simply run a large amount of programs in a given programming language with some inputs, and obtain a subset of the concrete semantics for each program. Concretely, we obtain $\mathcal{D} = \{\langle x^j, y^j\rangle\}_{j=1}^{N}$ consisting of pairs $\langle x^j, y^j\rangle$ where $x^j$ is a program and $y^j$ is the desired output of the analysis when applied to $x^j$. We note that our method is independent of how the labels are obtained: one can obtain the labels $y^j$ by running other analyzers on the programs $x^j$ in $\mathcal{D}$ (e.g., not only dynamic analysis) or can provide them manually.

*Language* $\mathcal{L}_{pt}$ To express a static analyzer, we use a domain-specific language that specifies the abstract transformers (inference rules) of the analysis. In this work, we define a loop-free language with branches, called $\mathcal{L}_{pt}$ (described in Section 8). This language can be used to express various interesting static analyses (e.g., points-to analysis).

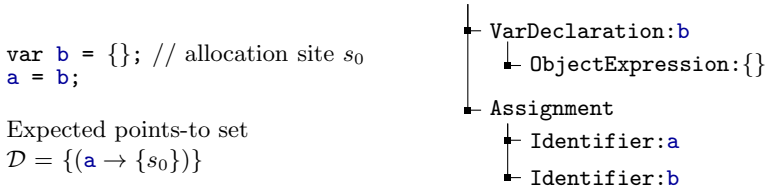
*Synthesizer* The synthesizer component takes as input the dataset $\mathcal{D}$ and a language $\mathcal{L}_{pt}$ for specifying transformers, and produces a candidate program analysis $pa \in \mathcal{L}_{pt}$ which correctly handles the pairs in $\mathcal{D}$. This component is described in Section 6. The synthesizer we propose phrases the problem of learning a static analysis over $\mathcal{L}_{pt}$ as a problem in learning decision trees over $\mathcal{L}_{pt}$. Thus, the approach is fairly general and can be applied to learn transformers of static analyses beyond those instantiated in this work.

*Oracle* Our goal is to discover a program analysis that not only behaves as described by the pairs in the dataset $\mathcal{D}$, but one that generalizes to programs beyond those in $\mathcal{D}$. To address this challenge, we introduce the oracle component and connect it with the synthesizer. The goal of the oracle component, *FindCounterExample*, is to take as input the learned analysis $pa$ and to try and find an example program $x$ for which $pa$ fails to produce the desired result $y$. This counter-example $\langle x, y \rangle$ is then fed back to the synthesizer which uses it to generate a new candidate analyzer as illustrated in Fig. 1. To produce a counter-example, the oracle must have a way to quickly and effectively test a (candidate) static analyzer. In Section 5, we present two techniques that make the testing process more effective by leveraging the current set $\mathcal{D}$ when generating a candidate program $x$. One of the two techniques is independent of the analyzer we are trying to learn while the other is parametric on the type of analysis. These techniques for testing a static analyzer are of independent interest, beyond the setting of learning considered in our work.

*Iterative learning* To learn a static analyzer $pa$, the synthesizer and the oracle are linked together in a counter-example guided loop. This type of iterative search is frequently used in program synthesis [34], though each instantiation heavily depends on the particular application task at hand. For example, in our setting, the examples in $\mathcal{D}$ are programs (and not say program states) and we deal with notions of approximation specific to our domain of static analysis. This also means that we cannot directly leverage off-the-shelf components (e.g., SMT solvers) or existing synthesis approaches. Importantly, the counter-example guided learning approach employed here is of interest to machine learning as it addresses the fundamental problem of overfitting with techniques beyond those typically used (e.g., regularization [23], which is insufficient here).

*Practical applicability* We implemented our approach and instantiated it to the task of learning a points-to analysis for JavaScript, a practical and relevant problem. Interestingly, our system learned inference rules missed by manually crafted state-of-the-art solutions, e.g., Facebook's Flow [5]. We contacted a developer on Flow's team who confirmed that extending Flow to cover the cases handled by our analysis is a highly requested feature.

Our approach can be used in various scenarios. For instance, a designer may first provide some base analysis rules and then use our approach to discover the remaining, trickier rules (e.g., those that model built-in APIs).




(a) Training data (b) AST representation of the code in (a)

**Fig. 2.** Example data for learning points-to analysis.

## 3 Overview

This section provides an intuitive explanation of our approach on a simple points-to analysis for JavaScript. Assume we are learning the analysis from one training data sample given in Fig. 2 (a). This sample consists of two variables `a`, `b` and one allocation site $s_0$ that allocates an empty object. Our goal is to learn that the variable `a` may point to that empty object (allocation site $s_0$).

Points-to analysis is typically done by applying inference rules until fixpoint. An example of an inference rule modeling the effect of assignment is:

$$\frac{\texttt{VarPointsTo}(v_2, h) \qquad \texttt{Assignment}(v_1, v_2)}{\texttt{VarPointsTo}(v_1, h)} \ [\textsc{Assign}]$$

This rule essentially says that if variable $v_2$ is assigned to $v_1$ and $v_2$ may point to an object $h$, then the variable $v_1$ may also point to this object $h$.

*Domain specific language (DSL) for transformers:* Consider the following general shape of inference rules:

$$\frac{\texttt{VarPointsTo}(v_2, h) \qquad v_2 = f(v_1)}{\texttt{VarPointsTo}(v_1, h)} \ [\textsc{General}]$$

Here, the function $f$ takes a program element (a variable) and returns another program element or $\bot$. The rule says: use the function $f$ to find a variable $v_2$ whose points-to set will be used to determine what $v_1$ points to. The Assign rule is an instance of the General rule with the following function $f$:

$$f(x) = \begin{cases} y & \textbf{if there is } \ \texttt{Assignment}(x, y) \\ \bot & \textbf{otherwise} \end{cases}$$

The function $f$ can be implemented by traversing the AST and checking if the parent node of $x$ is of type `Assignment` and if $x$ is its first child. In this case, the right sibling of $x$ is returned. Otherwise $f$ returns $\bot$. We describe a domain-specific language $\mathcal{L}_{pt}$ that can describe such functions $f$ in Section 8.

Consider Fig. 2 (b) which shows the AST of our example. In addition to Assign, we need to handle the case of variable initialization (first line in the

program). We can accomplish that by updating $f$ as follows:

$$f_{desired}(x) = \begin{cases} y & \textbf{if there is } \texttt{Assignment}(x, y) \\ y & \textbf{if there is } \texttt{VarDeclaration:x}(y) \\ \bot & \textbf{otherwise} \end{cases}$$

*Problem statement* The problem of learning a points-to analysis can now be stated as follows: find an analysis $pa \in \mathcal{L}_{pt}$ such that when analyzing the programs in the training data $\mathcal{D}$, the resulting points-to set is as outlined in $\mathcal{D}$.

*The overfitting problem* Note that the dataset $\mathcal{D}$ does not uniquely determine the best function $f$. In fact, instead of the desired one, other functions can be returned:

$$f_{overfit}(x) = \begin{cases} y & \textbf{if } y \textbf{ is a } \texttt{VarDeclaration:y} \textbf{ preceding } x \\ y & \textbf{if there is } \texttt{VarDeclaration:x}(y) \\ \bot & \textbf{otherwise} \end{cases}$$

This function inspects the statement prior to an assignment instead of at the assignment itself and yet it succeeds to produce the correct analysis result on our dataset $\mathcal{D}$. However, this is due to the specific syntactic arrangement of statements in the training data $\mathcal{D}$ and may not generalize to other programs, beyond those in $\mathcal{D}$.

*Our solution* As discussed earlier, to address the problem of overfitting to $\mathcal{D}$, we propose a counter-example guided procedure that biases the learning towards semantically meaningful analyses. That is, the oracle tests the current analyzer and tries to find a counter-example on which the analysis fails. Our strategy to generating candidate programs is to modify the programs in $\mathcal{D}$ in ways that can change both the syntax and the semantics of those programs. As a result, any analysis that depends on purely syntactic properties of the program (e.g., order of statements in our example) would be penalized at the next iteration of *Synthesize*. As we show in the evaluation, our approach results in a much faster oracle than had we blindly generated programs. This is critical as faster ways of finding counter-examples increases the size of the search space we can explore, enabling us to discover interesting analyzers in reasonable time.

For example, a possible way to exclude $f_{overfit}$ is to insert an unnecessary statement (e.g., `var c = 1`) before the assignment `a = b` in Fig. 2 (a). Here, the analysis defined by $f_{overfit}$ produces an incorrect points-to set for variable `a` (as it points-to the value 1 of variable `c`). Once this sample is added to $\mathcal{D}$, $f_{overfit}$ is penalized as it produces incorrect results and the next iteration will produce a different analysis until eventually the desired analysis $f_{desired}$ is returned.

*Correctness of the approach* An important point about our method is that produces an analyzer that is guaranteed to be sound w.r.t to all of the examples in $\mathcal{D}$ we have seen. This is because even if it cannot exactly satisfy all examples in $\mathcal{D}$ the synthesis procedure always returns an *over-approximation* of the desired

outputs. That is, when it cannot match the target output exactly, *Synthesize* learns to approximate (e.g., can return $\top$ in some cases). A formal argument together with a discussion on these points is provided in Section 6. However, our method is not guaranteed to be sound for all programs in the programming language. We see the problem of certifying the analyzer as orthogonal and complementary to our work: our method can be used to predict an analyzer which is likely correct, generalize well, and to sift through millions of possibilities quickly, while a follow-up effort can examine this analyzer and decide whether to accept it or even fully verify it. Here, an advantage of our method is that the learned analyzer is expressed as a program, which can be easily examined by an expert, as opposed to standard machine learning models where interpreting the result is nearly impossible (and therefore difficult to verify with standard methods).

## 4 Checking Analyzer Correctness

We begin by providing definitions about analysis correctness. These definitions are later used for defining the *Synthesize* and *FindCounterExample* procedures.

### 4.1 Analysis Correctness

First, following [4], we state what it means for our static analyzer to be correct. As these definitions are fairly standard, we discuss the parts which we use in our approach in a semi-formal manner. The concrete semantics of a program $p$ include all of $p$'s concrete behaviors and is captured by a function $[\![p]\!] \colon \mathbb{N} \to \wp(\mathcal{C})$. This function associates a set of possible concrete states in $\mathcal{C}$ with each position in the program $p$. A position in the program could for instance be a program counter or some node in the program's abstract syntax tree.

A static analysis $pa$ of a program $p$ computes an abstract representation of the program's concrete behaviors, captured by a function $pa(p) \colon \mathbb{N} \to \mathcal{A}$ where $(\mathcal{A}, \sqsubseteq)$ is typically an abstract domain, usually a lattice of abstract facts equipped with an ordering $\sqsubseteq$ between facts. An abstraction function $\alpha \colon \wp(\mathcal{C}) \to \mathcal{A}$ then establishes a connection between the concrete behaviors and the abstract facts. It defines how a set of concrete states in $\mathcal{C}$ are abstracted into an abstract element in $\mathcal{A}$. The function is naturally lifted to work point-wise on a set of positions in $\mathbb{N}$ (used in the definition below).

**Definition 1 (Analysis Correctness).** *We say that a static analysis pa is correct if the following condition holds:*

$$\forall p \in \mathcal{T}_{\mathcal{L}}.\ \alpha([\![p]\!]) \sqsubseteq pa(p) \tag{1}$$

Here we use $\mathcal{T}_{\mathcal{L}}$ to denote the set of all possible programs in the target programming language ($\mathcal{T}_{\mathcal{L}}$). That is, a static analysis is correct if it over-approximates the concrete behaviors of the program according to the particular lattice ordering. There are other definitions of correctness (e.g., instead of $\alpha$, use the concretization function $\gamma$). Note that the definition above does not mention the precision of the analysis $pa$ but is only concerned with soundness.

### 4.2 Checking Correctness

One approach for checking the correctness of an analyzer it to try and automatically verify the analyzer itself, that is, to prove that the analyzer satisfies Definition 1 (or a similar definition) via sophisticated reasoning (e.g., as the one found in [10]). If such an automated verifier exists, one can then provide it with a candidate analysis *pa*, and if *pa* is incorrect, the verifier would automatically find a counter-example program which violates Definition 1. Unfortunately, such automated verifiers do not currently exist (though, coming up with one is an interesting research challenge). However, even if they did exist, placing such a verifier in the middle of a counter-example learning loop where one has to discard thousands of candidate analyzers quickly, will be prohibitively expensive. Thus, the correctness definition that we use in our approach is:

**Definition 2 (Analysis Correctness on a Dataset and Test Inputs).** *We say that a static analysis pa is correct w.r.t to a dataset of programs $P$ and test inputs ti if the following condition holds:*

$$\forall p \in P.\ \alpha(\llbracket p \rrbracket_{ti}) \sqsubseteq pa(p) \tag{2}$$

The two restrictions over Definition 1 are: the use of a set $P \subseteq \mathcal{T}_{\mathcal{L}}$ instead of $\mathcal{T}_{\mathcal{L}}$ and the restriction $\llbracket p \rrbracket_{ti}$ instead of $\llbracket p \rrbracket$. Here, $\llbracket p \rrbracket_{ti} \subseteq \llbracket p \rrbracket$ denotes a subset of a program $p$'s behaviors obtained after running the program on some set of test inputs *ti*.

The advantage of using this definition is that checking it can be easily automated. That is, to obtain $\llbracket p \rrbracket_{ti}$, we simply run the program $p$ on its test inputs *ti* (and thus, obtain a finite set of executions) and then apply the function $\alpha$ on the resulting set. To obtain $pa(p)$, we run the analyzer *pa* on $p$; finally, we compare the two results via the inclusion operator $\sqsubseteq$.

Next, we discuss how to obtain the dataset $P$ (e.g., via counter-example learning). In Section 10, we elaborate on where to obtain test inputs *ti* required to run a program.

## 5 The Oracle: Testing an Analyzer

We next describe the oracle component of our approach shown in Fig. 1, called *FindCounterExample*. It takes as input a candidate analyzer *pa* and the current dataset $\mathcal{D}$ that was used to learn *pa* and outputs a counter-example program on which *pa* behaves incorrectly. That is, the oracle is a procedure that aims to effectively test the correctness of the analyzer – this means that further research advancements in testing of static analyzers can be leveraged by our approach.

More formally, if $P_{\mathcal{D}} = \{x \mid \langle x, y \rangle \in \mathcal{D}\}$, our goal is to find a counter-example program $p \in \mathcal{T}_{\mathcal{L}}$ such that $p \notin P_{\mathcal{D}}$ and the correctness condition in Definition 2 is violated for the given analysis *pa* and program $p$. That is, our oracle must generate new programs beyond those already present in $P_{\mathcal{D}}$.

*Key Challenge* A key problem to address when designing the oracle is that it must *quickly* find a counter-example in the search space of all possible programs, if the currently learned analysis *pa* is incorrect (as the oracle is in the middle of a counter-example guided search). As we show in Section 10, finding such a counter-example by blindly generating new programs does not work as the search space of programs in $\mathcal{T}_{\mathcal{L}}$ is massive (or infinite, if the length of programs is unbounded).

We address this challenge by *prioritizing* the search to first check certain programs based on ideas inspired by state-of-the-art testing techniques [11,21]. Note that these techniques are used to significantly speed-up the search performed by the oracle and do not affect the correctness guarantees of the approach.

*Speeding up the search* Our main idea that enables searching in the space of programs $\mathcal{T}_{\mathcal{L}}$ faster is to leverage the dataset $P_{\mathcal{D}}$ and to generate new programs by performing modifications of the programs in $P_{\mathcal{D}}$. These modification are not random: we leverage the structure of the current analysis *pa* in two ways: (i) to find the program and the position in that program to modify, and (ii) to determine what modification to perform at this position.

### 5.1 Choosing Modification Positions

Given a program $x \in P_{\mathcal{D}}$ and analysis *pa*, we prioritize positions that are *read* while executing the program analysis *pa* and changing them would trigger different *execution path* in the analyzer *pa* itself (not the analyzed program). Determining these positions is done by instrumenting the program analyzer and recording the relevant instructions affecting the branches the analyzer takes.

For example, for Fig. 2 (a), we defined the analysis by the function $f_{overfit}$. For this function, only a subset of all AST nodes determine which of the three cases in the definition of $f_{overfit}$ will be used to compute the result of the analysis. Thus, we choose the modification position to be one of these AST nodes.

### 5.2 Defining Relevant Program Modifications

We now define two approaches for generating interesting program modifications that are potential counter-examples for the learned program analysis *pa*.

**Modification via Equivalence Modulo Abstraction** Once a program position is selected, we need to decide how to modify the statement at that position. With our first technique, the goal is to ensure that the candidate analysis *pa* is robust to certain types of program transformations. We refer to these as Equivalence Modulo Abstraction (EMA) transformations. Here, we transform the statement at the particular program position in a semantically-preserving way, producing a set of programs. Moreover, while the transformation is semantic-preserving, it is also one that should not affect the result of the analysis *pa*.

More formally, an EMA transformation is a function $F_{ema} : \mathcal{T}_{\mathcal{L}} \times \mathbb{N} \to \wp(\mathcal{T}_{\mathcal{L}})$ which takes as input a program $p$ and a position in the program, and produces a set of programs that are a transformation of $p$ at position $n$. If the analysis $pa$ is correct, then these functions (transformations) have the following property:

$$\forall p' \in F_{ema}(p, n).pa(p) = pa(p') \tag{3}$$

The intuition behind such transformations is to ensure stability by exploring *local program modifications*. If the oracle detects the above property is violated, the current analysis $pa$ is incorrect and the oracle reports the counter-example program $p'$. Examples of such transformations are insertion of dead code, renaming variable names, replacing constants or reordering function declarations. The decision for whether to use a particular transformation can vary depending on the kind of analysis we are trying to learn. Some transformations are applicable regardless of the analysis while others are more specific. For instance, inserting dead code that shares variable names with the rest of the program can affect flow-insensitive program analysis, however, a flow-sensitive analysis should be stable under a transformation that inserts dead code referencing other variables. We also note that the EMA property is similar to notion of algorithmic stability used in machine learning where the output of a classifier should be stable under small perturbations of the input (e.g., rotating an image) as well as the concept of equivalence modulo inputs used to validate compilers [21].

**Modification via Global Jumps** The previous modifications always generated semantic-preserving transformations. However, to ensure better generalization we are also interested in exploring changes to programs in $P_{\mathcal{D}}$ that may not be semantic preserving, defined via a function $F_{gj} : \mathcal{T}_{\mathcal{L}} \times \mathbb{N} \to \wp(\mathcal{T}_{\mathcal{L}})$. The goal is to discover a new program which exhibits behaviors not seen by any of the programs in $P_{\mathcal{D}}$ and is not considered by the currently learned analyzer $pa$.

While the space of modifications here is richer than with EMA (it need not preserve semantics), we still only perform changes at certain places in the program as dictated by Section 5.1, thus leveraging our setting of learning analyzers. Further, unlike the EMA approach which uses a specific correctness criterion to generate counter-examples, here, for the newly generated program, the oracle uses the criteria described in Section 4.

Overall, as shown in Section 10, our approach for generating programs to test the analysis $pa$ via the functions $F_{gj}$ and $F_{ema}$ is an order of magnitude more efficient at finding counter-examples than naively modifying the programs in $P_{\mathcal{D}}$.

## 6 Learning an Analyzer

In this section we present our approach for learning a static analyzer from examples. We first describe a template of a general purpose language $\mathcal{L}$ used to express inference rules (the template is instantiated later in Section 8), then

discuss the input components of the analysis, and finally present the learning algorithm *Synthesize* from Fig. 1 based on decision tree learning.

### 6.1 Preliminaries

Let $\mathcal{D} = \{\langle x^j, y^j \rangle\}_{j=1}^{N}$ be a dataset of programs from a target programming language $\mathcal{T}_{\mathcal{L}}$ together with outputs that a program analysis should satisfy. That is, $x^j \in \mathcal{T}_{\mathcal{L}}$ and $y^j$ are the outputs to be satisfied by the learned program analysis. Then, we define what it means for an analysis to be correct on such a set of input/output examples.

**Definition 3 (Analysis Correctness on Examples).** *We say that a static analysis $pa \in \mathcal{L}$ is correct on $\mathcal{D} = \{\langle x^j, y^j \rangle\}_{j=1}^{N}$ if:*

$$\forall j \in 1 \ldots N \quad . \quad y^j \sqsubseteq pa(x^j) \tag{4}$$

The restriction over Definition 1 here is that instead of working on every program $p \in \mathcal{T}_{\mathcal{L}}$, we only use the programs in the training dataset $\mathcal{D}$. This definition is not the same as Definition 2, because the result of the analysis here is provided in $\mathcal{D}$ and need not be computed by running programs on test inputs.

Our goal is to find a program analysis that is correct on the given dataset $\mathcal{D}$. However, the above definition is not concerned with analysis precision and we use other means to avoid obvious, but useless solutions where the analysis always returns the $\top$ element of the lattice $(\mathcal{A}, \sqsubseteq)$. We address this problem by defining a precision metric.

*Precision metric* First, we define a function $r \colon \mathcal{T}_{\mathcal{L}} \times \mathcal{A} \times \mathcal{L} \to \mathbb{R}$ that takes a program in the target language, its desired program analysis output and a program analysis and indicates if the result of the analysis is exactly as desired:

$$r(x, y, pa) \quad = \quad \textbf{if } (y \neq pa(x)) \textbf{ then } 1 \textbf{ else } 0 \tag{5}$$

Using $r$, we define a function that computes a precision metric *cost* on the full dataset $\mathcal{D}$ as follows:

$$cost(\mathcal{D}, pa) = \sum_{\langle x,y \rangle \in \mathcal{D}} r(x, y, pa) \tag{6}$$

Using this precision metric, we can state the following lemma:

**Lemma 1.** *For a program analysis $pa \in \mathcal{L}$ and a dataset $\mathcal{D}$, if $cost(\mathcal{D}, pa) = 0$, then the analysis is correct according to Definition 3.*

**Proof:** The proof is direct. Because $cost(\mathcal{D}, pa) = 0$ and $r$ is positive, then for every $\langle x, y \rangle \in \mathcal{D}$, $r(x, y, pa) = 0$. This means that $y = pa(x)$ and so $y \sqsubseteq pa(x)$, which is as defined in Definition 3. □

(a) $a \in \mathit{Actions}$ $\quad g \in \mathit{Guards}$
$l \in \mathcal{L} ::= a \mid \texttt{if}\ \ g\ \ \texttt{then}\ \ l\ \ \texttt{else}\ \ l$

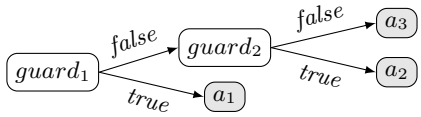
(b) 


**Fig. 3.** (a) Syntax of a template language $\mathcal{L}$ with branches for expressing transfer functions. (b) Example of a function from the $\mathcal{L}$ language shown as a decision tree.

### 6.2 Problem Formulation

We now state the requirements for the *Synthesize* procedure. Given a language $\mathcal{L}$ that describes inference rules (i.e., abstract transformers) and a dataset $\mathcal{D}$ of programs with the desired analysis results, the *Synthesize* procedure should return a program analysis $pa \in \mathcal{L}$ such that:

1. $pa$ is correct on the examples in $\mathcal{D}$ (Definition 3), and
2. $\mathit{cost}(\mathcal{D}, pa)$ is minimized.

The above statement essentially says that we would like to obtain a sound analysis which also minimizes the over-approximation that it makes. As the space of possible analyzers can be prohibitively large, we discuss a restriction on the language $\mathcal{L}$ and give a procedure that efficiently searches for an analyzer such that correctness is enforced and *cost* is (approximately) minimized.

### 6.3 Language Template for Describing Transformers

We now describe a template of the language $\mathcal{L}$ for describe transformers, shown in Fig. 3 (a). The template is simple and contains actions and guards that are to be instantiated later (Section 8). The statements in the language are either an action or a conditional **`if-then-else`** statements that can be applied recursively.

The abstract transformers of a static analyzer are expressed as a function built from statements in $\mathcal{L}$. As usual, the function is executed until a fixed point [4]. The semantics of the **`if`** statements in $pa$ is standard: guards are predicates (side-effect free) that inspect the program being analyzed and depending on their truth value, the corresponding branch of the **`if`** statement is taken. The reason such **`if`** statements are interesting is because they can express analysis rules such as the ones of our running example in Fig. 2.

### 6.4 Learning Algorithm

We now describe an algorithm which learns functions (i.e., program analyzers) over the language $\mathcal{L}$. A key challenge is that the search space of possible programs over $\mathcal{L}$ is massive as the number of possible combinations of branches and subprograms is too large.

However, note that elements of $\mathcal{L}$ can be represented as trees where internal nodes are guards of **`if`** statements and the leafs are actions as shown in Fig. 3 (b). Using this observation we can phrase the problem of learning an analyzer in $\mathcal{L}$ as the problem of learning a decision tree, allowing us to adapt existing algorithms used for decision tree learning to our setting.

```
def Synthesize(D)
    Input: Dataset D = {⟨x^j, y^j⟩}_{j=1}^N
    Output: Program pa ∈ L
    begin
        a_best ← genAction(D)
        if cost(D, a_best) = 0 then
            return a_best
        end
        g_best ← genBranch(a_best, D)
        if g_best = ⊥ then
            // D are noisy examples
            return approximate(D)
        end
        p1 ← Synthesize({⟨x, y⟩ ∈ D | g_best(x)})
        p2 ← Synthesize({⟨x, y⟩ ∈ D | ¬g_best(x)})
        return if g_best then p1 else p2
    end
```

**Algorithm 1:** Learning algorithm for programs from language $\mathcal{L}$.

*Decision tree learning* Decision tree learning is a well studied and widely used approach for learning classifiers used in many domains, including the tasks of learning program invariants [8] and building probabilistic models for code [28]. In our work we extend the ID3 [27] algorithm to handle action programs in the leafs and to enforce correctness of the resulting analysis $pa \in \mathcal{L}$. Similar to ID3, our algorithm is a greedy procedure that builds the decision tree in a top-down fashion and locally maximizes a metric called information gain.

### 6.5 ID3 for Correct Program Analysis

We show our learning procedure in Algorithm 1. The algorithm uses three helper functions that we define next. First, the *genAction* function takes a dataset $\mathcal{D}$ and returns the analysis $a_{best}$:

$$a_{best} = genAction(\mathcal{D}) = \underset{a \in Actions}{\arg\min}\ cost(\mathcal{D}, a) \tag{7}$$

That is, *genAction* returns the most precise program analysis consisting only of *Actions* (as we will see later, an action is just a sequence of statements, without branches). If $a_{best}$ is such that $cost(\mathcal{D}, a_{best}) = 0$, the analysis is both precise and correct (from Lemma 1), which satisfies our requirements stated in Section 6.2 and we simply return it. Otherwise, we continue by generating an **`if`** statement.

*Generating branches* The ID3 decision tree learning algorithm generates branches based on an information gain metric. To define this metric, we first use a standard definition of entropy. Let the vector $\boldsymbol{w} = \langle w_1, ..., w_k \rangle$ consist of elements from a set $C$. Then the entropy $H$ on $\boldsymbol{w}$ is:

$$H(\boldsymbol{w}) = -\sum_{c \in C} \frac{count(c, \boldsymbol{w})}{k} \log_2 \left( \frac{count(c, \boldsymbol{w})}{k} \right) \tag{8}$$

where $count(c, \boldsymbol{w}) = |\ \{i \in 1 \ldots k \mid w_i = c\}\ |$.

For a dataset $d \subseteq \mathcal{D}$, let $d = \{x_i, y_i\}_{i=1}^{|d|}$. Then, we define the following vector:

$$\boldsymbol{w}_d^{a_{best}} = \langle r(x_i, y_i, a_{best}) \mid i \in 1 \ldots |d| \rangle \tag{9}$$

That is, for every program in $d$, we record if $a_{best}$ is a precise analysis (via the function $r$ defined previously). Let $g \in Guards$ be a predicate that is to be evaluated on a program $x$. Let $\mathcal{D}^g = \{\langle x, y\rangle \in \mathcal{D} \mid g(x)\}$ and $\mathcal{D}^{\neg g} = \mathcal{D} \setminus \mathcal{D}^g$.

The information gain on a set of examples $\mathcal{D}$ for analysis $a_{best}$ and predicate guard $g$ is then defined as:

$$IG^{a_{best}}(\mathcal{D}, g) = H(\boldsymbol{w}_{\mathcal{D}}^{a_{best}}) - \frac{|\mathcal{D}^g|}{|\mathcal{D}|} H(\boldsymbol{w}_{\mathcal{D}^g}^{a_{best}}) - \frac{|\mathcal{D}^{\neg g}|}{|\mathcal{D}|} H(\boldsymbol{w}_{\mathcal{D}^{\neg g}}^{a_{best}}) \tag{10}$$

What the information gain essentially says is how many bits of information about the analysis correctness will be saved if instead of using the imprecise analysis $a_{best}$ directly, we split the dataset with a predicate $g$. Then we define *genBranch* as follows:

$$g_{best} = genBranch(a_{best}, \mathcal{D}) = \quad \operatorname*{arg\,max}_{g \in Guards}{}^{\perp}\, IG^{a_{best}}(\mathcal{D}, g) \tag{11}$$

Here, $\operatorname{arg\,max}^{\perp}$ is defined to return $\perp$ if the maximized information gain is 0, or otherwise to return the guard predicate $g$ which maximizes the information gain.

Back to Algorithm 1, if *genBranch* returns a predicate with positive information gain, we split the dataset with this predicate and call *Synthesize* recursively on the two parts. In the end, we return an **`if`** statement on the predicate $g$ and the two recursively synthesized analysis pieces.

*Approximation* If the information gain is 0 (i.e. $g_{best} = \perp$), we could not find any suitable predicate to split the dataset and the analysis $a_{best}$ has non-zero cost. In this case, we define a function *approximate* that returns an approximate, but correct program analysis – in our implementation we return analysis that loses precision by simply returning $\top$, which is always a correct analysis.

Note that, in practice, this approximation does not return $\top$ for the entire program analysis, but only for few of the branches in the decision tree, for which the synthesis procedure fails to produce a good program using both *genAction* and *getBranch*.

In terms of guarantees, for Algorithm 1, we can state the following lemma.

**Lemma 2.** *The analysis $pa \in \mathcal{L}$ returned by Synthesize is correct according to Definition 3.*

The proof of this lemma simply follows the definition of the algorithm and uses induction for the recursion. For our induction base, we have already shown that in case $cost(\mathcal{D}, a_{best}) = 0$, the analysis is correct. The analysis is also correct if *approximate* is called. In our induction step we use the fact that analyses $p_1$ and $p_2$ from the recursion are correct and must only show that the composed analysis **`if`** $g_{best}$ **`then`** $p_1$ **`else`** $p_2$ is also correct.

## 7 Points-to Analysis

In this section we present an instantiation of our approach to the task of learning transformers/rules for points-to analysis. The goal of points-to analysis is to answer queries of the type $q\colon V \to \wp(H)$, where $V$ is a set of program variables and $H$ is a heap abstraction (e.g., allocation sites). That is, the goal is to compute the set of (abstract) objects to which a variable may point-to at runtime. Similar to the example illustrated in Section 3, to answer such queries a common line of work [33,12,24] uses a declarative approach where the program is abstracted as a set of facts and the analysis is defined declaratively (e.g., as a set of Datalog rules) using inference rules that are applied until a fixed point is reached.

*Our goal* Our goal is to learn the inference rules that define the analysis, from data, as described in our approach so far. In particular, we would like to infer rules of the following general shape:

$$\frac{\texttt{VarPointsTo}(v_2, h) \qquad v_2 = f(v_1)}{\texttt{VarPointsTo}(v_1, h)} \quad [\textsc{General}]$$

where the goal of learning is to find a set of functions $f$ that, when used in the points-to analysis, produce precise results (as defined earlier). However, we focus our attention not on learning the standard and easy to define rules, as the one for assignment, but on rules that are hard and tricky to model by hand and are missed by existing analyzers. In particular, consider the following subset of inference rules that capture the points-to sets for the `this` variable in JavaScript. This rule has the following shape:

$$\frac{\texttt{VarPointsTo}(v_2, h) \qquad v_2 = f(\texttt{this})}{\texttt{VarPointsTo}(\texttt{this}, h)} \quad [\textsc{This}]$$

which is an instantiation of the general rule for the `this` variable by setting $v_1 =$ `this`. In JavaScript, designing such rules is a challenging task as there are many corner cases and describing those precisely requires more inference rules than the rest of the (standard) analysis rules. Further, because assigning a value to the `this` object is not allowed (i.e., using `this` as a left-hand side of an assignment expression), the value of `this` at runtime is not observed at the program level, yet assignments do occur internally in the interpreter and the runtime. Complicating matters, the actual values of the `this` reference can depend on the particular version of the interpreter.

*Example* Let us illustrate some of the complexity for determining the objects to which `this` points-to within the same method as shown in Fig. 4. Here, `this` points-to different objects depending on how the method is invoked and what values are passed in as arguments. In addition to the values shown in the example, other values may be seen during runtime if other APIs are invoked, or the method `isBig` is used as an object method or as a global method.

```
global.length = 4;
var dat = [5, 3, 9, 1];
function isBig(value) {
  return value >=
      this.length;
}
```

```
// this points to global
dat.filter(isBig); // [5, 9]
// this points to boxed 42
dat.filter(isBig, 42); // []
// this points to dat object
dat.filter(isBig, dat); // [5, 9]
```

**Fig. 4.** JavaScript code snippet illustrating subset of different objects to which `this` can point to depending on the context method `isBig` is invoked in. We show the analysis learned for `Function.prototype.filter` in Section 10.

### 7.1 Instantiating our Learning Approach

We now define the necessary components required to instantiate the learning approach described so far. Most of the instantiations are fairly direct except for the language $\mathcal{L}$, described separately in Section 8.

*Lattice of abstract heap locations* Fig. 5 shows the lattice $(\mathcal{H}, \sqsubseteq)$ used to represent the abstract domain of heap locations $H$. The abstraction function $\alpha\colon O \to H$ maps the concrete objects seen at runtime to abstract heap locations represented using a context-insensitive allocation site abstraction $H$. The lattice is quite simple and consists of the standard elements $\top$, $\bot$ and elements corresponding to individual heap locations $h_1 \cdots h_n$ that are not comparable.

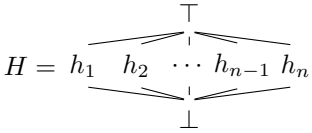



**Fig. 5.** Lattice of context-insensitive abstract heap locations $H$ for points-to analysis.

*Concrete and abstract program semantics* The concrete properties we are tracking and their abstract counterpart as described in Section 4 are instantiated by setting $\mathcal{C} := O$, $\mathcal{A} := H$ and $\mathbb{N} := \langle V, \mathbb{I}^* \rangle$. That is, all concrete program behaviors are captured by a function $[\![p]\!]\colon \langle V, \mathbb{I}^* \rangle \to \wp(O)$ that for each program variable $V$ sensitive to the $k$-most recent call sites $I$ computes a set of possible concrete objects seen at runtime $O$. The abstract semantics are similar except that we instantiate the abstract domain to be the lattice describing heap-allocated objects $H$. We discuss how we obtain the concrete behaviors $[\![p]\!]_{ti}$ after running the program on a set of test inputs $ti$ in Section 9.1.

*Program modifications* We list the program modifications used to instantiate the oracle in Table 1. The semantic preserving program modifications that should not change the result of points-to analysis $F_{ema}$ are inserted dead code and renamed variables and user functions (together with the parameters) as well as generated expressions that are side-effect free (e.g, declaring new variables). To explore new program behaviours by potentially changing program semantics we use program modifications $F_{gj}$ that change values of constants (strings and numbers), add methods arguments and add method parameters.

**Table 1.** Program modifications used to instantiate the oracle (Section 5) that generates counter-examples for points-to analysis.

| **Program Modifications** | |
|---|---|
| $F_{ema}$ | $F_{gj}$ |
| Adding Dead Code | Adding Method Arguments |
| Renaming Variables | Adding Method Parameters |
| Renaming User Functions | Changing Constants |
| Side-Effect Free Expressions | |

## 8 Language for Points-To Inference Rules

We now provide a definition of our domain specific language $\mathcal{L}_{pt}$, an instantiation of the template language $\mathcal{L}$ shown in Fig. 3. Our main goal was to design a language $\mathcal{L}_{pt}$ that is fairly generic: (i) it does not require the designer to provide specific knowledge about the transformers, and (ii) the language can be used to describe abstract transformers beyond those of points-to analysis. Point (i) is especially important as specifying tricky parts of the transformers by hand requires substantial effort, which is exactly the process we would like to automate. Indeed, we aim at a language that is expressive enough to capture complex transformers which use information from method arguments, fields, assignments, etc., yet can be automatically discovered during the learning.

To achieve this, the main idea is to define $\mathcal{L}_{pt}$ to work over Abstract Syntax Tree (AST) by providing means of navigating and conditioning on different parts of the tree. Further, we do not require the analysis to compute the results directly (e.g., a concrete points-to set for a given location). Instead, we allow the results to be specified indirectly by means of navigating to an AST position that determines the result. For example, such locations in the AST correspond to program positions with the same points-to set for points-to analysis, or to declaration sites for scope analysis or to program positions with the same type for type analysis. Next, we discuss the syntax and semantics of $\mathcal{L}_{pt}$.

*Syntax* The syntax of $\mathcal{L}_{pt}$ is summarized in Fig. 6 and consists of two kinds of basic instructions – `Move` instructions that navigate over the tree and `Write` instructions that accumulate facts about the visited nodes. We split the `Move` instructions into three groups where $\texttt{Move}_{\texttt{core}}$ include language and analysis independent instructions that navigate over trees, $\texttt{Move}_{\texttt{js}}$ include instructions that navigate to a set of interesting program locations that are specific to the JavaScript language. Finally we include $\texttt{Move}_{\texttt{call}}$ which allows learning of a call-site sensitive analysis. Using the `Move` and `Write` instructions we then define an action to be a sequence of `Move` instructions and a guard to be a sequence of both `Move` and `Write` instructions.

*Semantics* Programs from $\mathcal{L}_{pt}$ operate on a state $\sigma$ defined as follows: $\sigma = \langle t, n, ctx, i\rangle \in States$ where the domain $States = AST \times X \times Context \times \mathbb{I}^*$. In

$$m \in \texttt{Move}_{\texttt{core}} ::= \texttt{Up} \mid \texttt{Left} \mid \texttt{Right} \mid \texttt{DownFirst} \mid \texttt{DownLast} \mid \texttt{Top}$$
$$m \in \texttt{Move}_{\texttt{js}} ::= \texttt{GoToGlobal} \mid \texttt{GoToUndef} \mid \texttt{GoToNull} \mid \texttt{GoToThis} \mid \texttt{UpUntilFunc}$$
$$m \in \texttt{Move} ::= \texttt{Move}_{\texttt{core}} \cup \texttt{Move}_{\texttt{call}} \cup \texttt{Move}_{\texttt{js}} \qquad m \in \texttt{Move}_{\texttt{call}} ::= \texttt{GoToCaller}$$
$$w \in \texttt{Write} ::= \texttt{WriteValue} \mid \texttt{WritePos} \mid \texttt{WriteType} \mid \texttt{HasLeft} \mid \texttt{HasRight} \mid \texttt{HasChild}$$
$$a \in \mathit{Actions}_{pt} ::= \epsilon \mid \texttt{Move} \ ; \ a$$
$$g \in \mathit{Guards}_{pt} ::= \epsilon \mid \texttt{Move} \ ; \ g \mid \texttt{Write} \ ; \ g$$
$$\mathit{ctx} \in \mathit{Context} ::= (N \cup \Sigma \cup \mathbb{N})^*$$
$$l \in \mathcal{L}_{pt} ::= \epsilon \mid a \mid \textbf{if} \ \ g = \mathit{ctx} \ \ \textbf{then} \ \ l \ \ \textbf{else} \ \ l$$

**Fig. 6.** Language $\mathcal{L}_{pt}$ for expressing the result of points-to query by means of navigating over an abstract syntax tree.

a state $\sigma = \langle t, n, \mathit{ctx}, i \rangle$, $t$ is an abstract syntax tree, $n$ is the current position in the tree, $\mathit{ctx}$ is the currently accumulated context and $i$ is the current call trace. The accumulated context $\mathit{ctx} \in \mathit{Context} = (N \cup \Sigma \cup \mathbb{N})^*$ by a $\mathcal{L}_{pt}$ program is a sequence of observations on the tree where each observation can be a non-terminal symbol $N$ from the tree, a terminal symbol $\Sigma$ from the tree or a natural number in $\mathbb{N}$. Initially, execution starts with the empty context $[] \in \mathit{Context}$ and the AST $t$, initial node $n$ and current call trace $i$ supplied as arguments.

For a program $p \in \mathcal{L}_{pt}$, a tree $t \in \mathit{AST}$, and a position $n \in X$ in the tree, we say that program $p$ computes a position $n' \in X$, denoted as $p(t, n, i) = n'$, iff there exists a sequence of transitions from $\langle p, t, n, [], i \rangle$ to $\langle \epsilon, t, n', [], i \rangle$. That is, $n'$ is the last visited position by executing the program $p$ on a tree $t$ starting at position $n$. The context is empty both at the beginning and at the end of execution as it is used only to evaluate the **if** condition when deciding which branch to take. We provide the small-step semantics of `Move` and `Write` instructions as well as the **`if-then-else`** statement, in the Appendix, Section A.

*Example* Consider the following program in $\mathcal{L}_{pt}$ that encodes the Assign rule illustrated in Section 3:

$$f(t, n, i) = \begin{cases} \texttt{Right} & \textbf{if } \texttt{WritePos Up WriteType = 1 Assignment} \\ \texttt{Top} & \textbf{else} \end{cases}$$

When executed on a tree $t$ in Fig. 2 (b) starting at position $n =$ `Identifier:a`, the program $f$ first executes the guard `WritePos Up WriteType` which starts by writing value `1` as the node at current position is the first child, then navigates to the position of parent node and writes its type `Assignment`. This collected context `1 Assignment` is then compared to the one specified in the condition. The equality is satisfied and the program takes the **`if`** branch, resets the current position back to the position $n$ before executing the context inside the **`if`** branch, and then continues executing the code inside the **`if`**, `Right`, which navigates to its right sibling. This sibling is also the output of executing program $f(t, n, i)$.

# 9 Implementation

In this section we describe the implementation details of our approach.

## 9.1 Obtaining Programs's Concrete Behaviors $[\![p]\!]_{ti}$

We extract the relevant concrete behaviours of the program $p$ by instrumenting it such that when executed, $p$ produces a trace $\pi$ consisting of all object reads, method enters, method exits and call sites. Additionally, at each method entry, we record the reads of all the parameters and the value of `this`. Further, every element in the trace contains a mapping to the location in the program (in our case to the corresponding node in the AST) and object reads record the unique identifier of the object being accessed. Given such a trace $\pi$, we create a dataset $\mathcal{D}$ used for points-to analysis by generating one input/output example for each position in the trace $\pi$ at which the `this` variable was read. Further, we select only the first read of `this` in each scope as all such references point to the same object. An input/output example is a pair $\langle AST \times X \times \mathbb{I}^*, O \rangle$, where $t \in AST$ is an abstract syntax tree corresponding to the input program, $n \in X$ is a position in the tree where a given read was performed, $i \in \mathbb{I}^*$ is a call trace and $o \in O$ is the identifier of the concrete object seen during execution.

## 9.2 Checking Analysis Correctness

For a program analysis $pa$ and a dataset $\mathcal{D}$, we are interested in checking whether the analysis results computed for program $p$ are correct with respect to the concrete values seen during the execution of $p$. Recall (from Section 8) that executing the analysis $pa \in \mathcal{L}_{pt}$ on an input example $\langle t, n, i, o \rangle$ (as defined above) produces a position $n' = pa(t, n, i)$ in the program or the element $\top$. If the analysis returns $\top$ then it is trivially correct, otherwise we distinguish between two cases. If $n' = n$, we say that the analysis is correct if the value $o$ has not been seen in the trace $\pi$ before position $n$. This is true when position $n$ is a new allocation site. If $n' \neq n$, we say that the analysis is correct if the value $o$ has been seen previously in the trace at position $n'$.

## 9.3 Synthesising $\mathcal{L}_{pt}$ Programs

We instantiate the learning described in Section 6 using the following two program generators *genAction* and *genBranch* for the $\mathcal{L}_{pt}$ language. We instantiate *genAction* using an enumerative search that considers all programs up to size 5. To instantiate *genBranch* we also use enumerate search that considers as conditions all programs up to size 6 (with up to 5 move and 1 write instruction). To determine the concrete value used as a right-hand side of the condition, we collect the top 10 most common values observed when executing the condition and pick one that maximizes the information gain metric as defined in Section 6.5. For the dataset sizes used in our work such simple generators proved to be effective in practice. To scale for larger datasets one could use the idea of representative sampling [29] that was shown to work well for the domain of programs.

**Table 2.** The effect of using the learned analysis to guide the search for counter-examples compared to a "black box" approach.

| | Programs explored until first counter-example is found | |
|---|---|---|
| Difficulty | "Black Box" | Guided by Analysis |
| Easy ($\approx 60\%$) | 146 | 13 |
| Hard ($\approx 40\%$) | $> 3000$ | 130 |

### 9.4 JavaScript Restrictions

Finally, we remove from the training data programs that use the `eval` function, dynamic function binding using `Function.prototype.bind` and `Function` object constructor. These are language features that require combination of analyses to handle precisely and are therefore typically ignored by static analyzers [22].

## 10 Evaluation

In this section we provide an experimental evaluation of our approach instantiated to the problem of learning points-to analysis as described in Section 7 and Section 8. We show that:

- Our counter-example based learning is important for scalability and leads to an order of magnitude more efficient search for a counter-example.
- The approach can learn a practical points-to analysis for JavaScript's built-in objects and is easy to interpreted and incorporate into existing program analyzers that handle such cases only partially or not at all.
- The counter-example based learning is critical for ensuring that the learned analysis generalizes well and does not overfit to the training dataset.

The experiments were performed on a 28 core machine with 2.60Ghz Intel(R) Xeon(R) CPU E5-2690 v4 CPU, running Ubuntu 16.04. In our implementation we parallelized both the learning and the search for the counter-examples. Finally, we use `NodeJS` interpreter v4.2.6 to execute JavaScript programs.

*Training dataset* We use the official ECMAScript (ECMA-262) conformance suite[1], which is a collection of over 20 000 test cases specifying the behaviour of a JavaScript interpreter as defined in the ECMAScript ECMA-262 standard[2]. As this conformance suite is based on the latest version of the standard, all existing implementations support only a subset of the testcases. In particular, the `NodeJS` interpreter used in our evaluation can execute (i.e., does not throw a syntax error) 15 675 tests which we use as the training dataset for learning.

[1] `https://github.com/tc39/test262`
[2] `http://www.ecma-international.org/ecma-262/6.0/`

**Table 3.** Dataset size, number of counter-examples found and the size of the learned points-to analysis for JavaScript APIs that affect the points-to set of `this`.

| Function Name | Dataset Size | Counter-examples Found | Analysis Size* |
|---|---|---|---|
| **Function.prototype** | | | |
| `call()` | 26 | 372 | 97 (18) |
| `apply()` | 6 | 182 | 54 (10) |
| **Array.prototype** | | | |
| `map()` | 315 | 64 | 36 (6) |
| `some()` | 229 | 82 | 36 (6) |
| `forEach()` | 604 | 177 | 35 (5) |
| `every()` | 338 | 31 | 36 (6) |
| `filter()` | 408 | 76 | 38 (6) |
| `find()` | 53 | 73 | 36 (6) |
| `findIndex()` | 51 | 96 | 28 (6) |
| **Array** | | | |
| `from()` | 32 | 160 | 57 (7) |
| **JSON** | | | |
| `stringify()` | 18 | 55 | 9 (2) |

* Number of instructions in $\mathcal{L}_{pt}$ (Number of `if` branches)

### 10.1 Oracle Effectiveness for Finding Counter-examples

We begin by evaluating the effectiveness of our oracle in finding counter-examples for the learned analyzer *pa*. As a baseline we compare to an oracle that uses a "black box" approach, that is, it simply applies all possible modifications to a randomly selected program from the training dataset.

For both oracles we measure how many programs are explored on average before the first counter-example is found. As we can see in Table 2, our approach needs to generate on average an order of magnitude fewer programs. This shows that leveraging the learned program analysis to select which programs and program locations to modify, indeed helps the oracle significantly. Additionally, we compare two common cases that arise during learning: (i) a case where the program analysis is imprecise and finding a counter-example is easy, and (ii) a hard case where the counter-example is a corner case not yet covered by the analysis. Intuitively, finding a counter-example becomes harder as the analysis becomes more refined, in which case using the knowledge of the learned program analysis becomes even more important.

### 10.2 Learning Points-to Analysis for JavaScript

We now evaluate the effectiveness of our approach for the task of learning a points-to analysis for JavaScript built-in APIs that affect the binding of `this`. This is a particularly useful task in practice as upon manual inspection existing implementations either model this only partially [12,5] (by modelling only a subset of the behaviors of `Function.prototype` APIs) or not at all [24,15]. This

includes Facebook Flow's static type checker [5] used in industry which produces unsound results when analyzing APIs other than `Function.prototype`.

A summary of our learned analyzer is shown in Table 3. For each API we collected all its usages in the ECMA-262 conformance suite, ranging from only 6 to more than 600, and used them as initial training dataset for the learning. In all cases, a significant amount of counter-examples were needed to refine the analysis and prevent overfitting to the initial dataset. On average, for each API, the learning finished in 14 minutes, out of which 4 minutes were used to synthesise the program analysis and 10 minutes used in the search for counter-examples (cumulatively across all refinement iterations). The longest learning time was 57 minutes for the `Function.prototype.call` API for which we also learn the most complex analysis – containing 97 instructions in $\mathcal{L}_{pt}$. We note that even though the APIs in `Array.prototype` have very similar semantics, the learned programs vary slightly. This is caused by the fact that different number and types of examples were available as the initial training dataset which means that also the oracle had to find different types of counter-examples.

*Learned Program Analysis* To illustrate the complexity of the learned program analysis and the fact that it is easy for it to be interpreted by a human expert, we show the learned analysis for the API `Array.prototype.filter` in Fig. 7. By inspecting the programs in the branches we can see that the analysis learns three different locations in the program to which the `this` object can point-to: the `global` object, a newly allocated object, or the second argument provided to the `filter` function. The analysis also learns the conditions determining which location to select. For example, `this` points to a new allocation site only if the second argument is a primitive value, in which case it is boxed by the interpreter. Similarly, `this` points-to the second argument (if one is provided), except for cases where the second argument is `null` or `undefined`.

```
Array.prototype.filter ::=
 if caller has one argument then
  points-to global object
 else if 2nd argument is Identifier then
  if 2nd argument is undefined then
   points-to global object
  else
   points-to 2nd argument
 else if 2nd argument is This then
  points-to 2nd argument
 else if 2nd argument is null then
  points-to global object
 else //2nd argument is a primitive value
  points-to new allocation site
```

**Fig. 7.** Learned analysis for JavaScript API `Array.prototype.filter`.

For better readability we replaced the sequence of instructions in $\mathcal{L}_{pt}$ used as branch conditions and branch targets with their informal descriptions. For example, the learned sequence that denotes the second argument of the calling method is `GoToCaller DownFirst Right Right`. It is important to note that that we were not required to manually provide any such sequences in the language but that the learning algorithm discovered such relevant sequences automatically.

### 10.3 Analysis Generalization

First, we manually inspected the learned program analyses at the first iteration of the *Synthesize* procedure to check if we overfit to the initial dataset and found that indeed, the initial analysis would not generalize to programs outside the provided dataset. The two main reasons were conditioning on an unrelated regularity found in the dataset (such as particular variable name) and conditioning on a semantically relevant fact but in a way that does not generalize well. In the latter case, a typical example is retrieving a function argument by simply relying that it will be present at a certain position. Our oracle, however, eliminates such kinds of non-semantic analyses by introducing additional function arguments and statements in the test cases. Overfitting to the initial dataset was also caused by the large search space of possible programs in the DSL for the analysis. However, we decided not to restrict the language, because a more expressive language allows us to automatically learn relevant facts used in the analysis from the data instead of defining them manually up-front.

Next, we quantify the need for the counter-example based refinement loop used in our learning in order to achieve good generalization. For this purpose, we compare the learning with our refinement loop vs. learning without the refinement loop but with more data provided up-front.

In the first approach, we start with a dataset $\mathcal{D}$ that is iteratively refined with counter-examples to produce a dataset $\mathcal{D} \subseteq \mathcal{D}'$ and analysis $pa'$. In the second case, a new dataset $\mathcal{D} \subseteq \mathcal{D}''$ is generated by applying all possible program modifications on all programs in $\mathcal{D}$, before performing the learning which produces the analysis $pa''$ (without further changing $\mathcal{D}''$). We find that in our experiments $\mathcal{D}' \ll \mathcal{D}''$: $\mathcal{D}'$ is up to two orders of magnitude smaller than $\mathcal{D}''$.

An analysis that generalizes well should be sound and precise on all of the datasets $\mathcal{D}$, $\mathcal{D}'$ and $\mathcal{D}''$. For analysis $pa'$ that is learned using counter-examples this is indeed the case as on dataset $\mathcal{D}''$ it achieves precision 99.9% with 0.01% results approximated to the top element in the lattice (that is, it does not produce a trivially correct, but useless result).

Interestingly, the converse is not true as evaluating the analysis $pa''$ on dataset $\mathcal{D}'$ results in precision of only 70.1% with the remaining 29.1% results being unsound. This high number of unsound results is due to the fact the dataset $\mathcal{D}'$ is relatively small (as shown in Table 3) with a large number of corner cases. More importantly, the reason is that programs in $\mathcal{D}''$ are one modification away from the initial programs $\mathcal{D}$ while programs $\mathcal{D}'$ generally get "further away" from programs in $\mathcal{D}$, with each refinement iteration. This is a fundamental issue that cannot be solved by generating more data initially. Note that generating all programs with two modifications is already infeasible in practice as their number grows exponentially with respect to the number of modifications.

*Summary* Overall, our evaluation shows that the learning approach presented in our work can quickly (on average in $\approx$ 14 minutes) learn analyses that handle various corner cases in selected points-to tasks. The analysis can be interpreted by looking at the learned program which enables better integration within ex-

isting analysis tools (compared to using the analysis as a "black box") as well as evaluating the analysis correctness by hand. We believe that this is a useful first step towards learning more complex analyses in the future.

## 11 Related Work

We now discuss recent research that is most closely related to our work.

*Synthesis from examples* Over the last several years, the research community has introduced a number of synthesis from example approaches. Here, one typically starts with a domain-specific language (DSL) which captures a hypothesis space of possible programs together with a set of examples the program must satisfy. The synthesizer then tries to discover the wanted program, in the process potentially asking for more examples from the user/oracle. If a specification is available, an SMT solver or another oracle can provide additional data points in the form of counter-examples using CEGIS-like techniques [34]. Examples of this direction include discovery of bit manipulation programs [18], string processing in spreadsheets [13], functional programs [7], and data structure specifications [9]. Recent work has also showed how to generalize the setting of learning from examples to deal with both, large as well as noisy datasets [29]. Other recent works [14,17] synthesize models for library and framework code. Similarly to us, a dataset is obtained by collecting program traces which are then used as a specification for a synthesis algorithm that tries to generate a model that satisfies them. The key difference from our work is that, while we aim to explain the values seen in the trace (by means of synthesizing a static analysis), `Mimic` [14] and `Pasket` [17] try to generate a model that exhibits the same behavior seen in the trace (e.g., by using a predefined set of program templates [17]).

*Program analysis and machine learning* We can think of the probabilistic type prediction approaches [19,30] as being a kind of program analyzer which predicts types of variables with certain probability. Such approaches work for lightweight predictions shared among many projects (e.g., a `string` or an `integer`), but are not suitable to the more general setting of predicting program specific properties (e.g., the object a variable points to). Other work connecting analysis and machine learning is that of [25]. Here the authors start with the analysis rules already specified, in their case, in Datalog. Since the analysis tends to produce false positives, the authors then learn weights assigned to the rules which allow them to avoid deriving certain facts at the fixed point (and avoid some false positives). Another recent work [26] explores the use of machine learning for analysis, this time however, the goal is to find a strategy for deciding which variables should be tracked flow insensitively. This approach aims to optimize the performance of the analysis by tracking fewer variables flow sensitively. A key difference compared to our work is that we present a method to learn the static analysis rules which can then be applied in an iterative manner. This is a more complex task than [19,30] which do not learn rules that can infer program specific properties and [25,26] which assume the rules are already provided.

*Learning from "Big Code"* Another line of work emerging in recent years has been that of learning from large data sets of programs (termed "Big Code"). The goal here is to learn probabilistic models which capture common patterns found in "Big Code" and to then build probabilistic tools based on these models. Such tools can then make predictions about programs and their properties with high precision, simplifying program development and reasoning. Recent examples of tasks addressed using this approach include code synthesis [2,29,31], predicting names [1,30], predicting type annotations [19,30] and others.

*Learning invariants* In an orthogonal effort there has also been work on learning program invariants using dynamic executions. For recent representative examples of this direction, see [8,20,32]. The focus of all of these works is rather different: they work on a per-program basis, exercising the program, obtaining observations and finally attempting to learn the invariants. Counter-example guided abstraction refinement (CEGAR) [3] is a classic approach for learning an abstraction (typically via refinement). Unlike our work, these approaches do not learn the actual program analysis which once learned can be applied to any program (thus they also do not need to learn from multiple programs).

*Scalable program analysis* Another line of work considers scaling program analysis in hard to analyse domains such as JavaScript at the expense of analysis soundness [6,24] (while still producing results useful for many practical purposes). These works are orthogonal to us and follow the traditional way of designing the static analysis components by hand. Instead, they can benefit from the approach proposed here, especially since they are already defined as a set of rules deriving new facts about programs.

## 12 Conclusion and Future Work

We presented the first approach for learning static analyzers from examples. Our approach takes as input a language for the abstract transformers, an abstraction function and an initial dataset of programs. Then, we introduce a counter-example guided search to iteratively add new programs that the learned analyzer should consider. These programs aim to capture corner cases of the programming language being analyzed. The counter-example search is made feasible thanks to an oracle able to quickly generate candidate example programs for the analyzer.

We implemented our approach and applied it to the challenging setting of learning a points-to analysis for JavaScript. This is a very difficult problem for learning yet one that is of practical importance. We show that our learning approach was able to discover a new analyzer which covered corner cases missed by prior, manually crafted analyzers for JavaScript.

Overall, we believe this is an interesting research direction with several possible future work items. For instance, learning how to model the interface of large JavaScript libraries w.r.t to a given analysis, or learning the transformers for other analyzers (e.g., type analysis, numerical analysis, etc).

## References


1. M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 38–49, 2015.
2. P. Bielik, V. Raychev, and M. Vechev. PHOG: Probabilistic model for code. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 2933–2942, 2016.
3. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. *Counterexample-Guided Abstraction Refinement*, pages 154–169. Springer Berlin Heidelberg, 2000.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM.
5. Facebook. Facebook Flow: Static typechecker for javascript. `https://github.com/facebook/flow`, 2016.
6. A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for javascript ide services. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 752–761, 2013.
7. J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.
8. P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*, pages 499–512, 2016.
9. T. Gehr, D. Dimitrov, and M. T. Vechev. Learning commutativity specifications. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 307–323, 2015.
10. R. Giacobazzi, F. Logozzo, and F. Ranzato. Analyzing program analyses. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 261–273. ACM, 2015.
11. P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, Jan. 2012.
12. S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 151–168, 2009.
13. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 317–330, 2011.
14. 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, pages 710–720, 2015.
15. D. Jang and K.-M. Choe. Points-to analysis for javascript. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1930–1937, 2009.
16. S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

17. J. Jeon, X. Qiu, J. Fetter-Degges, J. S. Foster, and A. Solar-Lezama. Synthesizing framework models for symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 156–167, 2016.
18. S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. ICSE '10, pages 215–224, 2010.
19. O. Katz, R. El-Yaniv, and E. Yahav. Estimating types in binaries using predictive modeling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*, pages 313–326, 2016.
20. S. Kowalewski and A. Philippou, editors. *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2009*, volume 5505 of *LNCS*. Springer, 2009.
21. V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 216–226, 2014.
22. B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, Jan. 2015.
23. U. v. Luxburg and B. Schoelkopf. Statistical learning theory: Models, concepts, and results. In *Inductive Logic*, pages 651–706. 2011.
24. M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 499–509, New York, NY, USA, 2013. ACM.
25. R. Mangal, X. Zhang, A. V. Nori, and M. Naik. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 462–473, 2015.
26. H. Oh, H. Yang, and K. Yi. Learning a strategy for adapting a program analysis via bayesian optimisation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015*, pages 572–588, 2015.
27. J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, Mar. 1986.
28. V. Raychev, P. Bielik, and M. Vechev. Probabilistic model for code with decision trees. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA 2016, pages 731–747.
29. V. Raychev, P. Bielik, M. Vechev, and A. Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 761–774, 2016.
30. V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, 2015.
31. V. Raychev, M. T. Vechev, and E. Yahav. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, page 44, 2014.
32. R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 388–411, 2013.
33. Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.
34. A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006*, pages 404–415, 2006.

# Appendix

## A Language for Points-To Inference Rules

Here, we provide the semantics of all `Move` and `Write` instructions as presented in Section 8. Further, we provide small-step semantics of $\mathcal{L}_{pt}$ language.

### A.1 $\mathcal{L}_{pt}$: Semantics of Instructions

The semantics of the write instructions are described by the [Write] rule in Fig. 8. Each write accumulates a value $c$ to the context $ctx$ according to the function $wr$:

$$wr\colon \texttt{Write} \times AST \times X \times \mathbb{I}^* \to N \cup \Sigma \cup \mathbb{N}$$

defined as follows:

- $wr(\texttt{WriteType}, t, n, i)$ returns $x$ where $x \in N$ is the non-terminal symbol at node $n$.
- $wr(\texttt{WriteValue}, t, n, i)$ returns the terminal symbol at node $n$ if one is available or a special value 0 otherwise, and
- $wr(\texttt{WritePos}, t, n, i)$ returns a number $x \in \mathbb{N}$ that is the index of $n$ in the list of children kept by the parent of $n$.
- $wr(\texttt{HasLeft}, t, n, i)$ and $wr(\texttt{HasRight}, t, n, i)$ return 1 if the $n$ has a left (right) sibling and 0 otherwise.
- $wr(\texttt{HasChild}, t, n, i)$ returns 1 if the $n$ has atleast one children and 0 otherwise.
- $wr(\texttt{HasCaller}, t, n, i)$ returns 1 if the call trace is non-empty (i.e., $|i| > 0$) and 0 otherwise.

Move instructions are described by the [Move] and [Move-Fail] rules in Fig. 8 and use the function $mv$:

$$mv\colon \texttt{Move} \times AST \times X \times \mathbb{I}^* \to X \times \mathbb{I}^*$$

defined as follows:

- $mv(\texttt{Up}, t, n, i) = n' \times i$ where $n'$ is the parent node of $n$ in $t$ or $\bot$ if $n$ has no parent node in $t$. Note that the [Move] rule updates the node at the current position to be the parent.
- $mv(\texttt{Left}, t, n, i) = n' \times i$ where $n'$ is the left sibling of $n$ in $t$ or $\bot$ if $n$ has no left sibling. Similarly, $mv(\texttt{Right}, t, n)$ produces the right sibling or $\bot$ if $n$ has no right sibling.
- $mv(\texttt{DownFirst}, t, n, i) = n' \times i$ where $n'$ is the first child of $n$ in $t$ or $\bot$ if $n$ has no children. Similarly, $mv(\texttt{DownLast}, t, n, i)$ produces the last child of $n$ or $\bot$ if $n$ has no children.
- $mv(\texttt{GoToGlobal}, t, n, i) = n' \times i$ where $n'$ is a node corresponding to the `global` JavaScript object in the $t$. For this and other `GoTo` operations the value of $n'$ is independent of the starting node $n$.

$$t \in AST \qquad n \in X \qquad ctx \in Context \qquad i \in \mathbb{I}^* \qquad s \in \mathcal{L}_{pt}$$

$$\frac{op \in \texttt{Move} \qquad n' \times i' = mv(op, t, n, i) \qquad n' \notin \{\bot, \top\}}{\langle op :: s, t, n, ctx, i\rangle \rightarrow \langle s, t, n', ctx, i'\rangle} \text{ [Move]}$$

$$\frac{op \in \texttt{Move} \qquad n' \times i' = mv(op, t, n, i) \qquad n' \in \{\bot, \top\}}{\langle op :: s, t, n, ctx, i\rangle \rightarrow \langle \epsilon, t, n', ctx, i\rangle} \text{ [Move-Fail]}$$

$$\frac{op \in \texttt{Write} \qquad c = wr(op, t, n, i)}{\langle op :: s, t, n, ctx, i\rangle \rightarrow \langle s, t, n, ctx \cdot c, i\rangle} \text{ [Write]}$$

$$\frac{op \in \textbf{if} \;\; g = ctx \;\; \textbf{then} \;\; l_{true} \;\; \textbf{else} \;\; l_{false} \qquad \langle g, t, n, [], i\rangle \rightarrow \langle \epsilon, t', n', ctx', i'\rangle \qquad ctx = ctx'}{\langle op, t, n, ctx, i\rangle \rightarrow \langle l_{true}, t, n, ctx, i\rangle} \text{ [If-True]}$$

$$\frac{op \in \textbf{if} \;\; g = ctx \;\; \textbf{then} \;\; l_{true} \;\; \textbf{else} \;\; l_{false} \qquad \langle g, t, n, [], i\rangle \rightarrow \langle \epsilon, t', n', ctx', i'\rangle \qquad ctx \neq ctx'}{\langle op, t, n, ctx, i\rangle \rightarrow \langle l_{false}, t, n, ctx, i\rangle} \text{ [If-False]}$$

**Fig. 8.** $\mathcal{L}_{pt}$ language small-step semantics. Each rule is of the type: $\mathcal{L}_{pt} \times States \rightarrow \mathcal{L}_{pt} \times States$.

- $mv(\texttt{GoToThis}, t, n, i) = n' \times i$ where $n'$ is a node corresponding to the object to which `this` keyword points-to in the top-level scope. In a web browser this would be `window` object while in `Node.js` application it is `module.exports`.
- $mv(\texttt{GoToUndefined}, t, n, i) = n' \times i$ where $n'$ is a node corresponding to the `undefined` JavaScript object in the $t$. Similarly, for $mv(\texttt{GoToNull}, t, n, i) = n' \times i$ the $n'$ is the `null` value.
- $mv(\texttt{GoToCaller}, t, n, i \cdot i') = n' \times i'$ where $n'$ is the node corresponding to call site of the top method $i$ from call trace and $i'$ is the call trace with the method $i$ removed. If the call trace is empty then $n' = \bot$.
- $mv(\texttt{UpUntilFunc}, t, n, i) = n' \times i$ navigates recursively to the first $n'$ using the `Up` operation such that the parent of $n'$ is a function declaration or root of the tree is reached.
- $mv(\texttt{Top}, t, n, i) = \top \times i$ denotes that the analysis approximates the result to the $\top$ element in the lattice.

### A.2 $\mathcal{L}_{pt}$: Small-step Semantics of $\mathcal{L}_{pt}$ language.

Recall from Section 8 that $\mathcal{L}_{pt}$ programs operate on a state $\sigma$ defined as follows: $\sigma = \langle t, n, ctx, i\rangle \in States$ where the domain $States$ is defined as $States = AST \times X \times Context \times \mathbb{I}^*$. Initially, execution starts with the empty context $[] \in Context$ and for a program $p \in \mathcal{L}_{pt}$, a tree $t \in AST$, and a position $n \in X$ in the tree, we say that program $p$ computes a position $n' \in X$, denoted as $p(t, n, i) = n'$, iff there exists a sequence of transitions from $\langle p, t, n, [], i\rangle$ to $\langle \epsilon, t, n', [], i\rangle$. The small-step semantics of executing a $\mathcal{L}_{pt}$ program are shown in Fig. 8.