Efficient Bottom-Up Synthesis for Programs with Local Variables

XIAOULI∗, University of Michigan, USA
XIAOYU ZHOU∗, University of Michigan, USA
RUI DONG, University of Michigan, USA
YIIHONG ZHANG, University of Washington, USA
XINYU WANG, University of Michigan, USA

We propose a new synthesis algorithm that can efficiently search programs with local variables (e.g., those introduced by lambdas). Prior bottom-up synthesis algorithms are not able to evaluate programs with free local variables, and therefore cannot effectively reduce the search space of such programs (e.g., using standard observational equivalence reduction techniques), making synthesis slow. Our algorithm can reduce the space of programs with local variables. The key idea, dubbed lifted interpretation, is to lift up the program interpretation process, from evaluating one program at a time to simultaneously evaluating all programs from a grammar. Lifted interpretation provides a mechanism to systematically enumerate all binding contexts for local variables, thereby enabling us to evaluate and reduce the space of programs with local variables. Our ideas are instantiated in the domain of web automation. The resulting tool, Arboretist, can automate a significantly broader range of challenging tasks more efficiently than state-of-the-art techniques including WebRobot and Helena.

CCS Concepts: • Software and its engineering → Automatic programming; Programming by example.

Additional Key Words and Phrases: Program Synthesis, Observational Equivalence, Web Automation

ACM Reference Format:
https://doi.org/10.1145/3632894

1 INTRODUCTION

Web automation can automate web-related tasks such as scraping data and filling web forms. While an increasing number of populations have found it useful [Chasins 2019; Katongo et al. 2021; UiPath 2022], it is notoriously difficult to create web automation programs [Krosnick and Oney 2021]. Let us consider the following example, which we also use as a running example throughout the paper.

Example 1.1. https://haveibeenpwned.com/ is a website where one could check whether or not an email address has been compromised (i.e., “pwned”). Figures 1 and 2 show the webpage DOMs for an email with no pwnage detected and for a pwned email respectively; both figures are simplified from the original DOMs solely for presentation purposes. Consider the task of scraping the pwnage text for each email from a list of emails. Figure 3 shows an automation program P for this task.

∗Xiang Li and Xiangyu Zhou contributed equally to this work.
1This is a real-life task from the iMacros forum: https://forum.imacros.net/viewtopic.php?f=7&t=26683.

Authors’ addresses: Xiang Li, University of Michigan, USA, xkevli@umich.edu; Xiangyu Zhou, University of Michigan, USA, xiangyz@umich.edu; Rui Dong, University of Michigan, USA, ruidong@umich.edu; Yihong Zhang, University of Washington, USA, yz489@cs.washington.edu; Xinyu Wang, University of Michigan, USA, xwangsd@umich.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2473-1421/2024/1-ART52
https://doi.org/10.1145/3632894
foreach email in list_of_emails:
    EnterData(email, //input[@id='account'] )
    Click( //button[@id='searchPwnage'] )
    ScrapeText( //div[@aria-expanded='true']//h2 ) # enter each email to search box # click search button # scrape pwnage message

Figure 1. DOM when no pwnage detected.

Figure 2. DOM when pwnage detected.

While the high-level logic of $P$ is rather simple, implementing it turns out to be very difficult. First, one must implement the right control-flow structure, such as the loop in $P$ with three instructions. Second, each instruction must use a generalizable selector to locate the desired DOM element for all emails across all iterations. For example, line 4 from Figure 3 uses one generalizable selector that utilizes the aria-expanded attribute. This attribute is necessary for generalization, since both messages (pwned and not pwned) are always in the DOM but which one to render is determined by this attribute’s value. On the other hand, a full XPath expression /html/body/div/div/div/div/h2 locates the element with “Good news – no pwnage found!” in both Figure 1 and Figure 2, which is not desired. In general, one has to try many candidate selectors before finding a generalizable one — in other words, this is fundamentally a search problem.

Synthesizing web automation programs. In general, implementing web automation programs requires creating both the desired control-flow structure (with arbitrarily nested loops potentially) and identifying generalizable selectors for all of its instructions; this is very hard and time-consuming. WebRobot [Dong et al. 2022] is a state-of-the-art technique that allows non-experts to create web automation programs from a short trace of user actions (e.g., clicking a button, scraping text). A key underpinning idea is its trace semantics: given a program $P$, trace semantics outputs a sequence $A'$ of actions that $P$ executes, by resolving any free variables (such as the local loop variable from Figure 3). Then, one can check if $P$ satisfies $A$ by checking if $P$’s output trace $A'$ matches $A$.

Key challenge: synthesizing programs with local variables. While trace semantics significantly bridges the gap between programming-by-demonstration (PBD) and programming-by-example (PBE) by enabling a “guess-and-check” style synthesis approach for web automation [Chen et al. 2023; Dong et al. 2022; Pu et al. 2022, 2023], state-of-the-art synthesis algorithms unfortunately fail to scale to challenging web automation tasks due to the heavy use of local variables in those tasks. Consider the enormous space of potential loop bodies to be searched. All these bodies use loop variables and therefore must be evaluated under an extended context that also binds such local variables to values. Conventional observational equivalence (OE) from the program synthesis literature [Albarghouthi et al. 2013; Udupa et al. 2013] fundamentally cannot reduce this space: they track bindings for only input variables, but not for local variables. As a result, they cannot build equivalence classes for loop bodies, and hence fall back to enumeration. The only existing work (to our best knowledge) that pushed the boundary of OE is RESL [Peleg et al. 2020]. Briefly, its idea
is to “infer” bindings for local variables given a higher-order sketch, which enables applying OE over the space of lambda bodies. A fundamental problem, however, is the data dependency across iterations (for functions like fold). Its implication to synthesis is succinctly summarized by RESL as a “chicken-and-egg” problem: we need output values of programs in order to apply OE for more efficient synthesis, but we need the programs first in order to obtain their output values. This cyclic dependency fundamentally limits all prior work (such as RESL, among others [Feser et al. 2015; Smith and Albarghouthi 2016]) to sketch-based approaches with crafted binding-inference rules, which still resort to enumeration in many cases. The general problem of how to reduce the space of programs with local variables remains open [Peleg et al. 2020].

Our idea: lifted interpretation. The same problem occurs in our domain: as we will show shortly, loops in web automation use local variables and exhibit data dependency across iterations. In this work, we propose a new algorithm that can apply OE-based reduction for any programs, without requiring binding inference. We build upon the OE definition from RESL [Peleg et al. 2020]: two programs belong to the same equivalence class if they share the same context and yield the same output. Notably, “context” here is a binding context with all free variables including both input and local variables. Our key insight can be summarized as follows.

We can compute an equivalence relation of programs based on OE under all reachable contexts, by creating equivalence classes simultaneously while evaluating all programs from a given tree grammar (with respect to a given input). Furthermore, we can use (a generalized form of) finite tree automata to compactly store the equivalence relation.

Here, a reachable context is one that emerges during the execution of at least one program from the grammar, for a given input. Furthermore, programs rooted at the same grammar symbol share reachable contexts. For instance, the loop from Figure 3 introduces the same binding context for loop bodies that can be put inside it. However, computing reachable contexts requires program evaluation which again requires reachable contexts — this is the aforementioned “chicken-and-egg” problem described in RESL [Peleg et al. 2020]. To break this cycle, our key insight is to simultaneously evaluate all programs top-down from the grammar, during which we construct equivalence classes of programs bottom-up based on their outputs under their reachable contexts. This idea essentially lifts up an interpreter from evaluating one single program at a time to simultaneously evaluating all programs from a grammar, with respect to a given input. This lifted interpretation process allows us to systematically enumerate all reachable contexts, and hence build equivalence classes for all programs including those with local variables.

General idea, instantiation, and evaluation. In the rest of this paper, we first illustrate how our idea works in general in Section 3 using a small functional language. Then, in Section 4, we present an instantiation of our approach in the domain of web automation. We implement this instantiation in a tool called ARBORIST². Our evaluation results show that ARBORIST can solve more challenging benchmarks using much less time, significantly advancing the state-of-the-art for web automation.

Contributions. This paper makes the following contributions.

- We propose a new synthesis algorithm, based on lifted interpretation and finite tree automata, that can reduce the search space of programs with local variables.
- We instantiate this idea and develop a new synthesis algorithm for web automation.
- We implement our instantiation in a tool called ARBORIST and evaluate it on 131 benchmarks. Our results highlight that the idea of lifted interpretation yields a significantly faster synthesizer.

²ARBORIST is a specialist that can manage a lot of trees (i.e., programs), even if they have local variables.
2 PRELIMINARIES

In this section, we review the standard concepts of observational equivalence (OE) and finite tree automata (FTAs) from the literature, focusing on their application to program synthesis.

2.1 Synthesis using Observational Equivalence

Observational equivalence (OE) was originally proposed by Hennessy and Milner [1980] to define the semantics of concurrent programs, and has been used widely within the programming languages community. Intuitively, two terms are observationally equivalent whenever they are interchangeable in all observable contexts. In the field of programming-by-example (PBE), OE has been utilized to reduce the search space of programs, typically in bottom-up synthesis algorithms.

**Bottom-up synthesis.** Bottom-up algorithms synthesize programs by first constructing smaller programs which are later used as building blocks to create bigger ones. Specifically, the algorithm begins with an initial set \( W \) containing all atomic programs of size 1 (e.g., input variables, constants), and then iteratively grows \( W \) by adding new programs of larger sizes that are composed of those already in \( W \). The algorithm terminates when \( W \) has a program \( P \) that meets the given specification. For instance, for a language that includes variable \( x \) and integer constants 1 and 2, \( W \) is initially \{x, 1, 2\} but later will contain more terms such as \( x + 1 \) and \( x + 2 \), assuming + operator is allowed by the language. If the specification is given as an input-output example pair \((1, 3)\) meaning “return 3 when \( x = 1 \), then \( x + 2 \) is a correct program whereas \( x + 1 \) is not.

**Observational equivalence reduction.** Bottom-up synthesis often uses observational equivalence to reduce the program space in order to improve the search efficiency. The key idea is to not add a new program \( P \) to \( W \), if there already exists some \( P' \in W \) that behaves the same as \( P \) observationally. In particular, existing PBE work [Albarghouthi et al. 2013; Udupa et al. 2013] defines two programs \( P_1, P_2 \) to be observationally equivalent if they yield the same output on each input example. This idea keeps only programs that are observationally distinct (given input examples), thereby reducing the size of \( W \) and accelerating the search. RESL [Peleg et al. 2020] further generalizes OE to consider an extended context that also includes local variables: two programs are observationally equivalent if they yield the same output given a shared context (which may include local variables). However, conventional bottom-up synthesis algorithms no longer work under this OE definition, as they cannot evaluate programs with free local variables before knowing their binding context.

2.2 Synthesis using Finite Tree Automata

OE essentially defines an equivalence relation of programs, which can be stored using finite tree automata (FTAs) [Wang et al. 2017a].

**Finite tree automata.** Finite tree automata (FTAs) [Comon et al. 2008] deal with tree-structured data: they generalize standard finite (word) automata by accepting trees rather than words/strings.

**Definition 2.1 (Finite Tree Automata).** A (bottom-up) finite tree automaton (FTA) over alphabet \( F \) is a tuple \( \mathcal{A} = (Q, F, Q_f, \Delta) \), where \( Q \) is a set of states, \( Q_f \subseteq Q \) is a set of final states, and \( \Delta \) is a set of transitions of the form \( f(q_1, \ldots, q_n) \rightarrow q \) where \( q_1, \ldots, q_n, q \in Q \) and \( f \in F \). A term \( t \) is accepted by \( \mathcal{A} \) if \( t \) can be rewritten to a final state according to the transitions (i.e., rewrite rules). The language of \( \mathcal{A} \), denoted \( L(\mathcal{A}) \), is the set of terms accepted by \( \mathcal{A} \).

**Notations.** We also use \( \mathcal{A} = (Q_f, \Delta) \) as a simpler notation, since \( Q \) and \( F \) can be determined by \( \Delta \). We use \( \text{SubFTA}(q, \mathcal{A}) \) to mean the sub-FTA of \( \mathcal{A} \) that is rooted at state \( q \).

**Program synthesis using FTAs.** Given a tree grammar \( G \) defining the syntax of a language and given an input-output example \( \mathcal{E} = (\mathcal{E}_{\text{in}}, \mathcal{E}_{\text{out}}) \), we can construct an FTA \( \mathcal{A} = (Q, F, Q_f, \Delta) \), such that \( L(\mathcal{A}) \) contains all programs from \( G \) (up to a finite size) that satisfy \( \mathcal{E} \). In particular, the alphabet...
Figure 4. A simple functional language. Here, \( x \) is the input variable, which is a list of integers. We simplify the standard fold operator to use a default seed of 0 (which is implicit and not shown as an argument). Note that fold introduces two local variables: acc is the accumulator, and elem will be bound to each element from \( L \). \( E \) is the lambda body, which may use local variables acc and elem. The “add” and “mult” operators are the standard addition and multiplication.

\[
P ::= \text{fold}(L, (\text{acc}, \text{elem}) \Rightarrow E) \\
E ::= \text{acc} \mid 1 \mid 2 \mid \text{add}(E, E) \mid \text{mult}(E, E) \\
L ::= x
\]

\( F \) consists of all operators from \( G \). We have a state \( q^c_s \in Q \) if there exists a program rooted at symbol \( s \) from \( G \) that outputs \( c \) given input \( E_{\text{in}} \). We have a transition \( f(q^c_{s_1}, \cdots, q^c_{s_n}) \to q^c_{s_0} \in \Delta \) if applying function \( f \) on \( c_1, \cdots, c_n \) yields \( c_0 \). A state \( q^c_s \) is marked final if \( c = E_{\text{out}} \) and \( s \) is a start symbol of \( G \). Once \( A \) is constructed, one can extract a program \( P \) from \( L(A) \) heuristically (e.g., smallest in size) and return \( P \) as the final synthesized program [Wang et al. 2017a].

Remarks. Every state \( q^c_s \in Q \) represents an equivalence class of all programs rooted at grammar symbol \( s \) that produce the same value \( c \) on \( E_{\text{in}} \). In other words, \( q^c_s \) stores all observationally equivalent programs. To our best knowledge, all existing FTA-based synthesis techniques [Miltner et al. 2022; Wang et al. 2017a,b, 2018b; Yaghmazadeh et al. 2018] are based on the notion of OE that considers only input variables. In other words, all existing techniques resort to enumeration of programs with free local variables [Peleg et al. 2020].

3 LIFTED INTERPRETATION

This section illustrates how the general idea of lifted interpretation works on a simple functional language. Section 4 will later describe a full-fledged instantiation to the domain of web automation.

3.1 A Simple Programming-by-Example Task

Example 3.1. Given the simple functional language from Figure 4, let us consider the following programming-by-example (PBE) task: synthesize a program that returns 7 given input list \([1, 2, 4]\). Suppose the intended program is:

\[
P_1 : \text{fold}(x, (\text{acc}, \text{elem}) \Rightarrow \text{add}(\text{acc}, \text{elem}))
\]

which calculates the sum of all elements from the input list \( x \). Consider another program:

\[
P_2 : \text{fold}(x, (\text{acc}, \text{elem}) \Rightarrow \text{add}(\text{mult}(\text{acc}, 2), 1))
\]

which returns the same output 7 as \( P_1 \), given the example input \([1, 2, 4]\). Notably, while the lambda bodies in \( P_1 \) and \( P_2 \) are different, they share the same context-output behaviors (or footprint), for the given input list. Please see Table 1 which shows their local variable bindings and corresponding output values across all iterations. In what follows, we will illustrate how to synthesize \( P_1 \) and \( P_2 \) from the input-output example \([1, 2, 4] \mapsto 7\), using our lifted interpretation idea.

<table>
<thead>
<tr>
<th>iteration</th>
<th>local variable bindings</th>
<th>( P_1 : \text{add}(\text{acc}, \text{elem}) )</th>
<th>( P_2 : \text{add}(\text{mult}(\text{acc}, 2), 1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>iteration 1</td>
<td>( acc \mapsto 0, \text{elem} \mapsto 1 )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>iteration 2</td>
<td>( acc \mapsto 1, \text{elem} \mapsto 2 )</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>iteration 3</td>
<td>( acc \mapsto 3, \text{elem} \mapsto 4 )</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>
3.2 FTAs using Observational Equivalence

Let us first present a new FTA-based data structure that our lifted interpretation approach utilizes to succinctly encode equivalence classes of programs. The main ingredient is its generalization of the FTA state definition from prior work [Wang et al. 2017a]: our state includes a context $C$ which contains information (e.g., all variable bindings) to evaluate programs with free local variables. In particular, we define an FTA state $q$ as a pair:

$$ (s, \Omega) \quad \text{where} \quad \Omega = \{ C_1 \mapsto O_1, \ldots, C_l \mapsto O_l \} $$

Here, $s$ is a grammar symbol, and $\Omega$ is a footprint which maps a context $C_i$ to an output $O_i$. Each entry $C_i \mapsto O_i$ is called a behavior; so a footprint is a set of behaviors. Intuitively, if there exists a program $P$ rooted at $s$ that evaluates to $O_i$ under $C_i$ for all $i \in [1, l]$, then our FTA $\mathcal{A}$ has a state $(s, \{C_1 \mapsto O_1, \ldots, C_l \mapsto O_l\})$, and vice versa. A context is reachable if it can actually emerge, when executing programs in a given grammar for a given input. Given a finite grammar, if all programs terminate, then the number of reachable contexts is finite. Our work assumes a finite number of reachable contexts, which we believe is a reasonable assumption for program synthesis.

The remaining definitions are relatively standard. Our alphabet $F$ includes all operators from the programming language. A transition $\delta \in \Delta$ is of the form $f(q_1, \ldots, q_n) \rightarrow q$ which connects multiple states to one state. However, because our state definition is more general, the condition under which to include a transition now becomes different from prior work [Wang et al. 2017a]. Specifically, $\mathcal{A}$ includes a transition $\delta = f(q_1, \ldots, q_n) \rightarrow q$, if for every behavior $C \mapsto O$ in $q$, we have behavior $C_k \mapsto O_k$ in $q_k$ (for all $k \in [1, n]$), such that according to $f$’s semantics and given $C_1 \mapsto O_1, \ldots, C_n \mapsto O_n$, evaluating $f$ under context $C$ indeed yields output $O$.

3.3 Illustrating Lifted Interpretation

Now we are ready to explain how our lifted interpretation idea works for Example 3.1.

**Setup.** First, we build an FTA $\mathcal{A}_s = (\{q_1\}, \Delta_s)$ — see Figure 5 — for the grammar in Figure 4. We will later apply lifted interpretation to $\mathcal{A}_s$. Each state in $\mathcal{A}_s$ is annotated with a grammar symbol and an empty footprint. Notice the cyclic transitions around $q_3$, due to the recursive add and mult productions. This induces an infinite space of programs — our approach finitizes the grammar by bounding the size of its programs, as standard in the literature [Wang et al. 2017a].

**Applying lifted interpretation.** Then, we use lifted interpretation to “evaluate” $\mathcal{A}_s$ under an initial context, which would eventually produce another FTA $\mathcal{A}_e = (\{q_{11}\}, \Delta_e)$ (see a part of it in Figure 6). Different from $\mathcal{A}_s$, $\mathcal{A}_e$ will cluster programs (including all sub-programs) into equivalence classes based on OE. Figure 7 shows the annotations (i.e., grammar symbols and footprints) for $\mathcal{A}_e$.

At a high-level, lifted interpretation traverses $\mathcal{A}_s$ systematically, computes reachable contexts on-the-fly during traversal, and most importantly, constructs the equivalence classes simultaneously given these reachable contexts. In particular, given an initial context $C$ and an FTA $\mathcal{A}_e = (Q_f, \Delta)$ with final states $Q_f$ and transitions $\Delta$, lifted interpretation returns an FTA $\mathcal{A}_e = (Q'_f, \Delta')$ with final states $Q'_f$ and transitions $\Delta'$, such that if two programs from $L(\mathcal{A}_s)$ yield the same output under $C$, then they belong to the same (final) state in $\mathcal{A}_e$. More specifically, we have:

$$ Q'_f \ = \ \{ q' \mid q_i \in Q_f, C \vdash q_i ; \Delta \rightsquigarrow (q'_i, \Delta') \} $$

$$ \Delta' \ = \ \bigcup_{q_i \in Q_f} \Delta'_i \quad \text{where} \ C \vdash q_i ; \Delta \rightsquigarrow (q'_i, \Delta'_i) $$

That is, if any final state $q_i$ in $\mathcal{A}_s$ “evaluates to” a state $q'_i$ under $C$, then $q'_i$ is a final state of $\mathcal{A}_e$. This process also yields a set $\Delta'_i$ of transitions for each $q_i$, which are added as transitions to $\mathcal{A}_e$. 

Proc. ACM Program. Lang., Vol. 8, No. POPL, Article 52. Publication date: January 2024.
The key inference rule that implements this judgment is the Programs in transitions $\Delta$. It reads as follows: given context $C \vdash q; \Delta \rightsquigarrow (q', \Delta')$.

It reads as follows: given context $C$, state $q$ (with respect to transitions $\Delta$) evaluates to state $q'$ with transitions $\Delta'$. The guarantee is that: for all programs $P$ from $L\{q\}$, $\Delta$ that yield the same output under context $C$, we have one unique state $q'$ (and transitions $\Delta'$) such that these programs $P$ are in $L\{q\}, \Delta\}$. For instance, given $C = \{x \mapsto [1, 2, 4]\}$, $q_1$ from $\mathcal{A}_s$ may evaluate to $q_{11}$ in $\mathcal{A}_e$: all programs in $L\{q\}, \Delta\}$ that yield 7 given $C$ are merged into state $q_{11}$.

**Inference rules.** The key inference rule that implements this judgment is the TRANSITION rule:

$$(\text{TRANSITION}) \quad \delta = f(q_1, \ldots, q_n) \rightarrow q \in \Delta \quad C \vdash \delta; \Delta \rightsquigarrow (q', \Delta')$$

$C \vdash q; \Delta \rightsquigarrow (q', \Delta')$$
It says: evaluating a state \( q \) boils down to evaluating each of \( q \)'s incoming transitions \( \delta \). For instance, to evaluate \( q_1 \) in \( \mathcal{A}_4 \) (see Figure 5) under \( C = \{ x \mapsto [1, 2, 4] \} \), we would evaluate \( \text{fold}(q_2, q_3) \rightarrow q_1 \) under \( C \). The following inference rules describe how to actually evaluate a fold transition.

\[
\begin{align*}
\text{(Fold-1)} & \quad C \vdash q_1; \Delta \rightsquigarrow (q'_1, \Delta'_1) \quad C \vdash \text{lst} \in \text{Footprint}(q'_1) \quad C, \text{lst} \vdash q_2; \Delta \rightsquigarrow (q'_2, \Delta'_2) \\
& \quad v_0 = 0 \quad C, \text{acc} \rightsquigarrow v_{i - 1}, \text{elem} \rightsquigarrow \text{lst}[i] \rightarrow v_i \in \text{Footprint}(q'_2) \quad i \in [1, |\text{lst}|] \\
& \quad \Omega = \text{Footprint}(q) \quad \Omega' = \Omega \cup \{ C \rightsquigarrow v_{\text{lst}[i]} \} \quad q' = \text{MkState}(P, \Omega') \\
& \quad C \vdash \text{fold}(q_1, q_2) \rightarrow q; \Delta \rightsquigarrow (q', \Delta'_1 \cup \Delta'_2 \cup \{ \text{fold}(q'_1, q'_2) \rightarrow q'' \})
\end{align*}
\]

\[
\begin{align*}
\text{(Fold-2)} & \quad v_0 = 0 \quad q'_0 = q \quad \Delta'_0 = \Delta \quad i \in [1, |\text{lst}|] \\
& \quad C_{i - 1} = C[\text{acc} \rightsquigarrow v_{i - 1}, \text{elem} \rightsquigarrow \text{lst}[i]] \\
& \quad q' = \text{MkState}(L, \Omega') \\
& \quad C, \text{lst} \vdash q_1; \Delta \rightsquigarrow (q'_1, \text{lst}[i] | \Delta'_1 \rightarrow q'')
\end{align*}
\]

Let us take \( \text{fold}(q_2, q_3) \rightarrow q_1 \) from \( \mathcal{A}_4 \) as an example and explain how these two rules work.

The Fold-1 rule first evaluates \( q_2 \) from \( \mathcal{A}_4 \) to \( q_4 \) in \( \mathcal{A}_6 \), which, as mentioned above, boils down to evaluating \( q_2 \)'s incoming transition \( x \rightarrow q_2 \). This is done using the following Input-V ar rule.

\[
\begin{align*}
\text{(Input-V ar)} & \quad C[x] = \text{lst} \quad \Omega = \text{Footprint}(q) \quad \Omega' = \Omega \cup \{ C \rightarrow \text{lst} \} \\
& \quad q' = \text{MkState}(L, \Omega') \\
& \quad C \vdash x \rightarrow q; \Delta \rightsquigarrow (q', (x \rightarrow q'))
\end{align*}
\]

Specifically, Input-V ar first obtains the value \( \text{lst} \), which is \([1, 2, 4]\), that \( x \) binds to. Then it creates a footprint \( \Omega' \) that includes all behaviors from \( q \) and a new binding \( C \vdash \text{lst} \). Finally, a new state \( q' \) is created, with grammar symbol \( L \) and footprint \( \Omega' \). Here, MkState is simply a state constructor. In our example, Input-V ar yields \( q_4 \), which has footprint \( \{ \{ x \mapsto [1, 2, 4] \} \} \mapsto [1, 2, 4] \); it means all programs in \( L(\{ q_4 \}, \Delta_e) \) produce \([1, 2, 4]\) given context \( C = \{ x \mapsto [1, 2, 4] \} \).

Popping up to Fold-1, given \( q_4 \), it retrieves the output \( \text{lst} \) for \( q_4 \) given context \( C \), notably, by looking up \( q_4 \)'s footprint. It then uses an auxiliary rule, Fold-2, to evaluate \( q_3 \) from \( \mathcal{A}_6 \) — which corresponds to lambda bodies — given \( C \) and \( \text{lst} \). This yields \( q_7 \) in \( \mathcal{A}_e \), among potentially other states which are not shown in Figure 6. Again, the guarantee is: all programs in \( L(\{ q_7 \}, \Delta_e) \) share the same footprint. Now given \( q_7 \), Fold-1 finally creates \( q_{11} \) in \( \mathcal{A}_e \), as well as transition \( \text{fold}(q_4, q_7) \rightarrow q_{11} \). The key is to compute \( q_{11} \)'s footprint \( \Omega' \), which inherits everything from \( q_1 \)'s footprint \( \Omega \) but also includes additionally the behavior \( C \rightarrow v_{\text{lst}[i]} \). Here, \( v_{\text{lst}[i]} \) (which is \( v_3 \) in our example, as \( |\text{lst}| = 3 \)) is the output for the fold operation, under context \( C = \{ x \mapsto [1, 2, 4] \} \) which is the input example we are concerned with. We note that the computation of \( v_{\text{lst}[i]} \) is based on looking up \( q_7 \)'s footprint.

Now let us briefly explain how the Fold-2 rule evaluates \( q_3 \) to \( q_7 \). The evaluation is an iterative process that follows the fold semantics. It begins with context \( C_0 \) that binds the accumulator \( \text{acc} \) to the default seed \( 0 \) and binds \( \text{elem} \) to the first element of \( \text{lst} \), then recursively evaluates \( q'_0 \) (i.e., \( q_5 \) in our example) under \( C_0 \) which yields \( q'_1 \) (not shown in Figure 6), and finally obtains \( v_1 \) (which \( \text{acc} \) should be bound to in the next iteration) by (again) looking up the footprint of \( q'_1 \). Note that \( q'_1 \) is an intermediate state whose footprint has one behavior, since we have only seen \( C_0 \) so far. The second iteration will repeat the same process but for \( q'_1 \) and using \( C_1 \), which would yield \( q'_2 \) whose footprint has two behaviors. This process continues until we reach the end of \( \text{lst} \), eventually yielding \( q'_{|\text{lst}|}; q_7 \) in \( \mathcal{A}_e \) is one such state. As shown in Figure 7, \( q_7 \) has three behaviors in its footprint.

We skip the discussion of the other rules which are used to construct all the other states in \( \mathcal{A}_e \), and refer readers to the extended version [Li et al. 2023b] of our paper for a complete list of rules. In the end, given \( \mathcal{A}_e \), we will mark states whose footprint satisfies the specification as final states, and extract a program from \( \mathcal{A}_e \). In our example, \( q_{11} \) is final, and both \( P_1 \) and \( P_2 \) are in \( L(\{ q_{11} \}, \Delta_e) \).
Figure 8. Web automation language. Left is syntax, where \( str \) is a string, \( i \) is an integer, \( t \) is an HTML tag, and \( \tau \) is an HTML attribute. Right is a subset of the trace semantics rules; please find the complete set of rules in the extended version [Li et al. 2023b].

### 4 INSTANTIATION TO WEB AUTOMATION

This section presents a full-fledged instantiation of the lifted interpretation idea to web automation.

#### 4.1 Web Automation Language

Figure 8 presents our web automation language. The syntax is slightly different from the one in WebRobot [Dong et al. 2022]. First, our syntax looks more “functional”: for example, loop bodies are presented as lambdas. This is solely for the purpose of making it easier to later present our approach. Second, the language is also slightly more expressive: \( \text{ForSelectors} \) allows starting from the \( i \)-th child/descendant with \( i \geq 1 \), whereas WebRobot’s syntax requires \( i = 1 \). This extension is motivated by our observation when curating new benchmarks: many tasks require this more relaxed form of loop. We refer interested readers to the WebRobot work for more details about the syntax, but in brief, a web automation program \( P \) is always a sequence of statements. It supports different types of statements. For example, \( \text{Click} \) clicks a DOM element located by selector expression \( se \). We use an XPath-like syntax for selector expressions: \( se/\psi[i] \) gives the \( i \)-th child of \( se \) that satisfies \( \psi \), whereas \( // \) considers \( se \)’s descendants. \( x \) is an input variable that is bound to a user-provided data source (like the list of emails from Example 1.1), whereas \( y \) and \( z \) are local variables introduced by and internal to the program. \( \text{ForData} \) is a loop-like statement that iterates over a list of data entries (such as emails from Example 1.1) given by \( de \), binds \( z \) to each of them, and executes loop body \( P \). \( \text{ForSelectors} \) is quite similar, but it loops over a list of selector expressions returned by \( se \). \( \text{While} \) handles pagination, where it repeatedly clicks the “next page” button located by \( se \) and executes loop body \( P \), until it no longer exists on the webpage.

Figure 8 also presents a subset of the trace semantics rules, which are cleaner than WebRobot’s. The evaluation judgment is of the form:

\[
\Pi, \Gamma \vdash P : A', \Pi'
\]

which reads: given a context — consisting of a DOM trace \( \Pi \) and a binding environment \( \Gamma \) (that binds all free variables in scope) — evaluating program \( P \) yields an action trace \( A' \) and a DOM trace.
procedure Synthesize(\( A, \Pi, I \))

input: Action trace \( A = [a_1, \cdots, a_m] \), DOM trace \( \Pi = [\pi_1, \cdots, \pi_m, \pi_{m+1}] \), and input data \( I \).

output: Program \( P \) that generalizes \( A \), or null if no such program can be found.

1: \( \mathcal{A} := \text{InitFTA}([a_1, \cdots, a_m], [\pi_1, \cdots, \pi_m], I) \) where \( \mathcal{A} = (Q, F, Q_f, \Delta) \);
2: while \( \mathcal{A}\_)\_is\_not\_saturated \) and \( \text{not\_timeout} \) do
3: \( \Delta' := \Delta; \)
4: for all \( \delta_1, \cdots, \delta_{2l} \in \Delta' \) where \( \delta_i = \text{Seq}(q'_i, q_i) \rightarrow q_{i-1}, i \in [1, 2l] \) do
  \( \text{\# Find 2l "consecutive" transitions.} \)
5: \( \mathcal{A}_s := \text{SpeculateFTA}([\delta_1, \cdots, \delta_{2l}], \Delta); \)
6: \( \mathcal{A}_e := \text{EvaluateFTA}(\mathcal{A}_s, \text{GetContexts}(q_0)); \) \( \text{\# GetContexts gives all contexts in q_0’s footprint.} \)
7: \( \mathcal{A} := \text{MergeFTAs}(\mathcal{A}, q_0, \mathcal{A}_e); \)
8: return \( \text{Rank}(\mathcal{A}, \Pi, I) \);

Algorithm 1. Top-level synthesis algorithm.

\( \Pi' \). This evaluation does not execute \( P \) in browser; instead, it simulates the execution by “replaying” \( P \) given \( \Pi \). We refer interested readers to the WebRobot paper [Dong et al. 2022] for the design rationale. Here, we briefly explain a few representative rules. The \( \text{Seq} \) rule is standard: it evaluates \( E \) and \( P \) in sequence, and concatenates the resulting action traces. The \( \text{Click} \) rule is more interesting. It first evaluates \( se \) (which may use a variable) under \( \pi_1 \) and \( \Gamma \), yielding a selector \( \chi \) which is used to form the output action. Then, it removes the first DOM from \( \Pi \) to obtain the resulting DOM trace \( \Pi' \). In other words, the program under evaluation and the DOM trace are always “in sync”: the first action to be executed always corresponds to the first DOM. The \( \text{ForData} \) rules are perhaps the most interesting. \( \text{ForData-1} \) first evaluates \( de \) to a list \( \text{lst} \), and then invokes \( \text{ForData-2} \) which is a helper rule that executes all iterations of the loop until termination. The key observation is: similar to the fold function from Example 3.1, \( \text{ForData} \) also performs dependent iterations; that is, an iteration has to be executed under a context computed by its previous iteration. In particular, the input DOM trace carries the dependency. This data-dependent feature actually is not specific to \( \text{ForData} \): all loops in our language are data-dependent, and in fact, \( \text{Seq} \) is too. Prior work cannot reduce the space of our web automation programs. Our lifted interpretation idea is able to reduce this space, and we will present how it works next.

4.2 Top-Level Synthesis Algorithm

Algorithm 1 shows the top-level algorithm that synthesizes web automation programs from demonstrations. It shares the same interface as WebRobot’s algorithm and thus can be directly integrated with WebRobot’s front-end UI. At a high-level, it takes as input an action trace \( A \), a DOM trace \( \Pi \), and input data \( I \). It returns a program \( P \) that generalizes \( A \) — i.e., given \( \Pi \) and \( \Gamma = \{x \mapsto I\} \), evaluating \( P \) using trace semantics produces \( A' \) such that \( A \) is a strict prefix of \( A' \). This generalization is possible, because we require \( \Pi \) to have one more DOM than the number of actions in \( A \). To synthesize \( P \), we first find all programs that reproduce \( A \) (i.e., \( A' \) has \( A \) as a prefix) — notably, this process compresses a large number of programs in an FTA \( \mathcal{A} \). Then line 8 picks a smallest program \( P \), from \( \mathcal{A} \), that generalizes \( A \). If no such \( P \) exists in \( \mathcal{A} \), the algorithm returns null.

Now let us dive into the algorithm a bit more, though more details will be described in subsequent sections. Line 1 initializes \( \mathcal{A} \) based on the input traces, such that \( \mathcal{A} \) stores all loop-free programs that are guaranteed to reproduce \( A \). The reason that we base our synthesis on the input DOM trace \( \Pi \) is because selector expressions are not given a priori; they are known only when \( \Pi \) becomes available. The input action trace \( A \) is used to further guide synthesis. The following example briefly illustrates what an initial FTA looks like; we defer the more detailed explanation to Section 4.4.

Proc. ACM Program. Lang., Vol. 8, No. POPL, Article 52. Publication date: January 2024.
Example 4.1. Consider the following action trace \( A = [a_1, \ldots, a_6] \) for the task from Example 1.1.

\begin{verbatim}
1 EnterData(x[0], /html/.../div[1]/input)   # enter pwned@gmail.com
2 Click(/html/body/.../div[1]/span/button)  # click search button
3 ScrapeText(/html/body/div[1]/.../div/h2)  # scrape "Oh no – pwned!"
4 EnterData(x[1], /html/.../div[1]/input)   # enter not_pwned@gmail.com
5 Click(/html/body/.../div[1]/span/button)  # click search button
6 ScrapeText(/html/body/div[0]/.../div/h2)  # scrape "Good news - no pwnage found"
\end{verbatim}

In addition, the demonstration also includes a DOM trace \( \Pi = [\pi_1, \ldots, \pi_7] \), where \( a_i \) is performed on DOM \( \pi_i \). Suppose \( I = ["pwned@ma1l.com", "not_pwned@ma11.com"] \) is the list of emails.

Figure 9 shows the abstract syntax tree for \( A \), and Figure 10 gives the corresponding initial FTA \( \mathcal{A} \). Here, we use \( \chi_I \) as a shorthand to denote the selector in \( a_i \). Note that \( A \) and \( \mathcal{A} \) have similar semantics, except that each action in \( A \) has multiple selectors as "leaf transitions" (see each dashed circle), whereas each action in \( \mathcal{A} \) has only one selector. Section 4.4 will present more details around how \( \mathcal{A} \) is constructed.

Given the initial \( \mathcal{A} \), we then enter a loop (lines 2-7) which iteratively adds to \( \mathcal{A} \) loopy programs, until no new programs can be found (i.e., \( \mathcal{A} \) saturates) or a predefined timeout is reached. In each iteration, we non-deterministically pick 2l consecutive \( \text{Seq} \) transitions \( \delta_1, \ldots, \delta_{2l} \) (line 4), and then perform three key steps: (1) \text{SPECULATEFTA} guesses an FTA \( \mathcal{A}_s \) based on these 2l transitions, (2) \text{EVALUATEFTA} performs lifted interpretation over \( \mathcal{A}_s \), yielding another FTA \( \mathcal{A}_e \), and (3) \text{MERGEFTAs} merges \( \mathcal{A}_e \) into \( \mathcal{A} \). The resulting \( \mathcal{A} \) at line 7, compared to \( \mathcal{A} \) before the merge, includes new loopy programs that are synthesized during this iteration, with the same guarantee that all programs in \( \mathcal{A} \) still reproduce \( A \). The generalization step takes place in (1), where we reroll a slice of statements in a program from \( \mathcal{A} \) to a loop that is then stored in \( \mathcal{A}_s \). This loop rerolling step, however, is speculative, meaning some rerolled loops may not be correct. Therefore, in step (2), we use \text{EVALUATEFTA} to check all loops and retain only those that can indeed reproduce \( A \) — this \text{EVALUATEFTA} algorithm (i.e., lifted interpretation) is the key contribution of this paper.

In what follows, we explain how each step works in more detail. In particular, we will begin with \text{EVALUATEFTA} in Section 4.3, given \( \mathcal{A}_e \) and its corresponding input contexts. Then in subsequent sections, we explain how FTA initialization, speculation, merging, and ranking work, respectively.
4.3 Lifted Interpretation

As mentioned in Section 3, our lifted interpretation uses the following key judgment.

\[ C \vdash q; \Delta \leadsto (q', \Delta') \]

In our domain, a context \( C \) consists of a DOM trace \( \Pi \) and a binding environment \( \Gamma \). Figure 12 shows the \textsc{EvaluateFTA} rules that implement lifted interpretation. We suggest readers looking at Figure 11 at the same time. Rule (1) reduces multi-context evaluation to single-context evaluation. We note that the evaluation under \( C_i \) relies on the previous evaluation result for \( C_{i-1} \). To evaluate a state \( q \) under a single context \( C \), Rule (2) further reduces it to evaluating each of \( q \)'s incoming transitions. The remaining rules evaluate various types of transitions; they share the same principle as illustrated in Figure 11. In particular, given transition \( \delta = f(q_1, \cdots, q_n) \rightarrow q \) and context \( C \):

1. First, we recursively evaluate each argument state \( q_i \). The input context under which to evaluate each \( q_i \) and the order to evaluate them depends on the semantics of \( f \).
2. Given each resulting state \( q'_i \) for \( q_i \), we obtain the output \( O \) for \( q \) for \( C \). Note that \( O \) is computed compositionally, from the corresponding outputs associated with \( q'_i \) and per \( f \)'s semantics.
3. Finally, we construct state \( q' \) that \( q \) evaluates to. In particular, \( q' \) includes \( C \leadsto O \) as a behavior in its footprint \( \Omega \), as well as all behaviors from \( q \)'s footprint \( \Omega \).

Let us examine the rules in detail. Rule (3) is a base case for a nullary transition \( \delta \) with no argument states: it directly evaluates the selector expression \( s \) and yields a state \( q' \) with behavior \( \Pi, \Gamma \leadsto \chi \). Rule (4) is more interesting. We first evaluate \( q_1 \) to \( q'_1 \) under context \( \Pi, \Gamma \): note that here we use the context for \( q \) to evaluate \( q'_1 \), due to \textsc{Click}'s semantics. Then, given \( q'_1 \), we obtain its output \( \chi \) which is later used to construct the output trace \( A' \) for \( q \). Finally, we construct \( q' \) with footprint \( \Omega' \), which includes the new behavior we just created based on \( q'_1 \) and \textsc{Click}'s semantics, as well as everything from \( \Omega \). Rule (5) considers a \textsc{Seq} transition with two argument states. We first evaluate \( q_1 \) to \( q'_1 \). However, before evaluating \( q_2 \), we need to obtain \( \Pi'_1 \) from \( q'_1 \) to form the context to evaluate \( q_2 \). The output action traces \( A'_1, A'_2 \) for \( q'_1, q'_2 \) form the output action trace for \( q \). Finally, we construct \( q' \) with \( \Omega' \), same as previous rules. Rule (6) is another base case for \textsc{skip}. Rule (7) also concerns a base case: if the input DOM trace is empty, it yields the sub-FTA rooted at \( q \). Note that in general, \( q \) would also include \( \Box, \Gamma \leadsto \Box, \Box \) in its footprint. Rule (7) does not show this explicitly, because we assume all states implicitly have \( \Box, \Box \leadsto \Box, \Box \).

Now let us look at the most exciting rules for loops. Consider \textsc{ForData}: its first argument is a data expression that yields a list \( \text{lst} \), and the second argument (i.e., loop body) is evaluated with the loop variable \( z \) being bound to each element from \( \text{lst} \). To evaluate \( q \) with an incoming \textsc{ForData} transition, Rule (8) first evaluates \( q_1 \) to \( q'_1 \) and obtains \( \text{lst} \) from \( q'_1 \). Then, we evaluate “loop body” state \( q_2 \) under a context with \( \text{lst} \): Rule (9) presents how this evaluation works. Intuitively, Rule (9) has a series of \( n \) evaluations: \( q \) to \( q'_1, q'_1 \) to \( q'_2, \cdots, q'_{n-1} \) to \( q'_n \). Note that the output DOM trace for \( q'_i \) is part of the context for evaluating \( q'_{i+1} \), due to the data dependency across iterations. The output
\begin{align}
q_0^n &= q \quad \Delta_0^n = \Delta \\
C_1 \vdash q_{i-1}; \Delta_{i-1} \rightsquigarrow (q_i', \Delta_i') & \quad \text{(1)} \\
\frac{C_1, \ldots, C_n \vdash q; \Delta \rightsquigarrow (q_n', \Delta_n')}{P = [\pi_1, \ldots, \pi_n]} & \quad \text{(2)}
\end{align}

\Pi = [\pi_1, \ldots, \pi_m] \quad \Gamma, \pi_1 \vdash s : \chi \quad \Omega = \text{Footprint}(q) \quad \Omega' = \Omega \cup \{ \Pi, \Gamma \mapsto \chi \} \quad q' = \text{MkState}(se, \Omega')

\begin{align}
\Pi, \Gamma \vdash s \mapsto q; \Delta \rightsquigarrow (q', \Delta') & \quad \text{(3)} \\
\Pi, \Gamma \vdash q_1; \Delta \rightsquigarrow (q_1', \Delta_1') \quad \Pi, \Gamma \mapsto \chi \mapsto \text{Footprint}(q_1') & \quad \Omega = \text{Footprint}(q) \quad \Pi = [\pi_1, \ldots, \pi_m] \\
A' = [\text{Click}(\chi)] & \quad \Omega' = \Omega \cup \{ \Pi, \Gamma \mapsto A', \{\pi_2, \ldots, \pi_m\}\} \quad q' = \text{MkState}(E, \Omega')
\end{align}

\begin{align}
\Pi, \Gamma \vdash \text{Click}(q_1) \mapsto q; \Delta \rightsquigarrow (q', \Delta_1' \cup \Delta_2' \cup \{\text{Click}(q_1') \mapsto q') & \quad \text{(4)} \\
\Pi, \Gamma \vdash \text{Seq}(q_1, q_2) \mapsto q; \Delta \rightsquigarrow (q', \Delta_1' \cup \Delta_2' \cup \{\text{Seq}(q_1', q_2') \mapsto q') & \quad \text{(5)} \\
\Omega = \text{Footprint}(q) \quad \Omega' = \Omega \cup \{ \Pi, \Gamma \mapsto A_i', \Pi_1', \Pi_2' \in \text{Footprint}(q_1') \} & \quad \text{(6)}
\end{align}

\begin{align}
\Pi, \Gamma \vdash \text{skip} \mapsto q; \Delta \rightsquigarrow (q', \{\text{skip} \mapsto q') & \quad \text{(7)} \\
[\], \Gamma \vdash q; \Delta \rightsquigarrow (q, \Delta') & \quad \text{SubFTA}(q, \Delta) = ([q], \Delta') \\
\Pi, \Gamma \vdash \text{lst} \in \text{Footprint}(q_1') & \quad \text{(8)} \\
\Pi, \Gamma, \text{lst} \vdash q_3; \Delta \rightsquigarrow (q_3', \Delta_3') \quad \Pi, \Gamma \mapsto \text{lst} \mapsto \text{Footprint}(q_3') & \quad \Omega = \text{Footprint}(q) \quad \Omega' = \Omega \cup \{ \Pi, \Gamma \mapsto A_i' + \cdots + A_i_\text{lst}' + \Pi_1', \Pi_2' \} \quad q' = \text{MkState}(E, \Omega')
\end{align}

\begin{align}
q_0'^n &= q \quad \Delta_0'^n = \Delta \\
\Pi_1', \Gamma \vdash [z \mapsto v_1]; q_{i-1}; \Delta_{i-1} \rightsquigarrow (q_i', \Delta_i') & \quad \Pi_1', \Gamma \vdash [z \mapsto v_1] \mapsto A_i', \Pi_i' \in \text{Footprint}(q_1') \\
\Pi, \Gamma, [v_1, \ldots, v_n] \vdash q; \Delta \rightsquigarrow (q_n', \Delta_n') & \quad \text{(9)}
\end{align}

Figure 12. A subset of rules for \text{EvaluateFTA}; the complete set can be found in [Li et al. 2023b].

state is $q'_n$, which encapsulates information from all $n$ iterations. Popping up back to Rule (8): given $q'_2$, we obtain the output traces $A'_i$ for all iterations, which are then concatenated to form the output trace in $q'$. We skip the discussion on other loop types as they are very similar to \text{ForData}.

\section*{Example 4.2. Consider the task from Example 4.1. Suppose $\mathcal{A}_s = (\{p\}, \Delta)$ shown in Figure 13 is the FTA speculated from the initial FTA in Figure 10. Section 4.5 will later explain how \text{SpeculateFTA} generates this $\mathcal{A}_s$, but in brief, it contains $\text{ForData}$ loops inferred from the initial FTA. Each state in $\mathcal{A}_s$ is annotated with a grammar symbol and an empty footprint (see a few examples in Figure 13).

Given a context consisting of a DOM trace $[\pi_1, \ldots, \pi_6]$ and a binding environment $\Gamma = \{x \mapsto I\}$ (both of which are from Example 4.1), \text{EvaluateFTA} applies lifted interpretation to $\mathcal{A}_s$: the process is very similar to that in Example 3.1, except that now we use trace semantics for a different syntax. Figure 14 illustrates a part of the FTA $\mathcal{A}_e = (Q'_f, \Delta')$ returned by \text{EvaluateFTA}. Here, we show two states $r_1 \in Q'_f$ and $r_2 \in Q'_f$, with different footprints, that $p$ evaluates to:

$$[\pi_1, \ldots, \pi_6, \Gamma, \vdash p; \Delta \rightsquigarrow (r_1, \Delta_1') \quad \text{where} \quad \Gamma = \{P, \vdash a_1, 2, a_3, a_4, a_5\}$$

$$[\pi_1, \ldots, \pi_6, \Gamma, \vdash p; \Delta \rightsquigarrow (r_2, \Delta_2') \quad \text{where} \quad \Gamma = \{P, \vdash a_1, 2, a_3, a_4, a_5\}$$

Here, $[a_1, \ldots, a_6]$ is the desired action trace (see Example 4.1) where $a_3$ scrapes “Oh no – pwned” and $a_6$ scrapes “Good news – no pwnage found”. The action $a_3'$, however, scrapes “Good news – no pwnage found”, which is undesired. The reason we can have $a_3'$ is because programs in $L(\{r_2\}, \Delta')$
use an undesired selector expression (such as the full selector expression described in Example 1.1) that does not generalize. More specifically, evaluating $p'_5$ from $A_4$ (with a full selector expression in ScrapeText) can yield state $r_6$ in $A_4$:

$$r_6 : (E, \{[\pi_3, \ldots, \pi_6], \{x \mapsto I, z \mapsto \text{"pwned@gmail.com"}\} \mapsto \text{"a_3\"}, [\pi_4, \pi_5, \pi_6] \mapsto \{a_6\}, [\\})$$

That is, for both emails, the full selector always scrapes “Good news – no pwnage found”. This in turn means that $p_0$ from $A_3$ can evaluate to $r_4$ in $A_4$:

$$r_4 : (P, \{[\pi_1, \ldots, \pi_6], \{x \mapsto I, z \mapsto \text{"pwned@gmail.com"}\} \mapsto \{a_1, a_2, a_3\}, [\pi_4, \pi_5, \pi_6] \mapsto \{a_4, a_5, a_6\}, [\\})$$

Finally, this leads to the aforementioned state $r_2$.

### 4.4 FTA Initialization

Now let us circle back and describe the FTA initialization procedure, which we note is specific to PBD and web automation. Figure 15 shows the key initialization rule that constructs, from action trace $[a_1, \ldots, a_m]$ and DOM trace $[\pi_1, \ldots, \pi_m]$, $m$ consecutive $\text{Seq}$ transitions in the initial FTA $\mathcal{A}$. In particular, all $q_i$ ($i \in [0, m]$) states have grammar symbol $P$, and all $q_i'$ ($i \in [1, m]$) states have $E$. Each $\text{Seq}$ transition $\delta_i$ ($i \in [1, m]$) connects $q_i'$ and $q_i$ to $q_{i-1}$, forming $m$ consecutive $\text{Seq}$ transitions. The footprint in $q_i$ means that all programs in $\text{SubFTA}(q_i, \mathcal{A})$ yield $[a_{i+1}, \ldots, a_m], [\\]$ under context $[\pi_{i+1}, \ldots, \pi_m], \Gamma$. Similarly, for each $q'_i$, it maps the context $[\pi_i, \ldots, \pi_m], \Gamma$ to $[a_i], [\pi_{i+1}, \ldots, \pi_m]$.

In addition to $\text{Seq}$, we also have transitions for statements (like $\text{Click}$ and ScrapeText) and selectors. The following rule shows how to construct transitions for $\text{Click}$ and its candidate selectors, from action trace $[a_1, \ldots, a_m]$, DOM trace $[\pi_1, \ldots, \pi_m]$, and input data $I$.

\[
\begin{align*}
  i &\in [1, m] \quad a_i = \text{Click}(\chi) \quad \pi_i \vdash \text{se} : \chi \quad \Gamma = \{x \mapsto I\} \\
  q'_i &= \text{MkState}(E, \{[\pi_i, \ldots, \pi_m], \Gamma \mapsto [a_i], [\pi_{i+1}, \ldots, \pi_m]\}) \\
  q''_i &= \text{MkState}(\text{se}, \{\pi_i, \Gamma \mapsto \chi\}) \\
  \delta'_i &= \text{Click}(q'^{\pi_i}_i) \rightarrow q'_i \quad \delta''_i = \text{se} \rightarrow q''_i \\
  q'_i, q''_i &\in Q \quad \delta'_i, \delta''_i \in \Delta
\end{align*}
\]

Intuitively, this rule states: given a Click action $a_i$ (with a full selector expression $\chi$) that is performed on DOM $\pi_i$, for any candidate selector $\text{se}$ that refers to the same DOM element on $\pi_i$ as $\chi$, we create a transition for $\text{se}$. This is exactly why Figure 10 has multiple selectors in the dashed circle around each $\chi_i$. The construction rules for other actions are very similar; we skip the details here.
\[ \Gamma = \{ x \mapsto I \} \]
\[ q_0 = \text{MkState}\left( P, [\{ \pi_1, \ldots, \pi_m \}], \Gamma \mapsto [a_1, \ldots, a_m], \emptyset \right) \]
\[ q_1 = \text{MkState}\left( P, [\{ \pi_2, \ldots, \pi_m \}], \Gamma \mapsto [a_2, \ldots, a_m], \emptyset \right) \]
\[ \quad \vdots \]
\[ q_m = \text{MkState}\left( P, [\{ \}, \Gamma \mapsto [a_m], \emptyset \right) \]

\[ \delta_1 = \text{Seq}(q'_1, q_1) \mapsto q_0 \quad \delta_m = \text{Seq}(q'_m, q_m) \mapsto q_{m-1} \]

Figure 15. Key FTA initialization rule, given action trace \([a_1, \ldots, a_m]\) and DOM trace \([\pi_1, \ldots, \pi_m]\).

**Procedure** SpeculateForData([\(\delta_1, \ldots, \delta_{2l}\), \(\Delta\)])

**Input:** Each \(\delta_i\) is of the form \(\delta_i = \text{Seq}(q'_i, q_i) \mapsto q_{i-1}\), \(i \in [1, 2l]\). \(\Delta\) is a set of transitions.

**Output:** FTA with final state \(p\) and transitions \(\Delta'\), which represents a set of speculated \(\text{ForeData}\) loops.

1. \(U_i := \text{AntiUnifyForData}(q'_i, q'_{i+1}, \Delta)\); \(\cdots\); \(U_l := \text{AntiUnifyForData}(q'_l, q'_{2l}, \Delta)\);
2. \(U := U_1 \cup \cdots \cup U_l\); \(\{p'_1, \Delta'_1\} := \text{Parametrize}(q'_1, \Delta, U)\); \(\cdots\); \(\{p'_l, \Delta'_l\} := \text{Parametrize}(q'_l, \Delta, U)\);
3. Create fresh states \(p_0(P, \emptyset), \ldots, p_l(P, \emptyset)\), and \(p'(de, \emptyset), p(E, \emptyset)\);
4. \(\Delta' := \{\text{ForData}(p', p_0) \mapsto p\} \cup \{\text{Seq}(p'_i, p_i) \mapsto p_{i-1}\} | i \in [1, l]\) \(\cup \{u \mapsto p' | u \in U\} \cup \Delta'_1 \cup \cdots \cup \Delta'_l\) \(\triangleright\) Transitions corresponding to anti-unifiers \(\triangleright\) Transitions constructed from parametrization
5. **Return** \(\{p, \Delta'\}\);

Algorithm 2. Algorithm for SpeculateForData that speculates an FTA of \(\text{ForeData}\) loops.

Figure 16. Illustration of the \(2l\) consecutive transitions \(\delta_1, \ldots, \delta_{2l}\) (line 4, Algorithm 1).

Figure 17. Illustration of Algorithm 2, given the \(2l\) transitions \(\delta_1, \ldots, \delta_{2l}\) from Figure 16.

### 4.5 Speculating FTAs

Section 4.3 assumed a given speculated FTA \(\mathcal{A}_s\). In this section, let us unpack the SpeculateFTA algorithm that infers \(\mathcal{A}_s\). Here, \(\mathcal{A}_s\) is an FTA that contains candidate loops (potentially nested).

Figure 17 illustrates how to speculate candidate \(\text{ForeData}\) loops from the \(2l\) consecutive transitions \(\delta_1, \ldots, \delta_{2l}\) (illustrated in Figure 16). Algorithm 2 describes the algorithm more formally. We suggest readers simultaneously looking at Figures 16, Figure 17 and Algorithm 2.

The first step is anti-unification (line 1): for all \(i \in [1, l]\), we synchronously traverse \(\text{SubFTA}(q'_i, \Delta)\) and \(\text{SubFTA}(q'_i, \Delta)\), and compute a set \(U_i\) of anti-unifiers. For \(\text{ForeData}\), an anti-unifier \(u\) is simply a data expression that \(\text{ForeData}\) may iterate over. The second step is parametrization (lines 2): for all anti-unifiers from \(U\), we traverse \(\text{SubFTA}(q'_i, \Delta)\) \((i \in [1, l])\) and build a new FTA with final state \(p'_i\) and transitions \(\Delta'_i\). The third and last step is to construct the final speculated FTA \(\mathcal{A}_s\) (lines 3-4).
Each state in $\mathcal{A}_4$ is annotated with an empty footprint, because we are yet to evaluate any programs. In other words, $\mathcal{A}_4$ is purely a syntactic compression without using any OE at all.

We refer interested readers to our extended version [Li et al. 2023b] for a complete description of the speculation algorithm that handles other types of loops. In what follows, we explain our anti-unification and parametrization algorithms.

**Anti-unification.** Given two states $q_1,q_2$ from FTA $\mathcal{A}$ with transitions $\Delta$, anti-unification traverses expressions in $\text{SubFTA}(q_1, \mathcal{A})$ and $\text{SubFTA}(q_2, \mathcal{A})$ in a synchronized fashion, and returns a set $U$ of anti-unifiers that may be iterated over by the speculated loops. In particular:

$$\text{AntiUnify}(q_1,q_2,\Delta) = U = \{ u \mid \Delta + q_1,q_2 \Rightarrow u \}$$

That is, $u \in U$ if there is an anti-unifier $u$ for $q_1,q_2$ given $\Delta$. Figure 18 presents the detailed rules. Rule (1) says that, to anti-unify $q_1,q_2$ with incoming transition $f$, we anti-unify the argument states $q'_1,q'_2$. Rule (2) does the same but for loops: we anti-unify expressions (i.e., the first argument) being looped over. Rules (3) and (4) concern the anti-unification of data expressions — the idea is to look for an increment pattern beginning with 1. Rules (5) and (6) anti-unify selector expressions, where it uses a more flexible pattern that allows starting from $k$ (not necessarily $k = 1$).

**Example 4.3.** Consider states $q'_1$ and $q'_4$ in $\mathcal{A}$ from Figure 10. The anti-unifier for $q'_1, q'_4$ is $x$.

**Parametrization.** Given all anti-unifiers $U$ and a state $q$ from $\mathcal{A}$ with transitions $\Delta$, PARAMETRIZE constructs from $\text{SubFTA}(q, \mathcal{A})$ a fresh new $\mathcal{A}'$ (with transitions $\Delta'$ and final state $q'$) in two steps.

First, we make a fresh new copy of each state from $\text{SubFTA}(q, \mathcal{A})$, keeping the same grammar symbol and but resetting the footprint to empty. This results in a mapping $M$ that maps every state in $\text{SubFTA}(q, \mathcal{A})$ to a state in $\mathcal{A}'$. Therefore, for each transition $f(q_1, \ldots, q_n) \rightarrow q_0 \in \Delta$, we have $f(M(q_1), \ldots, M(q_n)) \rightarrow M(q_0) \in \Delta'$.

Then, we add new transitions labeled with parametrized selector/data expressions to $\mathcal{A}'$, given each anti-unifier $u \in U$, mapping $M$, and the state $q$ from $\mathcal{A}$ with transitions $\Delta$. In other words, the actual parametrization takes place in this step. Figure 19 presents our parametrization rules. Rule (1) parametrizes any transition in $\Delta$ that uses a selector expression of the form $u \oplus s e$; that is, this selector uses the anti-unifier $u$ as a prefix. This rule adds a new transition with $u$ being replaced by variable $y$ which points to state $M(q')$. Rule (2) parametrizes data expressions in a similar manner.

**Example 4.4.** Consider the initial FTA $\mathcal{A}$ in Figure 10. Consider the anti-unifier $x$ from Example 4.3. Let us explain how to parametrize state $q'_1$ from $\mathcal{A}$, given anti-unifier $x$, to generate state $p'_1$ in $\mathcal{A}_4$ from Figure 13. We first make a copy of $\text{SubFTA}(q'_1, \mathcal{A})$: for example, the copy of $q'_1$ is $p'_1$; that is,
Then, we invoke Rule (2) from Figure 19 with \( f \) being EnterData. This yields a new transition \( z \rightarrow M(q''_1) \) where \( p''_1 = M(q''_1) \), which indeed appears in \( \mathcal{A}_s \) in Figure 13.

**Constructing \( \mathcal{A}_s \).** Finally, lines 3-4 connect the created states and transitions, as shown in Figure 17.

### 4.6 Merging FTAs

Let us explain the last two procedures in Algorithm 1.

**Merging \( \mathcal{A}_e \) into \( \mathcal{A} \).** Intuitively, MERGEFTAs (line 7) incorporates loops from \( \mathcal{A}_e \) into SubFTA\((q_0, \mathcal{A})\). Figure 20 illustrates how this works. We first construct a state \( q' \) and a Seq transition \( \delta \) that connects a final state \( q \) from \( \mathcal{A}_e \) and \( q' \) to \( q_0 \). In particular, \( q' \) has grammar symbol \( P \) and is associated with the following footprint:

\[
\begin{align*}
\Pi'_1, \Gamma & \mapsto A'_2, \Pi'' \quad \Pi, \Gamma \mapsto A', \Pi'' \in \text{Footprint}(q_0) \\
\Pi, \Gamma & \mapsto A'_1, \Pi'' \in \text{Footprint}(q) \\
A'_1 + A'_2 = A'
\end{align*}
\]

In other words, \( q' \) is constructed based on footprints from \( q_0 \) and \( q \), according to Seq’s semantics. Note the constraint \( A'_1 + A'_2 = A' \): we only keep loops in SubFTA\((q, \mathcal{A})\) that yield a prefix of \( A' \), because otherwise \( A'_2 \) is not a reachable context for \( q' \), at least not so in this case. For such final states \( q \) of \( \mathcal{A}_e \), we create the aforementioned \( q' \) and \( \delta \), and add \( \delta \) together with all transitions in \( \mathcal{A}_e \) to the set of transitions of \( \mathcal{A} \). Observe that, if \( q' \) already exists in \( \mathcal{A} \), then MERGEFTAs effectively connects \( q \) and an existing state \( q' \) in \( \mathcal{A} \) to \( q_0 \) which is also in \( \mathcal{A} \).

**Ranking.** The RANKING procedure (line 8) is fairly straightforward. It first runs EVALUATEFTA on \( \mathcal{A} \) using \( \Pi, \{ x \mapsto 1 \} \) as the context. Note that the last DOM \( \pi_{m+1} \) in \( \Pi \) does not have a demonstrated action in the input action trace \( A \), because our goal is to use the synthesized program \( P \) to automate the unseen actions. EVALUATEFTA gives \( \mathcal{A}' \) containing programs with different predicted actions \( a_{m+1} \). We pick a program \( P \) with the smallest size and return \( P \) as the final synthesized program.

### 4.7 Soundness and Completeness

**Theorem 4.5.** Given action trace \( A \), DOM trace \( \Pi \) and input data \( I \), our synthesis algorithm always terminates. Moreover, if there exists a program in our grammar (shown in Figure 8) that generalizes \( A \) (given \( \Pi \) and \( I \)) and satisfies the condition that (1) every loop has at least two iterations exhibited in \( A \) and (2) its final expression is a loop, then our synthesis algorithm (shown in Algorithm 1) would return a program that generalizes \( A \) (given \( \Pi \) and \( I \)) upon FTA saturation in Algorithm 1.

### 4.8 Interactive Synthesis with Incremental FTA Construction

**Interactive programming-by-demonstration.** Same as WEBROBOT [Dong et al. 2022], our synthesis technique can also be applied in an interactive setting: given an action trace \( A_m = [a_1, \ldots, a_m] \) and a DOM trace \( \Pi_m = [\pi_1, \ldots, \pi_{m+1}] \), we synthesize a program \( P_m \) that produces \( a'_{m+1} \) given \( \Pi_m \).

If this prediction \( a'_{m+1} \) is not intended, the user would manually demonstrate a correct \( a_{m+1} \). This leads to new traces \( A_{m+1} = [a_1, \ldots, a_{m+1}] \) and \( \Pi_{m+1} = [\pi_1, \ldots, \pi_{m+2}] \), both of which are fed to our algorithm again. This process repeats until the user obtains an intended program.

**Incremental FTA construction.** We incrementalize our Algorithm 1 based on the interactive setup. Given \( A_{m+1} \) and \( \Pi_{m+1} \), and given FTA \( \mathcal{A}_m \) constructed from \( A_m \) and \( \Pi_m \), we still build \( \mathcal{A}_{m+1} \) with the same guarantee as in Algorithm 1 but not from scratch. Our key insight is that we can re-use the validated loops from \( \mathcal{A}_m \), but we need to evaluate them against \( \Pi_{m+1} \) as some of them may not reproduce \( A_{m+1} \). Essentially, our incremental algorithm is still based on guess-and-check, but we
have a second type of speculation that directly takes sub-FTAs from $\mathcal{A}_m$ as speculated FTAs — this generates high-quality speculated FTAs more efficiently. $\mathcal{A}_{m+1}$ is still initialized in the same way but this time using $A_{m+1}$ and $\Pi_{m+1}$. We still perform SpeculateFTA and EvaluateFTA on $\mathcal{A}_{m+1}$ but in a much smaller scope this time. That is, $\delta_{gt}$ must involve $a_{m+1}$, because otherwise $\mathcal{A}_m$ had already considered it. The MergeFTAs and Ranking procedures remain the same.

5 EVALUATION

This section describes a series of experiments designed to answer the following questions:

- **RQ1**: Can Arborist efficiently synthesize programs for challenging web automation tasks? How does it compare against state-of-the-art techniques?
- **RQ2**: How necessary is it to use an expressive language in order to have a generalizable program? In particular, is it important to consider a large space of candidate selectors?
- **RQ3**: How does Arborist scale with respect to the number of candidate selectors considered?
- **RQ4**: How useful are various ideas proposed in this work?

**Web automation tasks.** To answer these questions, we construct a collection of web automation tasks from two different sources. First, we include all 76 tasks that were used to evaluate WebRobot: details about these tasks can be found in [Dong et al. 2022]. Second, we curate 55 new tasks, each of which has an English description of the task logic over one or more websites. All of our new tasks are curated based on real-life problems (e.g., those from the iMacros forum) and involve modern, popular websites (such as Amazon, UPS, Craigslist, IMDb) with complex webpages, whereas many of WebRobot’s benchmarks involve legacy websites. These new websites have deeply nested DOM structures which would require searching with a significantly larger space of candidate selectors. In particular, all 131 tasks involve data extraction, 45 of them involve data entry, 80 require navigation across webpages, and 48 involve pagination. Some of these tasks involve multiple types: for instance, 31 of them involve data entry, data extraction, and webpage navigation.

**Ground-truth Selenium programs.** For each task, we obtain a ground-truth automation program $P_{gt}$ (using the Selenium WebDriver framework). In particular, we reuse the 76 ground-truth Selenium programs from WebRobot, and manually write 55 programs for our new tasks. On average, these Selenium programs consist of 43 lines of code, with a max of 147 lines. In general, it takes about half an hour to a few hours for us to write up an automation program, depending on the complexity of webpages and the task logic.

**Benchmarks.** In our evaluation, a benchmark is defined as a tuple $(A, \Pi, I)$, where $A$ is an action trace, $\Pi$ is a DOM trace, and $I$ is input data. We obtain one benchmark for each task by running the corresponding $P_{gt}$ in the browser (given input data $I$ if $P_{gt}$ involves data entry) — during execution, we record the trace $A = [a_1, \ldots, a_m]$ of actions that $P_{gt}$ executes as well as $A$’s corresponding DOM trace $\Pi = [\pi_1, \ldots, \pi_m]$. Here, $a_i$ is an action performed on $\pi_i$. Note that selectors in actions are recorded as full XPath expressions.

**Candidate selectors.** For any full XPath selector $\chi$ in an action $a_i$, we also record a set $S$ of candidate selectors for $\chi$. In particular, a candidate selector $se$ is a concrete selector expression (i.e., without variable $y$) from our grammar (see Figure 8) that locates the same DOM element on $\pi_i$ as $\chi$. In other words, $se$ evaluates to $\chi$ given $\pi_i$, or more formally, $\pi_i \vdash se : \chi$. As mentioned in Example 1.1 and Section 4.2, it is important to consider candidate selectors, since full XPath expressions typically do not generalize. Candidate selectors also affect the overall search space, as explained below.

---

3We terminate $P_{gt}$ after every loop from $P_{gt}$ has been executed for three full iterations or when $|A|$ reaches 500, whichever yields a longer action trace. This essentially serves as a “timeout” to avoid running $P_{gt}$ for an unnecessarily long time.
Program space. In this work, the search space of programs for a benchmark with action trace $A$ is defined jointly by the grammar from Figure 8 and the candidate selectors for all actions in $A$. This is because Figure 8 does not specify a priori the space of selectors $se$ and predicates $\psi$. Instead, they are defined once the traces $A$ and $\Pi$ are given: e.g., the space for $se$ includes all candidate selectors. It is very hard (if not impossible) to define this space a priori without the DOMs. Obviously, the overall search space of programs grows when we consider more candidate selectors.

5.1 RQ1: Can ARBORIST Automate Challenging Web Automation Tasks?

In this section, we evaluate ARBORIST against three metrics: (i) how many tasks it can successfully synthesize intended programs for, (ii) how much synthesis time it takes, and (iii) how many user-demonstrated actions it requires in order to synthesize intended programs. In other words, we evaluate ARBORIST’s effectiveness, efficiency, and generalization power. We also compare ARBORIST against the state-of-the-art, especially on our new tasks that are more challenging.

Setup. We use the same setup from the WEBROBOT paper [Dong et al. 2022]. In particular, given a benchmark with $A$ and $\Pi$ that have $m$ actions and DOMs respectively, we create $m - 1$ tests. The $k$th test consists of an action trace $A_k = [a_1, \ldots, a_k]$ and a DOM trace $\Pi_k = [\pi_1, \ldots, \pi_k, \pi_{k+1}]$. For each action, we consider all candidate selectors in our grammar with at most 3 predicates. For example, $a[1]/b[2]$ uses 2 predicates and therefore is included, but $c/d/a[1]/b[2]$ is not considered even if it can also locate the same DOM element. We run ARBORIST in a way to simulate an interactive PBD process, same as how WEBROBOT was evaluated. That is, we feed all tests to ARBORIST in sequence: we run ARBORIST on the $k$-th test (with $A_k$ and $\Pi_k$), obtain a synthesized program $P_k$, and check if $P_k$ predicts $a_{k+1}$ (i.e., $P_k$ yields $A_{k+1}$ given $\Pi_k$). If not, $a_{k+1}$ is counted as a user-demonstrated action; otherwise, $a_{k+1}$ can be correctly predicted and thus is not counted. We always count the first action $a_1$ as user-demonstrated. Furthermore, for each test we use a 1-second timeout and record the time it takes to return $P_k$. We run ARBORIST incrementally (as described in Section 4.8) in this experiment: for each test, it resumes synthesis based on FTAs from previous tests/iterations. Finally, we inspect if the synthesized program $P_{m-1}$ (in the last iteration) is an intended program: if so, the corresponding benchmark is counted as solved; otherwise, unsolved.

Baselines. Among three baselines, we focus on the following two.

- WEBROBOT, which is the original tool from the WEBROBOT work [Dong et al. 2022]. We note that, while its underlying algorithm is complete in theory, the implementation is not. For example, it restricts the number of parametrized selectors (to five) when parametrizing actions during speculation. These heuristics seemed to help avoid excessively slowing down the search, without severely hindering the completeness for those tasks considered in the WEBROBOT work.
- WEBROBOT-extended, which is an adapted version of WEBROBOT that uses the extended language from Figure 8 (which allows ForSelectors with $i \geq 1$). In this experiment, we range $i$ from 1 to 3. This baseline still keeps all heuristics from WEBROBOT (i.e., its search is incomplete).

We use the same space of candidate selectors as ARBORIST for these two baselines. The third baseline is Helena [Chasins 2019], which is also a PBD-based web automation tool.

RQ1 take-away:
- ARBORIST can synthesize intended programs for 93.9% of our benchmarks.
- It typically takes ARBORIST subseconds to synthesize programs from demonstration.
- ARBORIST uses a median of 12 user-demonstrated actions to generalize.
- ARBORIST can solve more benchmarks using less time than state-of-the-art techniques.
Main results. Figure 21 summarizes the main results for both Arborist and baselines: Figure 21(a) shows the percentage of benchmarks where intended programs can be synthesized, and Figure 21(b) reports the distribution of their corresponding synthesis times. Note that for both figures, we show both the aggregated data across all 131 benchmarks, and separately for prior and new tasks.

Let us inspect Figure 21(a) first. Across all 131 benchmarks, Arborist can synthesize an intended program for 93.9% (i.e., 123) of them, whereas baselines solve at most 68.7% (i.e., 90). This is a large gap, because baselines solve significantly fewer new tasks (which are very challenging): in particular, Arborist can solve 53 out of 55 (i.e., 96.4%), which is 2.5x more than that for baselines (i.e., 21/38.2%). These new tasks involve complex webpages and task logics, which require using nested loops and searching for selectors with more predicates in a larger space. This makes them significantly more challenging than prior tasks; as a result, WebRobot’s underlying enumeration-based algorithm fundamentally cannot scale to this level of complexity. For prior tasks (total 76) in the WebRobot paper (which baselines were developed and engineered on), Arborist still outperforms baselines by one more benchmark. It turns out this benchmark involves two loops that require the start index to be 2 and 3, which cannot be expressed in WebRobot’s original language. WebRobot-extended, however, solves this benchmark, since it uses a richer language. Arborist uses the same (extended) language, and therefore is able to synthesize an intended program as well.

In addition to solving a strict superset of benchmarks than baselines, Arborist is also significantly faster. Figure 21(b) reports statistics of the synthesis times. In particular, for each solved benchmark, we record the maximum synthesis time across all tests, and report the distribution of these times. For each tool, Figure 21(b) presents the quartile statistics of synthesis times. Across all 123 benchmarks solved by Arborist, 120 of them do not even use up the 1-second timeout. In contrast, baselines solve fewer benchmarks and time out on 15 benchmarks. Recall that both baselines have internal heuristics which unsoundly prune search space for faster search. We tested variants of them with these heuristics removed: the best one can solve 55 benchmarks, with 46 timeouts. In other words, for many benchmarks, baselines cannot exhaust the entire search space, although they stumped upon a correct program before timeout. With a longer timeout of 10 seconds, baselines can only solve 9 more benchmarks. On the other hand, with 1-second timeout, Arborist times out on 3 benchmarks — they all require doubly nested loops and are among the most challenging ones. Arborist solves them all (albeit reaching timeout), while baselines solve two.

Arborist uses a median of 12 user-demonstrated actions, which is in line with that for WebRobot. This is reasonable, as we use the same search space for all tools in this experiment. While Arborist
searches more programs than baselines (which oftentimes time out and hence search a small subset), our simple ranking heuristics seem to be quite effective at selecting generalizable programs.

**Discussion.** *Arborist* failed to solve 8 benchmarks, including 6 from prior work (due to limitations of the web automation language, as also explained in the *WebRobot* paper) and 2 new ones (which can be solved using a 10-second timeout). Careful readers may wonder why *WebRobot*-extended solves fewer benchmarks than *WebRobot*, albeit using a richer language. This is due to the poor performance of its underlying enumeration-based algorithm: using a 1-second timeout, *WebRobot*-extended cannot even find intended programs for some benchmarks that *WebRobot* solves.

**Detailed results.** Recall that *Arborist* internally has two key modules — namely, *SpeculateFTA* and *EvaluateFTA* — which typically use most of the running time. Among them, on average, the former takes 20% of the time, and the latter uses 80%, across all benchmarks. The final FTA has an average of 1714 states. The final synthesized programs on average have 6 expressions, and the largest one has 20. Among these programs, 76 use at least one doubly-nested loop, and 12 involve at least a three-level loop.

**Arborist vs. Helena.** Among the 76 *WebRobot* benchmarks, Helena’s PBD technique was able to synthesize intended programs from demonstrations (provided by us manually) for 13 benchmarks. For the 55 new tasks, Helena solved 11. By contrast, *Arborist* solved 70 and 53 respectively.

### 5.2 RQ2: How Important Is It To Consider Many Candidate Selectors?

The search space considered in RQ1 uses the grammar from Figure 8 with candidate selectors of size up to three (measured by the number of predicates). This is a *quite expressive* language containing at least one generalizable program for over 93% of our tasks (as *Arborist* was able to solve them). However, one might ask: is this high expressiveness really necessary? This is an important question, because if not, advanced synthesis techniques like *Arborist* may not be necessary in practice.

**Setup.** In this section, we investigate the impact of candidate selectors on the *expressiveness* of the resulting search space: that is, given a set $S$ of candidate selectors, whether or not the corresponding program space has at least one generalizable program. We choose to focus on candidate selectors in this experiment, because prior work \[Dong et al. 2022\] has already shown the necessity of operators from the grammar in Figure 8.

More specifically, given a benchmark with action trace $A$, for each $a_i$ with full XPath selector $\chi$, we use the following ways to construct the set $S$ of candidate selectors for $a_i$.

- $S$ includes all candidate selectors for $\chi$ up to a certain size. This is perhaps the simplest heuristic that one can design; RQ1 uses 3 as the max size which was shown to be sufficient for most of our benchmarks. Therefore, in this experiment, we vary the max size from 1 to 3, and investigate how it impacts the expressiveness. For each size, we run *Arborist* using the corresponding $S$, and record the number of benchmarks solved. Here, “solved” means an intended program can be found by *Arborist*, which is a witness that corresponding search space is expressive enough.\(^4\)
- $S$ contains candidate selectors sampled (uniformly at random) from all candidate selectors of size up to 3. We vary $|S|$ from 1 to 1500. For each $|S|$, we run *Arborist* with the corresponding $S$ and record the number of benchmarks solved; we repeat this 8 times. This gives us a finer-grained, more continuous view of the impact of candidate selectors.

While one could certainly craft other heuristics for constructing $S$, we do not consider them, because (i) it is impossible to enumerate all heuristics in the first place, and (ii) human-crafted heuristics from prior work \[Chasins et al. 2018; Dong et al. 2022\] have shown to be not as effective especially

\(^4\)Thanks to having access to a highly efficient synthesizer like *Arborist*, we are able to conduct this experiment using a realistic time limit to check if a given search space really contains a target program.
Figure 22. RQ2 results. Given X candidate selectors (sampled uniformly at random from all candidate selectors of size up to 3), we have Y benchmarks whose search space is confirmed to contain an intended program.

for challenging tasks. Another note is that, since we use Arborist merely as a means to check if the search space has a generalizable program, the synthesis time is not relevant in this experiment. For each benchmark with a given \( S \), we run Arborist incrementally until it reaches the end of the action trace, same as in RQ1. However, in RQ2, we use 10 seconds as the timeout per iteration, rather than 1 second from RQ1 — this allows Arborist to exhaustively search the program space such that we can more confidently conclude on its expressiveness. If the final synthesized program is intended, we count it as solved (meaning the corresponding search space is expressive enough).

RQ2 take-away:

In general, an expressive search space should consider a large number of selectors (at least at the level of hundreds) for each webpage.

Results. If using only the full XPath expressions from the input action trace, Arborist manages to solve 46 benchmarks (out of 131 total), while terminating on the remaining 85 without synthesizing an intended program. That is, the search spaces of the 85 benchmarks are exhausted before timeout. This confirms that full XPath expressions typically do not generalize, and we need to consider more candidate selectors in order to solve more tasks. If we additionally include all candidate selectors of size 1, the number of solved benchmarks bumps up to 87, which is 66% of all benchmarks. Arborist does not reach timeout for any of the rest, indicating the need to consider more selectors. Further including all candidate selectors of size 2 allows Arborist to solve 122 benchmarks, which is quite close to our results in RQ1. Again, no timeout is observed on the remaining benchmarks. We note that the selector space at this point is already quite large: on average, we have 305 selectors of size up to two (in our grammar) per DOM across our benchmarks; the median and max are 321 and 385 respectively. In other words, using a simple size-based heuristic, we necessarily need to consider multiple hundreds of candidate selectors per DOM, in order to automate a decent number of tasks. Finally, when considering all candidate selectors of size up to 3 (same as in RQ1), 125 benchmarks are confirmed to admit intended programs; no timeouts observed. The remaining 6 benchmarks, upon manual inspection, cannot be solved by Arborist’s language (to our best knowledge). While encouraging, this high expressiveness comes at the cost of searching among an average of 7627 selectors per DOM, with the median and max being 8066 and 9649, across all our benchmarks.

Figure 22 presents a finer-grained view of how the expressiveness increases when we range the number of candidate selectors from 0 to 1500 using a small increment. The x-axis is the number of selectors randomly sampled from the universe of all candidate selectors of size up to three. The y-axis is the percentage of benchmarks (out of all 131 benchmarks) solved by Arborist; that is,
their corresponding search spaces are confirmed to contain at least one desired program. For each $x$, the figure gives the max, min, and mean of the percentage of solved benchmarks across 8 runs. In total, there are only 6 benchmarks for which Arborist times out without returning an intended program. As also mentioned earlier, this is due to the limitation of the web automation language, rather than Arborist not being able to exhaust the search space. Therefore, we believe Figure 22 precisely describes all benchmarks whose program spaces contain a desired program.

Let us inspect Figure 22 more closely. First, $y$ grows pretty quickly when $x$ goes up from 0 to 100. At $x = 100$, across 8 runs, an average of 73 benchmarks are solved, while the max and min are 78 and 70 respectively. There are 41 benchmarks not solved in any of the runs: notably, for all these 41 benchmarks, Arborist terminates before timeout without returning a generalizable program. This confirms that with (only) 100 (randomly sampled) selectors, the corresponding search spaces of these 41 benchmarks do not contain any generalizable programs. Furthermore, 62 of the 90 solved benchmarks (in at least one run, using 100 selectors) are from prior work. Our observation is that these 62 benchmarks are relatively “easier” compared to our newly curated tasks: their task logics are relatively simpler and their solutions use relatively smaller selectors.

On the other hand, the growth significantly slows down after $x = 100$. We observe a pretty long tail of benchmarks that require multiple hundreds, or even more than a thousand, of selectors to be solved. For instance, increasing $x$ from 100 to 200 only grows $y$ from 56% to 71% (in terms of average). In order to have another 15% bump, we need at least 400 selectors. Notably, benchmarks solved in the $[100, 1000]$ range mostly come from our new tasks: these problems have to be solved with complex selectors chosen from a larger space which are used in complex nested loops.

Finally, we seem to reach a plateau after $x = 1400$, after which point increasing the number of selectors does not seem to help grow the number of solved benchmarks anymore. The maximum number of solved benchmarks we observed in this experiment (based on random sampling selectors) is 123 (i.e., 93.9% out of 131 total). (In the previous experiment that includes all candidate selectors based on their size, we observed 125 benchmarks solved when using all selectors up to size 3.)

5.3 RQ3: How Does Arborist Scale against Number of Candidate Selectors?

Following up RQ2, one may wonder how Arborist would scale with more candidate selectors than those considered in RQ1 and RQ2. In this experiment, we stress test Arborist’s search algorithm given a very large number of candidate selectors.

Setup. We define search efficiency as the amount of time for a search algorithm to exhaust a given search space. For Arborist, this means: given a benchmark with action trace $A$ and given candidate selectors for each $a_i \in A$, how long it takes for the FTA $\mathcal{A}$ to saturate (i.e., no more new programs can be found) before timeout. We choose to use the exhaustion time, rather than the time to discover a generalizable program (i.e., synthesis time), because the synthesis time oftentimes depends on the particular search order used while the exhaustion time is more stable. Also note that the exhaustion time does not reflect the synthesis time; the latter tends to be much shorter. This experiment does not evaluate Arborist’s synthesis time; see RQ1 for its synthesis times.

Specifically, we sample a set $S$ of candidate selectors from a universe of all candidate selectors of size up to four. This universe is extremely large, with an average of 139,886 selectors per DOM, with the median and max being 135,856 and 291,385 respectively. For each benchmark, we vary $|S|$ from 1000 to 10,000. Given each $|S|$, we run Arborist incrementally on each benchmark, until reaching the end of its action trace. We use the same 10-second time per iteration as in RQ2. However, in this experiment, we count the number of benchmarks that are exhausted. That is, Arborist terminates before reaching the timeout for every iteration. Given $|S|$, we run Arborist on each benchmark 5 times, and record the max, min, and mean number of exhausted benchmarks across all 5 runs. In
Figure 23. RQ3 results. Selectors are sampled uniformly at random from all candidate selectors of size up to four. Arborist exhausts a benchmark’s program space if it terminates before the timeout (i.e., 10 seconds).

addition, for each exhausted benchmark, we record its max exhaustion time among all iterations. We also report the distribution of these times across all exhausted benchmarks and across all runs, as a way to quantify Arborist’s search efficiency.

**RQ3 take-away:**

Arborist can search very efficiently: in particular, it can exhaust program spaces that consider multiple thousands of selectors within at most a few seconds.

**Results.** Figure 23(a) shows for each $|S|$, how many benchmarks Arborist can successfully exhaust in 10 seconds: since we have multiple runs for each $|S|$, we report the max, median, and min across all runs. Figure 23(b) presents the distribution of exhaustion times — same as in RQ1, we also report quartile statistics here in RQ3 — across all exhausted benchmarks for each $|S|$. The key take-away message is clear: Arborist scales quite well as the number of selectors is increased. For example, with 1,000 candidate selectors, Arborist is able to exhaust the program space for 95% of all 131 benchmarks, with a median exhaustion time of about 0.1 seconds. If we further increase $|S|$ from 1,000 to 5,000, we observe a small drop from 95% to 79% in terms of the percentage of benchmarks that can be exhausted, while the median exhaustion time is under 1 second. Finally, looking at the extreme of 10,000 selectors: 68% benchmarks exhausted with a median of 1-second exhaustion time.

While exhaustion is in general fast, it takes even less time to discover a generalizable program, as also mentioned earlier. For example, among those 68% (i.e., 89) exhausted benchmarks, Arborist can discover an intended program for 61 within 1 second (and 43 under 0.5 seconds). In contrast, WebRobot’s enumeration-based algorithm cannot exhaust more than 10 benchmarks (using the same 10-second timeout), even if fed with only 100 selectors. Furthermore, using 10,000 selectors, WebRobot’s median solving time (i.e., returning an intended program) is 10 seconds. These data points again highlight that Arborist’s underlying search algorithm is highly efficient.

5.4 **RQ4: Ablation Studies**

**Impact of observational equivalence.** We consider a variant of Arborist with the OE capability disabled. In other words, this ablation has to enumerate loop bodies (which use loop variables), and does not allow sharing across FTA states that correspond to loop bodies. We evaluate this variant under the RQ1 setup: it solves (i.e., generates an intended program for) 56 benchmarks, among which the median synthesis time is 1 second. In contrast, Arborist solves 123 benchmarks with a
median running time of 0.02 seconds across those solved. This again highlights the importance of OE for speeding up the search.

**Impact of incremental FTA construction.** This ablation builds the FTA $\mathcal{A}$ *from scratch* given new input traces, without reusing previous FTAs. That is, the incremental FTA construction optimization in Section 4.8 is disabled. We run this ablation using the RQ1 setup. It solves (i.e., returns an intended program for) 105 benchmarks — out of these solved benchmarks, 40 reach the 1-second timeout and the median running time is 0.3 seconds. *Arborist* solves 18 more benchmarks with only 3 timeouts in total and using a significantly less median time of 0.02 seconds.

### 5.5 Case Study: Large Language Models

Given the recent advances in large language models (LLMs) and exploding interests in applying them for program synthesis, we conduct a case study where we use LLMs to generate web automation programs from demonstrations. This is mainly a sanity check, and we refer interested readers to the extended version [Li et al. 2023b] for more details. In summary, our key take-away is that LLMs (in particular, GPT-3.5 [OpenAI 2022]) fail to generate semantically correct programs, even for some of the simplest benchmarks. The model can produce unstable results, claiming a benchmark is unsolvable in one run while outputting programs in another trial. While these results are poor, it is well-known that LLMs are sensitive to the prompting strategy [Si et al. 2022], and there might be a better method to prompt the model which we have not tried. Nevertheless, we believe these results indicate that our benchmarks are quite hard for state-of-the-art LLMs and require further research in relevant areas in order to better solve these problems.

### 6 RELATED WORK

In this section, we briefly discuss some closely related work.

**Observational equivalence (OE).** OE is a very general concept, which states the *indistinguishability* between multiple entities based on their *observed implications*. Hennessy and Milner [1980] proposed OE to define the semantics of concurrent programs, where two terms are observationally equivalent whenever they are interchangeable in *all observable contexts*. The idea of OE has also been adopted by programming-by-example (PBE) [Albarghouthi et al. 2013; Peleg et al. 2020; Udupa et al. 2013] to reduce a large search space of programs thereby boosting the synthesis efficiency. Building upon the concept of OE, our work extends OE-based reduction to also programs with local variables.

**Synthesis of programs with local variables.** Programs with local variables are evaluated under *non-static* contexts. To our best knowledge, there are no principled approaches to effectively reduce the space of such programs. Prior work [Chen et al. 2021, 2020; Feser et al. 2015; Peleg et al. 2020; Smith and Albarghouthi 2016; Wang et al. 2017b] typically falls back to some form of brute-force enumeration or utilizes domain-specific reasoning to prune the search space of programs with local variables (such as lambda bodies). We propose a principled approach — i.e., lifted interpretation — to reduce the space of such programs, thereby speeding up the search of them.

*RESL* [Peleg et al. 2020] is especially related to our work: it uses an *extended context* (same as ours) when searching lambdas; however, it does not present a general approach that can reduce such programs. It clearly articulated the key problem of applying OE in general: computing reachable contexts and evaluating programs depend on each other. Our work presents a new algorithm that computes contexts and evaluates programs *simultaneously*, by constructing the equivalence relation of programs *while* evaluating programs which facilitates the computation of reachable contexts. In contrast, RESL utilizes rules (manually provided) to infer reachable contexts for lambda bodies, for a given higher-order sketch. For data-dependent functions (like reduce and fold), RESL falls back to
enumeration. Our paper addresses the “infeasible hypothesis” from RESL (see D.3 in its appendix). We believe our work also opens up new ways to further study such program synthesis problems.

**Lifted Interpretation.** Our lifted interpretation idea can be viewed as a bidirectional approach: it traverses the grammar *top-down* to generate reachable contexts, during which it builds up programs *bottom-up* given contexts. Different from prior work [Gulwani et al. 2011; Lee 2021; Phothilimthana et al. 2016] that enumerates programs bidirectionally, we intertwines the enumeration of contexts and programs. Rosette [Torlak and Bodik 2014] is related, in that they also lift the interpretation from concrete programs to symbolic programs (e.g., defined by a program sketch). A key distinction is that our work directly performs program synthesis and uses finite tree automata to succinctly encode the program space, instead of reducing the search problem to SMT solving.

**Finite tree automata (FTAs) for program synthesis.** During lifted interpretation, programs are clustered into equivalence classes, succinctly compressed in an FTA. Compared to prior work [Handa and Rinard 2020; Koppel et al. 2022; Miltner et al. 2022; Wang et al. 2018a, 2017a,b, 2018b; Yaghmazadeh et al. 2018], states in our FTAs encode context-output behaviors, rather than input-output behaviors of programs. The lifted interpretation idea is not tied to FTAs: our algorithm is developed using FTAs, but we believe other data structures (such as VSAs [Gulwani 2011] or e-graphs [Willsey et al. 2021]), or an enumeration-based approach [Peleg et al. 2020] can also be leveraged.

**Program synthesis for web automation.** Our instantiation presents a new program synthesis algorithm for web automation. This is an important domain with a long line of work [Barman et al. 2016; Chasins et al. 2015, 2018; Chen et al. 2023; Dong et al. 2022; Fischer et al. 2021; Leshed et al. 2008; Lin et al. 2009; Little et al. 2007; Pu et al. 2022, 2023] in both human-computer interaction and programming languages. The most related work is **WebRobot** [Dong et al. 2022]: building upon its trace semantics, we further develop a novel synthesis algorithm that can automate a significantly broader range of more challenging tasks much more efficiently.

**Programming-by-demonstration (PBD).** In particular, our algorithm is a form of programming-by-demonstration that synthesizes programs from a user-demonstrated trace of actions. Different from prior PBD work [Chasins et al. 2018; Dong et al. 2022; Lau et al. 2003; Lieberman 1993; Mo 1990] that is based on either brute-force enumeration or heuristic search of programs, **ARBORIST** uses observational equivalence to reduce the search space and leverages finite tree automata to succinctly represent all equivalence classes of programs.

7 CONCLUSION

We proposed **lifted interpretation**, which is a general approach to reduce the space of programs with local variables, thereby accelerating program synthesis. We illustrated how lifted interpretation works on a simple functional language, and presented a full-fledged instantiation of it to perform programming-by-demonstration for web automation. Evaluation results in the web automation domain show that lifted interpretation allows us to build a synthesizer that significantly outperforms state-of-the-art techniques.

ACKNOWLEDGMENTS

We thank our shepherds, Hila Peleg and Nadia Polikarpova, for their extremely valuable feedback. We thank the POPL anonymous reviewers for their constructive comments. We also thank Anders Miltner, Chenglong Wang, Kasra Ferdowski, Ningning Xie, Shankara Pailoor, Yuepeng Wang for their feedback on earlier drafts of this work. This work was supported by the National Science Foundation under Grant Numbers CCF-2123654 and CCF-2236233.
DATA AVAILABILITY STATEMENT

An extended version of this paper can be found online [Li et al. 2023b], and an artifact supporting the results reported in this paper is available on Zenodo [Li et al. 2023a].

REFERENCES


Efficient Bottom-Up Synthesis for Programs with Local Variables

Received 2023-07-11; accepted 2023-11-07