Program Synthesis From Partial Traces

MARGARIDA FERREIRA, Carnegie Mellon University, USA and INESC-ID/IST, Portugal
VICTOR NICOLET, JOEY DODDS, and DANIEL KROENING, Amazon, USA

We present the first technique to synthesize programs that compose side-effecting functions, pure functions, and control flow, from partial traces containing records of only the side-effecting functions. This technique can be applied to synthesize API composing scripts from logs of calls made to those APIs, or a script from traces of system calls made by a workload, for example. All of the provided traces are positive examples, meaning that they describe desired behavior. Our approach does not require negative examples. Instead, it generalizes over the examples and uses cost metrics to prevent over-generalization. Because the problem is too complex for traditional monolithic program synthesis techniques, we propose a new combination of optimizing rewrites and syntax-guided program synthesis. The resulting program is correct by construction, so its output will always be able to reproduce the input traces. We evaluate the quality of the programs synthesized when considering various optimization metrics and the synthesizer's efficiency on real-world benchmarks. The results show that our approach can generate useful real-world programs.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; **Logic and verification**.

Additional Key Words and Phrases: Program synthesis

ACM Reference Format:
Margarida Ferreira, Victor Nicolet, Joey Dodds, and Daniel Kroening. 2025. Program Synthesis From Partial Traces. *Proc. ACM Program. Lang.* 9, PLDI (June 2025), 46 pages. https://doi.org/10.1145/3729316

1 Introduction

Program synthesis is for lazy people. The promise of program synthesis is that a user writes a simple specification and gets a complex program. If specifications are complex to write or synthesis tools are hard to apply, users will prefer to write the program directly. The best specification is one they already have with no changes needed. Unfortunately (for synthesis tool writers), these specs often take the form of traces of external side effects. These traces may be sequences of messages exchanged between networked servers, logs of calls made to an Application Programming Interface (API), or traces of system calls made by a workload. Program synthesis from traces can be widely applied across different tasks; there are many instances of this approach in Programming-By-Demonstration (PBD) work [11, 17, 21, 33], but all of this work is either limited to pure functional examples, simple trace replay, or assumes no computation occurs between effects visible in traces.

Real-world traces are an incomplete view of a task: function calls with no side effects may not be recorded. A program `a=F(); b=h(a); return G(b)` where only the calls to `F` and `G` are logged may produce traces `F()=0 :: G("id-0")=true` and `F()=42 :: G("id-42")=false`, with no occurrence of `h`. These *hidden* function calls not present in the traces are a significant challenge for trace-guided synthesis. In our example, the synthesizer infers from the data that there is a non-visible function that transforms `0` into `"id-0"` and `42` into `"id-42"`. These traces come from external recordings of program behavior. For example, the administrator of a cloud deployment

Authors' Contact Information: Margarida Ferreira, margarida@cmu.edu, Carnegie Mellon University, Pittsburgh, USA and INESC-ID/IST, Lisbon, Portugal; Victor Nicolet, victornl@amazon.com; Joey Dodds, jldodds@amazon.com; Daniel Kroening, dkr@amazon.com, Amazon, Seattle, Washington, USA.

ACM 2475-1421/2025/6-ART
https://doi.org/10.1145/3729316

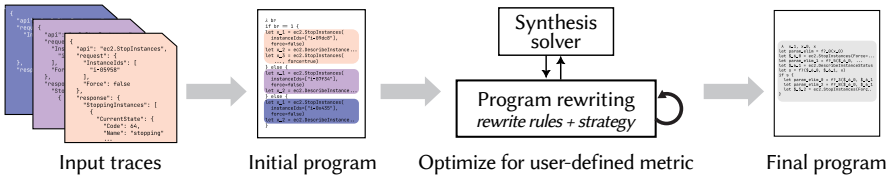


Fig. 1. Overview of our synthesis approach.

might repeatedly follow a sequence of steps to create a data store and then attach an access policy to that data store. For convenience, they create a policy name by appending the string "policy" to an ID generated by the creation of the data store. The log of these actions kept on the cloud will contain the creation of the data store, the policy, and the connection between the two. It will not contain the hidden function where the administrator bases the role name on the store's metadata.

To help automate such repetitive tasks, we propose a new synthesis technique, Syren. Syren is the first approach to synthesize programs from *partial* traces. It can infer both control flow and non-trivial hidden pure function calls with no additional input from the user. We combine optimizing program rewrites of an initial trivial solution with calls to a syntax-guided synthesizer (SyGuS) with input-output examples as specifications. Synthesis from input-output examples, also known as Programming-By-Example (PBE), has a long history of research and enables the synthesis of nontrivial functions. In Syren, we rely on a PBE synthesizer to synthesize hidden functions without additional input from the user. These hidden functions, in turn, let us perform further computation so that the rewrites can expose more intricate relations between data.

In example- and trace-based synthesis, we can assume the existence of a hidden *target program*, the program with the exact desired behavior. In the cloud administrator example above, the complete target program exists only in the administrator's mind. We can describe the *behavior* of a program as the set of traces that result from its execution on any possible input. When the synthesis specification is a set of traces, there is a trivial solution to the synthesis problem—the program that exactly reproduces the input traces. This trivial solution, though correct by construction (in the sense that it satisfies the specification), is likely not the target program that the user desires. The user probably wants a program that *generalizes* the provided traces. For example, our cloud administrator does not want a program that can reproduce all of the data store names they have used in the past, they want a program that takes a data store name as a parameter, allowing them to provide important information like the name, while saving them many repetitive clicks on a web interface. The trivially correct program is a lower bound on the behavior of all possible correct programs because it produces the minimal set of traces to be considered correct. The target program, on the other hand, is an upper bound of the desired behaviors—it would provide us with all the possible traces that are considered correct, but behaviors not exhibited by the target are undesirable. Since we do not have access to the target program, we need another way to quantify as accurately as possible how close to the target behavior a program in the space is. In other words, we need a *cost function* to efficiently traverse the program space.

Besides generalizing to traces beyond those observed (allowing more behavior), we considered readability when building Syren's cost functions. Like other synthesis works before us, we generally follow Occam's razor principle and favor shorter programs to achieve both goals. We evaluate Syren using two different cost functions we built, but our approach is agnostic to the cost function; a user can write different cost functions for Syren if they choose to optimize for something else. For example, a user might specify that a specific input to a given method call must be generalized, that they want to minimize the syntactic statements in the program, or a combination of both.

Figure 1 summarizes our approach: an initial set of traces is provided, under the assumption that this set describes a task to be performed. We use that set of traces to construct an initial program.

The core of our technique is the rewriting process guided by a user-defined program metric (the cost function) that may use an underlying syntax-guided synthesizer to generate some of the program's components. We present Syren's rewrite rules and how we use the synthesizer in §5.

While our synthesis problem could be encoded as an optimization Satisfiability Modulo Theories (SMT) problem or as a SyGuS problem, the large search space prevents off-the-shelf state-of-the-art solvers from solving it. In Syren, we efficiently traverse the search space by combining SyGuS with *program rewriting*. We start our search by building a trivial program, a lower bound on program behavior that is correct by construction. Then, we progressively rewrite it as long as we can decrease a cost function, generalizing the program and adding desired behavior, while *provably maintaining correctness*. Program rewriting is a natural approach when it comes to optimizing programs for a given cost. For example, compiler optimization and superoptimization [22, 23, 31, 36] uses rewrites to improve the performance of programs across various dimensions. A challenge to consider when developing a rewrite system is that an unsound rewrite can lead to incorrect programs. To reason about the correctness of our solution, we formally define a domain-specific language and a correctness statement that allow us to prove our rewrite rules correct w.r.t. the language and statement. The final result is guaranteed to be correct and is optimized for the user-defined metric.

To show the practical applicability of Syren, we implemented the algorithm and evaluated it on benchmarks gathered from cloud automation, filesystem manipulation, and document edition scripts. The 54 benchmarks were collected from custom tasks, existing AWS Automation Runbooks [3], Blink Automations [7] and related work [18]. We show that our approach generates scripts that accurately perform the task intended by the user for many tasks. This includes tasks with conditional control flow, loops, and hidden function calls between visible calls. We experiment with different strategies to apply rewrite rules and various metrics to optimize the final synthesized program. This shows that our approach is flexible and adaptable to different user requirements.

In summary, we make the following contributions:

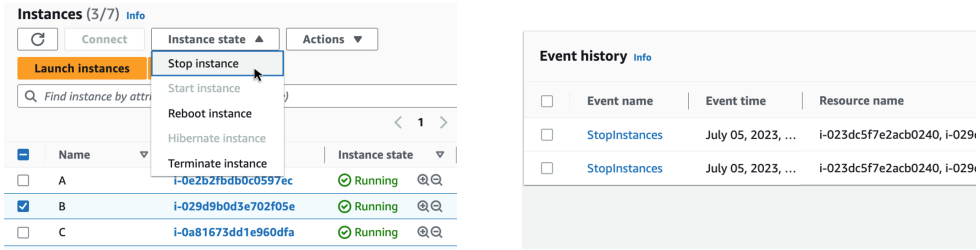
- We describe a synthesis problem where the specification is a set of traces containing visible function calls, and the program to synthesize must perform those function calls and additionally implement hidden function calls and control flow.
- We implement Syren, using a new approach that combines optimizing rewrites with traditional input-output guided program synthesis.
- We evaluate Syren on a set of benchmarks built from AWS automation runbooks, previous work on API synthesis, and publicly available libraries.

2 From Partial Traces to Programs

We start by illustrating one execution of our synthesis procedure. In this example, we synthesize a cloud management script that shuts down computing instances. The specification is a set of traces built from logs collected by the cloud provider (in this case, AWS [4], one of the largest cloud providers) as the user performed the desired task manually a few times using a visual interface. The logs contain calls to the AWS API. For our synthesizer, the API calls are *visible functions* in our program, whereas the data transformations with input and output data used for these API calls will be *hidden functions*.

Motivating Example. An administrator might stop computing instances using the AWS console (shown in Figure 2a) by:

(1) selecting the instances they want to stop,
(2) clicking "Instance state", and
(3) selecting "Stop instance" from the options in the drop-down menu that appears.



(a) Stopping instances in the AWS console (b) The resulting log showing API calls

Fig. 2. Performing and monitoring actions on the AWS console

```
λ instanceId.
let ids = list(instanceId)
let _ = ec2.StopInstances(instanceIds=ids, force=false)
let s = ec2.DescribeInstanceStatus(instanceIds=ids, includeAllInstances=true)
let status = extractInstanceStatus(s)
if status != "stopped" { let _ = ec2.StopInstances(instanceIds=ids, force=true) }
where
extractInstanceStatus := $.InstanceStatuses[0].InstanceState.Name
```

Fig. 3. Example of a program that stops an EC2 instance.

Clicks on the visual interface trigger calls to a cloud API, in this case, from the `ec2` service. A program could accomplish the same task by making the exact same API calls.

The program in Figure 3 automates a task slightly more complex than the 3 steps described before. To perform this more complex task in the visual interface, a user would have to, after the previous steps:

(4) click "refresh" to view the current status for their computing instances,
  (a) if the instance status is "stopped", then terminate the task here;
  (b) otherwise once again click "Instance state", and
(5) select "Force stop instance" from the drop-down menu.

Figure 2b shows the logs created in AWS by these actions. We use these logs, which from now on we will refer to as *traces*, as a specification for the synthesis problem. The synthesized program can be used to automate the task, reducing a many-click, repetitive, and error-prone task to a single press of a button. Our approach ultimately generates a program similar to the one shown in Figure 3 from the traces. We now give an overview of how our approach solves this specific problem.

Synthesis algorithm overview. We start with a set of traces, each containing the API calls made while carrying out a task. The following two traces exemplify the task described previously:

```
Trace #1 :=
  (ec2.StopInstances("InstanceIds": ["i-09dc8"], "force": false), { ... })
  (ec2.DescribeInstanceStatus("InstanceIds":["i-09dc8"]),
     {"Statuses": [{"InstanceState": {"Code":64, "Name":"stopping"}, ...}], ... })
  (ec2.StopInstances("InstanceIds": ["i-09dc8"], "force": true), { ... })
Trace #2 :=
  (ec2.StopInstances("InstanceIds": ["i-07f34"], "force": false), { ... })
  (ec2.DescribeInstanceStatus("InstanceIds": ["i-07f34"]),
     {"Statuses": [{"InstanceState": {"Code":80, "Name":"stopped"}, ...}], ...})
```

Each trace is a sequence of pairs, and each pair represents an API call: the first element shows the API method name and its inputs, and the second shows the response to that API call. Both example traces start with two calls to the API methods, `ec2.StopInstances` with the parameter `force` set to `false`, and `ec2.DescribeInstancesStatus`. In trace #1, the output of the call to `ec2.DescribeInstancesStatus` does *not* show the current status as `"stopped"`, so we see a second call to `ec2.StopInstances` with `force` set to `true`. The call to `ec2.DescribeInstancesStatus` shows the current status as `"stopped"` in trace #2, so no further calls are recorded.

The first step in our synthesis pipeline, as shown in Figure 1, is to build an initial program that provably generates all the input traces for some initial global state. We do this by branching the execution on the value of a fresh integer variable, `br`, and replaying each trace on a different branch. For our running example with two traces, we generate the following initial program:

```
λ br
if br == 1 {
   let x_1_1 = ec2.StopInstances(instanceIds=["i-09dc8"], force=false)
   let x_1_2 = ec2.DescribeInstanceStatus(instanceIds=["i-09dc8"])
} else {
   let x_2_1 = ec2.StopInstances(instanceIds=["i-07f34"], force=false)
   let x_2_2 = ec2.DescribeInstanceStatus(instanceIds=["i-07f34"])
}
```

Our approach progressively transforms the program by applying rewrite rules that decrease an optimization metric. In this example, we use a metric that measures the syntactic complexity of the program: we add 10 for each statement, 1 for each parameter, and 1 for each usage of `br`, which is a synthetic variable that should only be used by the initial program. Initially, the program has cost 62. The first rewrite applied to the program pulls the first call to `ec2.StopInstances` out of the if-statement and replaces its arguments, which were constants, with a ternary expression. A second application of the same rewrite rule extracts the call to `ec2.DescribeInstanceStatus`. Each rule reduces the cost by 9, eliminating one statement but introducing one usage of `br`. After applying these two rules, the intermediate program has cost 44:

```
λ br
let x_1_1 = ec2.StopInstances(instanceIds=(br==1)?["i-09dc8"]:["i-07f34"], force=false)
let x_1_2 = ec2.DescribeInstanceStatus(instanceIds=(br==1)?["i-09dc8"]:["i-07f34"])
if br == 1 { let x_1_3 = ec2.StopInstances(instanceIds=["i-09dc8"], force=true) }
```

To eliminate usages of `br` in the ternary expressions, we need to replace the conditional expression `br==1` with another expression that evaluates to the same value but does not use `br`. There are two ways to achieve this: either introduce a new input parameter that takes the value of the expression, or synthesize a function that will eventually evaluate to the conditional expression value. To synthesize a (nonconstant) function, Syren considers as potential inputs all variables bound in the scope of the expression being replaced. In the first appearance of the expression `(br==1)?["i-09dc8"]:["i-07f34"]`, there are no variables bound in the scope that could be used as input to a data transformation. So, we have no choice but to introduce a new input parameter, `i_1`. Syren replaces all usages of the original expression with `i_1`. Within the conditional branches, Syren also replaces the usages of the value the expression evaluates to (considering the conditional). This results in the following program, with cost 43:

```
λ br, i_1
let x_1_1 = ec2.StopInstances(instanceIds=i_1, force=false)
let x_1_2 = ec2.DescribeInstanceStatus(instanceIds=i_1)
if br == 1 { let x_1_3 = ec2.StopInstances(instanceIds=i_1, force=true) }
```

The final rewrite for this example replaces the last conditional that depends on `br` with the output of a new data transformation ϕ over all variables in scope. After this last rewrite rule, the synthesized program is parametric on an implementation of that data transformation:

```
Λ ϕ. λ i_1
let x_1_1 = ec2.StopInstances(instanceIds=i_1, force=false)
let x_1_2 = ec2.DescribeInstanceStatus(instanceIds=i_1)
let c = ϕ(i_1, x_1_1, x_1_2)
if c { let x_1_3 = ec2.StopInstances(instanceIds=i_1, force=true) }
```

This rewrite is valid only if we can provide an implementation `f` for ϕ that ensures the program can reproduce the input traces. During the rewrite process, we maintain a mapping from the identifiers in the program to corresponding values in the traces. Then, we use these mappings to compute a set of input-output constraints that `f` must satisfy. For the traces and program in this example, we can extract the following two input-output pairs for the desired implementation `f`:

```
f(["i-09dc8"], {"StoppingInstances": [...], "ResponseMetadata": {...}},
  {"Statuses":[{"InstanceState":{"Code":64, "Name":"stopping"},...}],...}) = true
```
for trace τ_1,

```
f(["i-07f34"], {"StoppingInstances": [...], "ResponseMetadata": {...}},
  {"Statuses":[{"InstanceState":{"Code":80, "Name":"stopped"},...}],...}) = false
```
for trace τ_2.

We encode the problem into a syntax-guided synthesis solver to generate a solution, which yields:[1]

```
f := (i_1, x_1_1, x_1_2) -> x_1_2.InstancesStatuses[0].InstanceState.Name != "stopped".
```

Substituting ϕ for `f` yields a program that is correct by construction, with a minimal cost of 41. This final program is syntactically equivalent to the one in Figure 3.

3 Background and Definitions

In this section, we formally introduce concepts necessary to explain our approach to synthesizing scripts that compose visible side-effecting function calls with conditionals, loops, and (hidden) pure function calls. In §3.1, we define the domain-specific language (DSL) syntax of our synthesized programs. This DSL is an intermediate representation that we can easily convert to most common scripting languages. Next, in §3.2, we define *program traces*, which we use as an input specification for synthesis. Finally, in §3.3, we define the semantics of our language, which relates a program to input and output states, as well as to traces. These semantics allow us to prove properties about the manipulation of the DSL to prove our approach correct.

3.1 Core Language Syntax

Figure 4 presents the syntax of our core DSL. A program $\mathcal{P}$ is a function with input variables[2] $\overline{x}$ and a set of hidden functions $\overline{f := \mathcal{F}}$. The body of the program is an instruction list $\mathcal{I}$, either empty (ϵ) or with statements. A statement $\mathcal{S}$ can be a simple binding, a conditional, a loop, or the instruction that marks the end of the script. A binding `let` $x = e_1$ s_2 binds e_1 to x in s_2. Conditionals `if` b $\{s_1\}$ `else` $\{s_2\}$ s_3 execute s_1 if b is true, otherwise s_2, and then s_3. Our language has two forms of loops. `retry` s `until` b (retry until) executes the instructions in s at least once, until b is true, or some predefined maximum number of retries is reached. `for` $x \in L$ $\{s\}$ iterates through the list L, binding x to each element and executing s.

[1] In practice, we need at least one more trace and its respective input/output example to synthesize this solution. If we consider only the two example traces shown, a simpler implementation is synthesized for `f`: `f := (i_1, x_1_1, x_1_2) -> i_1 == ["i-09dc8"]`.

[2] We write $\overline{x}$ to denote zero or more occurrences of x.

$$
\begin{array}{lcll}
\mathcal{P} & ::= & \lambda\overline{x}.\mathcal{I} \texttt{ where } \overline{f := \mathcal{F}} & \textit{Program} \\
\mathcal{I} & ::= & \epsilon \mid \mathcal{S}\ \mathcal{I} & \textit{Instructions} \\
\mathcal{S} & ::= & \texttt{let } x = \mathcal{E} & \textit{Pure binding} \\
 & \mid & \texttt{if } \mathcal{B} \{\mathcal{I}\} \texttt{ else } \{\mathcal{I}\} & \textit{Conditional} \\
 & \mid & \texttt{retry } \{\mathcal{I}\} \texttt{ until } \{\mathcal{B}\} & \textit{Retry until} \\
 & \mid & \texttt{for } x \in L \{\mathcal{S}\} & \textit{Foreach loop} \\
 & \mid & \texttt{return} & \textit{Return} \\
\mathcal{E} & ::= & \mathbb{A}(\overline{x}) & \textit{Visible function call} \\
 & \mid & f(\overline{x}) & \textit{Hidden function call} \\
 & \mid & \mathcal{B}\ \texttt{?}\ \mathcal{E}\ \texttt{:}\ \mathcal{E} & \textit{Ternary expression} \\
\mathcal{B} & ::= & \top \mid \bot \mid \mathcal{B} \vee \mathcal{B} \mid \mathcal{B} \wedge \mathcal{B} \mid \neg\mathcal{B} & \textit{Predicates} \\
 & \mid & x = C & \textit{Value check} \\
 & \mid & x (\geq \mid > \mid \leq \mid <) y & \textit{Value comparison} \\
\mathcal{F} & ::= & ?? & \textit{Pure Function} \\
C & ::= & s \in \texttt{string} \mid n \in \mathbb{Z} \mid b \in \{\texttt{true}, \texttt{false}\} & \textit{Constants}
\end{array}
$$

Fig. 4. Core scripting language.

Expressions $\mathcal{E}$ can be visible or hidden function calls. A visible function call $\mathbb{A}(\overline{x})$ is a call to some externally defined function $\mathbb{A}$ with arguments $\overline{x}$, and a hidden call $f(\overline{x})$ is a call to a pure function f whose implementation $\mathcal{F}$ is defined in the program. Visible functions can have side effects on the outside world, changing the results of future calls. However, they do not change the local state of the DSL execution. The hidden functions are *pure*, and their specific syntax depends on the chosen domain.

We clearly separate pure function implementations $\mathcal{F}$ from the rest of the program for two reasons: to simplify reasoning about variable usage and whole-program rewrites, and to highlight the fact that our DSL is agnostic of the hidden functions domain. In our implementation of Syren, we consider a minimal language of hidden functions, which includes a JSONPath as well as other basic operations over strings, numbers, and Booleans. However our approach can be generalized to any other language for hidden functions, as long as expressions in that language can be synthesized.

Example 3.1. The script in Figure 3 is an example of a script written in our DSL. The script takes a single input, `instanceIDs` and defines one data operation **`extractInstanceStatus`**. The visible functions are the API methods `StopInstances` (called twice), and `DescribeInstanceStatus`.

3.2 Program traces

Our synthesis starts from *observable* traces that can be produced by the program's execution. The *observable* traces contain only records of the visible function calls made by the program and their results; the hidden function calls do not appear in traces . Formally, a trace is a (possibly empty) finite sequence of records of all visible calls that the run of the program makes:

$$\tau := \langle\rangle \mid (\mathbb{A}(\overline{v}), e) :: \tau \quad \text{(traces)},$$

where $\langle\rangle$ is the empty trace, operator $\bullet$ performs concatenation, and $\langle\rangle \bullet \tau = \tau \bullet \langle\rangle = \tau$ for any trace τ. Each record is a pair $(\mathbb{A}(\overline{v}), e)$, where the first element states the name of the function $\mathbb{A}$ and the inputs to the call $\overline{v}$; the second element, e, is the response to the call. e is an expression in the same language as the hidden functions.

3.3 DSL Semantics

Next, we define the semantics of our language as the relation $\Rightarrow$, presented in Figure 5. The relation $\Rightarrow$ maps a pair of a program body and state to a triple of local state, trace and continuation token. In our DSL semantics, we refer to two different notions of state. The local state, σ, stores the bindings of every variable assigned in the program at a given point. The global state, G,

$$\textsc{Seq}\ \frac{(s,\sigma) \Rightarrow (\sigma', \tau, \texttt{cont}) \qquad (S',\sigma') \Rightarrow (\sigma'', \tau', cr)}{(s\ S', \sigma) \Rightarrow (\sigma'', \tau \bullet \tau', cr)} \qquad \textsc{Seq-S-Term}\ \frac{(s,\sigma) \Rightarrow (\sigma', \tau, \lightning)}{(s\ S', \sigma) \Rightarrow (\sigma', \tau, \lightning)} \qquad \textsc{Ret}\ \frac{}{(\texttt{return}, \sigma) \Rightarrow (\sigma, \langle\rangle, \lightning)}$$

$$\textsc{Emp}\ \frac{}{(\epsilon, \sigma) \Rightarrow (\sigma, \langle\rangle, \texttt{cont})} \qquad \textsc{Ite-}\top\ \frac{\sigma \models b \qquad (S_\top, \sigma) \Rightarrow (\sigma', \tau, cr)}{(\texttt{if}\ b\ \{S_\top\}\ \texttt{else}\ \{S_\bot\}, \sigma) \Rightarrow (\sigma', \tau, cr)}$$

$$\textsc{Ite-}\bot\ \frac{\sigma \not\models b \qquad (S_\bot, \sigma) \Rightarrow (\sigma'', \tau', cr)}{(\texttt{if}\ b\ \{S_\top\}\ \texttt{else}\ \{S_\bot\}, \sigma) \Rightarrow (\sigma'', \tau', cr)}$$

$$\textsc{Retry-Until-Continue}\ \frac{(S,\sigma) \Rightarrow (\sigma', \tau', cr) \qquad \sigma' \not\models b \wedge \#_\iota < K \qquad \texttt{retry}\ \{S\}\ \texttt{until}\ \{b\}, \sigma'[\#_\iota \to \#_\iota + 1]) \Rightarrow (\sigma'', \tau'', cr)}{({}^{\#\iota}\texttt{retry}\ \{S\}\ \texttt{until}\ \{b\}, \sigma) \Rightarrow (\sigma'', \tau' \bullet \tau'', cr)}$$

$$\textsc{Retry-Until-Stop}\ \frac{(S,\sigma) \Rightarrow (\sigma', \tau', cr) \qquad \sigma' \models b \vee \#_\iota \geq K}{({}^{\#\iota}\texttt{retry}\ \{S\}\ \texttt{until}\ \{b\}, \sigma) \Rightarrow (\sigma'[\#_\iota \to 0], \tau', cr)} \qquad \textsc{Retry-S-Term}\ \frac{(S,\sigma) \Rightarrow (\sigma', \tau', \lightning)}{({}^{\#\iota}\texttt{retry}\ \{S\}\ \texttt{until}\ \{b\}, \sigma) \Rightarrow (\sigma', \tau', \lightning)}$$

$$\textsc{For-Continue}\ \frac{(S, \sigma[x \to L[\#_\iota]]) \Rightarrow (\sigma', \tau', cr) \qquad \sigma' \models \#_\iota < |L| \qquad ({}^{\#\iota}\texttt{for}\ x \in L\ \{S\}, \sigma'[\#_\iota \to \#_\iota + 1]) \Rightarrow (\sigma'', \tau'', cr)}{({}^{\#\iota}\texttt{for}\ x \in L\ \{S\}, \sigma) \Rightarrow (\sigma'', \tau' \bullet \tau'', cr)}$$

$$\textsc{For-Stop}\ \frac{(S,\sigma) \Rightarrow (\sigma', \tau', cr) \qquad \sigma'\#_\iota \geq |L|}{({}^{\#\iota}\texttt{for}\ x \in L\ \{S\}, \sigma) \Rightarrow (\sigma'[\#_\iota \to 0], \tau', cr)} \qquad \textsc{For-S-Term}\ \frac{(S,\sigma) \Rightarrow (\sigma', \tau', \lightning)}{({}^{\#\iota}\texttt{for}\ x \in L\ \{S\}, \sigma) \Rightarrow (\sigma', \tau', \lightning)}$$

$$\textsc{Hidden}\ \frac{\sigma \models f := \mathcal{F} \wedge \exists e \cdot \mathcal{F}(\overline{y}) = e}{(\texttt{let}\ x = f(\overline{y}), \sigma) \Rightarrow (\sigma[x \to e], \langle\rangle, \texttt{cont})} \qquad \textsc{Visible}\ \frac{\sigma \models \exists \overline{v} \cdot \overline{y} = \overline{v} \qquad \mathbb{A}(G, \overline{v}) \downarrow e}{(\texttt{let}\ x = \mathbb{A}(\overline{y}), \sigma) \Rightarrow (\sigma[x \to e], (\mathbb{A}(G, \overline{v}), e), \texttt{cont})}$$

Fig. 5. Big-step semantics

represents external resources accessed by the visible functions in the trace. The notion of global state is necessary because visible functions are not pure functions of their inputs; depending on the resources they access, two calls to the same function with the same inputs might return different outputs. The continuation token is either `cont`, indicating that evaluation must continue, or $\lightning$, indicating that the evaluation must stop.

The rule Seq specifies how a statement s followed by instructions S' is evaluated sequentially and traces are concatenated. The rule Seq-S-Term handles the case where the first statement *terminates* the evaluation of the program. Ret states that the statement `return` always terminates early with an empty trace. Emp generates an empty trace, does not change the local state, and always continues evaluation. In Ite-$\top$ and Ite-$\bot$ for conditionals, either the branch with instructions $S_\bot$ or $S_\top$ are evaluated depending on whether the local state entails b or not. The continuation or termination token cr of the if-then-else is the same as the token in the evaluation of the branch, in particular, the statement terminates the evaluation when the branch terminates evaluation.

The rules Retry-Until-Continue, Retry-Until-Stop, and Retry-S-Term define how the retry-until statements are evaluated. Note that retry-until does not have the same semantics as a while loop: it will always terminate, and the predicate b is not guaranteed to hold when the loop ends. We assume a constant K that bounds the number of times the body of a retry-until statement can be "retried". For any set of traces, we can select a K higher than the longest trace. This ensures that

K is high enough that any trace can be regenerated by the transformed program without timing out. We discuss this more in §7.

Each retry-until statement is given a unique identifier, $\#_\iota$, and each of those identifiers is assigned 0 in the initial local state. The rule Retry-Until-Continue states that a retry-until statement with identifier $\#_\iota$ evaluates to state σ'' and trace $\tau' \bullet \tau''$ when one iteration results in σ' and τ', $b \wedge \#_\iota < K$ holds in state σ', and evaluating again the retry-until statement with the state σ' where $\#_\iota$ is incremented results in σ'' and τ''. The rule Retry-Until-Stop handles the case where $b \wedge \#_\iota < K$ does *not* hold after evaluating the body of the loop. The rule Retry-S-Term handles the case where the body of the retry loop returns, and therefore the entire program returns. The rules for for-loops (For-Continue, For-Stop and For-S-Term) are similar, the main difference being that the variable x is bound at each new iteration and the stop condition depends on the size of the list L, not the value of the Boolean b.

We differentiate binding on the type of expression they bind. If it is a call to a hidden function f (rule Hidden) then the local state is modified by binding x to the value $\mathcal{F}(\overline{y})$ evaluates to in the current state σ, according to the semantics of the data-transformation domain and assuming $\mathcal{F}$ is f's implementation. The trace is unchanged by the hidden function. If it is a call to a visible function (rule Visible), the arguments of the call are evaluated in σ, the result of the call is bound to x in the local state, and the call to $\mathbb{A}$ with the input values is recorded in the trace.

The relation $\mathbb{A}(G, \overline{v}) \downarrow e$ means that the call to externally defined function $\mathbb{A}$ with input $\overline{v}$ in global state G returns a response e. We implicitly update the global state as a function of each visible call and transfer it through sequences. This means that two programs that start in the same global state and execute the same visible calls receive the same responses to those calls. This formulation allows us to reason about the semantics of the program given existing pairs of input-output examples of calls (the traces) without actually executing any of the calls; we only need to assume the initial global state is the same as in the traces. A limitation of this approach is that time is not considered, so our approach will be unsound in situations where responses implicitly depend on time as opposed to ordering.

Finally, we introduce notation for relating traces with programs and states.

Definition 3.2 (Program Evaluation). Let $P := \lambda\overline{x}.S$ `where` $\overline{f := \mathcal{F}}$ a program and σ a state mapping every variable in $\overline{x}$ to some value and every f into the corresponding $\mathcal{F}$. Then, given a starting global state, there might exist exactly one trace τ and termination token cr such that $(S, \sigma) \Rightarrow (\sigma', \tau, cr)$. If and only if the trace and termination token exist, we say that τ is a *trace of* P with input σ and write $P(\sigma) = \tau$. While not all syntactically valid programs fully evaluate, all *synthesized* programs evaluate by construction. Given that all programs we discuss are synthesized, we no longer need to consider programs that do not successfully evaluate.

4 Synthesis Problem

As we illustrated with our motivating example, the synthesis problem solved in this paper consists in finding a program P in the language described in Figure 4 that reproduces a set of input traces T_{in}. In this section, we formalize this intuitive correctness constraint and define our synthesis problem as a combination of the correctness constraint and another constraint on the quality of the program.

4.1 Correct Solutions

We use the notion of *trace subsumption* to describe one program that can generate at least the same traces as another:

Definition 4.1 ($\sqsupseteq$). A program P' subsumes a program P ($P' \sqsupseteq P$) if and only if for every state σ and trace τ such that $P(\sigma) = \tau$, there exists a state σ' such that $P'(\sigma') = \tau$.

Note that $\sqsupseteq$ is a partial order on programs. If $P \sqsupseteq P'$ and $P' \sqsupseteq P$ then P and P' are *trace equivalent*. In general, we are interested in transformations that preserve subsumption (i.e. $P \rightsquigarrow P'$ only if $P' \sqsupseteq P$, where $\rightsquigarrow$ is a transformation), not just trace equivalence. Formally, our correctness constraint Ψ is

$$\Psi(P, T_{in}) \equiv \forall \tau_i \in T_{in} \cdot \exists \sigma \cdot P(\sigma) = \tau_i. \tag{1}$$

The set T_{in} is a set of finite input traces $\tau_1, \tau_2, ..., \tau_t$[3]. There is always a trivial solution to Ψ for a given set of traces T_{in}. It can be constructed using a single integer parameter `br` and $|T_{in}|$ branches, where each branch can be selected with a value for `br`, and the branch makes the API calls contained in the `br`-th trace of T_{in}. We show in §2 how this solution is built directly from the traces; it simply replays each of the traces, and the set of possible traces of the program is exactly T_{in}.

Another less trivial solution would consist of combining visible function calls when possible but leaving all the inputs of the visible calls as parameters of the program, thus always discarding the output of the visible function call. This is also not an acceptable solution. Although it generalizes to other inputs, the generalization only comes from the program being entirely parameterized.

4.2 Quality Constraint

Although trivial solutions exist, they usually will not be what a user would expect as output; there is an expectation that the solution is a generalization of the traces. There are also infinitely many correct (solutions to Ψ) programs that are very general; consider, for example, a program listing all possible syntactic productions that satisfy the correctness constraint in branches. However, those programs are also not typically what the user expects.

To address this challenge, we assume the existence of a *program cost function* χ that, given a program P and a set of traces T_{in}, returns a positive number. This program cost function reflects what the user expects; a good program is one with a low cost. For example, the program cost function could return the count of branches and the count of parameters of the program, indicating that the user desires a program with low complexity that is likely to generalize well. The goal of the synthesizer is to find a program that minimizes the cost function. Formally, the goal is to solve, given a fixed set of traces T_{in},

$$\min_{\forall P \cdot \Psi(P, T_{in})} \chi(P, T_{in}). \tag{2}$$

In this paper, we describe a generic algorithm that is parametric in χ, first by describing our rewrites in §5, and then by describing the search approach in §6.

5 Rewriting Programs

Our synthesis algorithm applies a succession of rewrite rules to transform an initial trivial program into a more general and user-friendly one. Each of these rewrite rules provably maintains the program's correctness constraint, Ψ, so that all intermediate programs can generate all input traces in T_{in}. We split our rules into two categories, *synthesis* rules and *refinement* rules, depending on how they maintain correctness.

Lemma 1. *Subsumption preserves correctness:* $\Psi(P, T_{in}) \bigwedge P' \sqsupseteq P \implies \Psi(P', T_{in})$.

Proof. If P can generate all traces in T_{in}, and P' can generate all traces that P can generate, then P' can generate all traces in T_{in}. □

Refinement rewrite rules preserve subsumption: for all P and P', if a refinement rule rewrites P into P', $P \rightsquigarrow P'$, then $P' \sqsupseteq P$. By Lemma 1, these rules preserve the correctness when applied to a correct program. The following is an example of the application of a refinement rule that

[3]We always assume that $|T_{in}| > 1$.

extracts an identical instruction $\mathcal{R}$ from both branches of an if-then-else statement. This rewrite does not change the semantics of the program but improves its readability by reducing its number of instructions.

$$\begin{array}{l}\lambda\ \overline{x}.\\ \mathcal{U}\ \texttt{if}\ \mathcal{C}\ \{\ \mathcal{R}\ \mathcal{S}\ \}\ \texttt{else}\ \{\ \mathcal{R}\ \mathcal{T}\ \}\ \mathcal{V}\end{array} \rightsquigarrow \begin{array}{l}\lambda\ \overline{x}.\\ \mathcal{U}\ \mathcal{R}\ \texttt{if}\ \mathcal{C}\ \{\ \mathcal{S}\ \}\ \texttt{else}\ \{\ \mathcal{T}\ \}\ \mathcal{V}\end{array}$$

Synthesis rewrite rules all follow the same pattern: replace an expression e with the output of a call to a to-be-synthesized pure function ϕ. ϕ is not visible in the traces, so we refer to it as a *hidden* function. All the bound variables available at the location are used as arguments to ϕ, except when the rule is trying to eliminate a parameter. The correctness of a synthesis rewrite rule is conditioned by the existence of a solution for the hidden function calls they introduce. Formally, we denote by $\Lambda\overline{\phi} \cdot P$ a program P parametric on a set of hidden functions $\overline{\phi}$. For a given set of implementations $\overline{f}$, $(\Lambda\overline{\phi} \cdot P)(\overline{f})$ is a valid program in our DSL. A single synthesis rule $\rightsquigarrow$ rewrites P to a program $P' := \Lambda\phi \cdot P_s$ parametric on some hidden function ϕ. The rewrite rule $P \rightsquigarrow P'(f)$ is correct for some function f only if $P'(f) \sqsupseteq P_{in}$ where P_{in} is the initial program. By Lemma 1, if $\Psi(P_{in}, T_{in})$ and $P'(f) \sqsupseteq P_{in}$, then $\Psi(P', T_{in})$. Note that we can chain multiple synthesis rules together and check for correctness only later, i.e. rewrite $P \rightsquigarrow \Lambda\phi \cdot P_s \rightsquigarrow \Lambda\phi, \phi' \cdot P'_s$ and then find f and f' later to instantiate ϕ and ϕ'. We explain how to find the implementation of hidden functions f in §5.2. For a list of Syren's rewrite rules, the reader can refer to Appendix B.

Example 5.1. We illustrate below how we apply a sequence of rewrite rules to generalize programs and produce an acceptable solution. Suppose that we have constants `c1,c2,c3`, visible functions `A`, `B` and some initial program P_{in} as shown below:

P_{in} :

```
λ br.
if br = 1 {
  let x1 = A(c1)
  let y = B(c2)
} else {
  let x2 = A(c3)
}
```

$\rightsquigarrow$

P_1 :

```
λ br.
let a = (br = 1) ? c1 : c3
if br = 1 {
  let x1 = A(a)
  let y = B(c2)
} else { let x2 = A(a) }
```

$\rightsquigarrow$

P_2 :

```
λ br.
let a = (br = 1) ? c1 : c3
let x = A(a)
if br = 1 { let y = B(c2) }
```

$\rightsquigarrow$

P_3 :

```
λ br, d.
let x = A(d)
if br = 1 {
  let y = B(c2)
}
```

$\rightsquigarrow$

P_4 :

```
Λ ϕ λ br, d.
let x = A(d)
let c = ϕ(d, x)
if c { let y = B(c2) }
```

$\rightsquigarrow$

P_5 :

```
Λ ϕ λ br, d.
let x = A(d)
let c = ϕ(d, x)
if c { let y = B(c2) }
```

We rewrite P_{in} using a refinement rule that introduces a new variable `a`, which is bound to the constants `c1` or `c3` in the conditional, and then used as argument to the calls to `A`. P_1 is the resulting program. Then, we apply to P_1 the refinement rule shown in §5, which factors the calls to `A` out of the conditional, resulting in P_2. The third rewrite eliminates the expression `let a = if br = 1 {c1} {c3}` which depends on `br` and introduces a parameter `d` that takes its value. This rewrite provably maintains correctness and produces a program, P_3, generalized to any input `d`. A fourth rewrite introduces a function parameter ϕ (to be synthesized) to eliminate `br` from the conditional, resulting in P_4. The final rewrite eliminates the unused parameter `br`.

5.1 Trace Valuation

Rewrites maintain correctness by ensuring that, given an implementation for the hidden function introduced, the program can still generate the initial set of traces. While refinement rewrite rules are correct for all inputs of the program, synthesis rewrite rules require more attention. To keep track of this correctness constraint, we maintain an augmented local state σ, the trace valuation of the program, which relates variables in the program with a specific trace and a concrete value (and an iteration number for variables in loops). The trace valuation stores the relationships necessary for the rewritten program to reproduce each trace $\tau \in T_{in}$, and each rewrite rule application modifies that state to maintain the invariant. This ensures that the value of all expressions of the program for a certain trace is always known, either because the variable's value is known, or the expression's value can be computed from those known values. Given an expression e, trace valuations σ and trace τ we denote the value of e in trace τ and state σ by $[\![e]\!]_{\sigma,\tau}$.

The initial program has only one variable `br`, and initially $[\![\texttt{br}]\!]_{\sigma,\tau_i} = i$ for each trace $\tau_i \in T_{in}$. Then, each rewrite rule $\rightsquigarrow$ in our system is accompanied with a trace valuation transformation t, which we denote by $\rightsquigarrow_t$. We introduce a new function I, which extracts the program parameters from a state, and overload $\rightsquigarrow$ to apply to sequences of instructions as well, instead of entire programs only. Each t and associated rewrite $\rightsquigarrow_t$ is correct when for each input trace, when a rewrite and corresponding trace valuation transformation are applied, if the program runs with the inputs of the updated state then it produces the same input trace:

$$\tau_i \in T_{in} \wedge t(\sigma_0') = \sigma_1' \wedge (S, \sigma_0) \Rightarrow (\sigma_0', \tau_i, cr) \wedge S \rightsquigarrow_t S' \implies (S', I(\sigma_1')) \Rightarrow (\sigma_1', \tau_i, cr)$$

The function t will encapsulate the parameter updates of the rewrite and any functions introduced. This rule requires the correctness of S (in the third conjunct of the hypothesis), and the conclusion directly implies the correctness of S', where $I(\psi_1')$ witnesses the existential needed by the correctness statement.

Synthesis rules replace an expression or number of expressions with a hidden function.

Example 5.2. The refinement rule of Example 5.1 can specified with its transformation t_3:

P_2:

```
λ br.
let a = if br = 1 { c1 } else { c3 }
let x = A(a)
if br = 1 { let y = B(c2) }
```

$\rightsquigarrow_{t_3}$

P_3:

```
λ br, d.
let x = A(d)
if br = 1 { let y = B(c2) }
```

where $t_3(\sigma) = \sigma[(d, \tau) \mapsto [\![\texttt{if br = 1 c1 else c3}]\!]_{\sigma,\tau}]$

That is, the trace valuation transformation t_3 corresponding to this rewrite assigns the resulting value of evaluating the eliminated expression `if br = 1 c1 else c3` to the new parameter `d`. Concretely, if the program above is synthesized from the two traces:

$$\tau_1 = (\texttt{A(c1)}, o_1) :: (\texttt{B(c2)}, o_2) \quad \text{and} \quad \tau_2 = (\texttt{A(c3)}, o_3)$$

Given that $[\![\texttt{br}]\!]_{\sigma,\tau_1} = 1$, we have $[\![\texttt{if br = 1 c1 else c3}]\!]_{\sigma,\tau_1} = \texttt{true}$, and therefore $[\![d]\!]_{t_3(\sigma),\tau_1} = \texttt{c1}$. For the second trace, we would have $[\![d]\!]_{t_3(\sigma),\tau_1} = \texttt{c3}$.

In a following step in Example 5.1, we apply the following synthesis rule to the program:

P_3:

```
λ br, d.
let x = A(d)
if br = 1 { let y = B(c2) }
```

$\rightsquigarrow_{t_4}$

P_4:

```
Λ ϕ λ br, d.
let x = A(d)
let c = ϕ(d, x)
if c { let y = B(c2) }
```

where $t_4(\sigma) = \sigma[(c, \tau) \mapsto [\![\texttt{br} = 1]\!]_{\sigma,\tau}]$

The trace valuation for program P_4 will map `d` to the correct boolean value in each trace, that is, $[\![c]\!]_{\sigma,\tau_1} =$ `true` and $[\![c]\!]_{\sigma,\tau_2} =$ `false`. Additionally, the values for the inputs of c and x will also be known from the state, for example for trace 1 $[\![d]\!]_{\sigma,\tau_1} =$ `c1` and $[\![x]\!]_{\sigma,\tau_1} = o_1$ (see trace τ_1).

Example 5.3. The rewrite rules that introduce retry loops are synthesis rules because the condition on which to stop the loop needs to be synthesized. Syntactically, the rewrite identifies a sequence of statements, possibly with conditionals, and rolls them into a loop. The following is an example of a loop introduction rewrite:

```
λ br, y⃗.                          Λ ϕ λ br, y⃗.
let a = A(c1)                     let a = A(c1)
let b1 = B(c2)          ⇝t        retry {
let b2 = B(c2)                      let b = B(c2)
if br=1 { let b3 = B(c2) }          let s = ϕ(b, a, y⃗)
                                  } until s
```

where $t(\sigma) = \sigma[(\texttt{(b,b,b)}, \tau) \mapsto [\![\texttt{(b1,b2,b3)}]\!]_{\sigma,\tau}] \cup [(\texttt{(s,s,s)}, \tau) \mapsto \texttt{(false,}[\![\texttt{br != 1}]\!]_{\sigma,\tau}\texttt{,true)}]$

In the rewritten program syntax, a new variable `s` is bound to the result of the hidden function ϕ and used as a stopping condition for the retry loop. The key in ensuring this is a correct rewrite is in the valuation transformation t. The new trace valuation maps *iterations* of `b` to the values of each statement that has been captured in the loop, represented by the vector `(b1,b2,b3)`. When evaluating the program for a trace, the variable `b3` will not be defined for the traces where `br` $\neq$ 1, in which case the value is null. The valuation of condition `s` is also a vector `(s,s,s)` that is computed by assigning the truth value of whether the statement in the trace should be the last one; at the end of the second iteration, `s` is `true` for the trace where `br!=1`.

5.2 Synthesizing Hidden Functions

The correctness of the result of applying a synthesis rewrite rule $P \rightsquigarrow P'(f)$ depends on satisfying a set of constraints imposed on f by the condition $\exists f.P'(f) \sqsupseteq P_{in}$. As we explained in the previous section, all rewrite rules update an extended state that keeps track of the valuations of the variables in a correct program. Given the value in the extended state, the synthesis of f is reducible to a standard programming-by-example (PBE) synthesis problem, deducible from σ only. Those constraints are solved with an off-the-shelf synthesizer to produce either an implementation for f or an unsatisfiability result. In the latter case, the synthesis rule cannot be applied while maintaining correctness.

$\mathcal{B}$	::=	$\mathcal{J} == \mathcal{V}$	*string or integer equality*
	|	$empty(\mathcal{J})$	*emptiness check*
	|	$!\mathcal{B}$	*negation*
$\mathcal{J}$	::=	\$	*input*
	|	$\mathcal{J}.\mathcal{K}$	*select child by name*
	|	$\mathcal{J}..\mathcal{K}$	*select descendants by name*
	|	$\mathcal{J}[\mathcal{I}]$	*select by index*
	|	$\mathcal{J}[\mathcal{I} : \mathcal{I}]$	*slice by index*
	|	$length(\mathcal{J})$	*length*
	|	$\mathcal{V} + \mathcal{J}$	*numerical addition*
	|	$\mathcal{V} \bullet \mathcal{J}$	*/ string concatenation*
$\mathcal{K}$	::=	$k \in keys$	
$\mathcal{V}$	::=	$v \in values$	
$\mathcal{I}$	::=	$i \in indices$	

Fig. 6. Hidden functions synthesis DSL.

Generating input/output examples. Synthesis rewrite rules replace an expression e in the program with a function call $\phi(\overline{x})$ whose result is bound to a new variable `y`. Before the rewrite, for each trace τ we had some value for e, i.e., $[\![e]\!]_{\sigma,\tau} = v_\tau$. To maintain trace subsumption, the transformation t ensures $[\![\texttt{y}]\!]_{t(\sigma),\tau} = v_\tau$ by mapping the new variable `y` to the appropriate value. A correct implementation for ϕ must satisfy for each trace τ the input-output constraint $\phi([\![\overline{x}]\!]_{\sigma,\tau}) = v_\tau$.

Example 5.4. Recall Example 5.2. Program P_4 is parametric on ϕ, which appears in the statement `let c =` ϕ`(d,x)`, and is correct for a specific implementation of ϕ iff for all traces τ, $\phi([\![d]\!]_{\sigma,\tau}, [\![x]\!]_{\sigma,\tau}) = [\![c]\!]_{\sigma,\tau}$. Since we have two traces we have the constraints $\phi($ `c1`$, o_1) =$ `true` and $\phi($ `c3`$, o_3) =$ `false`.

Synthesizing solutions. The input-output pairs for each hidden function are used to synthesize the expression for that hidden function. To achieve this, we encode the problems of our synthesis domain into an existing example-based program synthesizer. The hidden functions synthesis can be done in any domain, as long as it is supported by the example-based synthesizer. In this section, we illustrate using the domain of our running example: cloud automation scripts. The visible functions are typically APIs that accept parameters and return responses in JSON format [9]. Thus, our domain targets JSON data manipulation scripts, including small predicates for generating conditions. This domain covers the majority of use cases of hidden functions between visible function calls in the automation scripts we observed. JSON is a lightweight, language-agnostic data interchange format widely used in web applications and APIs. The main (recursive) datatypes are lists and dictionaries, which map unique string keys to other JSON objects. The base datatypes are booleans, strings and numbers. Our synthesis domain is summarized in Figure 6, which presents a grammar that includes basic comparison between objects and values, and JSONPath [16] operations. The non-terminal $\mathcal{B}$ in the grammar symbolizes the boolean expressions we consider in our DSL, and $\mathcal{J}$ the JSONPath expressions. Those operations allow the selection of specific indices, members, or descendants of JSON data structures. For example, the path `$.element[0]` selects the `element` field of the object, and then the first element in that list.

To the best of our knowledge there is no synthesizer that targets this domain, despite the ubiquity of JSON to represent data in applications. Solving it requires encoding our problem into a domain supported by a general purpose synthesizer that allows specifications using input-output examples. In Syren, we encode the JSONPath synthesis problem grammar into Rosette [34], a solver-aided programming language with synthesis constructs. Rosette does not support symbolic strings, thus in our encoding, all strings are constant values extracted from the input and output examples. The string values are used for *keys* and *values* in the grammar in Figure 6, and are the result of enumerating all keys in the dictionaries in input-output constraints, and all values, respectively. In some problems, the size of this set of constants becomes a bottleneck because the objects returned by API calls contain hundreds of keys and values. We parallelize the search for a solution by producing sub-grammars for the problem, using different sets of keys and values, and different grammar sizes [8, 15]. Rosette was able to synthesize most Boolean expressions in our benchmarks in the grammar including a subset of JSONPath and string operations, shown in Figure 6.

We considered an alternative approach to using Rosette: encoding the synthesis problem into SMT theories, and using a SyGuS solver supporting those theories. The SyGuS language [29] allows users to specify synthesis problems with input/output pairs as specifications. We encoded JSON data structures and JSONPath operations using a combination of list and user-defined datatypes for dictionaries and lists, and string and integer theories for the base types. We tested this encoding on CVC5 [5] alongside Rosette [34] on our benchmarks, and found that Rosette consistently outperforms CVC5. CVC5 was unable to solve the problems in our JSONPath benchmarks in reasonable time. We also experimented with PBE problems in the domain of arithmetic operations, and CVC5 and the SyGuS encoding outperformed the grammar defined in Rosette. We conclude

```
λ br
if br == 1 {
   let y_1 = A¹₁(x¹₁) let y_2 = A¹₂(x¹₂) ... let y_N1 = A¹_N1(x¹_N1) (** Replays trace τ1 *)
} else if br == 2 {
   let y_1 = A²₁(x²₁) let y_2 = A²₂(x²₂) ... let y_N2 = A²_N2(x²_N2) (** Replays trace τ2 *)
} else if br == 3 {
   let y_1= A³₁(x³₁) let y_2 = A³₂(x³₂) ... let y_N3 = A³_N3(x³_N3) (** Replays trace τ3 *)
} else ...
```

Fig. 8. The initial program P_{in} takes a single integer parameter `br`, and has $|T_{in}|$ branches, where each branch i simply replays the visible function calls in trace $\tau_i \in T_{in}$. A_i^q is the i-th function call in trace #q, and x_i^q its corresponding input.

that the performance of the PBE solver to which Syren offloads the hidden function synthesis depends very highly on the domain. Syren is agnostic to it, and we provide support for using either CVC5 or Rosette, as well as for parallel portfolio solving.

6 Rewrite Strategies

The synthesis algorithm is a rewrite process that starts with an initial program P_{in} that trivially satisfies the correctness criterion Ψ, but is not likely to minimize the cost function χ. The goal is to transform the initial program by applying refinement and synthesis rules until a program minimizing χ is found. Naturally, a naive solution would be to enumerate all possible ways of rewriting P_{in}. However, depending on the program and the set of rewrites available, there may not be a finite set of programs. We consider different strategies to explore the search space of all rewrites efficiently, with the goal of optimizing for χ.

Initial Program. We start with a trivially correct program P_{in} that provably generates all input traces T_{in}. This program is constructed by introducing a single parameter `br` and a program body that consists of $|T_{in}|$ branches. Each branch is guarded by a condition `br` $== i$, with $0 \leq i < |T_{in}|$. The statements in the then-branch are the API calls of trace τ_i, written as API call bindings to fresh local variables. The else-branch contains the other branches. Figure 8 shows the constructed program.

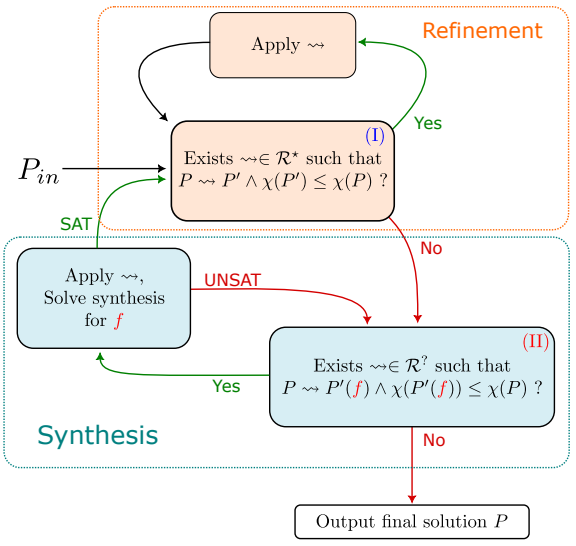


Fig. 7. Cost-directed alternating rewrite rule application.

In Section 5, we distinguish two types of rewrite rules: *refinement rules* $\mathcal{R}^\star$, which simply rewrite the program maintaining trace subsumption, and *synthesis rules* $\mathcal{R}^?$ which introduce a data transform synthesis constraint and are correct by construction of the solution of these constraints. Intuitively, refinement rules use less memory and computation, whereas synthesis rules should be applied more carefully. For scalability, one should use refinement rules as much as possible until applying synthesis rules is necessary.

Alternating Refinement and Synthesis. Figure 7 illustrates our main algorithm, which alternates between refinement rule and synthesis rule applications until no rewrite rule is applicable. We start with the initial program P_{in} and look for refinement rules (in $\mathcal{R}^\star$) to apply in a way that reduces the cost of the program (step (I)). The rule that yields the lowest cost is selected first. If such a rule can be found, we apply it to the current program. We repeat these two steps until no refinement rule can be found. In that case, the algorithm moves inside the bottom loop. It searches for a synthesis rule in $\mathcal{R}^?$ that reduces the most the cost of the program (step (II)). If no rule can be found, the synthesis terminates with the current program. Otherwise, the algorithm attempts to apply the rewrite rule $\exists f \cdot P \rightsquigarrow P'(f)$ and solve for f using the synthesis process described in Section 5.2. There are two possible answers: either a solution is found (SAT) or the synthesizer returned UNSAT. In the first case, the algorithm returns to the upper loop and repeats the entire process. In the other case, the algorithm backtracks on the synthesis rewrite and attempts to find another synthesis rule to apply. When no synthesis rules apply, the algorithm returns the final program.

This algorithm applies synthesis rules parsimoniously compared to refinement rules. A synthesis problem is solved only when no refinement rules can further lower the program cost, and as soon as a synthesis rule is applied, the algorithm attempts to use more refinement rules.

6.1 Baselines

We give a brief overview of baseline algorithms we implemented as a basis for testing our hypothesis, starting with the observation that motivates them.

Refine-then-Synthesize. Experimentally, we observe that, in many cases, it may be sufficient to apply rules in *only three phases*. First, apply all possible refinement rules to simplify the trivial program P_{in}. Then, we apply every possible synthesis rule that reduces the program's cost. Finally, a final round of refinement rewriting is necessary to clean up the program with the new data transformations. The intuition is that refinement rules operate mostly on the control flow of the program, while synthesis rules operate on the data flow, and interaction between the two is minimal.

With that insight in mind, the *refine-then-synthesize* algorithm (denoted RTS) first applies refinement rules, reducing the cost of the program until no refinement rule is applicable, and then synthesis rules until no rule can be found, and finally another round of refinement rules. In other words, it is a modification of the algorithm in Figure 7 where the SAT arrow instead points to step (II) and updates the program, and the No arrow returns to refinement for one round.

k-Bounded Search. One problem of the two previous algorithms is that they *may get stuck on a local minimum of the cost function*. A completely different approach that does not have this problem is a bounded exhaustive search starting from P_{in}. In the k-bounded search algorithm (denoted by k-search), rewrite rules from both the refinement set $\mathcal{R}^\star$ and synthesis set $\mathcal{R}^?$ are applied, independently of their effect on program cost. Rules are applied again to the resulting programs until all programs resulting from applying k rules (where k is a constant) are obtained.

Once all possible rewrites are enumerated, the algorithm ranks all rewrites by increasing cost and attempts to find the program with the lowest cost whose underlying synthesis constraints are satisfiable. Note that in this version of the algorithm, we do not solve the synthesis problem when a synthesis rule is applied. The enumeration is done without a single call to the synthesis solver, which is used only for the programs with low scores.

7 Evaluation

We implemented our synthesis approach in a tool, Syren, and evaluated the different algorithms and cost functions against a set of benchmarks, showing a promising approach for synthesizing real-world API composing functions. Since no existing tool can solve the problem out of the box,

we compare against the baseline algorithms introduced in §6: *refine-then-synthesize* (RTS) and k-bounded search (k-search). Although comparison against a monolithic syntax-guided synthesis approach may be possible (e.g., encoding the problem in Rosette), the limitations in the scalability of Rosette to solve even only the subproblems indicate that it would not scale to the entire problem.

7.1 Implementation

All experiments were run on a 2022 Macbook Pro with an M1Pro processor (10 physical cores) and 32GB memory. Syren is implemented in OCaml and Python 3.12 and uses the Rosette [34] solver-aided language (version 4.1 running on Racket version 8.11) with its default solver Z3 (version 4.12) [12] to synthesize data transformations. The implementation uses a synthesis constraint cache to avoid repeated calls to the solver with the same constraints. This is especially useful since the algorithm will attempt many synthesis rewrites that will not have a solution.

7.2 Benchmarks

We test Syren on a set of 54 benchmarks that implement various tasks that require branching and looping (in the form of retries) using various APIs. The full description of each task is available in Appendix D. We grouped our benchmarks into four different categories to indicate their origin. The first category is a set of *custom benchmarks* that we wrote to perform some tasks on cloud infrastructure, some shell scripts, and SVG manipulation scripts. We then collected tasks from the Blink automation library [7], where various APIs are interfaced. A similar set of tasks comes from AWS Systems Manager Automation Runbooks [3]. Our final category consists of tasks adapted from previous literature; we adapt the nested loop-free benchmarks from ApiPhany [18] that use Stripe and Slack APIs[4]. In general, in the benchmarks used to evaluate Syren, there is a clear separation between the visible function calls (cloud API calls, system calls or library calls) and the local operations (which can be encoded into some solver's theory). When constructing the set of traces, two important parameters have to be considered: whether the different sequences of visible calls exemplify the desired program's control flow paths, and whether the various input values to the calls are sufficient to infer the hidden function's implementation in a given domain (e.g. JSON transformations requires fewer examples than arithmetic).

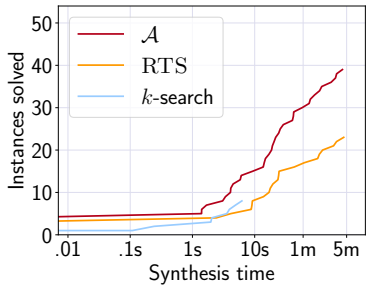
We wrote programs for each benchmark and collected the inputs for the synthesizer by simulating the traces those programs would produce. We ran each program for enough different inputs that the produced traces exercise all program paths; we manually inspected the synthesized program and added more traces when it did not exemplify all the behaviors of the target benchmark program. We collected between 2 and 10 traces (median 4) for each benchmark. This is not the smallest number of traces necessary to describe the task, but a reasonable amount that the user could provide.

Our benchmarks include synthesis tasks of varying complexity so we can gauge how well Syren scales. Although we cannot predict how complex a given task is to synthesize, we can estimate it by the complexity of the smallest program that performs that task. We do so by considering the number of conditionals, loops, and hidden functions in the program.

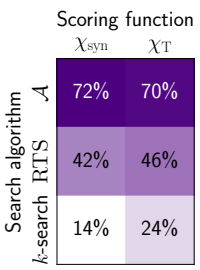
7.3 Cost Functions

We ran experiments with two different cost functions to evaluate the flexibility of our approach with respect to different user-defined notions of "best program." We follow the general idea that good programs are simple programs that generalize well. The rewrite rules, especially the refinement rules, are generally geared towards syntactic simplification of the program. The cost functions

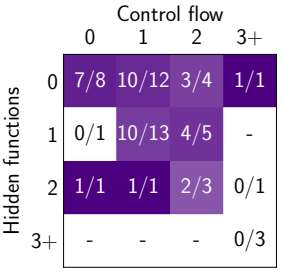
[4]We collect benchmarks from ApiPhany [18] by simulating traces from their solutions that do not have list comprehensions. However, we cannot make a direct comparison since our specifications are traces and theirs are types.



(a) Number of benchmarks solved (optimal solution synthesized) for each algorithm of Syren with score function χ_{syn} against synthesis time.



(b) Percentage of benchmarks for which Syren synthesized the optimal solution by search algorithm and scoring function.



(c) Count of optimally synthesized / total benchmarks, grouped by complexity measures (# hidden functions, # control flow statements) of the program.

Fig. 9. Comparison of synthesis times and quality of synthesized programs using different search algorithms and cost functions. The background color in the heat map in 9b reflects the same data as the labels. In 9c, we show Syren's ability to scale to complex benchmarks. The darker the color the the better Syren preformed for benchmarks of that complexity.

match this high-level goal and generally assign a lower cost to simpler programs; the exact meaning of simplicity depends on the user.

Syntactic Complexity. The first method we use, denoted by χ_{syn}, is a straightforward cost function that describes the syntactic complexity of the program. This intuitively corresponds to a user who desires a program that is syntactically as simple as possible. This function computes a weighted sum of the number of conditionals, loops, and parameters in the program and a penalty for using the dummy branching variable `br` introduced in the initial program. We use a simplified version of the function in the running example of §2. This function can easily be customized, for example, by modifying the weights of each characteristic in the summation. Typically, we prioritize fewer statements, then fewer parameters, and set the weights accordingly. The penalty for the `br` variable ensures that the algorithm will prioritize eliminating this parameter over all else.

Reuse Across Traces. Our second cost function, denoted by χ_{T}, measures how many times API call statements are reused with respect to the input set of traces. For each trace in T_{in}, we count how many times each API call statement in the program must be called to produce the trace. The cost is the total number of API calls in the traces plus the number of statements minus the sum of the counts for all statements and all traces. Intuitively, a program with a lower cost means that the API call statements are reused more often; for example, unrolling a loop would increase the cost. We also add a penalty for using the `br` variable. This is another measure of the program's simplicity. This cost function is coarser in that it assigns the same cost to many different programs.

7.4 Results

The goal is to synthesize a program that is equivalent to the one used to collect traces from; when we report synthesis success, this means the synthesized program is syntactically equivalent to the desired program (modulo variable renaming). Since Syren may terminate with a correct solution that is not optimal (i.e. has a larger cost than our desired program), we also report when the tool terminated but did not solve the benchmark.

Figure 9 plots the synthesis time required for each benchmark and combination of cost function (χ_{syn} or χ_{T}) and algorithms ($\mathcal{A}$ for our algorithm, RTS and k-bounded for the baselines). Each experiment runs 10 times with a timeout of 10 minutes. We use a fixed $k = 6$.

Our proposed algorithm $\mathcal{A}$ finds the most solutions across benchmarks and cost functions (39 or 72% optimally synthesized for syntactic cost, and 38 or 70% for trace reuse cost out of 54 benchmarks). The simplified algorithm, RTS, produces fewer optimal solutions, 23 (42%) for χ_{syn} and 25 (46%) for χ_T); in many cases the solution produced is not optimal because the algorithm did not attempt enough rewrites. This is especially the case for the more complex benchmarks containing loops. We found that the choice of scoring function has little impact on solving time. However, χ_T is more coarse in the sense that more programs may have the same score, and when manually inspecting solutions, we found that they are usually farther from the ideal solution. χ_{syn} is better at characterizing the ideal solution. The solutions obtained required from 4 to 36 rewrites. The tables in the Appendix A list detailed synthesis times for all benchmarks.

The bounded search performs poorly across the benchmarks (13 optimally solved for χ_T), either yielding a poor solution (because of a small k) or timing out. The size of the search space for the bounded search is a combination of both the number of rewrite rules and the complexity of the initial program. In general, we observed that scaling k to the same number used to find a solution using the other algorithms would produce an intractable search space. Interestingly, we observe many timeouts for the χ_T cost function; there are many programs with the same cost, but most of them have unrealizable synthesis subproblems.

Even when the best synthesizable solution does not require a data transformation, the rewrite process might still attempt to find some. Synthesis time is dominated by the number of synthesis rewrites attempted rather than the number that ends up being used, so time often does not correspond to the total number of rewrites. For example, synthesizing a solution for the CreateTable benchmark takes 78s, despite not requiring any data transformation. However, since there are two API calls with many arguments, the algorithm must ensure, by attempting to synthesize data transformations, that none of the arguments of the second API call can be computed from the results of the first API call.

Comparison against Large Language Models. Large Language Models (LLMs) allow users to generate code from natural language specifications. One can speculate whether specifications could also be given as a list of traces, as in our problem. To compare Syren against LLMs, we queried Claude 3.5 Sonnet [2] with a prompt explaining the problem to solve, followed by the same traces used by Syren, in JSON format (see the full prompt in Appendix C). The output is expected to be a Python script that should be correct (in the sense of Equation 1) and semantically close to the ideal program. With that success criterion in mind, the LLM synthesizes a correct and optimal solution for 29 benchmarks (53% of total).

Note that the evaluation of correctness and equivalence is manual work on our part. This highlights a crucial difference between the two methods. When Syren succeeds and generates a program, the user trusts it is correct, in the sense that it will reproduce the traces. If Syren cannot generate a good program, it will fail or otherwise generate a program that is not general enough, but the output is always safe to use. Specifically, it will never make API calls different from those in the traces, or calls with new inputs. On the other hand, LLMs can fail to synthesize a correct program silently; their output can be subtly incorrect, and verifying this output would require effort (that Syren avoids by generating a program correct by construction). For example, in the task described in §2, Claude synthesizes the condition `status == "running"` instead of `status != "stopped"`, which does not satisfy traces where the instance status is `"stopping"`.

Limitations. Our evaluation compares Syren's default search strategy to our own baselines. To the best of our knowledge, there is no other tool that currently solves this problem, in which the specification is a set of *partial* traces.

Our diverse range of benchmarks includes a few examples where Syren times out or does not return a good solution. Some of those are due to the limited expressive power of the syntax-guided synthesis approach used to create the data transformations. For example, the benchmark ReportLongRunningInstancesToSlack requires reasoning about dates and time intervals, and our solver cannot currently synthesize a solution. The objective program cannot be represented in our synthesis DSL, so we cannot indicate the number of ifs or loops such a program would have. Similarly, the CreateImage benchmark requires reasoning about string operations. Improving the underlying PBE solver to handle more and more complex date, integer, and string operations is an interesting and promising direction of future research. Such improvements would result in an improvement of Syren's performance without further modification. The rewrite system also has limitations when considering benchmarks with complex control flow. For example, our tool fails to synthesize the automation AWSSupport-CopyEC2Instance, which has a loop, 5 conditionals, and 16 data transformations to synthesize. Our approach also cannot synthesize any control flow that happens before the first API call in the automation to be synthesized. For example, the benchmark AWS-ConfigureS3BucketVersioning contains empty traces that correspond to situations where the user ran the script, but it terminated before performing any API call. Our approach is unable to infer this behavior, since it cannot observe what the potential inputs to the decision are.

Our current representation of loops limits the programs we can synthesize. We are unable to synthesize programs that loop a fixed number of times and then time out. If Syren is given the traces that result from this pattern, it will instead attempt to discover a condition that is consistent across all the final API calls, which will not exist in general. We omitted the benchmarks from ApiPhany[18] that could not be generated because either they contain nested loops or the loop iterates on parameter of the script. We believe those limitations could be lifted in future work, allowing Syren to synthesize programs with more complex structures.

8 Related Work

In this section, we provide a more in-depth overview of previous work related to ours. Although we found strong work on problems related to the one Syren solves, there is no previous solution to solve the same problem. We compare our work against techniques that target similar outputs, i.e. programs with API calls. Then, we look at work that consider similar trace-based specifications, which we can mainly classify in programming-by-demonstration. Finally, we look at work similar in their approach: rewriting techniques and syntax-guided synthesis.

Synthesis with API calls. SyPet [14], TYGAR [19], RbSyn [20] and ApiPhany [18] propose type-guided approaches to synthesis of programs containing sequences of API calls. All three address component-based synthesis, which focuses on finding a composition of components (API calls or library functions) that implements some desired task. These systems differ from Syren in more than one aspect. First, SyPet, TYGAR, RbSyn, and ApiPhany take as a specification the desired input and output types of a program. Syren, on the other hand, synthesizes a program using logs of the desired task executed manually. The contrasts between their approach and ours go beyond the input specification: in their approach, the main challenge SyPet tackles is the large search space of API calls that it has to consider, which results from the ever-increasing expressivity and size of API libraries. TYGAR and ApiPhany propose additional techniques to better represent and understand API calls, thus effectively reducing this search space. The challenge in our problem is not to discover which API calls need to be made because they are present in the traces, and thus, the API size does not impact our problem. However, we consider more complex interactions between API calls and synthesize data operations that correspond to the non-observable part of the traces. Conversely, due to the nature of our approach, where we synthesize scripts based on

logs, we already know which API calls are made, so the complexity of our synthesis procedure is not affected by the size of the underlying APIs.

DemoMatch [37] discovers code snippets explaining API usage, but in the authors' own words "While DemoMatch produces code, it is not a program synthesis tool; it is an API discovery tool."

Programming-by-demonstration (PBD). There are examples of work in PBD that target the automation of web tasks, one of the domains to which we applied Syren, but there are significant differences in the setting and method that justify our claim of novelty. Ringer [6] explores ways to represent one recording of a user interacting with a web page and outputs a script that reproduces a single execution of the demonstrated behavior. Our work attempts to generalize multiple executions into a program from logs; the generalization over data required in our setting is not a problem for Ringer. Approaches such as Konure [32], DemoMatch [37], PUMICE [24], and Sketch-n-Sketch [10] are distinct from ours because we synthesize programs only from the logs of the API calls they make, *without* examples of the local operations of the program to be synthesized. None of the works cited above attempts to synthesize hidden functions, especially with conditions that depend on the outputs of visible function calls. Furthermore, some of the approaches rely on interactive demonstrations to generalize their data (WebRobot [13]) or queries (Konure [32]), or on an existing program to extract dynamic traces from (Chisel [26]). Instead, we rely only on a fixed set of examples. Inferring conditionals and loops when a user or algorithm can test the program paths through demonstrations is a different task from inferring them only by optimizing for a cost function. Like Syren, WebRobot [13] and Arborist [25] use a rewriting strategy to synthesize programs, with speculative rewrites to generalize patterns. However, their method is interactive and relies on the user to validate the heuristic speculations made by the algorithm. In contrast, we rely on data transformation synthesis to validate the applicability of a synthesis rewrite rule (for conditions and loops) and on the cost function to direct the application of rewrite rules. Neither attempts to synthesize decision-making control flow, i.e., if-statements, or loop conditionals (they synthesize for-each loops and while(true) loops, whereas we focus on loops that are stopped by a condition). The synthesis and generalization of these structures is the main challenge in our approach. Our work differentiates itself from Konure [32] in two significant ways. First, Konure is an active learning system that requires querying of the system that needs modeling. Our system is completely passive (closer to PBE than to PBD) and relies only on a fixed set of examples. Second, Konure has full observability over what it generates. For example, the predicates in their DSL are SQL queries, which are observable in the trace. The authors mention that to support more sophisticated implementations in their DSL they suppose that they would need to have a more fine-grained observation of the application. Instead, we propose that fine-grained data operations can be synthesized using a synthesis solver for small expressions. Our paper shows that this difficulty requires carefully exploring the search space of rewrite rules.

Program Rewriting. There is a rich history of work in applying rewriting strategies for program analysis, refactoring, or optimization [27, 35]. Superoptimization [22, 23, 31] problems, where programs must be rewritten to optimize for a given cost, are classic applications of such techniques. Our application is different in that many of our rewrites are whole-program rewrites that need to consider the state of the program, as opposed to small local rewrites. Note that our approach would not scale when applying many rewrites in an exploratory search: we also need to solve the underlying syntax-guided synthesis problems, which are much more computationally expensive than syntactic rewrite rules. To the best of our knowledge, this work is the first to introduce a system where the application of a rewrite rule is conditional on solving an input-/output-based synthesis problem. Rewrite approaches have also been used to solve other synthesis problems, such as automatic parallelization in Mold [30]. Although they also consider a two-phase approach with

refinement and exploration, they use solely fixed rewrite rules instead of our synthesis rules, and the exploratory phase does not have the same expressive power as our synthesis-based solution. Szalinski [27] describes an approach that takes a flat, hard-to-read program and introduces map and fold operators using optimizing rewrites. Although its goal is similar to the rewrite part of our approach, Szalinski has no way to synthesize hidden logic in control flow.

Off-the-shelf SyGuS solver. The problem Syren tackles could, in theory, be encoded into Syntax-Guided Synthesis (SyGuS) [1] and solved using an off-the-shelf solver, such as CVC5 [5]. We attempted to implement this approach and ran into two problems. First, it is challenging to encode side-effecting functions as uninterpreted functions because the output is not a direct function of the inputs alone. Their output depends on some global state, updated by other functions in the program. More complex encodings can be designed, but they would be a novel contribution on their own, in particular, if they could be made to perform well. With any straightforward encoding, the resulting formula would be too complex to be solved in a reasonable time by CVC5. As explained in §5.2, we were unable to use CVC5 for even the JSON synthesis subproblems.

9 Conclusion and Future Work

In this paper we described a novel approach for the synthesis of scripts with calls to side-effecting methods, such as web API method calls, from partial execution traces containing only those calls. The described approach combines a search through a space of rewrite rules that mutate the program without jeopardizing its correctness, and syntax-guided synthesis, which allows us to fill in the expressions that we cannot observed in the traces. We implement our approach in Syren, and test it on 54 benchmarks that combine hand-written examples based on common visual interface tasks, scripts from publicly available libraries, and benchmarks from previous work. We show that our approach can successfully synthesize 39/54 scripts in under 5 minutes.

The main bottleneck in Syren's results is in the data-transformations synthesis, so performance improvements should focus on syntax-based synthesis within the domain of *JSONPath*. Another improvement would be writing more specific grammars for the data transforms synthesis, by leveraging additional information about the types of the inputs and outputs, or even their semantic types, as suggested by ApiPhany [18]. Another interesting orthogonal line of work would be to allow traces which show errors on the calls, and synthesize scripts with error handling. Syren could also be extended to cover more looping behavior, particularly timeouts.

We believe the problem Syren solves is an important and very general problem, yet we had to construct a new set of benchmarks to evaluate our approach. In related work, the benchmarks included either types, synthesized programs or complete traces, where a clear separation between API-level and local operations was not clear. Our benchmark set could be extended with additional use cases where that separation is clear, and the collection of traces is plausible because the system collecting the traces is external to the hidden program. For example, one can collect traces of library calls, system calls or various cloud API calls. The quantity and quality required to synthesize a given program will depend on the control-flow complexity of that program, and the ability of an underlying hidden function synthesizer to synthesize data transformations from few examples.

Acknowledgments

We are grateful to Ruben Martins and Ricardo Brancas for their feedback on multiple drafts of this paper. We also thank Ricardo for his very insightful suggestions that improved Syren's evaluation. Finally, we thank the anonymous reviewers and shepherd for their comments and guidance. This work was partially done during an internship at Amazon. Margarida Ferreira is supported by

the Portuguese Foundation for Science and Technology under the Carnegie Mellon Portugal PhD fellowship SFRH/BD/151467/2021.

Data-Availability Statement

The source code for Syren is available on Zenodo [28], along with all the data we used to test it. The artifact contains comprehensive instructions and scripts to reproduce the experiments reported in this paper.

References

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD*. IEEE, Portland, OR, USA, 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[2] Anthropic. 2024. Introducing Claude 3.5 Sonnnet. https://www.anthropic.com/news/claude-3-5-sonnet.

[3] AWS. 2023. AWS Automation Runbooks Reference. https://docs.aws.amazon.com/systems-manager-automation-runbooks/latest/userguide/automation-runbook-reference.html.

[4] AWS. 2023. What is AWS? https://aws.amazon.com/what-is-aws/.

[5] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *TACAS*. Springer, Munich, Germany, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[6] Shaon Barman, Sarah E. Chasins, Rastislav Bodík, and Sumit Gulwani. 2016. Ringer: web automation by demonstration. In *OOPSLA*. ACM, Amsterdam, The Netherlands, 748–764. https://doi.org/10.1145/2983990.2984020

[7] Blink. 2023. Blink | The Security Automation Copilot. https://www.blinkops.com/.

[8] Ricardo Brancas, Miguel Terra-Neves, Miguel Ventura, Vasco Manquinho, and Ruben Martins. 2024. Towards Reliable SQL Synthesis: Fuzzing-Based Evaluation and Disambiguation. In *FASE*. Springer, Luxembourg City, Luxembourg, 232–254. https://doi.org/10.1007/978-3-031-57259-3_11

[9] Tim Bray. 2017. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259. https://doi.org/10.17487/RFC8259

[10] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. In *PLDI*. ACM, Santa Barbara, CA, USA, 341–354. https://doi.org/10.1145/2908080.2908103

[11] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky (Eds.). 1993. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA.

[12] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science, Vol. 4963)*. Springer, Budapest, Hungary, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[13] Rui Dong, Zhicheng Huang, Ian Iong Lam, Yan Chen, and Xinyu Wang. 2022. WebRobot: web robotic process automation using interactive programming-by-demonstration. In *PLDI*. ACM, San Diego, CA, USA, 152–167. https://doi.org/10.1145/3519939.3523711

[14] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *POPL*. ACM, Paris, France, 599–612. https://doi.org/10.1145/3009837.3009851

[15] Margarida Ferreira, Ranysha Ware, Yash Kothari, Inês Lynce, Ruben Martins, Akshay Narayan, and Justine Sherry. 2024. Reverse-Engineering Congestion Control Algorithm Behavior. In *IMC*. ACM, Madrid, Spain, 401–414. https://doi.org/10.1145/3646547.3688443

[16] Jeff Friesen. 2019. *Extracting JSON Values with JsonPath: Document Processing for Java SE*. Apress Berkeley, CA, Berkeley, CA, USA, 299–322. https://doi.org/10.1007/978-1-4842-4330-5_10

[17] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Found. Trends Program. Lang.* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[18] Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. 2022. Type-directed program synthesis for RESTful APIs. In *PLDI*. ACM, San Diego, CA, USA, 122–136. https://doi.org/10.1145/3519939.3523450

[19] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2020. Program synthesis by type-guided abstraction refinement. In *POPL*. ACM, New Orleans, LA, United States, 12:1–12:28. https://doi.org/10.1145/3371080

[20] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2021. RbSyn: Type- and Effect-Guided Program Synthesis. In *PLDI*. ACM, Virtual, Canada, 344–358. https://doi.org/10.1145/3453483.3454048

[21] Natasha Yogananda Jeppu, Thomas F. Melham, Daniel Kroening, and John O'Leary. 2020. Learning Concise Models from Long Execution Traces. In *DAC*. IEEE, San Francisco, CA, USA, 1–6. https://doi.org/10.1109/DAC18072.2020.9218613

[22] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-Directed Superoptimizer. In *PLDI* (Berlin, Germany). ACM, Beijing, China, 304–314. https://doi.org/10.1145/512529.512566

[23] Joel Kuepper, Andres Erbsen, Jason Gross, Owen Conoly, Chuyue Sun, Samuel Tian, David Wu, Adam Chlipala, Chitchanok Chuengsatiansup, Daniel Genkin, Markus Wagner, and Yuval Yarom. 2023. CryptOpt: Verified Compilation with Randomized Program Search for Cryptographic Primitives. In *PLDI*. ACM, Orlando, FL, United States, 1268–1292. https://doi.org/10.1145/3591272

[24] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and Brad A. Myers. 2019. PUMICE: A Multi-Modal Agent that Learns Concepts and Conditionals from Natural Language and Demonstrations. In *UIST*. ACM, New Orleans, LA, USA, 577–589. https://doi.org/10.1145/3332165.3347899

[25] Xiang Li, Xiangyu Zhou, Rui Dong, Yihong Zhang, and Xinyu Wang. 2024. Efficient Bottom-Up Synthesis for Programs with Local Variables. In *POPL*. ACM, London, UK, 1540–1568. https://doi.org/10.1145/3632894

[26] Benjamin Mariano, Ziteng Wang, Shankara Pailoor, Christian Collberg, and Işil Dillig. 2024. Control-Flow Deobfuscation using Trace-Informed Compositional Program Synthesis. In *OOPSLA*. ACM, Pasadena, CA, USA, 2211–2241. https://doi.org/10.1145/3689789

[27] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *PLDI*. ACM, London, UK, 31–44. https://doi.org/10.1145/3385412.3386012

[28] Victor Nicolet and Margarida Ferreira. 2025. *Program Synthesis From Partial Traces (Software Artifact)*. https://doi.org/10.5281/zenodo.15047359

[29] Saswat Padhi, Elizabeth Polgreen, Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. 2023. The SyGuS Language Standard Version 2.1. https://doi.org/10.48550/arXiv.2312.06001 arXiv:2312.06001 [cs.PL]

[30] Cosmin Radoi, Stephen J. Fink, Rodric M. Rabbah, and Manu Sridharan. 2014. Translating imperative code to MapReduce. In *OOPSLA*. ACM, Portland, OR, USA, 909–927. https://doi.org/10.1145/2714064.2660228

[31] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *ASPLOS*. ACM, Houston, Texas, USA, 305–316. https://doi.org/10.1145/2451116.2451150

[32] Jiasi Shen and Martin C. Rinard. 2019. Using active learning to synthesize models of applications that access databases. In *PLDI*. ACM, Phoenix, AZ, USA, 269–285. https://doi.org/10.1145/3314221.3314591

[33] Richard Shin, Illia Polosukhin, and Dawn Song. 2018. Improving Neural Program Synthesis with Inferred Execution Traces. In *NeurIPS*. Curran Associates, Inc., Montréal, Canada, 8931–8940. https://proceedings.neurips.cc/paper/2018/hash/7776e88b0c189539098176589250bcba-Abstract.html

[34] Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*. ACM, Edinburgh, UK, 530–541. https://doi.org/10.1145/2594291.2594340

[35] Eelco Visser. 2001. A Survey of Rewriting Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science* 57 (2001), 109–143. https://doi.org/10.1016/S1571-0661(04)00270-1

[36] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. 2021. Synthesizing safe and efficient kernel extensions for packet processing. In *SIGCOMM*. ACM, Virtal (online), 50–64. https://doi.org/10.1145/3452296.3472929

[37] Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. 2017. DemoMatch: API discovery from demonstrations. In *PLDI*. ACM, Barcelona, Spain, 64–78. https://doi.org/10.1145/3062341.3062386

A Benchmarks and Detailed Results

Table 1. List of benchmarks, measures of complexity, and outcome of synthesis using Syren ($\chi_{\text{syn}}$and $\mathcal{A}$) or an LLM. #Traces indicates the number of traces used to generate the program. #If (#Loop, #Hidden-f) report the number of conditionals (resp. loops, hidden functions) in the best synthesizable solution. The outcome columns report the synthesis outcome for either Syren (Timeout, Terminated or Optimal) or the LLM (Success or Failure).

Name	#Traces	#If	#Loop	#Hidden-f	Syren outcome	LLM outcome
Custom Benchmarks						
Start Instances	7	0	0	0	Optimal	Success
Stop Instances	9	0	0	0	Optimal	Success
Create Image	4	0	0	1	Timeout	Failure
Create Image From Ssm Parameter And Log	3	0	0	4	Optimal	Success
Create Table	7	0	0	0	Optimal	Failure
Delete Table	7	0	0	0	Optimal	Failure
Create Bucket Then Folder	3	0	0	1	Optimal	Success
Put Object If Not Present	6	1	0	1	Optimal	Failure
Stop Instances Cond	6	1	0	1	Optimal	Failure
Create Table Insert Item	8	0	1	1	Timeout	Success
Backup Then Delete Table	6	0	1	2	Optimal	Success
Start Instances With Tags	7	0	0	1	Optimal	Success
Stop All Running Instances	4	0	0	1	Optimal	Success
Move Ddb Item	7	0	0	1	Optimal	Failure
Move Ddb Item If Present	8	1	0	2	Optimal	Success
Copy S3Objects	3	1	1	2	Optimal	Failure
Tag Instances With Dry Run	4	1	0	1	Optimal	Failure
Send Email On Input	3	0	1	1	Optimal	Success
Retrieve Channel Members	3	0	1	2	Optimal	Failure
List and move files	3	1	1	0	Optimal	Failure
Clean current dir	2	1	1	1	Optimal	Success
Clean regular files in dir	2	1	1	2	Optimal	Success
Conversation members	4	1	1	3	Timeout	Failure
SVG-Increase Circle Radius-json	3	0	0	2	Terminated	Failure
SVG-Increase Circle Radius	3	0	0	1	Optimal	Success
Blink Automation						
Copy Instance To New Region	4	0	1	2	Timeout	Success
Report Long Running Instances	8	??	??	-1	Timeout	Failure
Create Iam User And Notify	3	0	0	1	Optimal	Failure
AWS Automation Runbooks						
Stop EC2Instance	5	0	0	0	Optimal	Success
Start EC2Instance	5	1	0	2	Optimal	Failure
Configure S3Bucket Versioning	9	0	0	1	Terminated	Success
Configure Cloud Watch On EC2	8	1	0	0	Terminated	Failure
Copy EC2Instance	3	5	1	16	Timeout	Failure
Resize Instance	8	3	2	3	Timeout	Failure
Set Required Tags	4	0	0	0	Optimal	Success
Literature benchmarks						
Api Phany Ex. 1.1	4	1	2	5	Timeout	Success
Api Phany Ex. 1.2	3	0	0	2	Optimal	Failure
Api Phany Ex. 1.6	3	0	0	1	Optimal	Success
Api Phany Ex. 1.8	3	0	0	1	Optimal	Success
Api Phany Ex. 2.1	3	0	1	1	Optimal	Success
Api Phany Ex. 2.3	3	0	0	2	Optimal	Success
Api Phany Ex. 2.5	3	0	1	1	Optimal	Success
Api Phany Ex. 2.6	3	0	0	1	Optimal	Failure
Api Phany Ex. 2.7	4	0	1	1	Optimal	Success
Api Phany Ex. 2.10	2	0	1	1	Timeout	Success
Api Phany Ex. 2.11	3	0	0	1	Optimal	Failure
Api Phany Ex. 2.13	3	0	0	2	Optimal	Success
Api Phany Ex. 3.1	2	0	0	0	Optimal	Failure
Api Phany Ex. 3.3	3	0	0	0	Terminated	Failure
Api Phany Ex. 3.4*	2	0	1	1	Optimal	Failure
Api Phany Ex. 3.6	3	0	1	1	Optimal	Success
Api Phany Ex. 3.7	2	0	1	1	Optimal	Success
Api Phany Ex. 3.8	2	0	2	2	Timeout	Success
Api Phany Ex. 3.9	3	0	1	1	Timeout	Failure

Table 2. List of Custom benchmarks, with synthesis time and number of SyGuS solver calls. For each algorithm ($\mathcal{A}$, RTS and k-search) we report the synthesis time in seconds, the number of total SyGuS solver calls, and the subset of those calls that returned SAT, in parenthesis. When Syren times out (TO), it does not report the number of solver calls (N/A).

Custom Benchmarks												
	χ_{syn}						χ_T					
	$\mathcal{A}$		RTS		k-search		$\mathcal{A}$		RTS		k-search	
Name	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)
Start Instances	<0.01	0 (0)	<0.01	0 (0)	TO	N/A	<0.01	0 (0)	<0.01	0 (0)	TO	N/A
Stop Instances	<0.01	0 (0)	<0.01	0 (0)	TO	N/A	<0.01	0 (0)	<0.01	0 (0)	TO	N/A
Create Image	TO	N/A	TO	N/A	0.16	0 (0)	TO	N/A	TO	N/A	TO	N/A
Create Image From Ssm Parameter And Log	207.02	8 (4)	206.53	8 (4)	201.87	0 (0)	210.21	8 (4)	205.58	8 (4)	TO	N/A
Create Table	78.33	4 (0)	123.40	7 (0)	0.44	0 (0)	77.97	4 (0)	123.41	7 (0)	TO	N/A
Delete Table	<0.01	0 (0)	<0.01	0 (0)	TO	N/A	<0.01	0 (0)	<0.01	0 (0)	TO	N/A
Create Bucket Then Folder	25.76	3 (2)	4.14	2 (3)	0.11	6 (61)	25.85	3 (2)	4.13	2 (3)	0.06	1 (1)
Put Object If Not Present	256.88	5 (1)	269.27	8 (1)	0.74	0 (0)	13.65	3 (1)	268.51	8 (1)	175.23	10 (1)
Stop Instances Cond	57.80	7 (7)	51.84	5 (6)	0.37	0 (0)	50.71	3 (3)	54.64	5 (7)	104.34	7 (2)
Create Table Insert Item	TO	N/A	TO	N/A	310.20	6 (3)	TO	N/A	TO	N/A	310.29	6 (1)
Backup Then Delete Table	198.66	11 (4)	200.20	9 (4)	2.08	0 (0)	196.25	8 (3)	TO	N/A	2.08	0 (0)
Start Instances With Tags	14.22	2 (1)	17.83	2 (1)	0.35	1 (1)	19.32	2 (1)	20.56	2 (1)	0.43	1 (0)
Stop All Running Instances	24.30	1 (1)	24.24	1 (1)	TO	N/A	16.29	1 (1)	21.72	1 (1)	TO	N/A
Move Ddb Item	119.49	6 (1)	109.31	7 (1)	0.38	0 (0)	89.38	6 (1)	108.40	7 (1)	TO	N/A
Move Ddb Item If Present	91.50	7 (2)	102.89	8 (2)	2.18	0 (0)	32.67	3 (1)	102.27	8 (3)	TO	N/A
Copy S3Objects	110.38	4 (2)	TO	N/A	3.32	2 (2)	110.17	4 (2)	TO	N/A	3.18	2 (2)
Tag Instances With Dry Run	74.95	9 (3)	62.35	8 (3)	0.10	0 (0)	28.36	3 (1)	62.51	8 (4)	61.31	12 (263)
Send Email On Input	5.86	3 (2)	8.88	4 (2)	10.14	1 (1)	5.75	3 (2)	8.83	4 (2)	10.14	1 (1)
Retrieve Channel Members	6.08	2 (2)	0.67	7 (4)	TO	N/A	6.02	2 (2)	TO	N/A	TO	N/A
List and move files	14.89	9 (3)	199.48	11 (3)	TO	N/A	14.92	10 (3)	199.76	11 (3)	6.49	0 (0)
Clean current dir	1.68	1 (1)	TO	N/A	TO	N/A	1.70	1 (1)	TO	N/A	TO	N/A
Clean regular files in dir	9.27	2 (2)	6.53	3 (3)	TO	N/A	11.32	2 (2)	8.60	3 (3)	TO	N/A
Conversation members	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A
SVG-Increase Circle Radius-json	165.69	6 (4)	207.81	6 (4)	0.09	0 (0)	165.94	6 (4)	207.65	6 (4)	8.33	3 (4)
SVG-Increase Circle Radius	42.38	4 (3)	2.41	4 (4)	0.09	1 (1)	42.44	4 (3)	2.40	4 (4)	0.09	14 (8)

Table 3. List of Blink Automation benchmarks, with synthesis time and number of SyGuS solver calls. For each algorithm ($\mathcal{A}$, RTS and k-search) we report the synthesis time in seconds, the number of total SyGuS solver calls, and the subset of those calls that returned SAT, in parenthesis. When Syren times out (TO), it does not report the number of solver calls (N/A).

Blink Automation												
	χ_{syn}						χ_T					
	$\mathcal{A}$		RTS		k-search		$\mathcal{A}$		RTS		k-search	
Name	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)
Copy Instance To New Region	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A
Report Long Running Instances	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A
Create Iam User And Notify	41.83	2 (1)	43.17	2 (1)	0.05	2 (1)	40.75	2 (1)	41.26	2 (1)	0.05	2 (2)

Table 4. List of AWS Automation Runbooks benchmarks, with synthesis time and number of SyGuS solver calls. For each algorithm ($\mathcal{A}$, RTS and k-search) we report the synthesis time in seconds, the number of total SyGuS solver calls, and the subset of those calls that returned SAT, in parenthesis. When Syren times out (TO), it does not report the number of solver calls (N/A).

AWS Automation Runbooks												
	χ_{syn}						χ_T					
	$\mathcal{A}$		RTS		k-search		$\mathcal{A}$		RTS		k-search	
Name	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)
Stop EC2Instance	4.51	0 (1)	9.03	0 (1)	0.24	0 (1)	4.55	0 (1)	9.05	0 (1)	0.18	0 (1)
Start EC2Instance	19.26	4 (3)	13.83	5 (2)	0.30	1 (1)	11.50	2 (1)	15.33	4 (2)	351.02	14 (58)
Configure S3Bucket Versioning	71.48	3 (0)	80.84	3 (0)	77.89	3 (0)	50.56	1 (0)	77.11	1 (0)	56.80	1 (0)
Configure Cloud Watch On EC2	13.72	2 (0)	13.72	2 (0)	TO	N/A	8.77	1 (0)	13.73	2 (0)	TO	N/A
Copy EC2Instance	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A
Resize Instance	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A
Set Required Tags	16.93	2 (0)	16.97	2 (0)	4.91	13 (43)	16.97	2 (0)	16.97	2 (0)	243.93	3 (0)

Table 5. List of Literature benchmarks, with synthesis time and number of SyGuS solver calls. For each algorithm ($\mathcal{A}$, RTS and k-search) we report the synthesis time in seconds, the number of total SyGuS solver calls, and the subset of those calls that returned SAT, in parenthesis. When Syren times out (TO), it does not report the number of solver calls (N/A).

Literature benchmarks												
	χ_{syn}						χ_T					
	$\mathcal{A}$		RTS		k-search		$\mathcal{A}$		RTS		k-search	
Name	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)	Syren runtime	SyGuS calls (sat)
Api Phany Ex. 1.1	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A
Api Phany Ex. 1.2	18.64	4 (2)	19.64	4 (2)	0.07	0 (0)	17.72	4 (2)	19.43	4 (2)	84.93	1 (1)
Api Phany Ex. 1.6	28.84	6 (2)	24.64	6 (3)	0.09	1 (1)	27.49	6 (2)	24.45	6 (3)	174.70	1 (1)
Api Phany Ex. 1.8	21.74	4 (3)	9.18	3 (3)	0.16	1 (1)	23.76	4 (3)	11.60	3 (3)	0.18	1 (1)
Api Phany Ex. 2.1	1.41	1 (1)	TO	N/A	2.04	2 (2)	1.42	1 (1)	TO	N/A	1.68	2 (2)
Api Phany Ex. 2.3	168.98	12 (2)	184.82	15 (2)	0.04	0 (0)	161.13	12 (2)	184.17	15 (2)	161.41	10 (1)
Api Phany Ex. 2.5	1.40	1 (1)	TO	N/A	6.23	2 (2)	1.33	1 (1)	TO	N/A	6.07	2 (2)
Api Phany Ex. 2.6	20.50	4 (2)	23.62	4 (2)	0.14	0 (0)	30.28	4 (2)	31.88	4 (2)	TO	N/A
Api Phany Ex. 2.7	4.13	1 (1)	TO	N/A	TO	N/A	4.38	1 (1)	TO	N/A	TO	N/A
Api Phany Ex. 2.10	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A
Api Phany Ex. 2.11	13.72	4 (3)	8.79	3 (3)	0.11	1 (1)	13.90	4 (3)	8.21	3 (3)	0.12	1 (1)
Api Phany Ex. 2.13	40.39	9 (3)	31.81	7 (2)	0.08	0 (0)	40.53	9 (3)	31.15	7 (2)	28.94	0 (0)
Api Phany Ex. 3.1	<0.01	0 (0)	TO	N/A	1.97	1 (1)	<0.01	0 (0)	TO	N/A	1.61	1 (1)
Api Phany Ex. 3.3	162.15	3 (1)	165.80	2 (1)	181.45	2 (0)	162.83	3 (1)	164.49	2 (1)	183.63	3 (0)
Api Phany Ex. 3.4*	4.08	1 (1)	TO	N/A	3.92	1 (1)	3.86	1 (1)	TO	N/A	4.03	5 (6)
Api Phany Ex. 3.6	3.39	1 (1)	TO	N/A	2.85	3 (3)	3.57	1 (1)	TO	N/A	3.02	3 (3)
Api Phany Ex. 3.7	3.06	1 (1)	106.23	2 (1)	3.57	1 (1)	3.07	1 (1)	105.35	2 (1)	TO	N/A
Api Phany Ex. 3.8	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A
Api Phany Ex. 3.9	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A	TO	N/A

B Rewrite rules

B.1 Refinement Rewrite Rules

Recall our two sets of rewrite rules: refinement rewrite rules and synthesis rewrite rules . We ignore the data transformation definitions associated with the program, since they do not intervene in the rewrite.

Control Flow Manipulation. The following rewrite rule pulls calls to the same API out of a conditional:

$$\begin{aligned}\lambda\overline{z}\ \mathcal{S}\ \texttt{if}\ c\ \{\texttt{let}\ x = \mathbb{A}(\overline{y})\ \mathcal{S}'\}\ \texttt{else}\ \{\texttt{let}\ x' = \mathbb{A}(\overline{y}')\ \mathcal{S}''\}\ \mathcal{R}\\ \rightsquigarrow_t \lambda\overline{z}\ \mathcal{S}\ \texttt{let}\ x = \mathbb{A}(c?\,\overline{y}\ :\ \overline{y}')\ \texttt{if}\ c\ \{\mathcal{S}'\}\ \texttt{else}\ \{\mathcal{S}''\ [x' \to x]\}\ \ \mathcal{R}\ [x' \to x]\\ \text{with } t(\sigma) = \sigma[x \to c?\,\sigma(x)\ :\ \sigma(x')] \quad (3)\end{aligned}$$

Note that in this transformation, the condition is used both in transforming the arguments of the API call moved out of the branches (in $c?\,\overline{y}\ :\ \overline{y}'$) and in the update to the state (i.e. the mapping to x becomes the evaluated expression $c?\,\sigma(x)\ :\ \sigma(x')$.

A symmetric rewrite rule pushes calls out of branches:

$$\lambda\overline{z}\ \mathcal{S}\ \texttt{if}\ c\ \{\mathcal{S}'\ \texttt{let}\ x = \mathbb{A}(\overline{y})\}\ \texttt{else}\ \{\mathcal{S}''\ \texttt{let}\ x' = \mathbb{A}(\overline{y}')\}\ \mathcal{R}$$
$$\rightsquigarrow_t\ \lambda\overline{z}\ \mathcal{S}\ \texttt{if}\ c\ \{\mathcal{S}'\}\ \texttt{else}\ \{\mathcal{S}''\}\ \texttt{let}\ x = \mathbb{A}(c?\,\overline{y}\ :\ \overline{y}')\ \mathcal{R}\ [x' \to x]$$
$$\text{with } t(\sigma) = \sigma[x \to \sigma(x) \cup \sigma(x')] \quad (4)$$

The following rewrite rule eliminates empty conditional statements:

$$\lambda\overline{z}\ \mathcal{S}\ \texttt{if}\ c\ \{\}\ \texttt{else}\ \{\}\ \mathcal{R} \rightsquigarrow_t \lambda\overline{z}\ \mathcal{S}\ \mathcal{R}$$
$$\text{with } t(\sigma) = \sigma \quad (5)$$

And this rewrite rule inverts the branches of a conditional statement when the then branch is empty:

$$\lambda\overline{z}\ \mathcal{S}\ \texttt{if}\ c\ \{\}\ \texttt{else}\ \{\mathcal{S}'\}\ \mathcal{R} \rightsquigarrow_t \mathcal{S}\ \texttt{if}\ \neg c\ \{\mathcal{S}'\}\ \texttt{else}\ \{\}\mathcal{R}$$
$$\text{with } t(\sigma) = \sigma \quad (6)$$

More complex rewrite rules manipulate the control flow of the program to combine conditional statements in order to group API calls:

$$\lambda\overline{z}\ \mathcal{S}\ \texttt{if}\ c\ \{\texttt{let}\ x = \mathbb{A}(\overline{y})\}\ \texttt{else}\ \{\texttt{if}\ c'\ \{\texttt{let}\ x' = \mathbb{A}(\overline{y}')\}\ \texttt{else}\ \{\mathcal{S}'\}\}\ \mathcal{R}$$
$$\rightsquigarrow_t\ \lambda\overline{z}\ \mathcal{S}\ \texttt{if}\ c \vee c'\ \{\texttt{let}\ x = \mathbb{A}\left(c?\,\overline{y}\ :\ \overline{y}'\right)\}\ \texttt{else}\ \{\mathcal{S}'\}\ \ \mathcal{R}\ [x' \to x]$$
$$\text{with } t(\sigma) = \sigma[x \to c?\,\sigma(x)\ :\ (c'?\,\sigma(x')\ :\ \emptyset)] \quad (7)$$

A symmetric rule handles statements where the branches in the nested conditionals are reversed:

$$\lambda\overline{z}\ \mathcal{S}\ \texttt{if}\ c\ \{\texttt{let}\ x = \mathbb{A}(\overline{y})\}\ \texttt{else}\ \{\texttt{if}\ c'\ \{\mathcal{S}'\}\ \texttt{else}\ \{\texttt{let}\ x' = \mathbb{A}(\overline{y}')\}\}\ \mathcal{R}$$
$$\rightsquigarrow\ \lambda\overline{z}\ \mathcal{S}\ \texttt{if}\ c \vee \neg c'\ \{\texttt{let}\ x = \mathbb{A}\left(c?\,\overline{y}\ :\ \overline{y}'\right)\}\ \texttt{else}\ \{\mathcal{S}'\}\ \ \mathcal{R}\ [x' \to x]$$
$$\text{with } t(\sigma) = \sigma[x \to c?\,\sigma(x)\ :\ (c'?\,\emptyset\ :\ \sigma(x'))] \quad (8)$$

In general, all rules have similar variation that consider symmetries in the program. We do not list all of those.

Nested conditionals can be sequenced if the appropriate branches are empty:

$$\lambda\overline{z}\ \mathcal{S}\ \texttt{if}\ c\ \{\texttt{let}\ x = \mathbb{A}(\overline{y})\ \texttt{if}\ c'\ \{\texttt{let}\ x' = \mathbb{A}(\overline{y}')\}\ \texttt{else}\ \{\}\}\ \texttt{else}\ \{\}$$
$$\rightsquigarrow_t\ \lambda\overline{z}\ \mathcal{S}\ \texttt{if}\ c\ \{\texttt{let}\ x = \mathbb{A}(\overline{y})\}\ \texttt{else}\ \{\}\ \texttt{if}\ c' \wedge c\ \{\texttt{let}\ x' = \mathbb{A}(\overline{y}')\}\ \texttt{else}\ \{\}\ \mathcal{R}$$
$$\text{with } t(\sigma) = \sigma \quad (9)$$

And similarly nested conditionals can be simplified:

$$\lambda\overline{z}\ \mathcal{S}\ \texttt{if}\ c\ \{\texttt{if}\ c'\ \{\mathcal{S}'\}\ \texttt{else}\ \{\}\}\ \texttt{else}\ \{\}$$
$$\rightsquigarrow_t\ \lambda\overline{z}\ \mathcal{S}\ \texttt{if}\ c' \wedge c\ \{\mathcal{S}'\}\ \texttt{else}\ \{\}\ \mathcal{R}$$
$$\text{with } t(\sigma) = \sigma \quad (10)$$

Parameter Elimination. A rewrite rule eliminates a parameter x if it is not used in the body of the program:

$$x \notin FV(\mathcal{S}) \implies \lambda x, \overline{y}\ \mathcal{S} \rightsquigarrow_t \lambda\overline{y}\ \mathcal{S} \text{ where } t(\sigma) = \sigma$$

where $x \notin FV(\mathcal{S})$ means that x is not in the free variables of $\mathcal{S}$, i.e. not used in $\mathcal{S}$

Data Transform Elimination. Simple data transformations can be eliminated by inlining then in the program. In general, we inline data transformations that return constants or return one of their arguments. For example, a program $\lambda\overline{x}\ \mathcal{S}$ `where` $f := (x, y) \rightarrow y$ is transformed into a program $\lambda\overline{x}\ \mathcal{S}'$, in which $\mathcal{S}'$ is the same as $\mathcal{S}$ modulo the inlining of f. Inlining consists in substituting all variables bound to the result of f by the second argument of the binding.

Expression Simplification. There are additional refinement rules that operate only on the expressions that appear in the program and have no impact on the control flow (and their state transformation functions are identity).

B.2 Synthesis Rewrite rules

The following synthesis rewrite rule replaces the branching condition of an if-then-else instruction whose condition $C(\texttt{br})$ depends on `br` with a hidden function ϕ: in place of $C(\texttt{br})$, the branching condition becomes the output of a function ϕ. This function may take as input all parameters besides `br`, or any variable bound in the previous instructions of the program.

$$\begin{array}{l}\lambda\ \texttt{br}, \overline{y}. \\ \mathcal{S} \\ \texttt{if}\ C(\texttt{br})\ \{\mathcal{T}\} \\ \texttt{else}\ \{\mathcal{R}\}\end{array} \rightsquigarrow_t \begin{array}{l}\lambda\ \phi.\ \lambda\ \texttt{br}, \overline{y}. \\ \mathcal{S} \\ \texttt{let b = } \phi(\overline{y}, \texttt{BV}(\mathcal{S})) \\ \texttt{if b}\ \{\mathcal{T}\} \\ \texttt{else}\ \{\mathcal{R}\}\end{array}$$

where $t(\sigma) = \sigma \cup [b \rightarrow C(\texttt{br})]$.

In the rewrite above, $BV(\mathcal{S})$ denotes the set of bound variables in the statements of $\mathcal{S}$. The rewrite introduces a fresh variable `b`, which is assigned the value of the function ϕ and is used as the if-condition. To maintain correctness, t updates the augmented program state to ensure the value of `b` is the same as $C(\texttt{br})$ for every input trace.

Retry loops are introduced by synthesis rules because the condition of the loop needs to be synthesized in order for the loop to be valid. Retry loop introduction rewrites have the following form:

$$\begin{array}{l}\lambda\ \overline{y}. \\ \mathcal{R} \\ \mathcal{S} \\ \mathcal{S}' \\ \texttt{if}\ C\ \{\mathcal{S}''\} \\ \mathcal{T}\end{array} \rightsquigarrow_t \begin{array}{l}\lambda\ \phi\ \lambda\ \overline{y}. \\ \mathcal{R} \\ \texttt{retry}\ \{ \\ \quad \mathcal{S} \\ \quad \texttt{let b = } \phi(\texttt{BV}(\mathcal{S}), \texttt{BV}(\mathcal{R}), \overline{y}) \\ \}\ \texttt{until b} \\ \mathcal{T}\end{array}$$

where $t(\sigma) = \sigma[\overline{BV(\mathcal{S}), BV(\mathcal{S}), BV(\mathcal{S})} \rightarrow \overline{BV(\mathcal{S}), BV(\mathcal{S}'), BV(\mathcal{S}'')}] \cup [\overline{b} \rightarrow \overline{?\mathcal{S}, ?\mathcal{S}', ?\mathcal{S}''}]$

A new variable `b` is introduced, bound to the result of the hidden function ϕ, and then used as a stopping condition for the retry loop. Statements $\mathcal{S}$, $\mathcal{S}'$, and $\mathcal{S}''$ must all be calls to the same API method. The state transformation now maps *iterations* of the bound variables of $\mathcal{S}$ to the values of each statement that has been captured in the loop, represented by the vector $\overline{BV(\mathcal{S}), BV(\mathcal{S}'), BV(\mathcal{S}'')}$. When evaluating the program for a trace, the variables $BV(\mathcal{S}'')$ will not be defined for the traces where C is false, in which case the value is null. The valuation of condition `b` is also a vector $\overline{?\mathcal{S}, ?\mathcal{S}', ?\mathcal{S}''}$ that is computed by assigning the truth value of whether the statement replaced by the iteration is the last statement in the trace (denoted by $\overline{?S}$). In practice,

we generalize this rule by considering patterns where matching statements are in sequence, and the matching can happen within conditionals, as is the case with the last statement in this example rule.

C LLM experiment prompt

The following is the prompt used in every call to Claude 3.5 Sonnet in our comparison against LLMs experiment described in §7. We describe the task that Syren solves, including the goals of the synthesized program and lightweight restrictions on the output language. We choose to let the LLM express the script in any language to improve its chances of success. We also emphasize that the desired program must be able to replicate every provided trace, as that is our formal definition of correctness. We also added text that attempts to counteract a tendency to include unnecessary loops in the program.

````
You are a helpful assistant. You are given a set of traces from a program execution, and you
 need to generate a script that the user can then use in some automation.
The traces are in the format of a list of json events, where each event is of the form:
```
{
   "api": <THE API NAME>,
   "request": {
      <ARGUMENT 1>: <VALUE 1>,
      <ARGUMENT 2>: <VALUE 2>,
      ...
   },
   "response": <RESPONSE VALUE>
}
```
That is, the event contains an API name in the "api" field, the request parameters in the "
request" field and the response value in the "response" field. The "request" is a JSON
object, where each member is one of the arguments of the API call.

The script you generate can use retry loops, list maps and conditionals. Each statement in
the script should either be a control flow statement (conditional or loop), an API call (
with the same name as in the traces and same arguments) or a data transformation function.
Data transformation functions can transform the output of an API call into another value
that can be used in a conditional or in another API call's arguments. You need to write the
script as part of a function with clearly identified parameters.

The script you generate MUST reproduce at least the same traces that it was given, for some
value of the parameters of the script.
The script SHOULD be also more general, meaning that different parameter values can be used,
 and they would generate similar traces as the original ones.
You SHOULD NOT add loops when the traces do not show the need for loops, for example, if
each trace calls an API once, then the script should only call once.
````

D Detailed Benchmarks

This sections shows our detailed benchmarks of Cloud automation scripts. These scripts were manually written to reflect Cloud automation tasks, and were used to evaluate Syren.

D.1 Custom Benchmarks

Hand-written benchmarks that perform common cloud automation tasks.

D.1.1 *BackupThenDeleteTable.*

```
λ tableName, backupName
let backupCreation = dynamodb.CreateBackup(TableName=tableName, BackupName=backupName)
let backupArns = getBackupArn(backupCreation)
let backupArn = first(backupArns)
retry {
  let backupDesc = dynamodb.DescribeBackup(BackupArn=backupArn)
  let backupStatus = getBackupStatus(backupDesc)
} until (backupStatus[0] == "AVAILABLE")

let tableDeletion = dynamodb.DeleteTable(TableName=tableName)

where
getBackupArn := "$.BackupDetails.BackupArn"
getBackupStatus := "$.BackupDescription.BackupDetails.BackupStatus"
first := (x) -> x[0]
```

D.1.2 *CreateBucketThenFolder.*

```
λ bucketName
let bucketCreation = s3.CreateBucket(bucket=bucketName)
let folderCreation = s3.PutObject(bucket=bucketName, key="custom-logs/")
```

D.1.3 *CreateImage.*

```
λ instanceId, noReboot
let instanceIds = list(instanceId)
let description = ec2.DescribeInstances(instanceIds=instanceIds)
let requestId = getRequestId(description)
let imageName = buildName(instanceId, requestId)
let result = ec2.CreateImage(instanceId=instanceId, noReboot=noReboot, name=imageName)
where
buildName := (X, Y) -> (X + "_auto_") + Y[0]
getRequestId := "$..x-amzn-requestid"
```

D.1.4 *CreateImageFromSsmParameterAndLog.*

```

λ instances, x_3
let instanceDesc = ec2.DescribeInstances(InstanceIds=instances)
let parameter = ssm.GetParameter(Name="image-builder")
let instanceId = getExistingInstance(instances)
let imageName = getParameterValue(parameter)
let mustReboot = checkReboot(x_3)
let imageCreationRes = ec2.CreateImage(
  InstanceId=instanceId, Name=imageName, NoReboot=mustReboot)
let imageId = getImageId(imageCreationRes)
let api_res_2_3 = console.Log(message=imageId)
where
getExistingInstance := (x) -> x[0]
getParameterValue := (x) -> x..Value[0]
```

```
checkReboot := (x) -> x == "ami-e27e9b0896ea30tb2"
getImageId := (x) -> x.ImageId
```

D.1.5 *CreateTable.*

```
λ tableName, definitions, keys, provisioning
let tableCreating = dynamodb.CreateTable(tableName=tableName,
   attributeDefinitions=definitions,
   keySchema=keys,
   provisionedThroughput=provisioning)
```

D.1.6 *CreateTableThenInsertItem.*

```
λ tableName, definitions, keys, item
let provisionedThroughput = throughputSettings()
let tableCreating = dynamodb.CreateTable(TableName=tableName,
   AttributeDefinitions=definitions,
   KeySchema=keys,
   ProvisionedThroughput=provisionedThroughput)

retry {
  let tableDescription = dynamodb.DescribeTable(TableName=tableName)
  let tableStatus = extractStatus(tableDescription)
} until (tableStatus[0] == "ACTIVE")
let tableInserting = dynamodb.PutItem(TableName=tableName,Item=item)

where
extractStatus := "$.Table.TableStatus"
throughputSettings := () -> { "ReadCapacityUnits" : 5, "WriteCapacityUnits" : 5 }
```

D.1.7 *DeleteTable.*

```
λ tableName
let tableDeletion = dynamodb.DeleteTable(TableName=tableName)
```

D.1.8 *MoveDdbItem.*

```
λ originTableName, destinationTableName, keyValue, keyId
let itemKey = composeKey(keyValue, keyId)
let itemResponse = dynamodb.GetItem(TableName=originTableName,Key=itemKey)
let items = selectItem(itemResponse)
let item = first(items)
let response = dynamodb.PutItem(TableName=destinationTableName,Item=item)
where
composeKey := (x, y) -> {y : {"S": x}}
first := (x) -> x[0]
selectItem := "$.Item"
```

D.1.9 *MoveDdbItemIfPresent.*

```
λ originTableName, destinationTableName, keyValue, keyId
let itemKey = composeKey(keyValue, keyId)
let itemResponse = dynamodb.GetItem(TableName=originTableName,Key=itemKey)
let items = selectItem(itemResponse)
if len(items) > 0 {
  let item = first(items)
  let response = dynamodb.PutItem(TableName=destinationTableName,Item=item)
}
where
composeKey := (x, y) -> {y : {"S": x}}
first := (x) -> x[0]
selectItem := "$.Item"
```

D.1.10 PutObjectIfNotPresent.

```
λ bucketName, objectName
let listObjectsResult = s3.ListObjectsV2(Bucket=bucketName,Prefix=objectName)
let counts = getCount(listObjectsResult)
let count = fst(counts)
if count == 1 {
  let contentKeys = getContentKey(listObjectsResult)
  let contentKey = fst(contentKeys)
  if contentKey == objectName {
    return
  }
}
let createObject = s3.PutObject(Bucket=bucketName,Key=objectName)
where
getCount := "$.KeyCount"
getContentKey := "$.Contents[0].Key"
fst := (X) -> X[0]
```

D.1.11 CopyS3Objects.

```
λ originBucket, dest
let list_objects_result = s3.ListObjects(Bucket=originBucket, Prefix="*")
let keys = getObjectsKeys(list_objects_result)
for (key) in keys {
  let objectInfo = s3.HeadObject(Bucket=dest, Key=key)
  let c = getObjectVersion(objectInfo)
  if isFile {
    let moved = s3.Copy(
      SourceBucket=originBucket,
      SourceKey=key,
      DestinationBucket=dest,
      DestinationKey=key
    )
  }
}
where
 getObjectsKeys := (x) -> x..key
 getObjectVersion := (x) -> x..version == "v1"
```

D.1.12 *StartInstances.*

```
λ instanceIds
let x = ec2.StartInstances(InstanceIds=instanceIds)
```

D.1.13 *StartInstancesWithTags.*

```
λ key, value
let filters = makeFilterTag(key,value)
let x = ec2.DescribeInstances(Filters=filters)
let iids = extractIids(x)
let resp = ec2.StartInstances(InstanceIds=iids)
where
extractIids := "$..InstanceId"
makeFilterTag := (k,v) -> [{"Name": "tag:" + k, "Values" : [v]}]
```

D.1.14 *StopAllRunningInstances.*

```
λ
let filters = runningInstancesFilter()
let runningInstances = ec2.DescribeInstances(Filters=filters)
let instanceIds = extractInstanceIds(runningInstances)
let res = ec2.StopInstances(InstanceIds=instanceIds)
where
runningInstancesFilter := () -> [dict(Name="instance-state-name",Values=["running"])]
extractInstanceIds := "$..InstanceId"
```

D.1.15 *StopInstances.*

```
λ instanceIds
let x = ec2.StopInstances(InstanceIds=instanceIds)
```

D.1.16 *StopInstancesCond.*

```
λ instanceId
let ids = list(instanceId)
let _ = ec2.StopInstances(InstanceIds=ids, Force=False)
let s = ec2.DescribeInstanceStatus(InstanceIds=ids, IncludeAllInstances=True)
let statuses = extractInstanceStatus(s)
let status = first(statuses)
if status != "stopped" {
   let _ = ec2.StopInstances(InstanceIds=ids, Force=True)
}
where
first := (X) -> X[0]
extractInstanceStatus := "$.InstanceStatuses[0].InstanceState.Name"
```

D.1.17 *TagInstancesWithDryRun.*

```
λ instanceIds, key, value
let tags = makeTags(key,value)
let x = ec2.CreateTags(Resources=instanceIds, Tags=tags, DryRun=True)
if x == "ok" {
  let _ = ec2.CreateTags(Resources=instanceIds, Tags=tags)
}
where
makeTags := (k,v) -> [dict(Key=k, Value=v)]
```

D.1.18 *WaitForInputThenSendEmail.*

```
λ  user_email
retry {
  let payload = os.QueryInput()
  let b = isNonEmpty(_r)
} until ( b )
let _ = messages.SendEmail(payload=payload, userId=isNonEmpty)
where
isNonEmpty := (x) -> ! (x == "")
```

D.1.19 *RetrieveChannelMembers.*

```
λ
let slackResponse = slack.GetConversationsList()
let channelId = getFirstChannelId(slackResponse)
let conversationMembersResponse = slack.GetConversationMembers(channel=channelId)
let responseUserList = getMembersFunction(conversationMembersResponse)
for (memberUser) in responseUserList {
  let user_info = slack.GetUserInfo(user=memberUser)
}
 where
  getFirstChannelId := (x) -> x..id[0]
  getMembersFunction := (x) -> x.members
```

D.1.20 *SVG-IncreaseCircleRadius.*

```
λ elementId
let elt = js.getElementAttributeById(Attribute=0, Id=elementId)
let incr = f(api_res_2_0)
let api_res_2_1 = js.setElementAttribute(Attribute=0, Id=elementId, Value=incr)
 where
  f := (x) -> 1 + x
```

D.1.21 *SVG-IncreaseCircleRadius-json.*

```
λ document
let circle = js.getElementById(Document=document,Id="circle1")
let radius = js.getElementAttribute(Element=circle, Name="r")
let result = js.setElementAttribute(Element="circle1", Name="r", Value=radius)
where
add10 := (circle) -> circle + 10
```

D.1.22 SVG-ScaleAndMove.

```
λ  x_1
let api_res_2_0 = js.getElementAttributeById(Attribute=0, Id=x_1)
let param_elim_12 = f?_22(api_res_2_0)
let api_res_2_1 = js.setElementAttribute(Attribute=0, Id=x_1, Value=param_elim_12)
 where
  f?_22 := (arg0_2) -> 1 + arg0_2
```

D.1.23 List and Move Files.

```
λ source, dest
let paths = os.Ls(Dir=source, Pattern="/*")
for (path) in paths {
  let isFile = os.IsFile(Path=path)
  if isFile {
    let moved = os.Mv(Path=path, Dest=dest)
  }
}
```

D.1.24 Clean Current Dir.

```
λ dirName
let files = os.OpenAndGetdents(Dir=dirName)
let names = f1(files)
for (path) in (names) {
  let isfile_res = os.IsFile(Path=path)
  if isfile_res {
    let rm_res = os.Rm(Path=path)
  }
}
where:
f1 : x -> $..d_name
```

D.1.25 Clean Regular Files in Dir.

```
λ dirName
let getdents_res = os.OpenAndGetdents(Dir=dirName)
let f1_res = f1(getdents_res)
for (path) in (f1_res) {
  let fstat_res = os.OpenAndGetdentsFstat(Path=path)
  let f2_res = f2(fstat_res)
  if f2_res {
    let mv_res = os.Rm(Path=path)
  }
}
where:
f1 : x -> $..d_name
f2 : x -> $.st_type == "S_IFREG"
```

D.2 Blink automation benchmarks

Tasks from the Blink automation library [7], where various APIs are interfaced.

D.2.1 CopyEC2InstanceToNewRegion.

```
λ  instanceId, imageName, sourceRegion, destRegion, destName
let createImageResponse = ec2.CreateImage(InstanceId=instanceId, Name=imageName)
let imageId = extractImageId(createImageResponse)
let imageIds = list(imageId)
retry {
  let describeImagesResponse = ec2.DescribeImages(ImageIds=imageIds)
  let state = extractImageStatus(describeImagesResponse)
} until (state == "available")
let copyResponse = ec2.CopyImage(
   Name=destName, SourceImageId=imageId, SourceRegion=sourceRegion, Region=destRegion)

where
extractImageId := "$.ImageId"
extractImageStatus := "$..State"
```

D.2.2 CreateIamUserAndNotify.

```
λ userName, notificationEmail
let createUserResponse = iam.CreateUser(UserName=userName)
let user = extractUser(createUserResponse)
let sendEmail = blink.SendEmail(Content=)
where
extractUser := "$.User"
```

D.3 AWS Automation Runbooks Benchmarks

Scripts collected from AWS Systems Manager Automation Runbooks.

D.3.1 AWS-ConfigureCloudWatchOnEC2Instance.

```
λ instanceId, status, properties
let instanceIds = list(instanceId)
if status == "Enabled" {
  let _ = ec2.MonitorInstances(InstanceIds=instanceIds)
} else {
  let _ = ec2.UnmonitorInstances(InstanceIds=instanceIds)
}
```

D.3.2 AWS-ConfigureS3BucketVersioning.

```
λ bucketName, versioningState
if ((versioningState != "Enabled") && (versioningState != "Suspended")) {
  return
}
let config = makeConfig(versioningState)
let result = s3.PutBucketVersioning(Bucket=bucketName, VersioningConfiguration=config)
where
makeConfig := (x) -> dict(MFADelete="Disabled",Status=x)
```

D.3.3 AWS-ResizeInstance.

```
λ instanceId, instanceType
let instanceIds = list(instanceId)
let description = ec2.DescribeInstances(InstanceIds=instanceIds)
let actualType = getInstanceType(description)
if (actualType != instanceType) {
  let _ = ec2.StopInstances(InstanceIds=instanceIds)
  retry {
    let x = ec2.DescribeInstanceStatus()
    let status = extractInstanceStatus(x)
  } until (status == "stopped")
  let v = makeValue(instanceType)
  let _ = ec2.ModifyInstanceAttribute(InstanceId=instanceId, InstanceType=v)
  let _ = ec2.StartInstances(InstanceIds=instanceIds)
  retry {
    let x = ec2.DescribeInstanceStatus()
    let status = extractInstanceStatus(x)
  } until (status == "running")
}
where
extractInstanceStatus := "$.InstanceStatuses[0].InstanceState.Name"
getInstanceType := "$.Reservations[0].Instances[0].InstanceType"
makeValue := (x) -> {'Value' : x}
```

D.3.4 AWS-SetRequiredTags.

```
λ tags, resources
let taggedResourcesResponse = resourcegroupstaggingapi.TagResources(
  resourceARNList=resources,
   tags=tags)
```

D.3.5 AWS-StartEC2Instance.

```
λ instanceId
let ids = list(instanceId)
let s = ec2.DescribeInstances(InstanceIds=ids)
let status = extractInstanceStatus(s)
if status != "running" {
  let _ = ec2.StartInstances(InstanceIds=ids)
}
where
first := (X) -> X[0]
extractInstanceStatus := "$.Reservations[0].Instances[0].State.Name"
```

D.3.6 AWS-StopEC2Instance.

```
λ instanceId
let ids = list(instanceId)
let _ = ec2.StopInstances(InstanceIds=ids, Force=False)
let _ = ec2.StopInstances(InstanceIds=ids, Force=True)
```

D.3.7 AWSSupport-CopyEC2Instance.

```
λ instanceId,
   keyPair,
   region,
   subnetId,
   instanceType,
   securityGroupIds,
   keepImageSourceRegion,
   keepImage,
   keepImageDestinationRegion,
   noRebootInstanceBeforeTakingImage

(** Extract information from instance *)
let instanceIds = list(instanceId)
let instanceInfo = ec2.DescribeInstances(InstanceIds=instanceIds)
if instanceInfo == null {
  return
}
let sourceInstanceTypes = extractSourceInstanceType(instanceInfo)
let sourceAvailabilityZones = extractSourceAvailabilityZone(instanceInfo)
let sourceSubnetIds = extractSourceSubnetId(instanceInfo)
let sourceKeyPairs = extractSourceKeyPair(instanceInfo)
let sourceSecurityGroupIds = extractSourceSecurityGroupIds(instanceInfo)
let sourceRootDeviceNames = extractSourceRootDeviceName(instanceInfo)

(** Select *)
let sourceInstanceType = first(sourceInstanceTypes)
let sourceAZ = first(sourceAvailabilityZones)
let sourceSubnetId = first(sourceSubnetIds)
let sourceKeyPair = first(sourceKeyPairs)

(** Prepare the parameters for the new instance *)
let regionToUse = selectFirstNonEmpty(region, sourceAZ)
let isSameRegion = isSame(regionToUse, sourceAZ)
let instanceTypeToUse = selectFirstNonEmpty(instanceType, sourceInstanceType)
let subnetIdToUse = select(isSameRegion, subnetId, sourceSubnetId)
let securityGroupIdsToUse =
  select(isSameRegion, securityGroupIds, sourceSecurityGroupIds)
// Create new image
let imageName = makeName(instanceId)
let newImage = ec2.CreateImage(
  InstanceId=instanceId, NoReboot=noRebootInstanceBeforeTakingImage, Name=imageName)
let newImageIds = extractImageId(newImage)
retry {
   let imageInfo = ec2.DescribeImages(ImageIds=newImageIds)
```

```
  let imageState = extractImageState(imageInfo)
} until (imageState[0] == "available")
if (imageState[0] != "available") {
  return
}
(** Tag image *)
let tags = makeTags(instanceId)
let _ = ec2.CreateTags(Resources=newImageIds, Tags=tags)
(** Launch instance *)
let tagSpecs = makeTagSpecs(instanceId)
let newImageId = first(newImageIds)
if isSameRegion {
 if keyPair == "" {
    let r1 = ec2.RunInstances(
       ImageId=newImageId,
       SubnetId=subnetIdToUse,
       InstanceType=instanceTypeToUse,
       SecurityGroupIds=securityGroupIdsToUse,
       TagSpecifications=tagSpecs,
       MinCount=1,
       MaxCount=1)
 } else {
    let r1 = ec2.RunInstances(
       ImageId=newImageId,
       SubnetId=subnetIdToUse,
       InstanceType=instanceTypeToUse,
       SecurityGroupIds=securityGroupIdsToUse,
       KeyName=keyPair,
       TagSpecifications=tagSpecs,
       MinCount=1,
       MaxCount=1)
 }
 let newInstanceIds = getInstanceId(r1)

 // Wait for the new instance to be running
 let newInstanceId = first(newInstanceIds)
 retry {
   let newInstances = ec2.DescribeInstanceStatus(InstanceIds=newInstanceIds)
   let newInstanceStatus = extractInstanceStatus(newInstances)
 } until ((len(newInstanceStatus) > 0) && (newInstanceStatus[0] == "running"))

 if !keepImageSourceRegion {
   let deregisterResult = ec2.DeregisterImage(ImageId=newImageId)
   let snapshots = getSnapshots(imageInfo)
   let snapshot = first(snapshots)
   let deleteSnapshotResult = ec2.DeleteSnapshot(SnapshotId=snapshot)
 }
}

(** Transformation Definitions *)
where
extractSourceInstanceType := "$.Reservations[0].Instances[0].InstanceType"
```

```
extractSourceAvailabilityZone :=
  "$.Reservations[0].Instances[0].Placement.AvailabilityZone"
extractSourceSubnetId := "$.Reservations[0].Instances[0].SubnetId"
extractSourceKeyPair := "$.Reservations[0].Instances[0].KeyName"
extractSourceSecurityGroupIds :=
  "$.Reservations[0].Instances[0].SecurityGroups..GroupId"
extractSourceRootDeviceName := "$.Reservations[0].Instances[0].RootDeviceName"
selectFirstNonEmpty := (X, Y) -> X == "" ? Y : X
select := (B, X, Y) -> B ? (X == "" ? Y : X) : X
isSame := (X,Y) -> X == Y
first := (X) -> X[0]
makeName := (X) -> "Api_Doc_AWSSupport-CopyEC2Instance_LocalAmi_for_" + X
extractImageId := "$.ImageId"
extractImageState := "$.Images[0].State"
getSnapshots := "$.Images[0]..SnapshotId"
getInstanceId := "$.Instances[0].InstanceId"
extractInstanceStatus := "$.InstanceStatuses[0].InstanceState.Name"
makeTags := (X) -> [
  {"Key" : "Name", "Value" : "AWSSupport-CopyEC2Instance_LocalAmi_for" + X },
  {"Key" : "AWSSupport-CopyEC2Instance", "Value" : "some-unique-id" },
  {"Key" : "CreatedBy", "Value": "AWSSupport-CopyEC2Instance" }
]
makeTagSpecs := (X) -> [{
   "ResourceType": "instance",
   "Tags" : [
      { "Key" : "Name", "Value" : "AWSSupport-CopyEC2Instance_Source:_" + X },
      { "Key": "CreatedBy", "Value": "AWSSupport-CopyEC2Instance" }
   ]
}]
```

D.4 ApiPhany Benchmarks

These are tasks adapted from previous literature on API-composing programs Synthesis, ApiPhany [18], that use Stripe and Slack APIs.

D.4.1 ApiPhanyExample.1.1.

```
λ location_id
let res = stripe.GetTransactions(Location=location_id)
let transactions = extractTransactionsOrderIds(res)
for transaction in transactions {
  let _ = console.Log(Message=transaction)
}
where
extractTransactionsOrderIds := (x) -> x..OrderId
```

D.4.2 ApiPhanyExample.1.2.

```
λ user_email
let userLookupResult = slack.LookupByEmail(name=user_email)
let user_id = getUserIdFromLookupResult(userLookupResult)
```

```
let user_conversations = slack.ConversationsOpen(users=user_id)
let channelId = getChannelIdFromConversations(user_conversations)
let slack_response = slack.PostMessage(channel=channelId)
 where
  getChannelIdFromConversations := (x) -> ((x[0])[2]).channelId
  getUserIdFromLookupResult := (x) -> x.user..id
```

D.4.3 *ApiPhanyExample.1.6.*

```
λ channelId, messageTimestamp
let slackResponse = slack.ChatPostMessage(channel=channelId, ts=messageTimestamp)
let messageThreadId = getThreadFromResponse(slackResponse)
let response = slack.ChatUpdate(channel=channelId, ts=messageThreadId)
 where
  getThreadFromResponse := (x) -> x..thread[1]
```

D.4.4 *ApiPhanyExample.1.8.*

```
λ channelId
let conversationInfoResponse = slack.GetConversationInfo(channel=channelId)
let latestMessageTimestamp = getLatestMessageTimestamp(conversationInfoResponse)
let channelId = getChannelId(conversationInfoResponse)
let conversationHistoryResponse = slack.GetConversationsHistory(
   channel=channelId, oldest=latestMessageTimestamp)
 where
  getChannelId := (x) -> x.id
  getLatestMessageTimestamp := (y) -> y.latest
```

D.4.5 *ApiPhanyExample.2.1.*

```
λ customer, productId
let api_res_2_0 = v1.GetPrices(currency="USD", productId=productId)
let typ = listTypes(api_res_2_0)
for (typ) in (types) {
 et _r = v1.PostSubcriptions(customer=customer, productId=productId, type=typ)
}
 where
  listTypes := (x) -> x..type
```

D.4.6 *ApiPhanyExample.2.3.*

```
λ customerId, productCurrency, productName, unitAmount
let productResponse = v1.PostProducts(name=productName)
let productId = getProductIdFromProductResponse(productResponse)
let productPriceResponse = v1.PostPrices(
  currency=productCurrency, product=productId, unit_amount=unitAmount)
let productId = getPriceId(productPriceResponse)
let productInvoiceItemsResponse = v1.PostInvoiceItems(customer=customerId, price=productId)
 where
  getPriceId := (x) -> x..id[0]
  getProductIdFromProductResponse := (y) -> y..id[0]
```

D.4.7 *ApiPhanyExample.2.5.*

```
λ customer
let invoiceApiResponse = v1.GetInvoices(customer=customer)
let invoiceIds = getInvoiceIdFromResponse(invoiceApiResponse)
for (invoiceId) in (invoiceIds) {
  let chargesResponse = v1.GetCharges(invoiceId=invoiceId)
}
where
  getInvoiceIdFromResponse := (x) -> x.data
```

D.4.8 *ApiPhanyExample.2.6.*

```
λ subscriptionId
let subscriptionResponse = stripe.GetSubscription(SubscriptionExposedId=subscriptionId)
let subscription_id = getSubscriptionId(subscriptionResponse)
let subscriptionInvoice = stripe.GetInvoice(invoice=subscription_id)
let invoiceId = getInvoiceAmountPaid(subscriptionInvoice)
let refundedInvoice = stripe.Refund(charge=invoiceId)
 where
  getInvoiceAmountPaid := (x) -> x.amount_paid
  getSubscriptionId := (x) -> x..id[1]
```

D.4.9 *ApiPhanyExample.2.7.*

```
let customerApiResponse = v1.GetCustomers()
let filteredCustomers = getCustomerEmails(customerApiResponse)
for (customerEmail) in (filteredCustomers) {
  let customerEmail = mail.Show(email=customerEmail)
}
where
  getCustomerEmails := (x) -> x..email
```

D.4.10 *ApiPhanyExample.2.10.*

```
λ customerId, defaultPaymentId
let api_res_1_0 = v1.GetSubscriptions(customer=customerId)
let subscriptionIds = getSubscriptionIds(api_res_1_0)
for (subcriptionId) in (subscriptionIds) {
  let _r =
    v1.PostSubscriptionPayment(
      default_payment=defaultPaymentId,
        subscriptionExposedId=subscriptionId)
}
where
 getSubscriptionIds := (x) -> x.data..id
```

D.4.11 *ApiPhanyExample.2.11.*

```
λ customer_id
let getCustomerResult = stripe.GetCustomer(customer=customer_id)
let default_source = get_default_source(getCustomerResult)
let _ = stripe.DeletePaymentSource(customer=customer_id, id=default_source)
 where
  get_default_source := (x) -> x.default_source
```

D.4.12 ApiPhanyExample.2.13.

```
λ customerId, price
let invoiceItemsResponse = stripe.InvoiceItems(customer=customerId, price=price)
let invoiceId = getInvoiceId(invoiceItemsResponse)
let invoice = stripe.Invoice(customer=customerId, invoice=invoiceId)
let invoice = stripe.SendInvoice(invoice=invoiceId)

where
getInvoiceId := (x) -> x..invoiceId[0]
```

D.4.13 ApiPhanyExample.3.1.

```
λ locationId
let invoice_response = square.GetInvoices(locationId=locationId)
```

D.4.14 ApiPhanyExample.3.3.

```
λ tax_id
let list_res = slack.GetCatalog()
let objects = extractObjects(list_res)
for (object) in objects {
  let tax_ids = taxIds(object)
  for (id) in tax_ids {
    let p = tax_predicate(tax_id, id)
    if (p)) {
      let x = console.Log(object)
    }
  }
}

where
extractObjects := (x) -> x.objects
taxIds := (x) -> x.item_data.tax_ids
tax_predicate := (x,y) -> x == y
```

D.4.15 ApiPhanyExample.3.4.

```
let catalogList = square.ListCatalog()
let catalogMessages = getDiscountTypeFromResponse(catalogList)
for (message) in (catalogMessages) {let discountDetails = console.Log(message=message) }
where
  getDiscountTypeFromResponse := (x) -> x..discountType
```

D.4.16 ApiPhanyExample.3.6.

```
let paymentsResponse = square.GetPayments()
let paymentMessages = getPaymentNotes(paymentsResponse)
for (paymentMessage) in (paymentMessages) {
  let paymentNotes = console.Log(message=paymentMessage)
}
where
  getPaymentNotes := (x) -> x..payment_note
```

D.4.17 ApiPhanyExample.3.7.

```
λ location_id
let res = stripe.GetTransactions(Location=location_id)
let transactions = extractTransactionsOrderIds(res)
for transaction in transactions {
  let _ = console.Log(Message=transaction)
}
where
extractTransactionsOrderIds := (x) -> x..OrderId
```

D.4.18 ApiPhanyExample.3.8.

```
λ locationId, transactionId
let res = square.BatchRetrieveOrders(LocationId=locationId, TransactionId=transactionId);
let orders = getOrders(orders)
for order in orders {
  let line_items = getLineItems(order)
  for line_item in line_items {
    let _ = console.Log(Message=line_item)
  }
}

where
getOrders := (x) -> x.orders
getLineItems := (x) -> x.line_items
```

Received 2024-11-15; accepted 2025-03-06