

Program Synthesis from Partial Traces

Margarida Ferreira, Victor Nicolet, Joey Dodds, Daniel Kroening



Actions performed in a cloud computing console produce records of the underlying API calls

Instances (3/7) [Info](#)

[C](#) [Connect](#) [Launch instances](#) [Find instance by attribute](#)

Instance state ▲ Actions ▼

Stop instance

Start instance

Reboot instance

Hibernate instance

Terminate instance

Name	Instance ID	State	Actions
A	i-0e2b2fbdb0c0597ec	Running	+Q Q-
<input checked="" type="checkbox"/> B	i-029d9b0d3e702f05e	Running	+Q Q-
<input type="checkbox"/> C	i-0a81673dd1e960dfa	Running	+Q Q-
<input checked="" type="checkbox"/> D	i-0bcc4e4901c33c6b5	Running	+Q Q-
<input checked="" type="checkbox"/> E	i-023dc5f7e2acb0240	Running	+Q Q-
<input type="checkbox"/> F	i-02bbc787ec7ac3aca	Running	+Q Q-
<input type="checkbox"/> G	i-01d7-27d12-e-7-6-	Running	+Q Q-

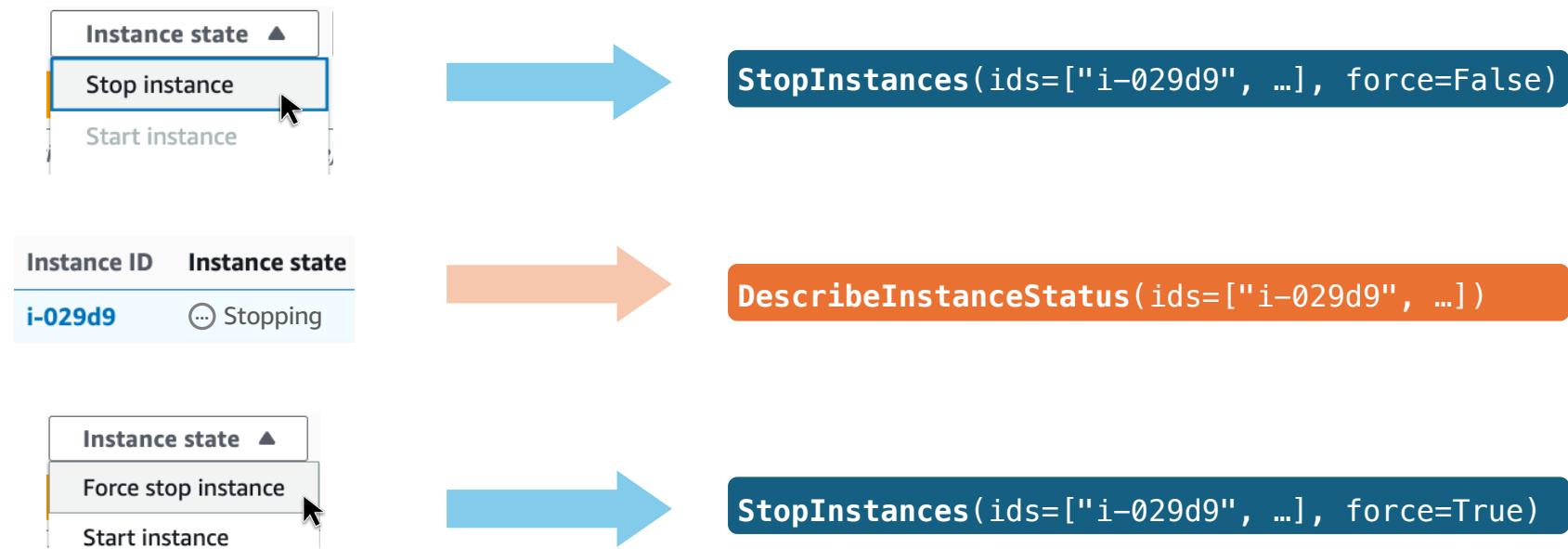
Event history [Info](#)

	Event name	Event time	Resource name
<input type="checkbox"/>	StopInstances	July 05, 2023, ...	i-023dc5f7e2acb0240, i-029d9b0d3e702f05e, i-0bcc4e4

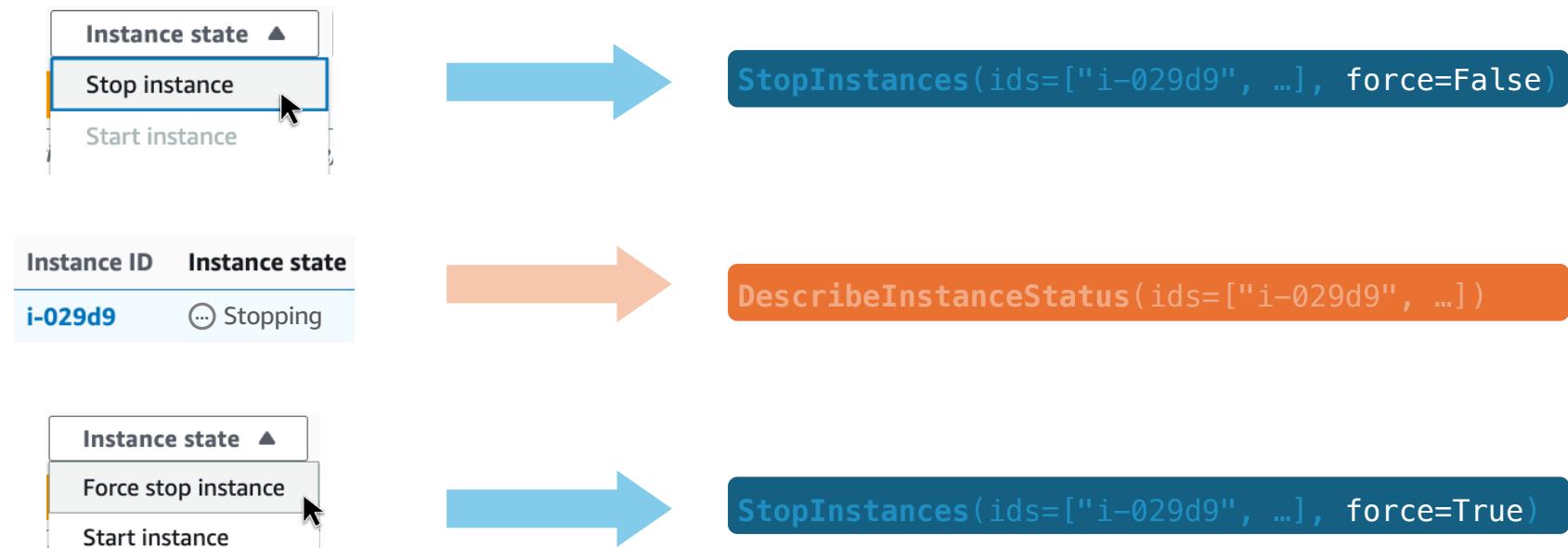
Consider the task of stopping some compute instances using the cloud computing interface



Each action performed gets recorded as an API method call, along with inputs and output



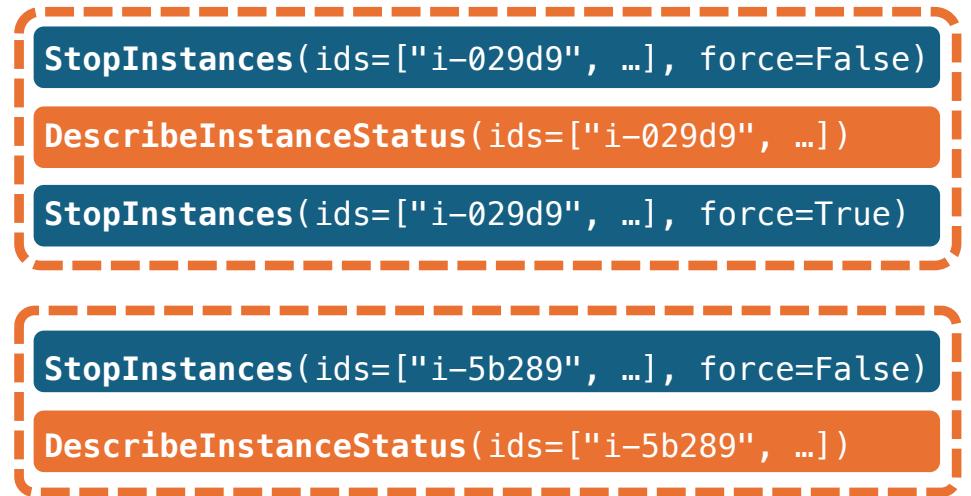
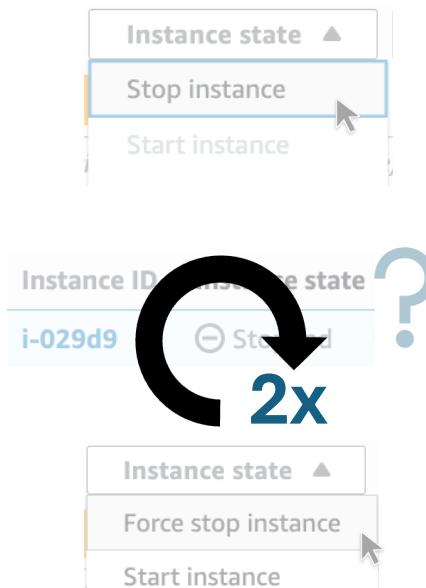
Each action performed gets recorded as an API method call, along with inputs and output



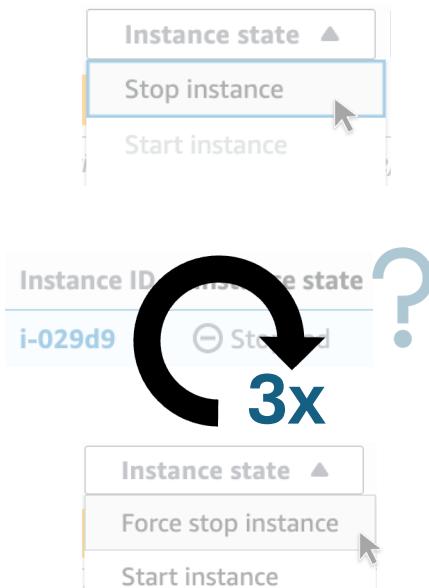
The sequence of API method calls is a trace that represents the task



Multiple executions of the same task produce different traces



Multiple executions of the same task produce different traces



```
StopInstances(ids=["i-029d9", ...], force=False)
DescribeInstanceStatus(ids=["i-029d9", ...])
StopInstances(ids=["i-029d9", ...], force=True)
```

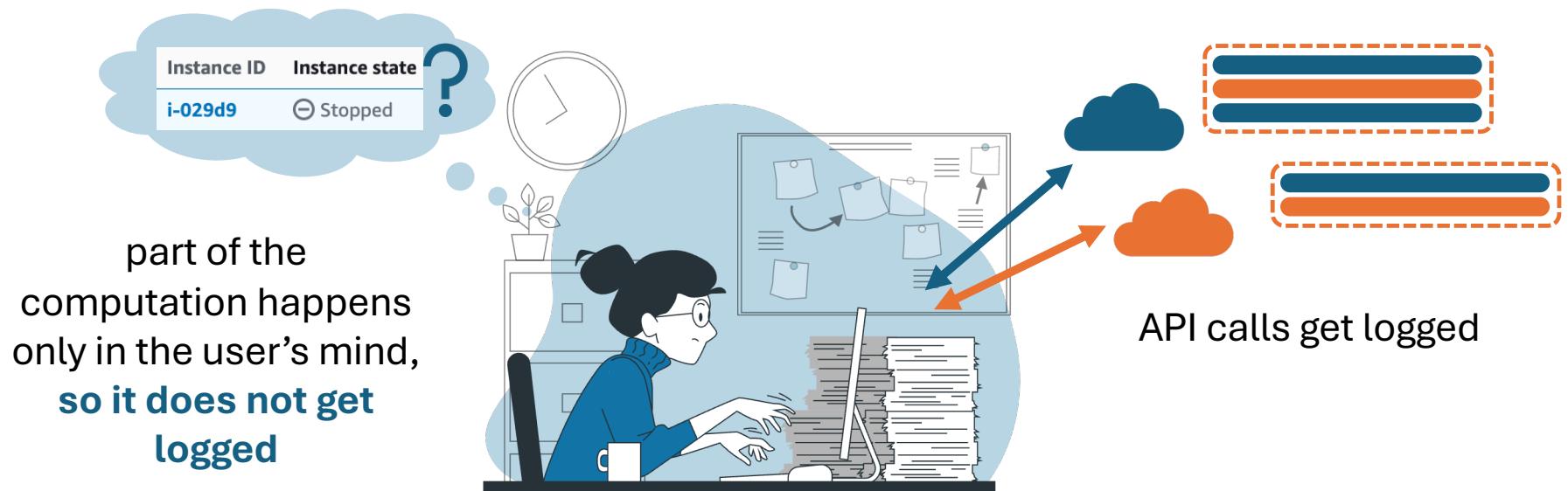
```
StopInstances(ids=["i-5b289", ...], force=False)
DescribeInstanceStatus(ids=["i-5b289", ...])
```

```
StopInstances(ids=["i-9ab4e", ...], force=False)
DescribeInstanceStatus(ids=["i-9ab4e", ...])
StopInstances(ids=["i-9ab4e", ...], force=True)
```

Our goal is to synthesize a program that executes a task represented by partial program traces

This program can then be offered to the user as a 1-click automation of their task

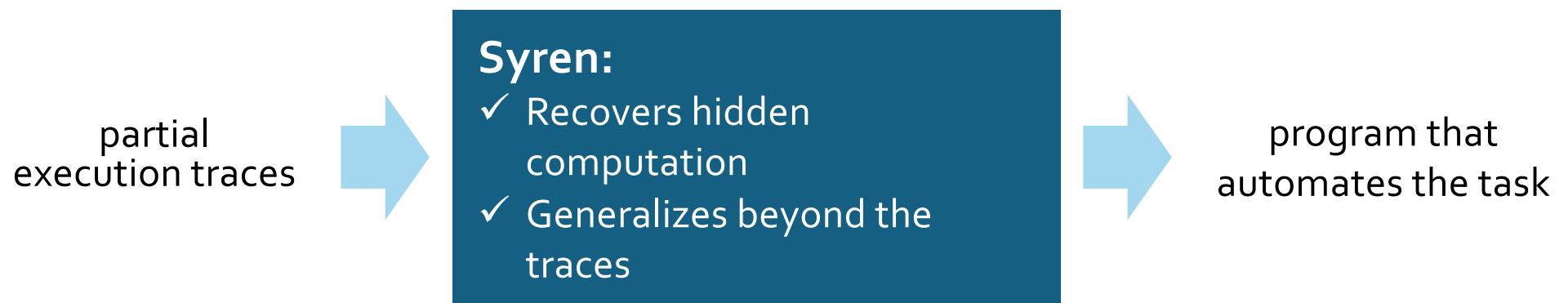
Information in traces is incomplete



Syren's Synthesis Pipeline



Syren's Synthesis Pipeline



At any point in the synthesis, Syren's candidate program is *correct*

Definition: Program correctness

At any point in the synthesis, Syren's candidate program is *correct*

Definition: Program correctness

A program P is correct w.r.t. the input traces T_{in}

$$\Psi(P, T_{in}) \equiv$$

At any point in the synthesis, Syren's candidate program is correct

Definition: Program correctness

A program P is correct w.r.t. the input traces T_{in} if for every trace $\tau_i \in T_{in}$

$$\Psi(P, T_{in}) \equiv \forall \tau_i \in T_{in}$$

At any point in the synthesis, Syren's candidate program is correct

Definition: Program correctness

A program P is correct w.r.t. the input traces T_{in} if for every trace $\tau_i \in T_{in}$ there is an input σ

$$\Psi(P, T_{in}) \equiv \forall \tau_i \in T_{in} \exists \sigma$$

At any point in the synthesis, Syren's candidate program is correct

Definition: Program correctness

A program P is correct w.r.t. the input traces T_{in} if for every trace $\tau_i \in T_{in}$ there is an input σ such that $P(\sigma)$ produces τ_i .

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

Syren builds an initial program from the partial traces



Syren builds an initial program from the partial traces

```
λ.  
if (??) {  
    let x1 = StopInstances(ids=["i-029d9", ...], force=False)  
    let x2 = DescribeInstanceState(ids=["i-029d9", ...])  
    let x3 = StopInstances(ids=["i-029d9", ...], force=True)  
} else if (??) {  
    let x1 = StopInstances(ids=["i-5b289", ...], force=False)  
    let x2 = DescribeInstanceState(ids=["i-5b289", ...])  
} else {  
    let x1 = StopInstances(ids=["i-9ab4e", ...], force=False)  
    let x2 = DescribeInstanceState(ids=["i-9ab4e", ...])  
    let x3 = StopInstances(ids=["i-9ab4e", ...], force=True)  
}
```

Syren builds an initial program from the partial traces

```
λ branch.  
if (branch == 0) {  
    let x1 = StopInstances(ids=["i-029d9", ...], force=False)  
    let x2 = DescribeInstanceState(ids=["i-029d9", ...])  
    let x3 = StopInstances(ids=["i-029d9", ...], force=True)  
} else if (branch == 1) {  
    let x1 = StopInstances(ids=["i-5b289", ...], force=False)  
    let x2 = DescribeInstanceState(ids=["i-5b289", ...])  
} else {  
    let x1 = StopInstances(ids=["i-9ab4e", ...], force=False)  
    let x2 = DescribeInstanceState(ids=["i-9ab4e", ...])  
    let x3 = StopInstances(ids=["i-9ab4e", ...], force=True)  
}
```

Syren's initial program is correct by construction

```
λ branch.  
  
if (branch == 0) {  
    let x1 = StopInstances(ids=["i-029d9", ...], force=False)  
    let x2 = DescribeInstanceState(ids=["i-029d9", ...])  
    let x3 = StopInstances(ids=["i-029d9", ...], force=True)  
}  
else if (branch == 1) {  
    let x1 = StopInstances(ids=["i-5b289", ...], force=False)  
    let x2 = DescribeInstanceState(ids=["i-5b289", ...])  
}  
else {  
    let x1 = StopInstances(ids=["i-9ab4e", ...], force=False)  
    let x2 = DescribeInstanceState(ids=["i-9ab4e", ...])  
    let x3 = StopInstances(ids=["i-9ab4e", ...], force=True)  
}
```

Definition: Program correctness

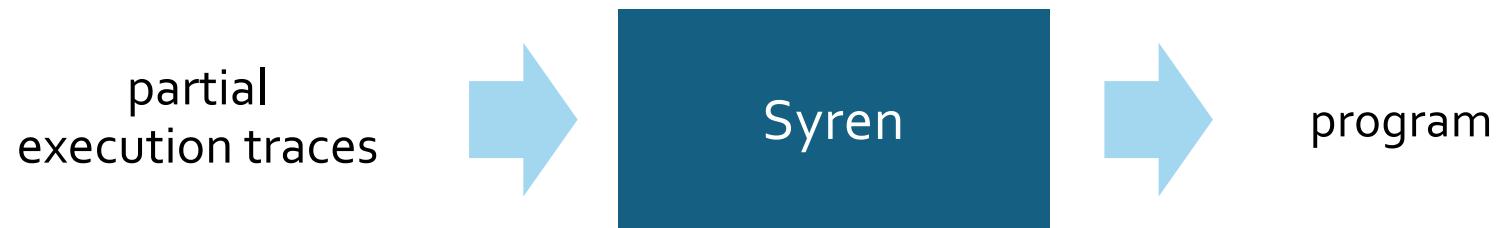
$$\Psi(P, T_{in}) \equiv \forall \tau_i \in T_{in} \exists \sigma P(\sigma) = \tau_i$$

The initial program is correct

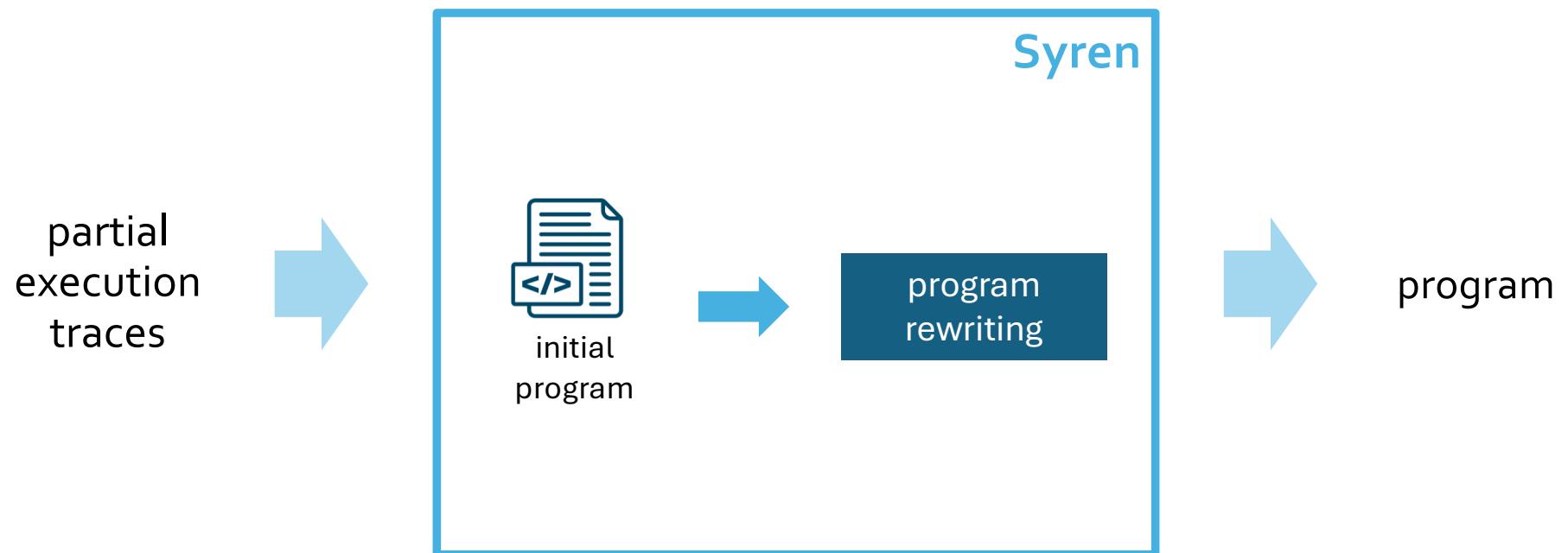
branch = 0 produces the first trace
branch = 1 produces the second trace
branch = 2 produces the third trace

But that is not enough!

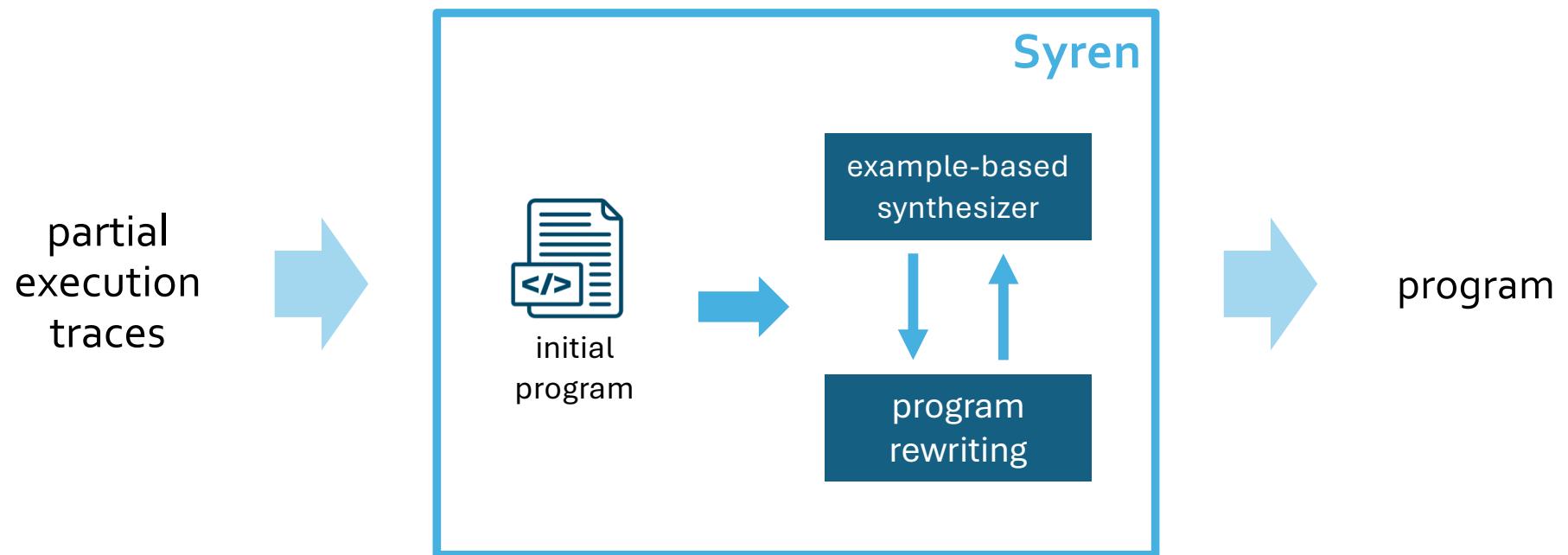
Syren's Synthesis Pipeline



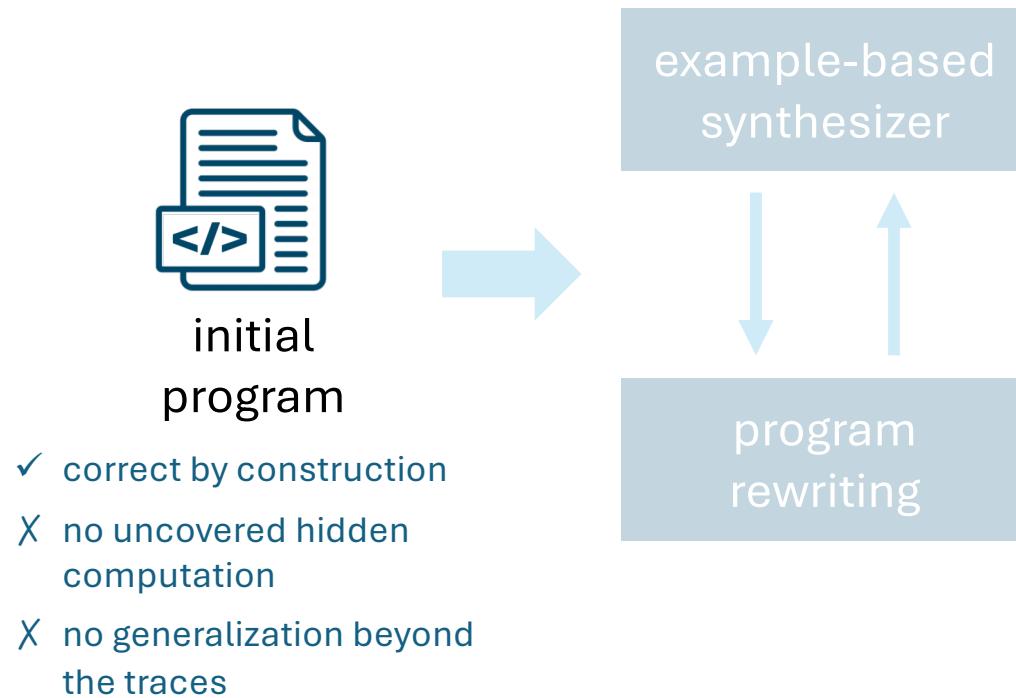
Syren's Synthesis Pipeline



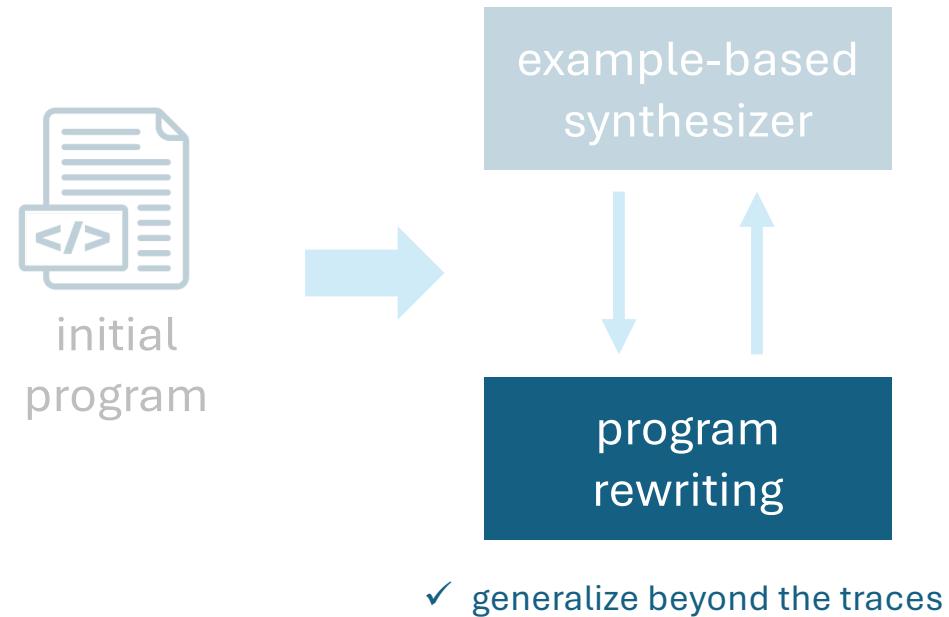
Syren's Synthesis Pipeline



Syren's Synthesis Pipeline



Syren's Synthesis Pipeline



Syren performs a search over a library of program optimizing rewrites

Each rewrite

- maintains program correctness
- improves the program by lowering a cost metric

Syren implements cost metrics that:

- Penalize syntactic complexity
- Incentivize reuse across traces

Syren applies compiler-like cost-reducing rewrites

```
λ branch.  
  if (branch == 0) {  
    let x1 = StopInstances(ids=["i-029d9", ...], force=False)  
    let x2 = DescribeInstanceState(ids=["i-029d9", ...])  
    let x3 = StopInstances(ids=["i-029d9", ...], force=True)  
  } else if (branch == 1) {  
    let x1 = StopInstances(ids=["i-5b289", ...], force=False)  
    let x2 = DescribeInstanceState(ids=["i-5b289", ...])  
  } else {  
    let x1 = StopInstances(ids=["i-9ab4e", ...], force=False)  
    let x2 = DescribeInstanceState(ids=["i-9ab4e", ...])  
    let x3 = StopInstances(ids=["i-9ab4e", ...], force=True)  
  }
```

Syren applies compiler-like cost-reducing rewrites

```
λ branch.  
  if (branch == 0) {  
    let c1 = ["i-029d9", ...]  
    let x1 = StopInstances(ids=c1, force=False)  
    let x2 = DescribeInstanceState(ids=c1)  
    let x3 = StopInstances(ids=c1, force=True)  
  } else if (branch == 1) {  
    let c1 = ["i-5b289", ...]  
    let x1 = StopInstances(ids=c1, force=False)  
    let x2 = DescribeInstanceState(ids=c1)  
  } else {  
    ...  
  }
```



Lower syntactic complexity

Syren applies compiler-like cost-reducing rewrites

```
λ branch.  
  if (branch == 0) {  
    let c1 = ["i-029d9", ...]  
    let x1 = StopInstances(ids=c1, force=False)  
    let x2 = DescribeInstanceState(ids=c1)  
    let x3 = StopInstances(ids=c1, force=True)  
  } else if (branch == 1) {  
    let c1 = ["i-5b289", ...]  
    let x1 = StopInstances(ids=c1, force=False)  
    let x2 = DescribeInstanceState(ids=c1)  
  } else {  
    ...  
  }
```

Syren applies compiler-like cost-reducing rewrites

```
λ branch.  
  if (branch == 0) { let c1 = ["i-029d9", ...] }  
  else if (branch == 1) { let c1 = ["i-5b289", ...] }  
  else { let c1 = ["i-9ab4e", ...] }  
  
  let x1 = StopInstances(ids=c1, force=False)  
  let x2 = DescribeInstanceState(ids=c1)  
  
  if (branch == 0) {  
    let x3 = StopInstances(ids=c1, force=True)  
  } else if (branch == 1) {} else {  
    let x3 = StopInstances(ids=c1, force=True)  
  }
```

 Lower syntactic complexity

Syren's rewrites generalize the program

```
λ branch.  
  if (branch == 0) { let c1 = ["i-029d9", ...] }  
  else if (branch == 1) { let c1 = ["i-5b289", ...] }  
  else { let c1 = ["i-9ab4e", ...] }
```



c1 value should be
an input to the script

```
let x1 = StopInstances(ids=c1, force=False)  
let x2 = DescribeInstanceState(ids=c1)  
  
if (branch == 0) {  
  let x3 = StopInstances(ids=c1, force=True)  
} else if (branch == 1) {} else {  
  let x3 = StopInstances(ids=c1, force=True)  
}
```

Syren's rewrites generalize the program

```
λ branch, i1.  
  let x1 = StopInstances(ids=i1, force=False)  
  let x2 = DescribeInstanceState(ids=i1)  
  
  if (branch == 0) {  
    let x3 = StopInstances(ids=i1, force=True)  
  } else if (branch == 1) {} else {  
    let x3 = StopInstances(ids=i1, force=True)  
  }
```

-  Lower syntactic complexity
-  Removes dependency on branch

⇒ More general program

Syren's rewrites generalize the program

```
λ branch, i1.  
  let x1 = StopInstances(ids=i1, force=False)  
  let x2 = DescribeInstanceState(ids=i1)  
  
  if (branch == 0) {  
    let x3 = StopInstances(ids=i1, force=True)  
  } else if (branch == 1) {} else {  
    let x3 = StopInstances(ids=i1, force=True)  
  }
```



Can we infer the user's intention from the data we can see?

Syren's rewrites generalize the program

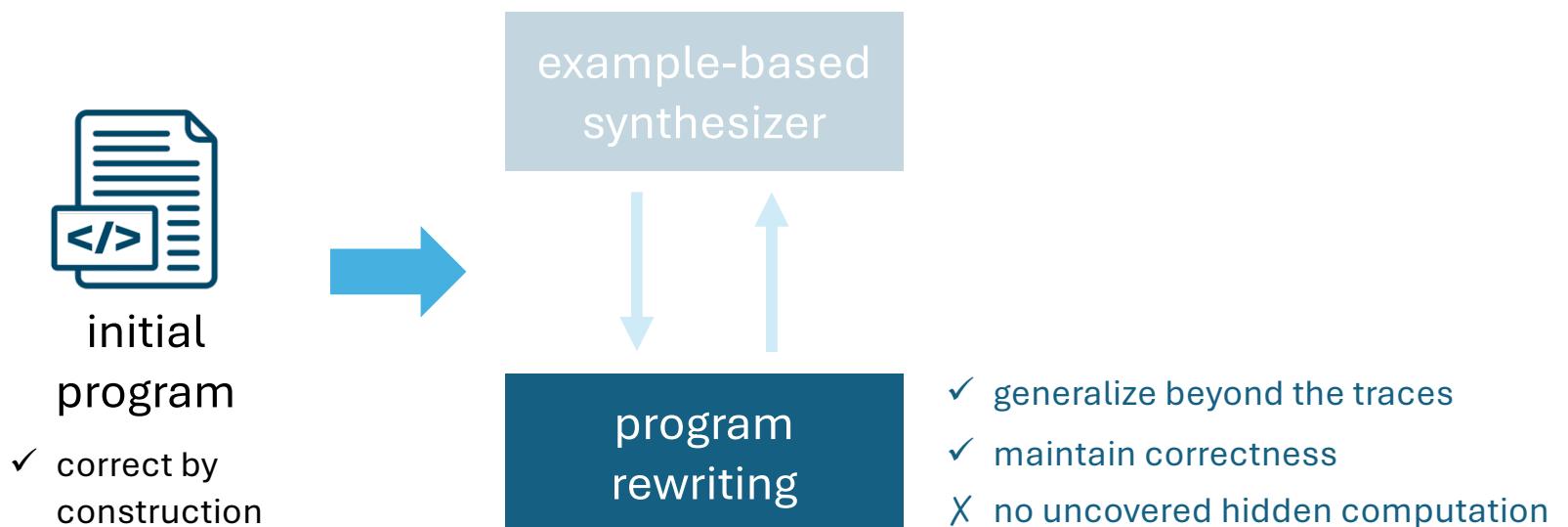
```
λ i1.  
  let x1 = StopInstances(ids=i1, force=False)  
  let x2 = DescribeInstanceState(ids=i1)  
  
  let x4 = _f(i1, x1, x2)  
  if x4 {  
    let x3 = StopInstances(ids=i1, force=True)  
  }  
  where  
    _f := ??
```



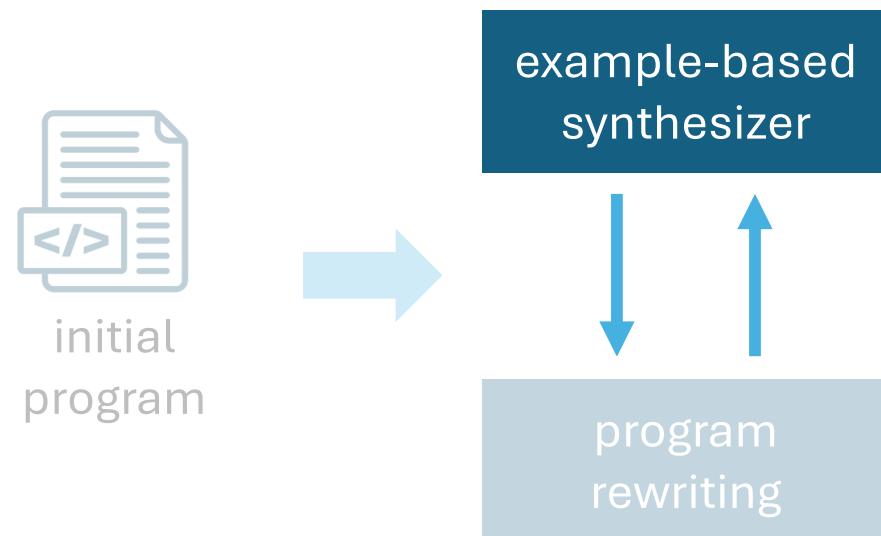
- Lower syntactic complexity
- Removes dependency on branch

⇒ More general program

Syren's Synthesis Pipeline



Syren's Synthesis Pipeline



The last rewrite introduced an undefined function

_f

```
λ i1.  
let x1 = StopInstances(ids=i1, force=False)  
let x2 = DescribeInstanceState(ids=i1)  
  
let x4 = _f(i1, x1, x2).....  
if x4 {  
    let x3 = StopInstances(ids=i1, force=True)  
}  
where  
_f := ??
```

_f is a new function that takes as input every constant defined in the program up until that point.
How do we find an implementation for _f?

The last rewrite introduced an undefined function

_f

```
λ i1.  
let x1 = StopInstances(ids=i1, force=False)  
let x2 = DescribeInstanceState(ids=i1)  
  
let x4 = _f(i1, x1, x2) .....  
if x4 {  
    let x3 = StopInstances(ids=i1, force=True)  
}  
  
where  
_f := ??
```

With _f is such that:

$_f(\tau_0[i1], \tau_0[x1], \tau_0[x2])$
= True

$_f(\tau_1[i1], \tau_1[x1], \tau_1[x2])$
= False

$_f(\tau_2[i1], \tau_2[x1], \tau_2[x2])$
= True

... for all input traces τ_i

We synthesize $_f$ from input-output examples

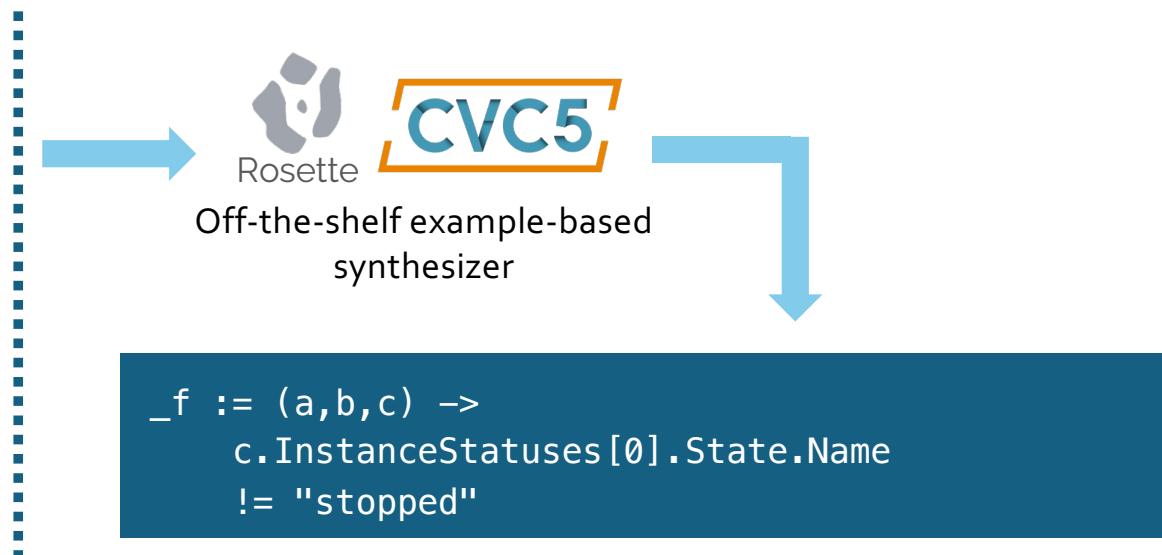
$_f$ is such that:

```
 $\_f(\tau_0[i1], \tau_0[x1], \tau_0[x2])$ 
= True
```

```
 $\_f(\tau_1[i1], \tau_1[x1], \tau_1[x2])$ 
= False
```

```
 $\_f(\tau_2[i1], \tau_2[x1], \tau_2[x2])$ 
= True
```

... for all input traces τ_i

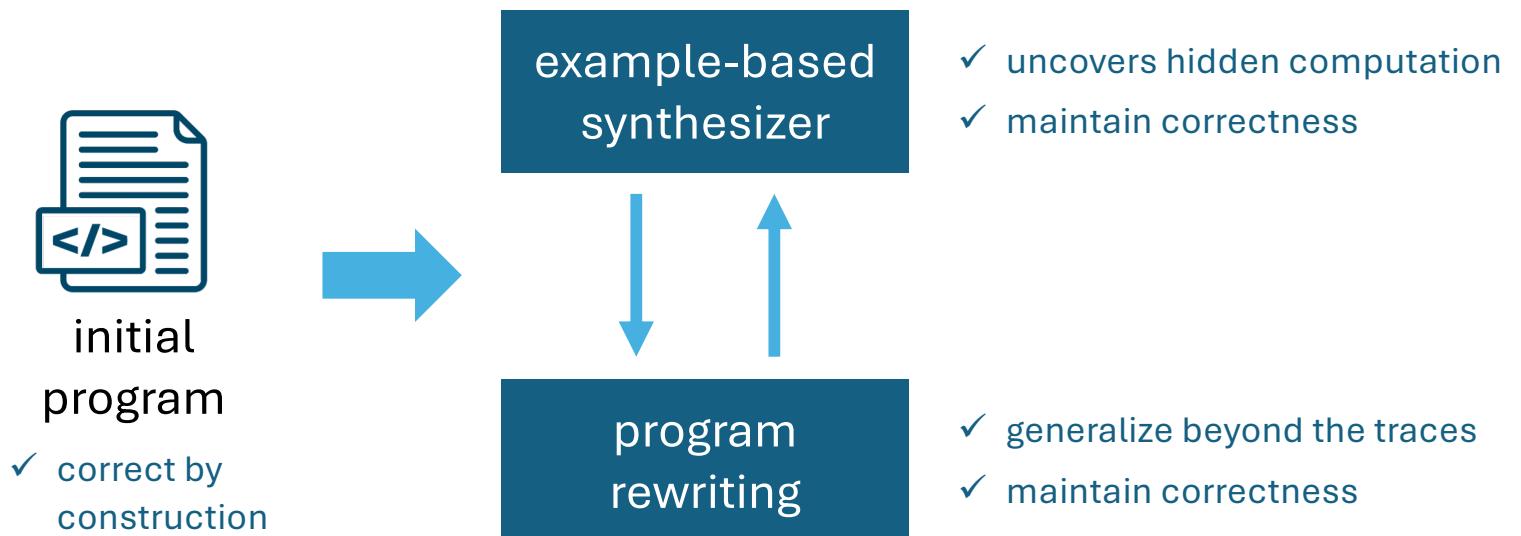


We synthesize **f** from input-output examples

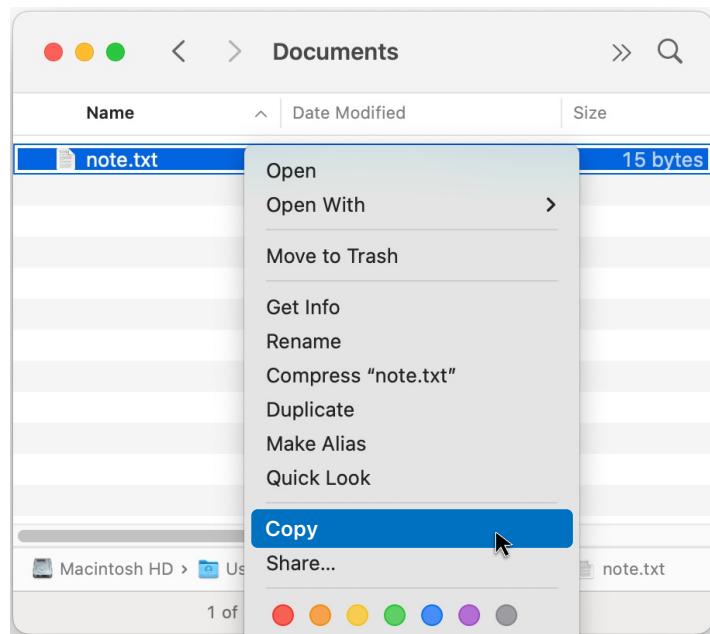
```
λ i1.  
let x1 = StopInstances(ids=i1, force=False)  
let x2 = DescribeInstanceState(ids=i1) ←  
let x4 = _f(i1, x1, x2)  
if x4 {  
    let x3 = StopInstances(ids=i1, force=True)  
}  
where  
_f := (a,b,c) ->  
    c.InstanceStatuses[0].State.Name // selects the third input and extracts the status  
    != "stopped"
```

Instance ID	Instance state	?
i-029d9	⊖ Stopped	?

Syren's Synthesis Pipeline

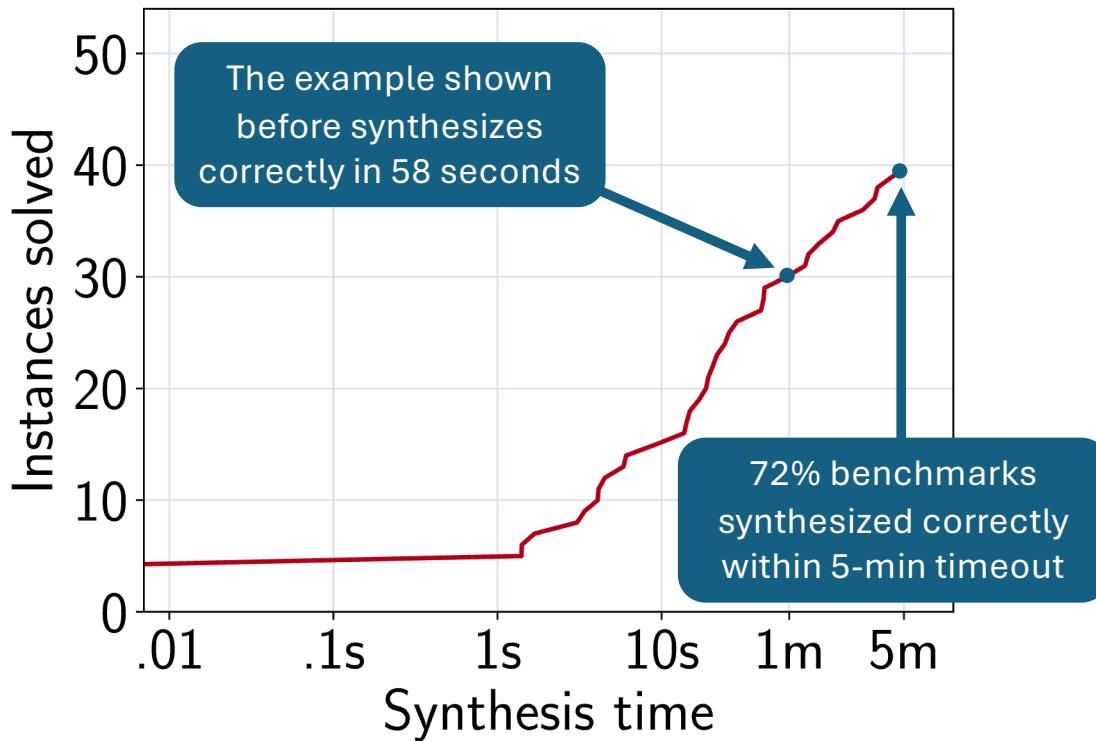


Syren can be used beyond traces of API calls: Filesystem manipulations get recorded as system traces



```
[...] mkdir      [...]/Documents/NewFolder    0.000007
[...] setattrlist [...]/Documents/NewFolder 0.000005
[...] setattrlist [...]/Documents/NewFolder 0.000006
[...] open       [...]/Documents/NewFolder/note.txt 0.000023
[...] write      [...]/Documents/NewFolder/note.txt 0.000032
[...] fsync      [...]/Documents/NewFolder/note.txt 0.000048
[...] close      [...]/Documents/NewFolder/note.txt 0.000005
[...] open       [...]/Documents/NewFolder/note.txt 0.000009
[...] read       [...]/Documents/NewFolder/note.txt 0.000011
[...] open       [...]/Desktop/note copy.txt   0.000007
[...] write      [...]/Desktop/note copy.txt   0.000014
[...] fsync      [...]/Desktop/note copy.txt   0.000006
[...] close      [...]/Desktop/note copy.txt   0.000004
```

Evaluation overview



54 benchmarks

- Cloud automation, filesystem manipulation, and document editing scripts
- AWS cloud automation scripts, Blink automations, related work

Synthesized programs with

- ≤ 4 control flow structures (conditionals and loops)
- ≤ 2 hidden functions synthesized from input-output examples

LLM (Claude 3.5 Sonnet) found the intended program for only 53% of the benchmarks

Program Synthesis from Partial Traces

We presented Syren, a synthesis method that enriches compiler-like optimizing rewrites with calls to an off-the-shelf example-based synthesizer to uncover control flow and hidden functions, and produce general-purpose scripts from partial traces of their execution.

Do you have an application
where Syren may be applied?

Margarida Ferreira
margarida@cmu.edu

