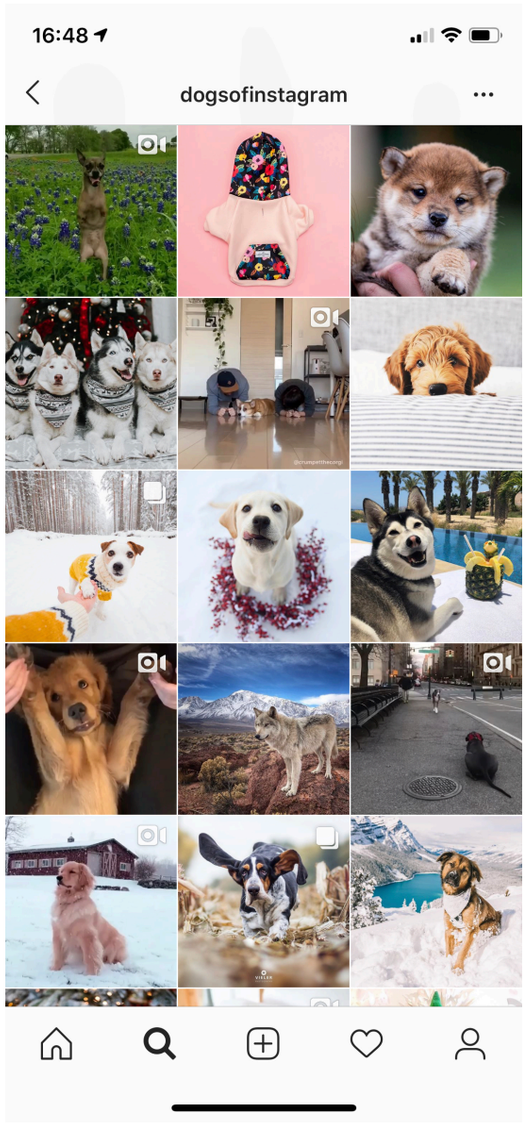
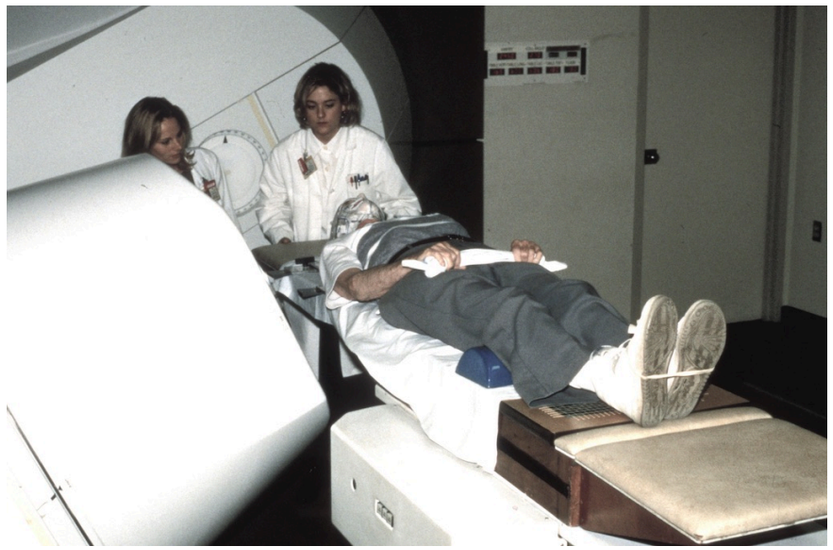
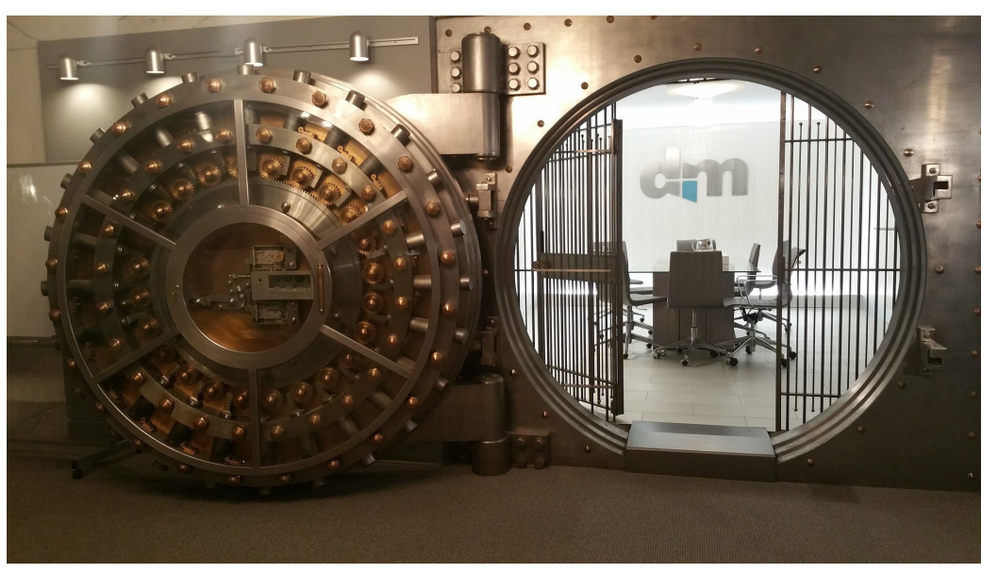
Optimizing the Automated Programming Stack

James Bornholt
University of Washington

Software is everywhere







Bugs are everywhere

7,768 views | Nov 20, 2018, 10:25am

Facebook And Instagram Go Down In Second Snag This Week

Janet Burns
Consumer Tech
I cover AI, cybersecurity, culture, drugs, and more.

Global Facebook users have reported outages this morning on the web's largest social media platform, as well as on its sister platform Instagram.

On Tuesday, social media users took to Twitter and other sites to report frequent log-in and site loading issues on Facebook, which the company says it is currently working to fix.

Bugs are everywhere

7,768 views | Nov 20, 2018, 10:25am

Facebook And Instagram Go Down In Second Snag This Week

Janet Burn
Consumer Te
I cover AI, cyb

Global Facebook us
largest social media

On Tuesday, social
frequent log-in and
says it is currently w

Facebook Security Breach Exposes Accounts of 50 Million Users

By Mike Isaac and Sheera Frenkel

Sept. 28, 2018

288

SAN FRANCISCO
handles the privat
on its computer ne
50 million users.

The breach, which
company's 14-year
code to gain acces

ext4 and data loss

By **Jonathan Cor**
March 11, 2009

example: Ubuntu's
luckless ext4 user r

Today, I was
system crash
much any file
was 0 bytes.

GLOBAL 500

Here's How IBM Crashed Australia's First Online Census

By **REUTERS** November 25, 2016

(SYDNEY) – International Business Machines (IBM, +0.88%) failed in its handling of the A$10 million ($7.4 million) IT contract for Australia's first predominantly online census, Australian Prime Minister Malcolm Turnbull said on Friday.

Automated programming tools

Automated programming tools

Automated programming successes

Verified
SQL optimizers
[Chu et al, VLDB'18]

Synthesized
network configs
[McClurg et al, PLDI'15]

Synthesized
crypto primitives
[Erbsen et al, Oakland'19]

Verified
operating systems
[Nelson et al, SOSP'17]

Synthesized
biology experiments
[Köksal et al, POPL'13]

Synthesized
memory models
[Bornholt et al, PLDI'17]

Synthesized
educational models
[Butler et al, VMCAI'18]

Challenges in automated programming

Intractability

Most problems in automated programming are intractable (many undecidable).

Specification

Automated programming requires a specification, which is often difficult to construct and audit.

Challenges in automated programming

Intractability

Most problems in automated programming are intractable (many undecidable).

Specification

Automated programming requires a specification, which is often difficult to construct and audit.

Domain specialization

Specialization reduces the size of the search space, eliminating irrelevant programs/behaviors.

Specialization allows for concise and expressive specifications that capture programmer intent.

Automated programming stack

Domain-specific tools

Automated programming stack

Domain-specific tools

SAT/SMT solving
improvements in scale and expressiveness

Automated programming stack

Domain-specific tools

Symbolic evaluation
algorithms to translate programs to SAT/SMT

SAT/SMT solving
improvements in scale and expressiveness

Automated programming stack

Domain-specific tools

Solver-aided languages
front-end abstractions for verification/synthesis

Symbolic evaluation
algorithms to translate programs to SAT/SMT

SAT/SMT solving
improvements in scale and expressiveness

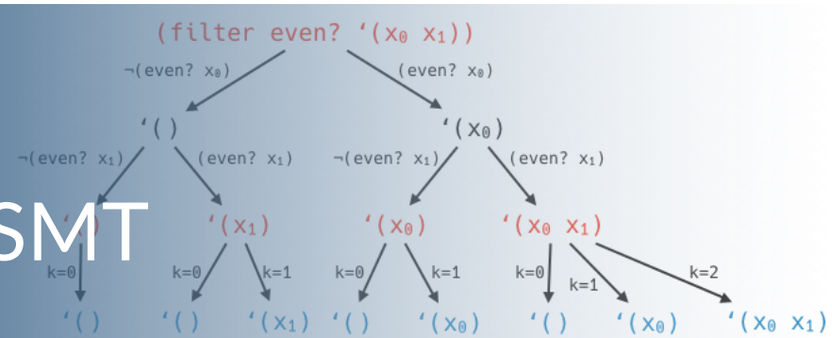
Automated programming stack

Domain-specific tools

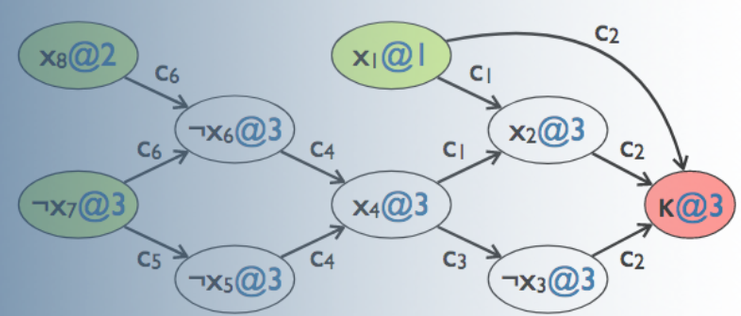
Solver-aided languages
front-end abstractions for verification/synthesis

```
(define (interpret prog inputs)
  (make-registers prog inputs)
  (for ([stmt prog])
    (match stmt
      [(list out opcode in ...)
       (define op (eval opcode))
       (define args (map load in))
       (store out (apply op args))]))
  (load (last)))
```

Symbolic evaluation
algorithms to translate programs to SAT/SMT



SAT/SMT solving
improvements in scale and expressiveness



New abstractions and tools can empower programmers to build specialized automated programming tools that improve software reliability.

MemSynth [PLDI'17]: an automated tool for synthesizing memory consistency models

Ferrite [ASPLOS'16]: a tool for synthesizing crash-safe file system code

New abstractions and tools can empower programmers to build specialized automated programming tools that improve software reliability.

MemSynth [PLDI'17]: an automated tool for synthesizing memory consistency models

Ferrite [ASPLOS'16]: a tool for synthesizing crash-safe file system code

New abstractions and tools can empower programmers to build specialized automated programming tools that improve software reliability.

Metasketches [POPL'16]: a strategy abstraction for synthesis problems

SymPro [OOPSLA'18]: a technique for systematically building scalable tools

MemSynth [PLDI'17]: an automated tool for synthesizing memory consistency models

Ferrite [ASPLOS'16]: a tool for synthesizing crash-safe file system code

New abstractions and tools can empower programmers to build specialized automated programming tools that improve software reliability.

Metasketches [POPL'16]: a strategy abstraction for synthesis problems

SymPro [OOPSLA'18]: a technique for systematically building scalable tools

Automated tools are ***worth building***

The case of memory models [PLDI'17]

Building them can be ***made systematic***

Symbolic profiling [OOPSLA'18]

The future is ***more automation***

Automating the automated programming stack

Automated tools are ***worth building***

The case of memory models [PLDI'17]

Building them can be ***made systematic***

Symbolic profiling [OOPSLA'18]

The future is ***more automation***

Automating the automated programming stack

Memory models define the memory ordering behavior of multiprocessors

Thread 1	Thread 2
1 `X = 1`	3 `Y = 1`
2 `if Y == 0:` `  print “hello”`	4 `if X == 0:` `  print “goodbye”`

All variables initialized to 0

Memory models define the memory ordering behavior of multiprocessors

Thread 1	Thread 2
1 `X = 1`	3 `Y = 1`
2 `if Y == 0:` `  print “hello”`	4 `if X == 0:` `  print “goodbye”`

All variables initialized to 0

Can this print… `hello`?

Memory models define the memory ordering behavior of multiprocessors

Thread 1	Thread 2
❶ `X = 1`	❸ `Y = 1`
❷ `if Y == 0:` `  print “hello”`	❹ `if X == 0:` `  print “goodbye”`

All variables initialized to 0

Can this print… `hello`? ❶ ❷ ❸ ❹

Memory models define the memory ordering behavior of multiprocessors

Thread 1	Thread 2
❶ `X = 1`	❸ `Y = 1`
❷ `if Y == 0:` `  print “hello”`	❹ `if X == 0:` `  print “goodbye”`

All variables initialized to 0

Can this print… `hello`? ❶ ❷ ❸ ❹

`goodbye`?

Memory models define the memory ordering behavior of multiprocessors

Thread 1	Thread 2
1 `X = 1`	3 `Y = 1`
2 `if Y == 0:` `  print “hello”`	4 `if X == 0:` `  print “goodbye”`

All variables initialized to 0

Can this print… `hello`? 1 2 3 4

`goodbye`? 3 4 1 2

Memory models define the memory ordering behavior of multiprocessors

Thread 1	Thread 2
❶ `X = 1`	❸ `Y = 1`
❷ `if Y == 0:` `  print “hello”`	❹ `if X == 0:` `  print “goodbye”`

All variables initialized to 0

Can this print…

`hello`? ❶ ❷ ❸ ❹

`goodbye`? ❸ ❹ ❶ ❷

nothing?

Memory models define the memory ordering behavior of multiprocessors

Thread 1	Thread 2
❶ `X = 1`	❸ `Y = 1`
❷ `if Y == 0:` `  print “hello”`	❹ `if X == 0:` `  print “goodbye”`

All variables initialized to 0

Can this print…		
	`hello`?	❶ ❷ ❸ ❹
	`goodbye`?	❸ ❹ ❶ ❷
	nothing?	❶ ❸ ❷ ❹

Memory models define the memory ordering behavior of multiprocessors

Thread 1	Thread 2
❶ `X = 1`	❸ `Y = 1`
❷ `if Y == 0:` `  print “hello”`	❹ `if X == 0:` `  print “goodbye”`

All variables initialized to 0

Can this print…

`hello`?	❶ ❷ ❸ ❹
`goodbye`?	❸ ❹ ❶ ❷
nothing?	❶ ❸ ❷ ❹
both?	

Memory models define the memory ordering behavior of multiprocessors

Thread 1	Thread 2
1 `X = 1`	3 `Y = 1`
2 `if Y == 0:` `  print “hello”`	4 `if X == 0:` `  print “goodbye”`

All variables initialized to 0

Can this print…

- `hello`? 1 2 3 4
- `goodbye`? 3 4 1 2
- nothing? 1 3 2 4
- both?  No! (sequential consistency)

Memory models define the memory ordering behavior of multiprocessors

Thread 1	Thread 2
1 `X = 1`	3 `Y = 1`
2 `if Y == 0:` `  print “hello”`	4 `if X == 0:` `  print “goodbye”`

All variables initialized to 0

Can this print…

`hello`? 1 2 3 4

goodbye? 3 4 1 2

nothing? 1 3 2 4

both?

No! (sequential consistency)

Yeah!
We wanna go fast!

Memory models define the memory ordering behavior of multiprocessors

…correctness of my compiler…

Compiler writers 👨‍💻

…rules to verify against…

Verification tools ✅

…possible low-level behaviors…

Kernel/library developers 👩‍🔬

Memory models define the memory ordering behavior of multiprocessors

…correctness of my compiler…

Compiler writers 👨‍💻

…rules to verify against…

Verification tools ✅

…possible low-level behaviors…

Kernel/library developers 👩‍🔬

Litmus tests and prose

Memory models define the memory ordering behavior of multiprocessors

…correctness of my compiler…

Compiler writers 👨‍💻

…rules to verify against…

Verification tools ✅

…possible low-level behaviors…

Kernel/library developers 👩‍🔬

Litmus tests and prose

$$\forall \vee \Rightarrow \bowtie \exists \in \cap \subset \wedge \cup$$

Formal specifications

Memory models define the memory ordering behavior of multiprocessors

…correctness of my compiler…

Compiler writers 👨‍💻

…rules to verify against…

Verification tools ✅

…possible low-level behaviors…

Kernel/library developers 👩‍🔬

Litmus tests and prose

x86 [Sewell et al, CACM'10]

PowerPC [Alglave et al, CAV'10, etc]

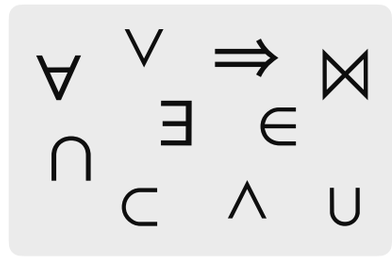
ARM [Flur et al, POPL'16]

$\forall \vee \Rightarrow \bowtie \exists \in \cap \subset \wedge \cup$

Formal specifications

MemSynth: automated programming for memory consistency models

Litmus tests

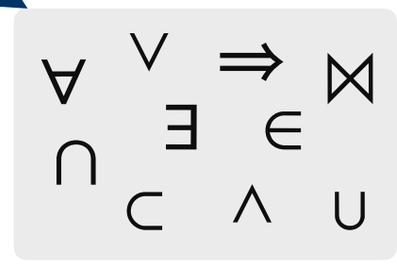


Formal specifications

MemSynth: automated programming for memory consistency models

Litmus tests

Synthesize specifications from litmus tests



Formal specifications

MemSynth: automated programming for memory consistency models

Litmus tests

Synthesize specifications from litmus tests

Detect ambiguities in synthesized models

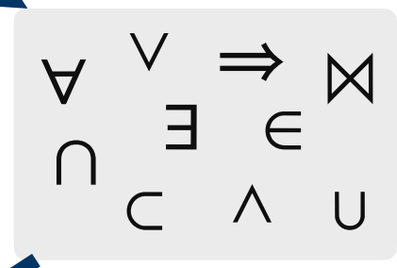
Formal specifications

MemSynth: automated programming for memory consistency models

x86: 2 seconds
PowerPC: 12 seconds

Synthesize specifications from litmus tests

Litmus tests

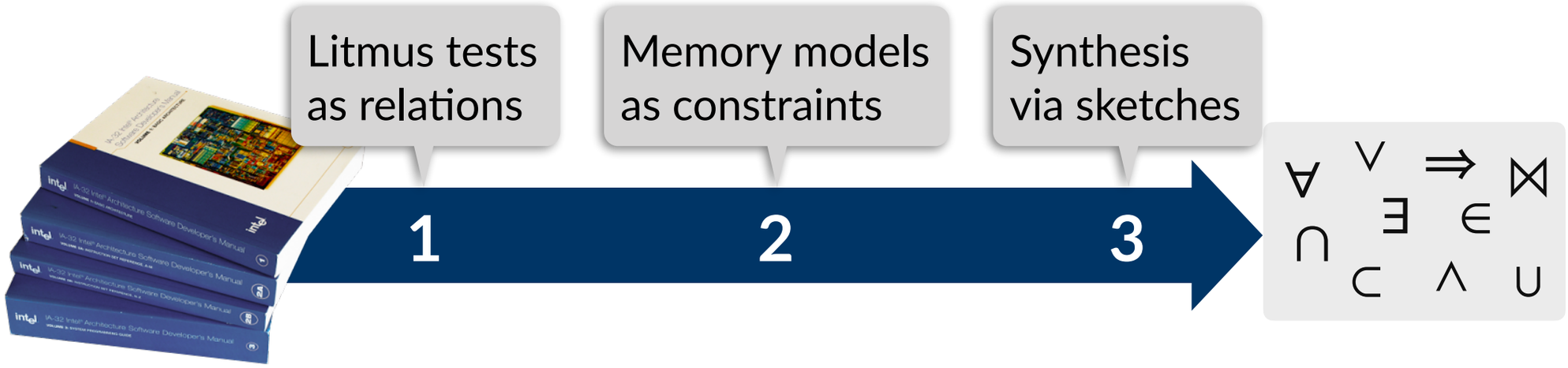


Formal specifications

Detect ambiguities in synthesized models

x86: 4 ambiguities
PowerPC: 9 ambiguities

MemSynth: automated programming for memory consistency models



Litmus tests

Formal specifications

Litmus tests as relations

Thread 1	Thread 2
❶ `X = 1`	❸ `Y = 1`
❷ `if Y == 0:` `  print “hello”`	❹ `if X == 0:` `  print “goodbye”`

All variables initialized to 0

Litmus tests as relations

Thread 1	Thread 2
❶ `X = 1`	❸ `Y = 1`
❷ `r0 = Y`	❹ `r1 = X`

All variables initialized to 0

Litmus tests as relations

Thread 1	Thread 2
❶ `X = 1`	❸ `Y = 1`
❷ `r0 = Y`	❹ `r1 = X`

All variables initialized to 0

Encode programs and behaviors as **relations** in **relational logic** (like Alloy)

Litmus tests as relations

Thread 1	Thread 2
❶ `X = 1`	❸ `Y = 1`
❷ `r0 = Y`	❹ `r1 = X`

All variables initialized to 0

Encode programs and behaviors as **relations** in **relational logic** (like Alloy)

Program relations
extracted from program text:

po = {(❶,❷), (❸,❹)}

Program order:
(*a*,*b*) ∈ po if *b* is after *a* on the same thread

Litmus tests as relations

Thread 1	Thread 2
❶ `X = 1`	❸ `Y = 1`
❷ `r0 = Y`	❹ `r1 = X`

All variables initialized to 0

Encode programs and behaviors as **relations** in **relational logic** (like Alloy)

Program relations
extracted from program text:

`po` = {(❶,❷), (❸,❹)}

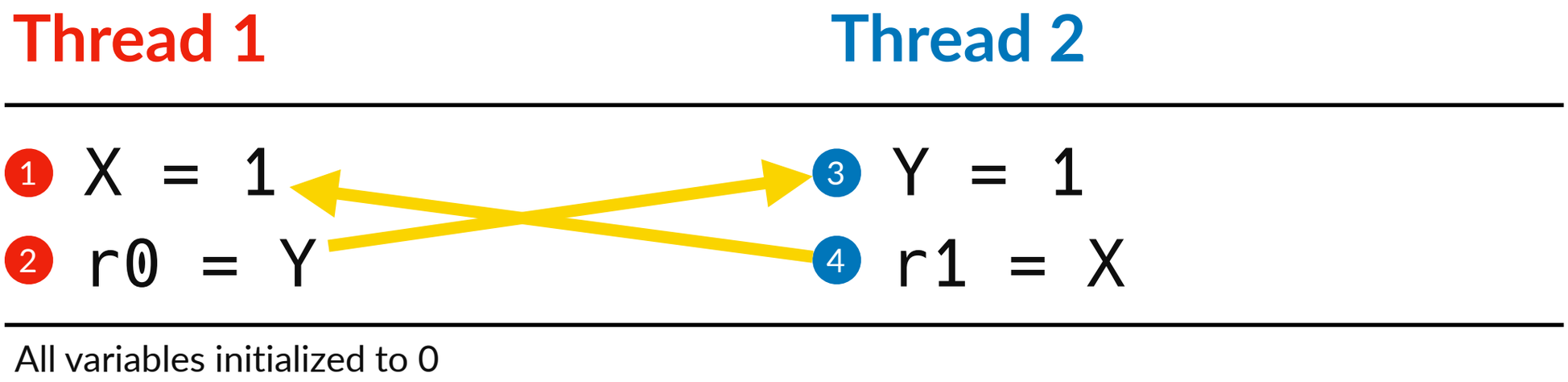
Program order:
(*a*,*b*) ∈ `po` if *b* is after *a* on the same thread

Execution relations
describe dynamic behavior:

`rf` = {(❷,❸), (❹,❶)}

Reads-from:
(*r*,*w*) ∈ `rf` if *r* reads the value written by *w*

Litmus tests as relations



Encode programs and behaviors as **relations** in **relational logic** (like Alloy)

Program relations
extracted from program text:

po = {(1,2), (3,4)}

Program order:
(*a*,*b*) ∈ po if *b* is after *a* on the same thread

Execution relations
describe dynamic behavior:

rf = {(2,3), (4,1)}

Reads-from:
(*r*,*w*) ∈ rf if *r* reads the value written by *w*

Memory models as relational constraints

Program relations
extracted from program text:

po = {(1,2), (3,4)}

Program order

Execution relations
describe dynamic behavior:

rf = {(2,3), (4,1)}

Reads-from

A memory model **constrains the allowed executions** of a program

Written as a **predicate** in relational logic

Memory models as relational constraints

Program relations
extracted from program text:

po = {(1,2), (3,4)}

Program order

Execution relations
describe dynamic behavior:

rf = {(2,3), (4,1)}

Reads-from

A memory model **constrains the allowed executions** of a program

Written as a **predicate** in relational logic

$M(T, E) \triangleq$

Memory models as relational constraints

Program relations extracted from program text:

po = {(1,2), (3,4)}

Program order

Execution relations describe dynamic behavior:

rf = {(2,3), (4,1)}

Reads-from

A memory model constrains the allowed executions of a program

Written as a predicate in relational logic

$M(T, E) \triangleq$

```
(&&
 (&&
  (&&
   (in rf (& (-> Writes Reads) (join loc (~ loc)) (join data (~ data))))
   (no (- (join rf (~ rf)) iden))
   (all ((r4 (- Reads (join Writes rf)))) (= (join r4 data) Zero)))
  (&&
   (in ws (& (-> Writes Writes) (join loc (~ loc))))
   (no (& iden ws))
   (in (join ws ws) ws)
   (all
    ((r5 Writes))
    (all ((r6 (- (& Writes (join loc (join r5 loc))) r5))) (or (in (-> r5 r6) ws) (in (-> r6 r5) ws))))
   (in ws (join loc (~ loc)))))
 (no
  (&
   (^
    (+ rf ws (+ (join (~ rf) ws) (& (-> (- Reads (join Writes rf)) Writes) (join loc (~ loc))))))
   (& po (join loc (~ loc))))))
 iden))
(no (& (^ (+ po rf)) iden))
(all
 ((r7 Writes))
 (=>
  (&& (in r7 (- (join univ ws) (join ws univ))) (some (join (join r7 loc) finalValue)))
  (= (join r7 data) (join (join r7 loc) finalValue))))
(no
 (&
  (^
   (+
    (& po dp)
    ws
    (+ (join (~ rf) ws) (& (-> (- Reads (join Writes rf)) Writes) (join loc (~ loc))))
    (-> none none)
    (+
     (^ (+ (+ (join (:> po Syncs) po) (join (join (:> po Syncs) po) rf)) (join rf (join (:> po Syncs) po))))
     (^
      (+
       (+
        (& (join (:> po Lwsyncs) po) (+ (-> Writes Writes) (-> Reads MemoryEvent)))
        (:> (join rf (& (join (:> po Lwsyncs) po) (+ (-> Writes Writes) (-> Reads MemoryEvent)))) Writes))
       (<: Reads (join (& (join (:> po Lwsyncs) po) (+ (-> Writes Writes) (-> Reads MemoryEvent))) rf)))))))
  iden)))
```

Memory models as relational constraints

Program relations
extracted from program text:

po = {(1,2), (3,4)}

Program order

Execution relations
describe dynamic behavior:

rf = {(2,3), (4,1)}

Reads-from

A memory model **constrains the allowed executions** of a program

Written as a **predicate** in relational logic

$M(T, E) \triangleq$ `(`**`no`**` (& (^ (+ po rf)) iden))`

Memory models as relational constraints

Program relations
extracted from program text:

po = {(1,2), (3,4)}

Program order

Execution relations
describe dynamic behavior:

rf = {(2,3), (4,1)}

Reads-from

A memory model **constrains the allowed executions** of a program

Written as a **predicate** in relational logic

$M(T, E) \triangleq$ `(`**`no`**` (& (^ (+ po rf)) iden))`

…by forbidding cycles involving rf ∪ po

Constraining the possible values of rf…

Memory models as relational constraints

Program relations
extracted from program text:

po = {(1,2), (3,4)}

Program order

Execution relations
describe dynamic behavior:

rf = {(2,3), (4,1)}

Reads-from

A memory model **constrains the allowed executions** of a program

Written as a **predicate** in relational logic

A memory model **allows** a test T if there exists an execution E that satisfies the predicate

$M(T, E) \triangleq$ `(no (& (^ (+ po rf)) iden))`

…by forbidding cycles involving rf ∪ po

Constraining the possible values of rf…

Synthesis from a memory model sketch

$M(T, E) \triangleq$ `(no (& (^ (+ po rf)) iden))`

Synthesis from a memory model sketch

$M(T, E) \triangleq$ `(no (& (^ (+ ?? ??)) iden))`

Expression holes for a synthesizer to complete

Synthesis from a memory model sketch

$M(T, E) \triangleq$ `(no (& (^ (+ ?? ??)) iden))`

Expression holes for a synthesizer to complete

```
po
rf
po + rf
po & rf
po - rf
...
```

Synthesis from a memory model sketch

A sketch specifies things we **know** (e.g., want a happens-before ordering)…

$M(T, E) \triangleq$ `(no (& (^ (+ ?? ??)) iden))`

Expression holes for a synthesizer to complete

```
po
rf
po + rf
po & rf
po - rf
…
```

Synthesis from a memory model sketch

A sketch specifies things we **know** (e.g., want a happens-before ordering)…

…and defines the shape of the parts we **don't know**

$M(T, E) \triangleq$ `(no (& (^ (+ ?? ??)) iden))`

Expression holes for a synthesizer to complete

```
po
rf
po + rf
po & rf
po - rf
…
```

Memory model frameworks

$M(T, E) \triangleq$ `(no (& (^ (+ ws rf ppo grf)) iden))`

[Alglave et al, CAV'10]

Memory model frameworks

$M(T, E) \triangleq$ `(no (& (^ (+ ws rf ppo grf)) iden))`

Preserved program order:
same-thread reorderings

Global reads-from:
inter-thread reorderings

[Alglave et al, CAV'10]

Memory model frameworks

$M(T, E) \triangleq$ `(no (& (^ (+ ws rf ppo grf)) iden))`

Preserved program order: same-thread reorderings

Global reads-from: inter-thread reorderings

	`ppo`	`grf`
Sequential consistency	`po`	`rf`
Total store order (x86)	`po - (Wr→Rd)`	`rf & SameThd`

[Alglave et al, CAV'10]

Memory model frameworks

$M(T, E) \triangleq$ `(no (& (^ (+ ws rf ?? ?? )) iden))`

Preserved program order: same-thread reorderings

Global reads-from: inter-thread reorderings

Sequential consistency	`po`	`rf`
Total store order (x86)	`po - (Wr→Rd)`	`rf & SameThd`

[Alglave et al, CAV'10]

Ocelot DSL for relational logic with holes

Expression holes for a synthesizer to complete

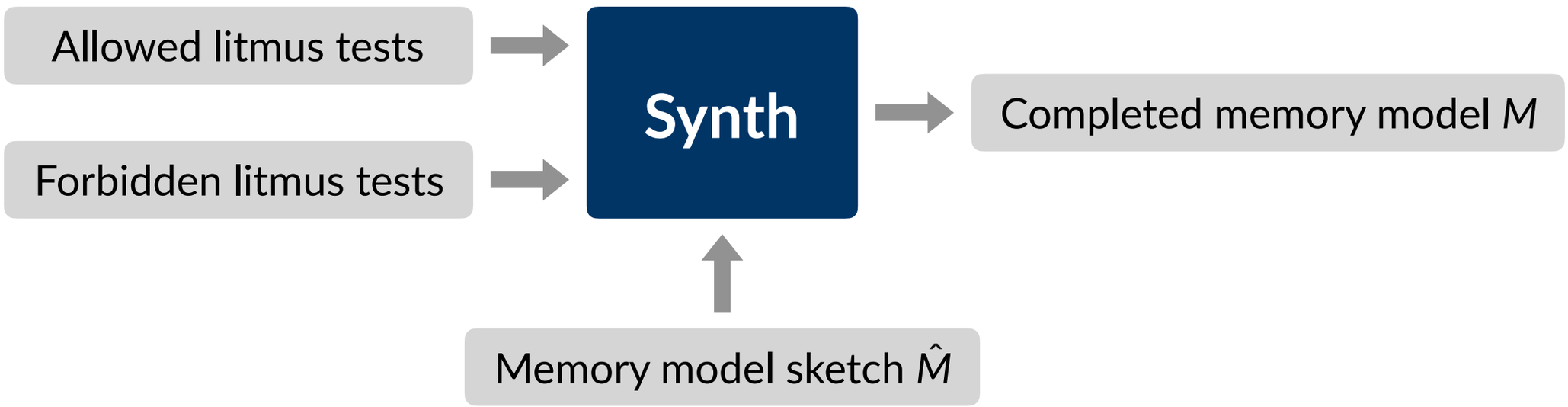
$M(T, E) \triangleq$ `(no (& (^ (+ ws rf ?? ??)) iden))`

Ocelot embeds relational logic in the **Rosette** solver-aided language [Torlak & Bodik 2014]

Also in use for SQL query synthesis and protocol reasoning

`http://ocelot.tools`

The synthesis query



The synthesis query

x86

2 allowed tests

3 5

1 2 4 6

7 8 9 10

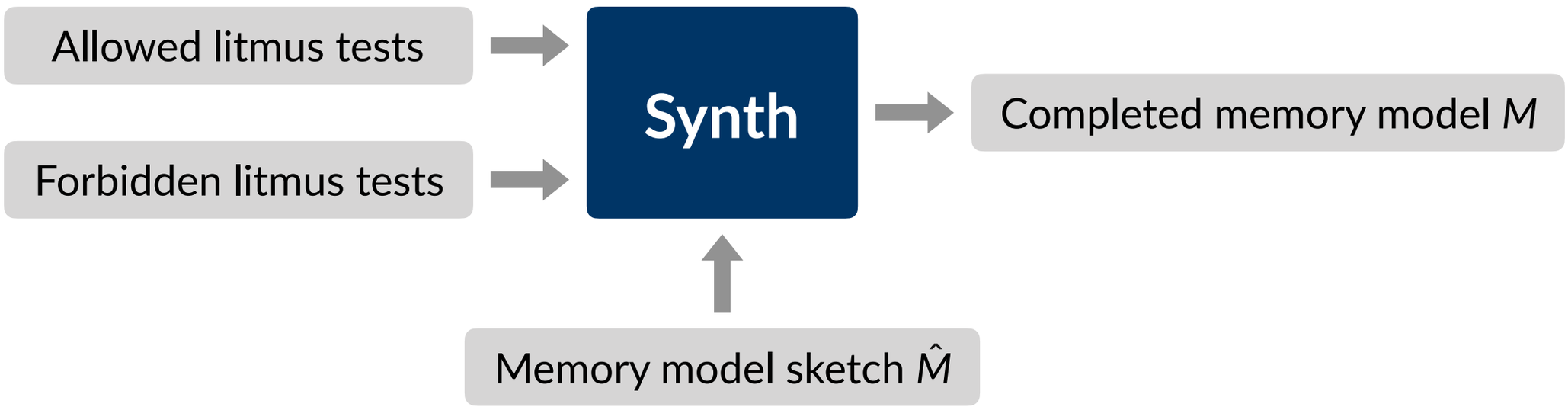
8 forbidden tests

Synth

Total store order

Memory model sketch $\hat{M}$

The synthesis query



The synthesis query

Allowed litmus tests →

Forbidden litmus tests →

$\exists\, M.\ \forall\, T \in \mathrm{T}^{+}.$ M allows T

→ Memory model M

↑ Memory model sketch $\hat{M}$

The synthesis query

Allowed litmus tests → $\exists\, M.\ \forall\, T \in \mathrm{T}^{+}.$ M allows T

Forbidden litmus tests → $\exists\, M.\ \forall\, T \in \mathrm{T}^{-}.$ M forbids T

→ Memory model M

↑ Memory model sketch $\hat{M}$

The synthesis query

Standard exists-forall quantifier pattern for synthesis

Allowed litmus tests → $\exists\, M.\ \forall\, T \in \mathrm{T}^{+}.$ M allows T

Forbidden litmus tests → $\exists\, M.\ \forall\, T \in \mathrm{T}^{-}.$ M forbids T

→ Memory model M

↑ Memory model sketch $\hat{M}$

The synthesis query

M **allows** *T*:
∃ *E*. *M*(*T*, *E*)

Standard exists-forall quantifier pattern for synthesis

Allowed litmus tests → ∃ *M*. ∀ *T*∈T⁺. M allows T

Forbidden litmus tests → ∃ *M*. ∀ *T*∈T⁻. M forbids T

→ Memory model *M*

↑ Memory model sketch $\hat{M}$

The synthesis query

Standard exists-forall quantifier pattern for synthesis

M **allows** *T*:
$\exists\, E.\ M(T, E)$

Allowed litmus tests → $\exists\, M.\ \forall\, T \in \mathsf{T}^{+}.\ \exists\, \mathsf{E}.\ M(T,E)$

Forbidden litmus tests → $\exists\, M.\ \forall\, T \in \mathsf{T}^{-}.\ \forall\, \mathsf{E}.\ \neg M(T,E)$

→ Memory model *M*

↑ Memory model sketch $\hat{M}$

The synthesis query

M **allows** *T*:
$\exists\, E.\ M(T, E)$

Standard exists-forall quantifier pattern for synthesis

Allowed litmus tests → $\exists\, M.\ \forall\, T \in \mathsf{T}^{+}.\ \exists\, \mathrm{E}.\ M(T,E)$ → Memory model *M*

Forbidden litmus tests → $\exists\, M.\ \forall\, T \in \mathsf{T}^{-}.\ \forall\, \mathrm{E}.\ \neg M(T,E)$

Higher-order quantification over relations! 😱

Memory model sketch $\hat{M}$

The synthesis query

M **allows** *T*:
$\exists\, E.\ M(T, E)$

Allowed litmus tests → $\exists\, M.\ \forall\, T \in \mathsf{T}^{+}.\ \exists\, \mathrm{E}.\ M(T,E)$

Forbidden litmus tests → $\exists\, M.\ \forall\, T \in \mathsf{T}^{-}.\ \forall\, \mathrm{E}.\ \neg M(T,E)$

→ Memory model *M*

↑ Memory model sketch $\hat{M}$

The synthesis query

M **allows** *T*:
$\exists\, E.\ M(T, E)$

Allowed litmus tests → $\exists\, M.\ \forall\, T \in \mathsf{T}^{+}.\ \exists\, \mathrm{E}.\ M(T,E)$ → Memory model *M*

Forbidden litmus tests → $\exists\, M.\ \forall\, T \in \mathsf{T}^{-}.\ \forall\, \mathrm{E}.\ \neg M(T,E)$

Memory model sketch $\hat{M}$

Handled by a quantified boolean formula (QBF) solver

The synthesis query

M **allows** *T*:
$\exists\, E.\ M(T, E)$

Allowed litmus tests → $\exists\, M.\ \forall\, T \in \mathsf{T}^{+}.\ \exists\, \mathrm{E}.\ M(T,E)$ → Memory model *M*

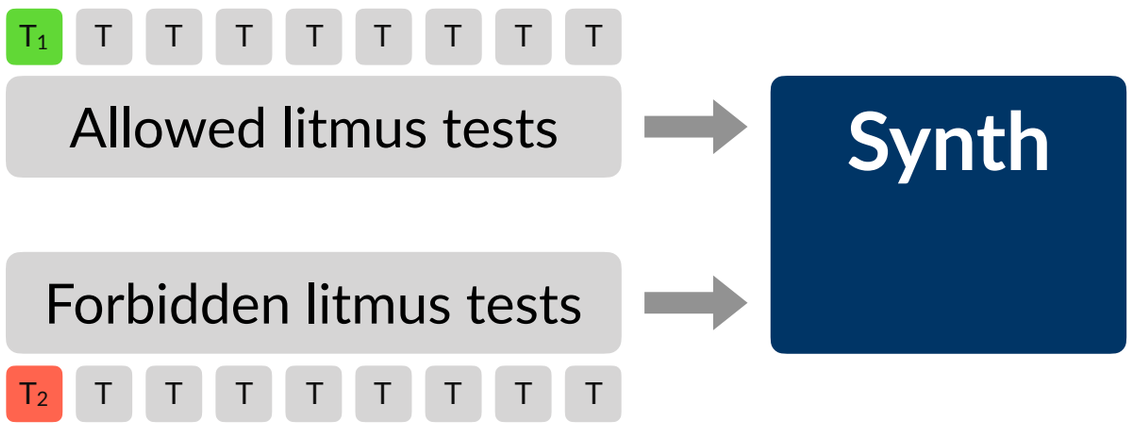
Forbidden litmus tests → $\exists\, M.\ \forall\, T \in \mathsf{T}^{-}.\ \forall\, \mathrm{E}.\ \neg M(T,E)$

Handled by incremental synthesis engine

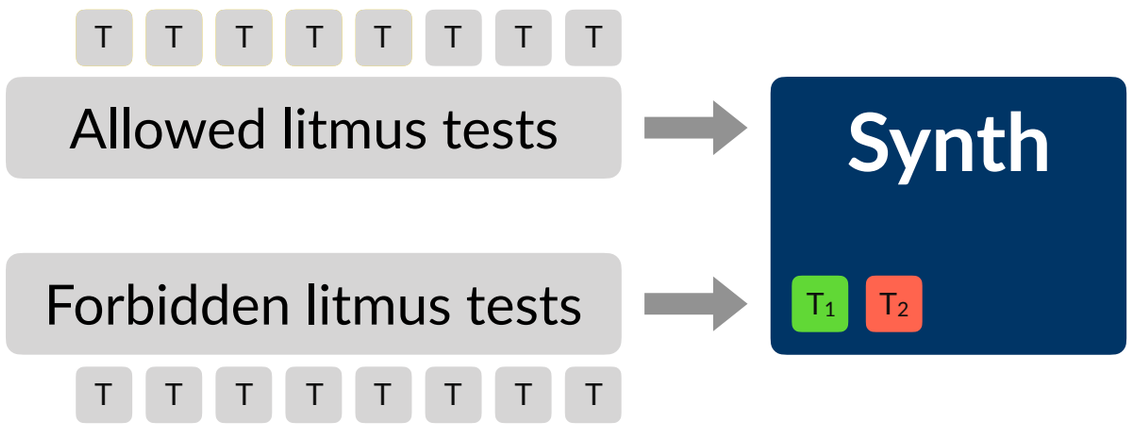
Memory model sketch $\hat{M}$

Handled by a quantified boolean formula (QBF) solver

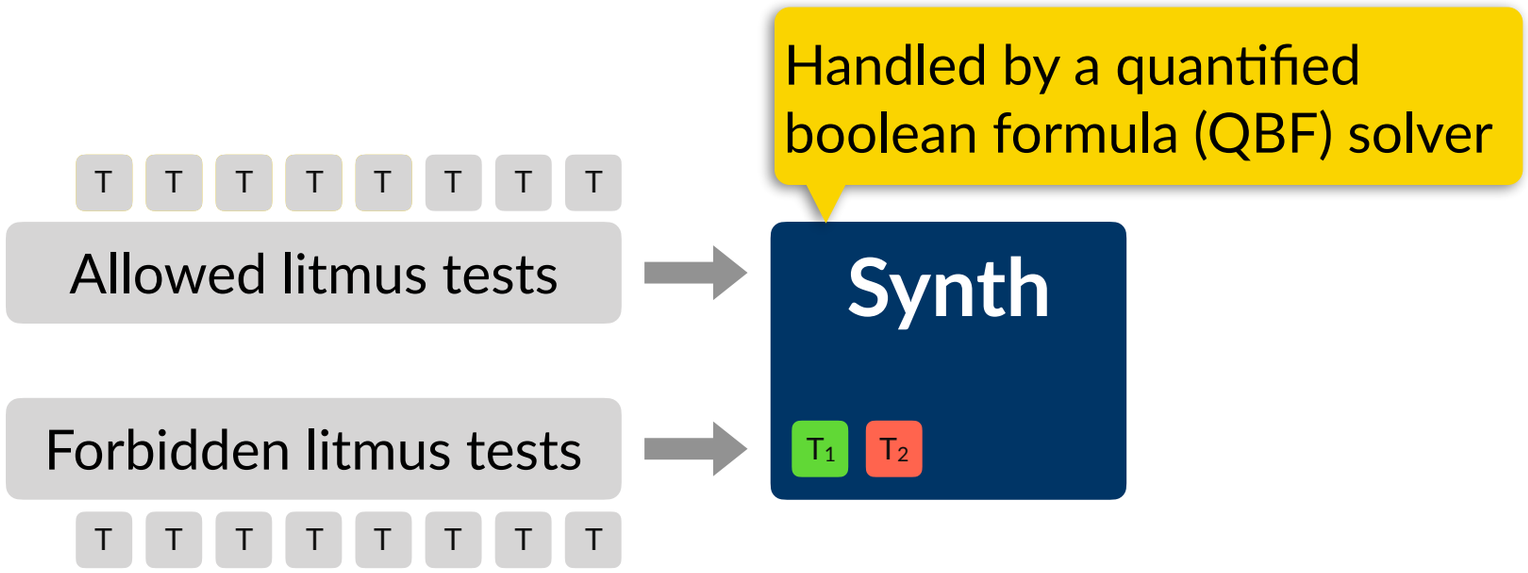
Incremental synthesis



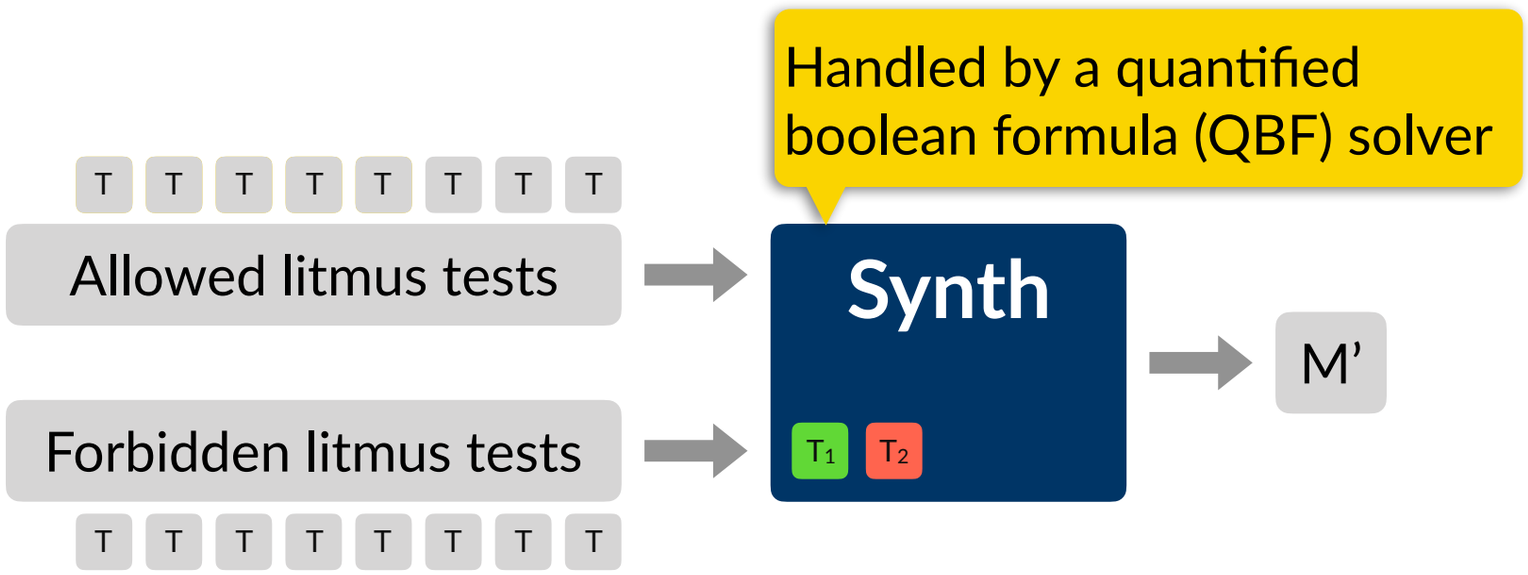
Incremental synthesis



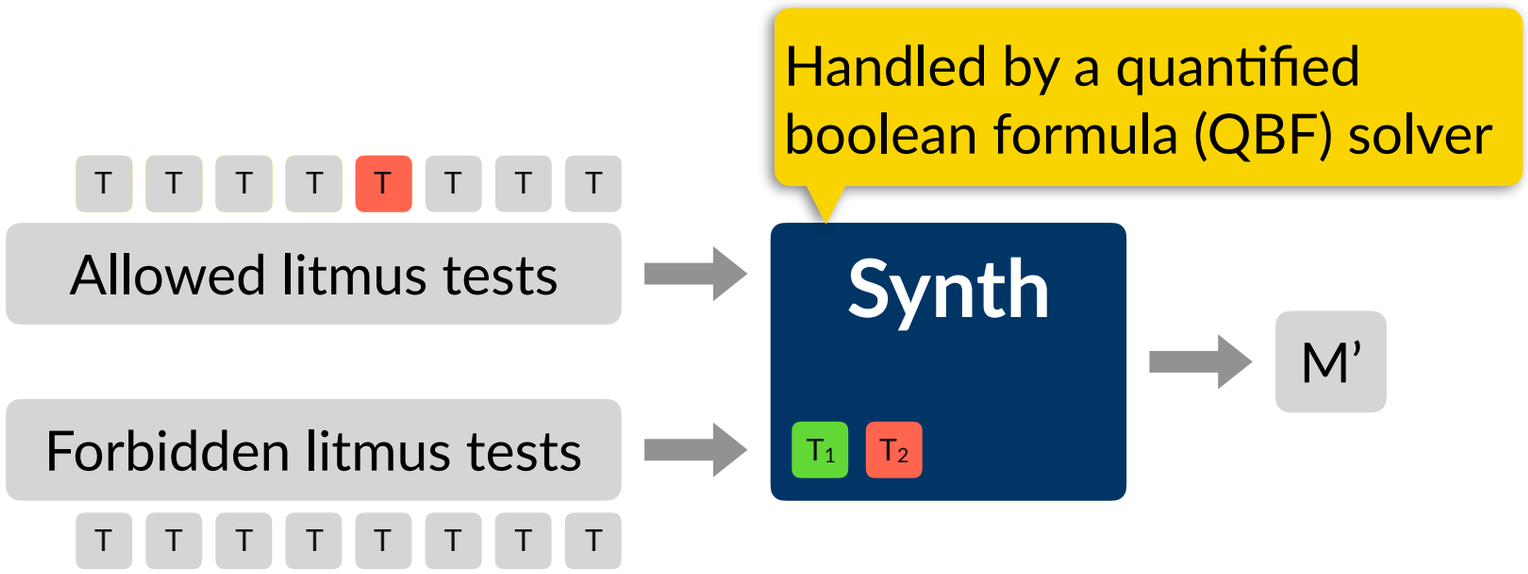
Incremental synthesis



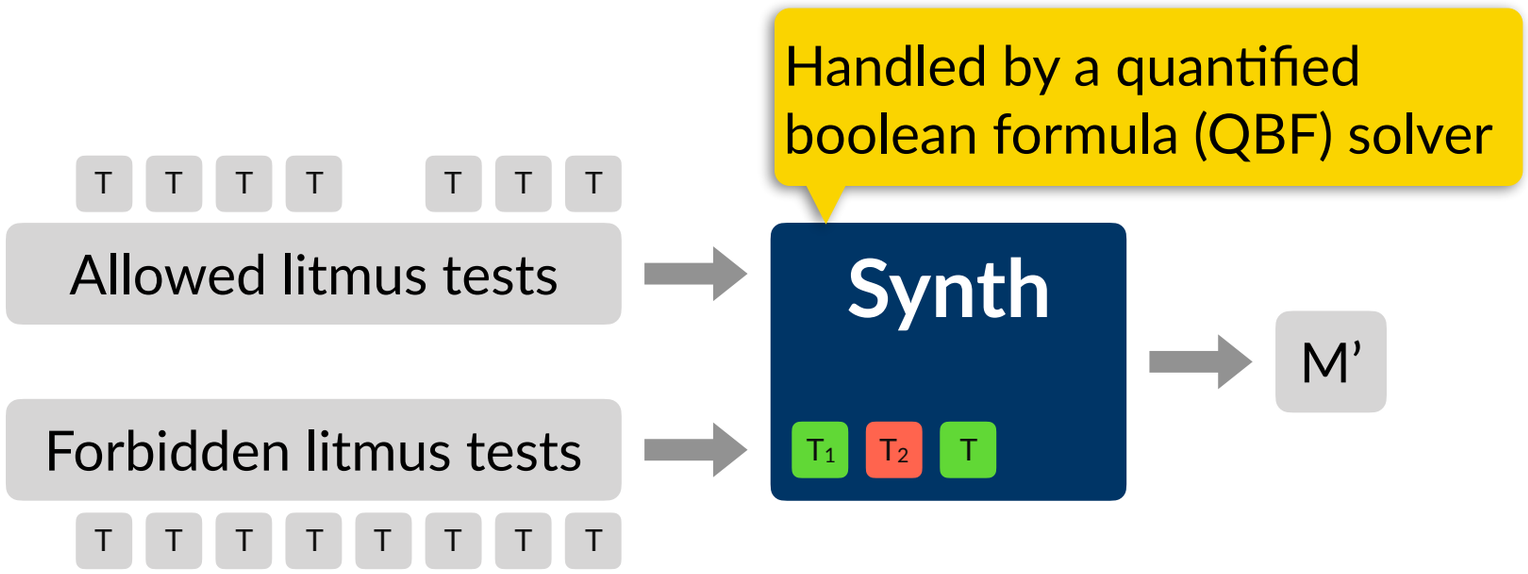
Incremental synthesis



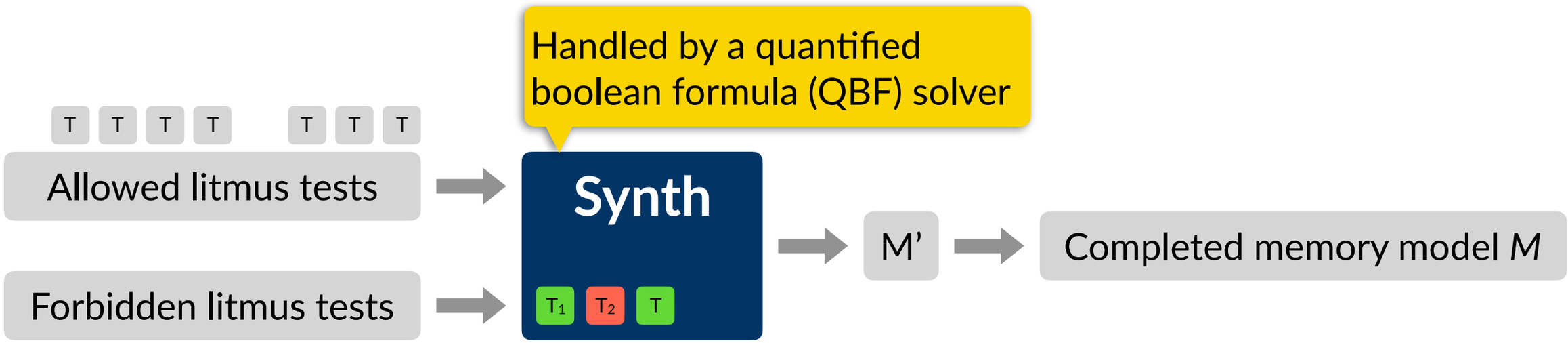
Incremental synthesis



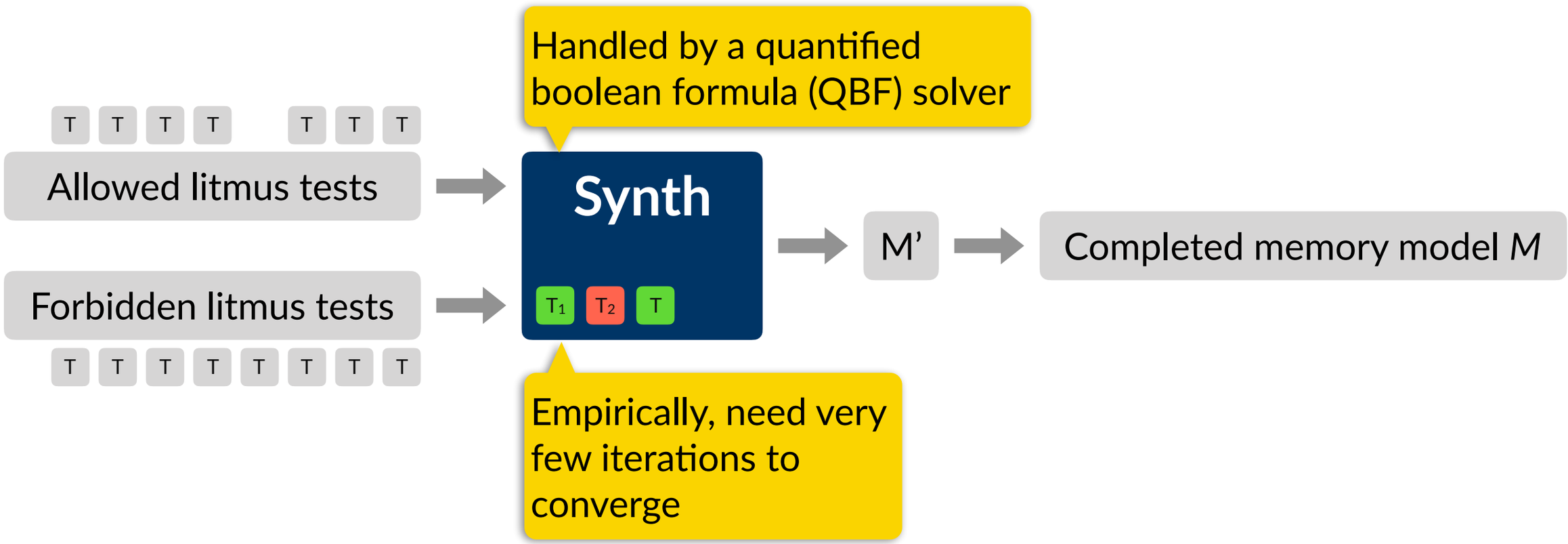
Incremental synthesis



Incremental synthesis



Incremental synthesis

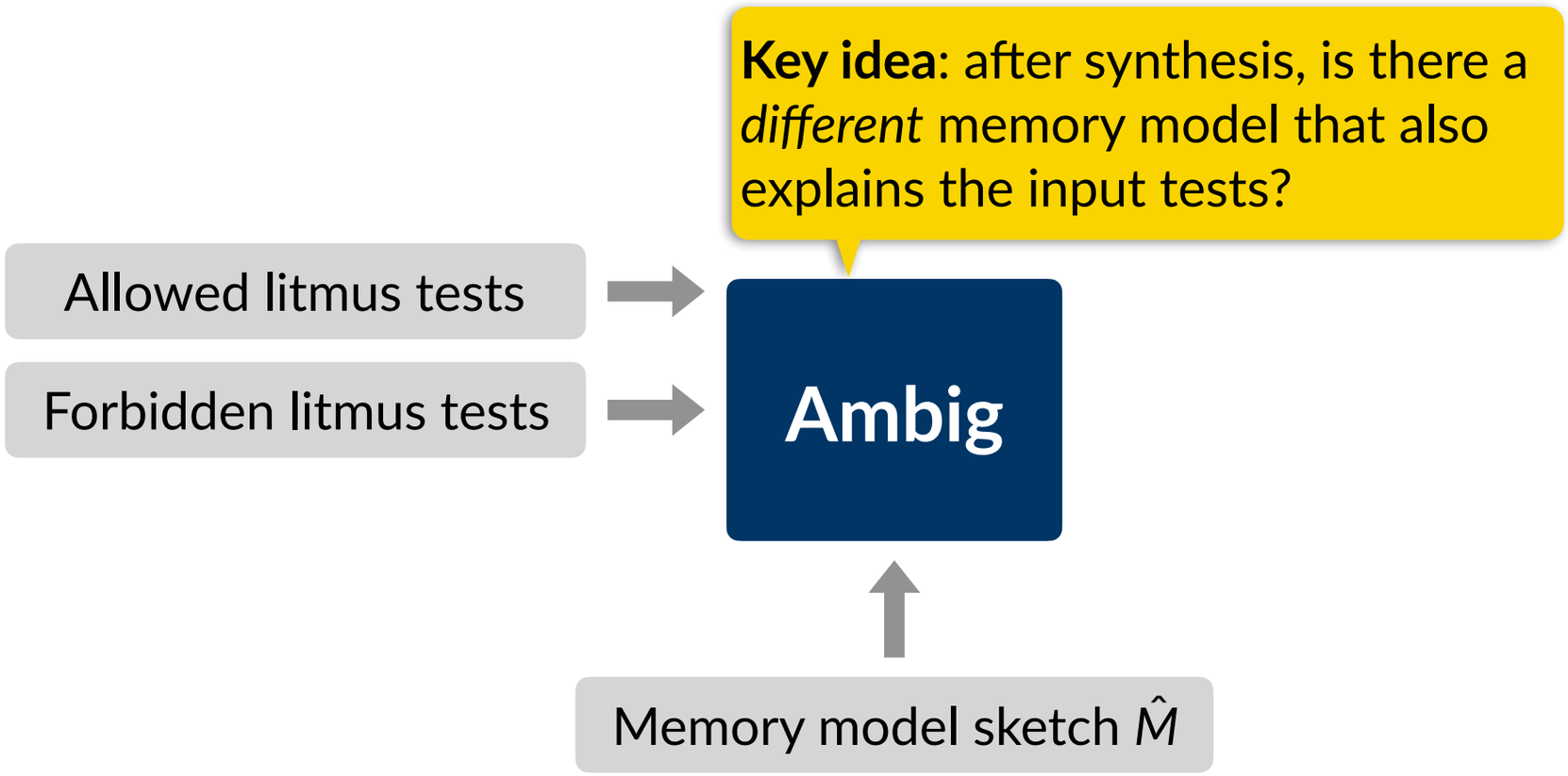


Disambiguating synthesized models

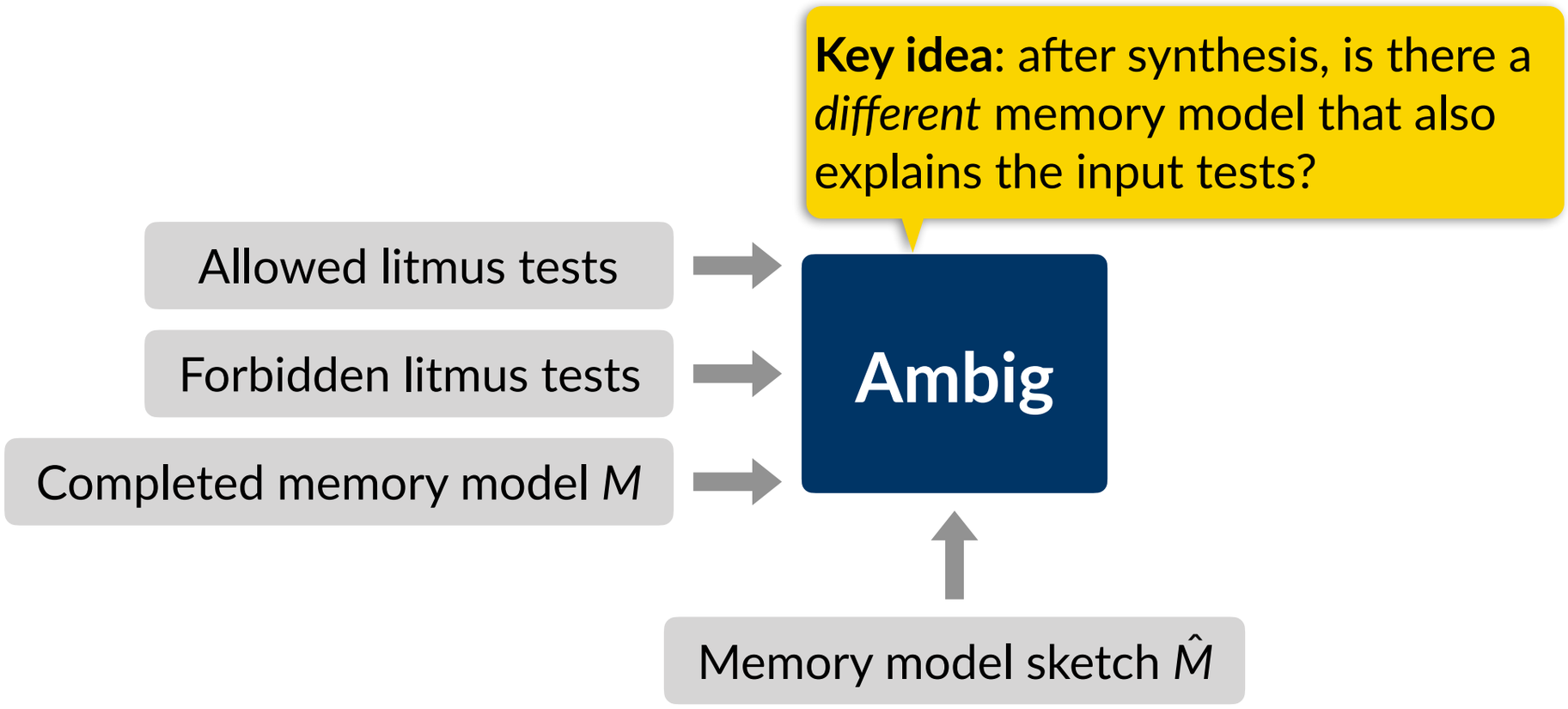
Key idea: after synthesis, is there a *different* memory model that also explains the input tests?

Ambig

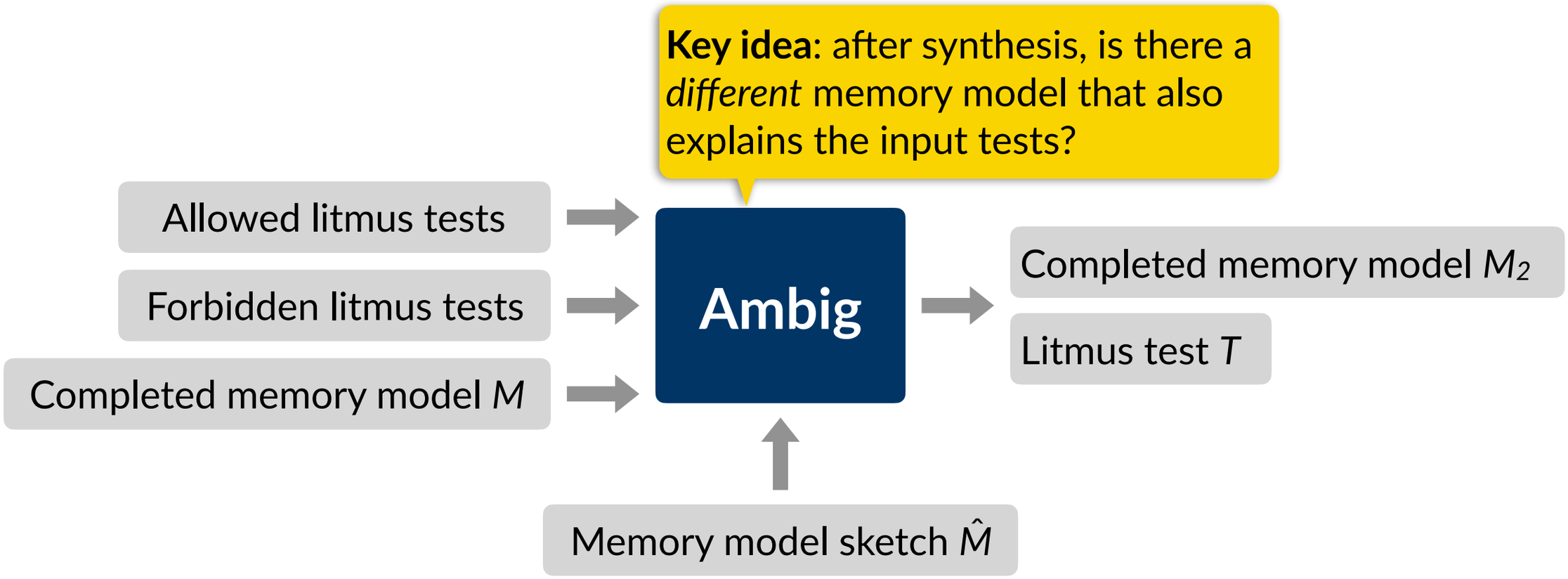
Disambiguating synthesized models



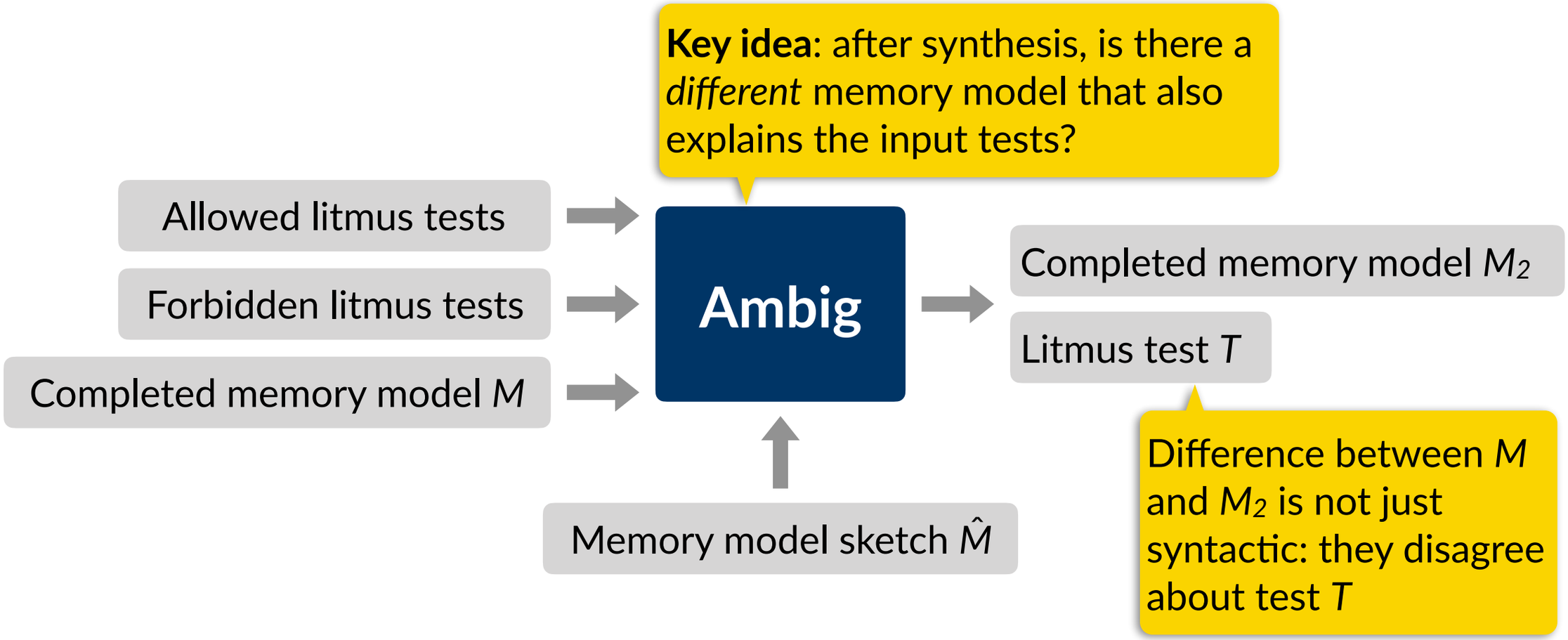
Disambiguating synthesized models



Disambiguating synthesized models



Disambiguating synthesized models



Synthesizing existing memory models

x86

PowerPC

Synthesizing existing memory models

x86 10 tests

PowerPC 768 tests

[Alglave et al, CAV'10]

Synthesizing existing memory models

		Synthesis
x86	10 tests 	✓ 2 seconds
PowerPC	768 tests [Alglave et al, CAV'10]	✓ 12 seconds

Synthesizing existing memory models

		Synthesis
x86	10 tests 	✓ 2 seconds

Not equivalent to TSO!

PowerPC	768 tests [Alglave et al, CAV'10]	✓ 12 seconds

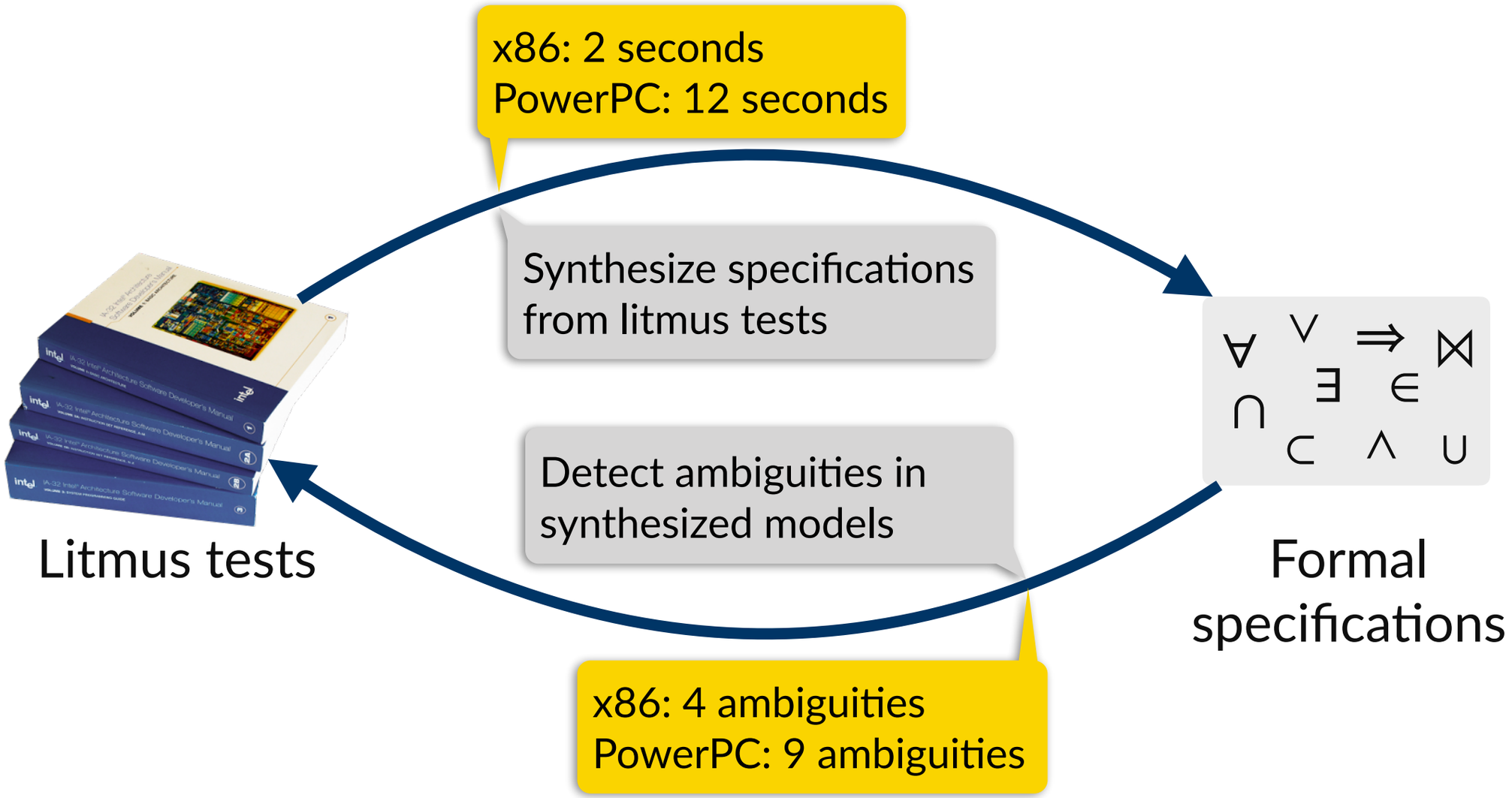
Synthesizing existing memory models

		Synthesis
x86	10 tests 	✓ 2 seconds Not equivalent to TSO!
PowerPC	768 tests [Alglave et al, CAV'10]	✓ 12 seconds Not equivalent to published model!

Synthesizing existing memory models

		Synthesis	Ambiguity
x86	10 tests 	✓ 2 seconds Not equivalent to TSO!	4 new tests `mfence`, `xchg`
PowerPC	768 tests [Alglave et al, CAV'10]	✓ 12 seconds Not equivalent to published model!	9 new tests `sync`, `lwsync`

MemSynth: automated programming for memory consistency models



Automated tools are ***worth building***

The case of memory models [PLDI'17]

Building them can be ***made systematic***

Symbolic profiling [OOPSLA'18]

The future is ***more automation***

Automating the automated programming stack

Automated tools are ***worth building***

The case of memory models [PLDI'17]

Building them can be ***made systematic***

Symbolic profiling [OOPSLA'18]

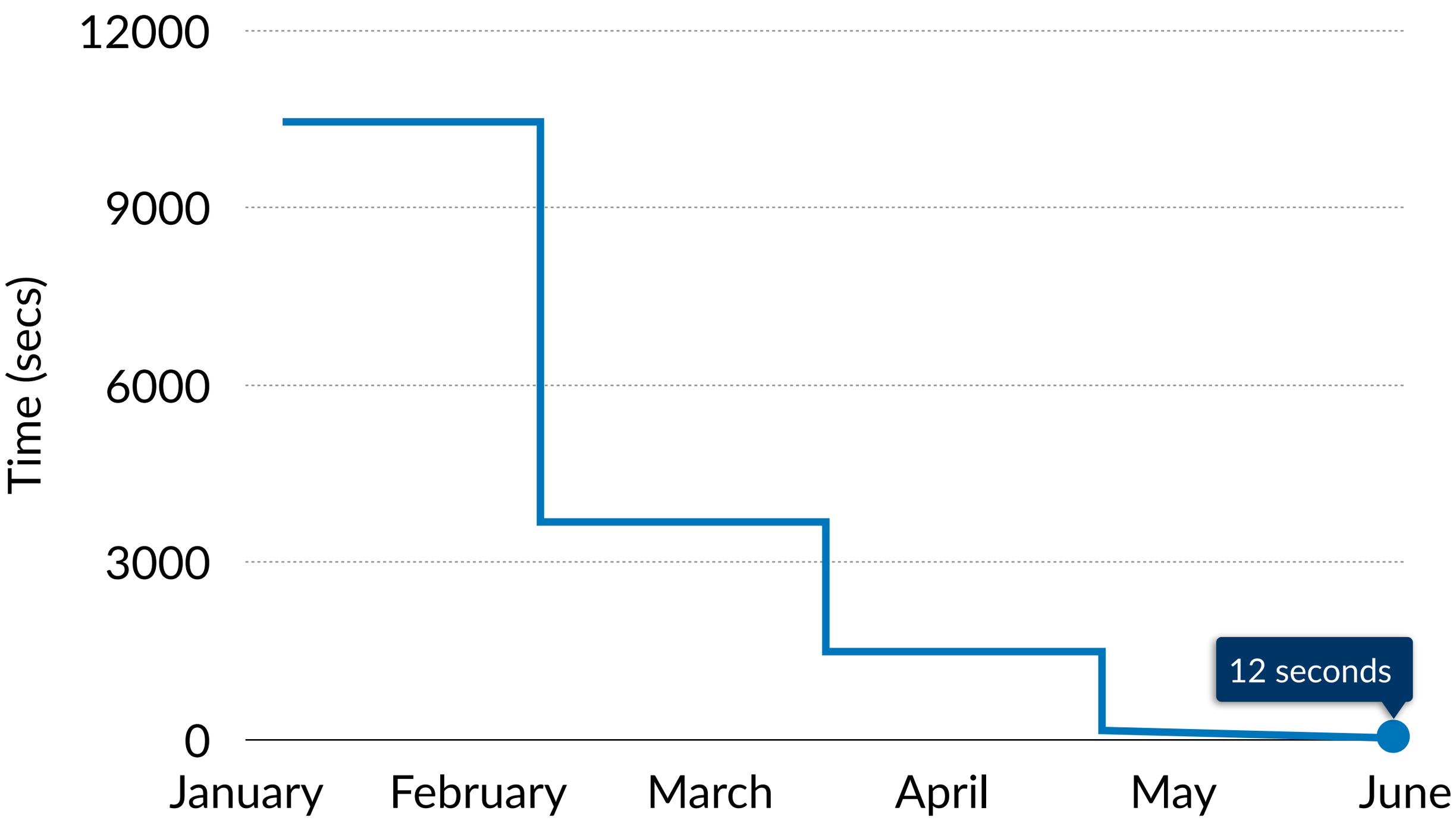
The future is ***more automation***

Automating the automated programming stack

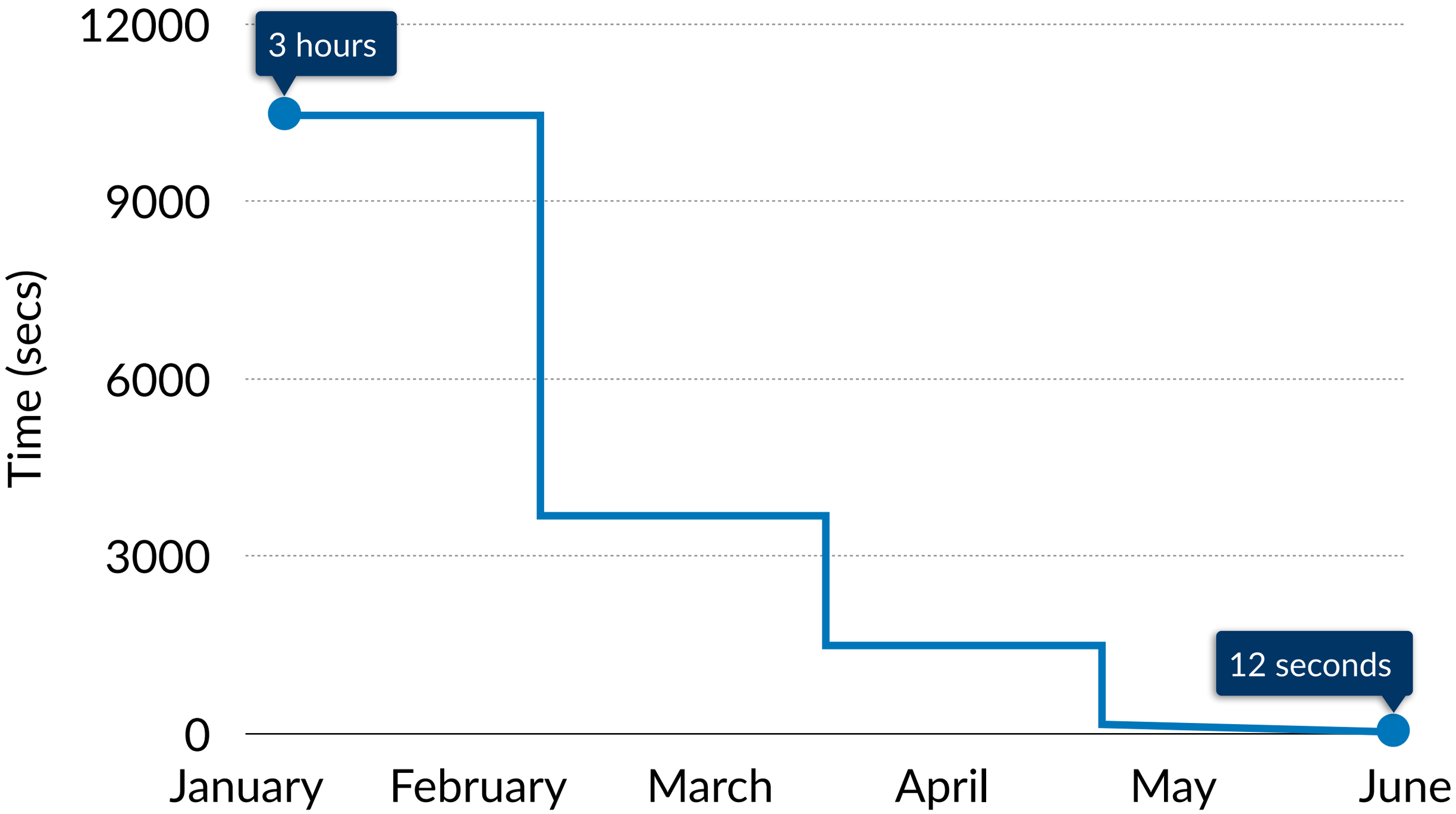
Scaling a synthesis tool is hard work

12 seconds

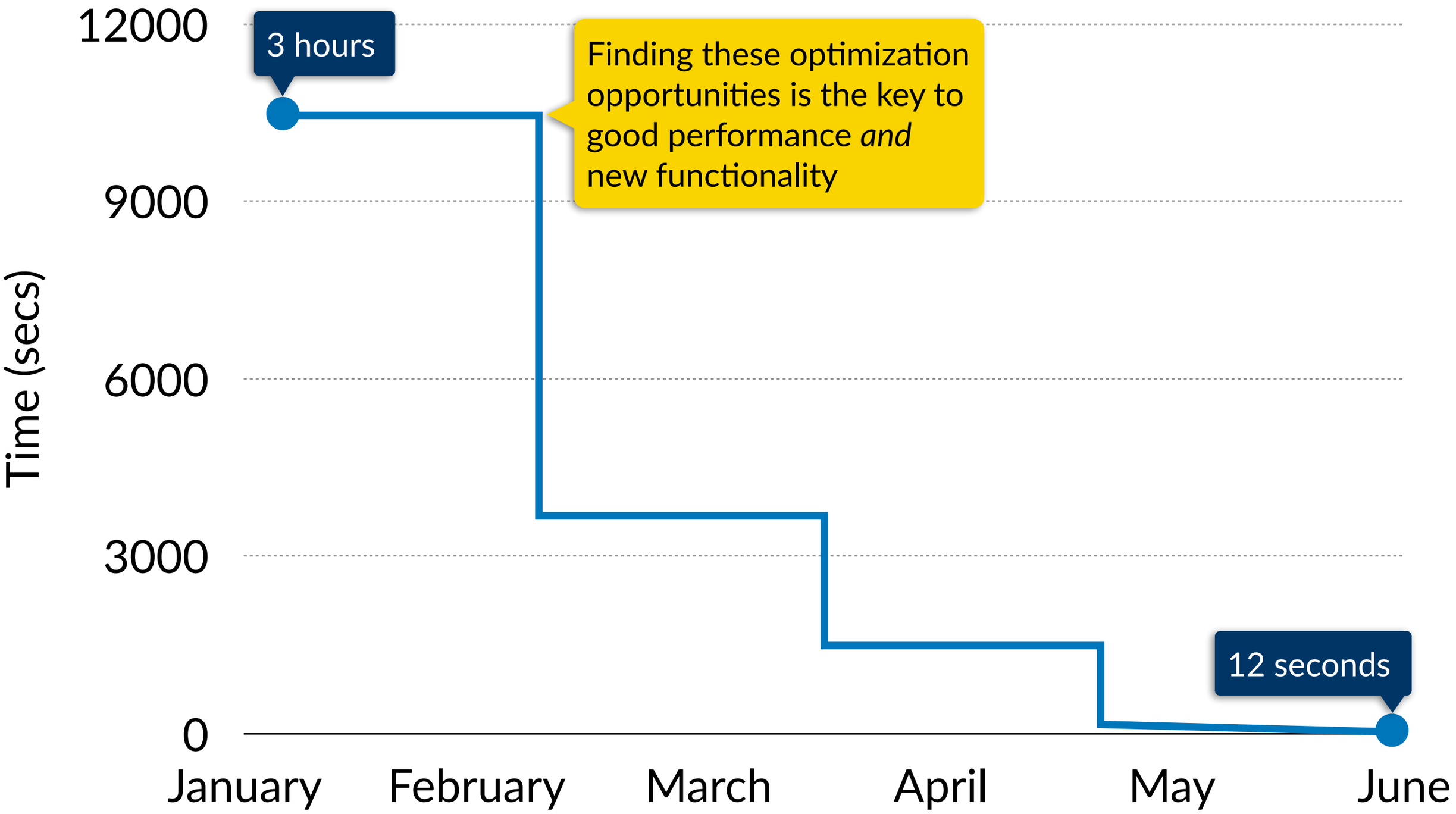
Scaling a synthesis tool is hard work



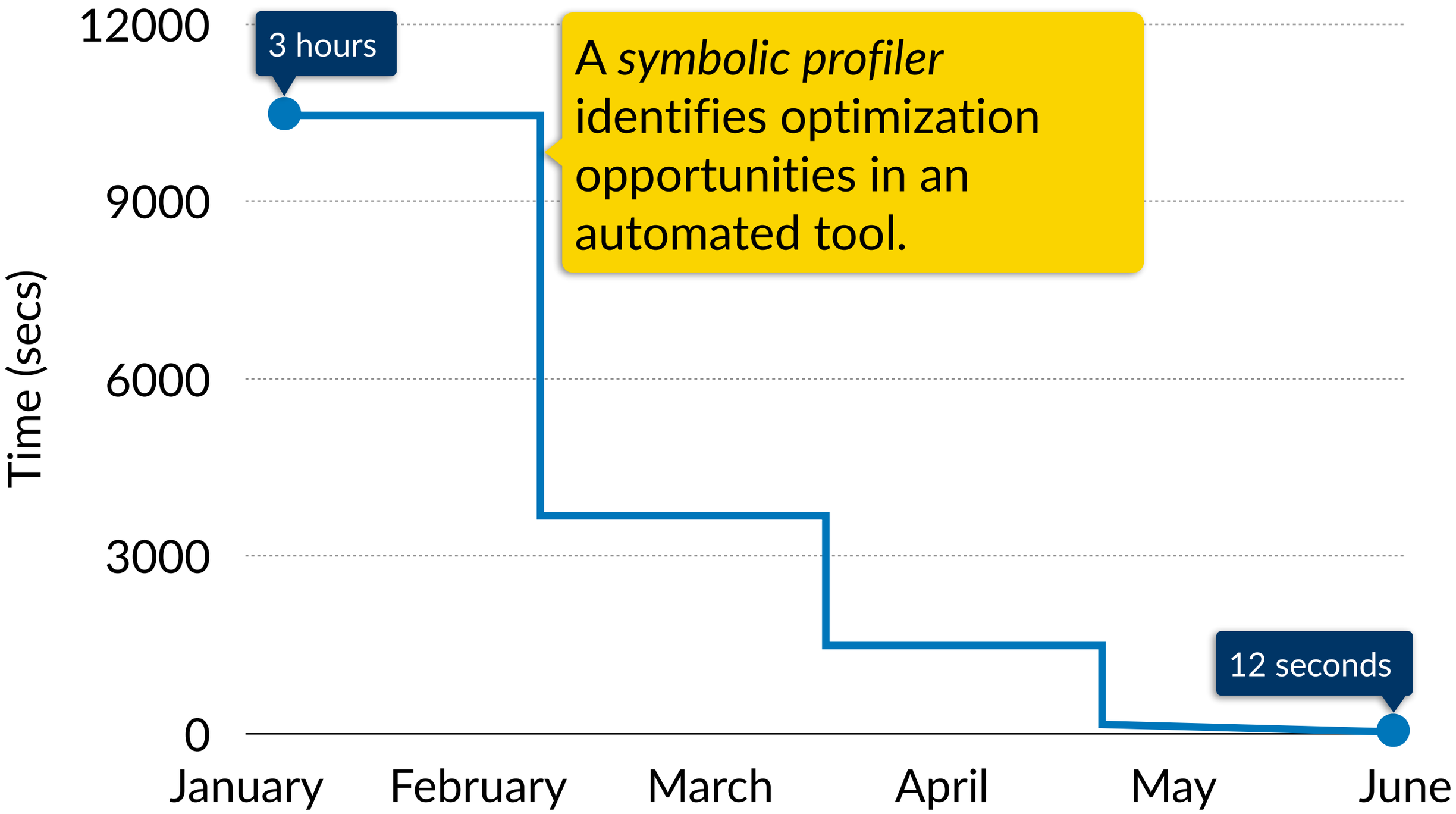
Scaling a synthesis tool is hard work



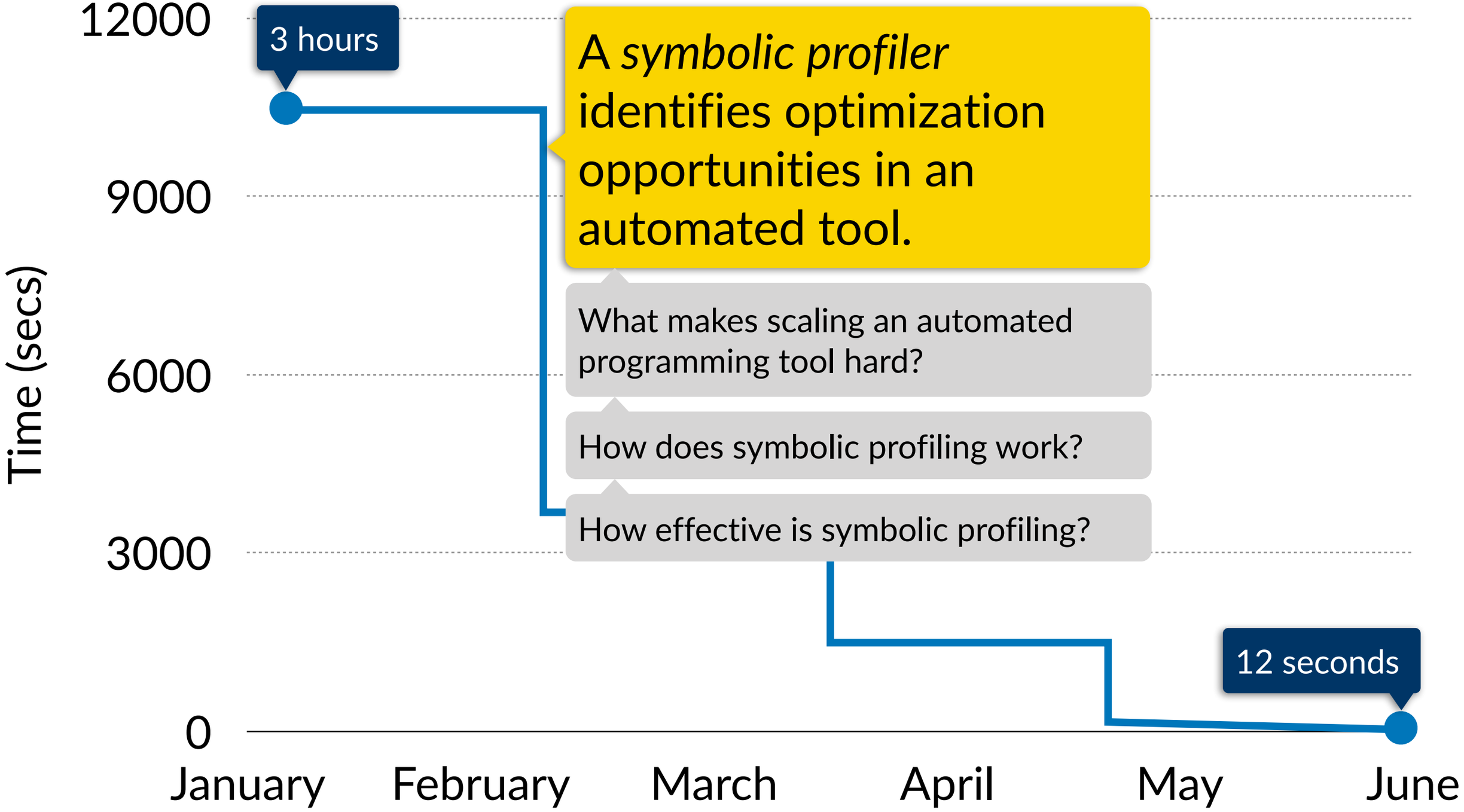
Scaling a synthesis tool is hard work



Symbolic profiling



Symbolic profiling



Symbolic evaluation executes *all paths* through a program

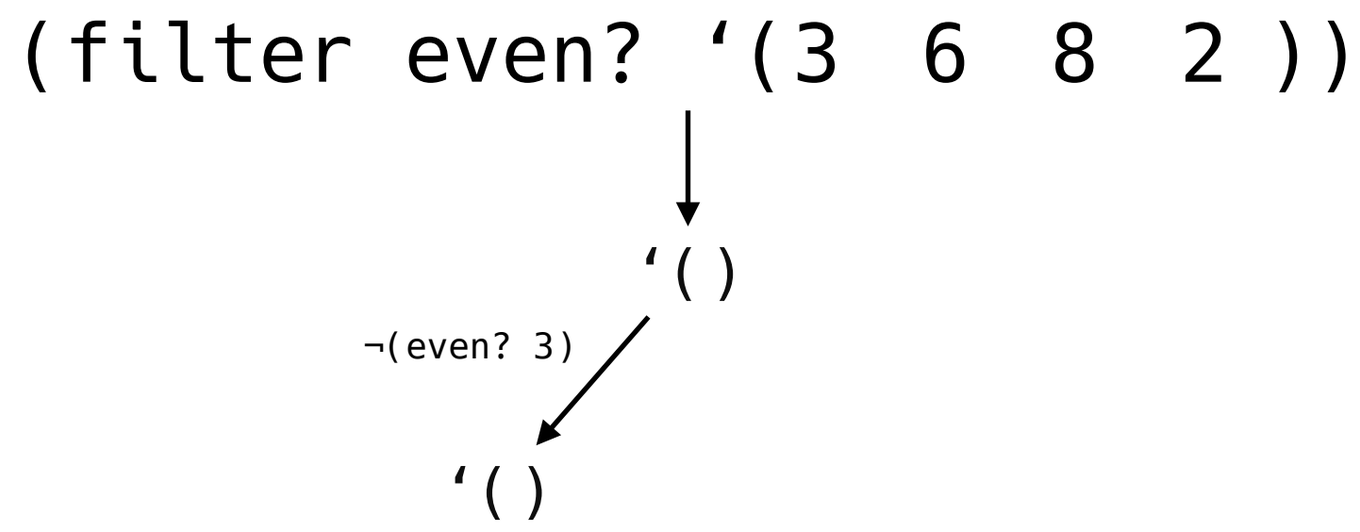
```
(filter even? '(3  6  8  2 ))
```

Symbolic evaluation executes *all paths* through a program

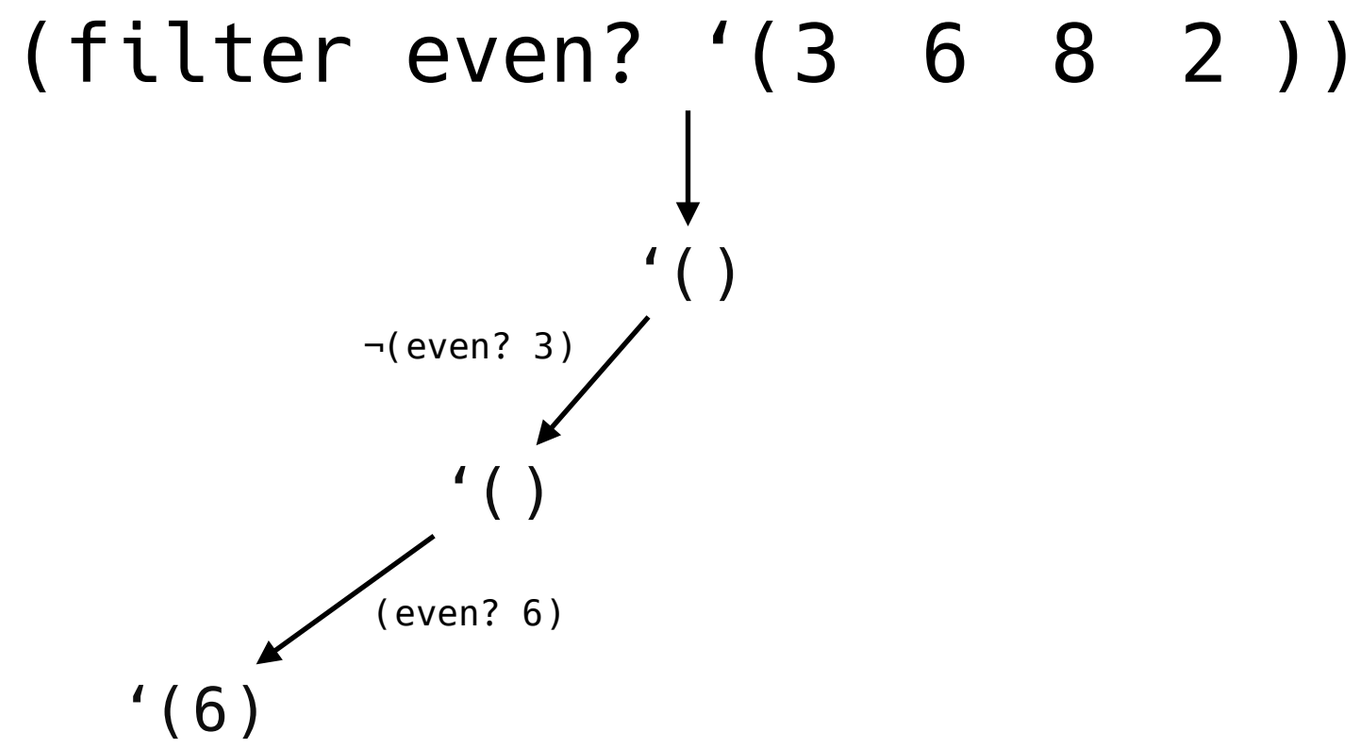
```
(filter even? '(3  6  8  2 ))
```

```
'()
```

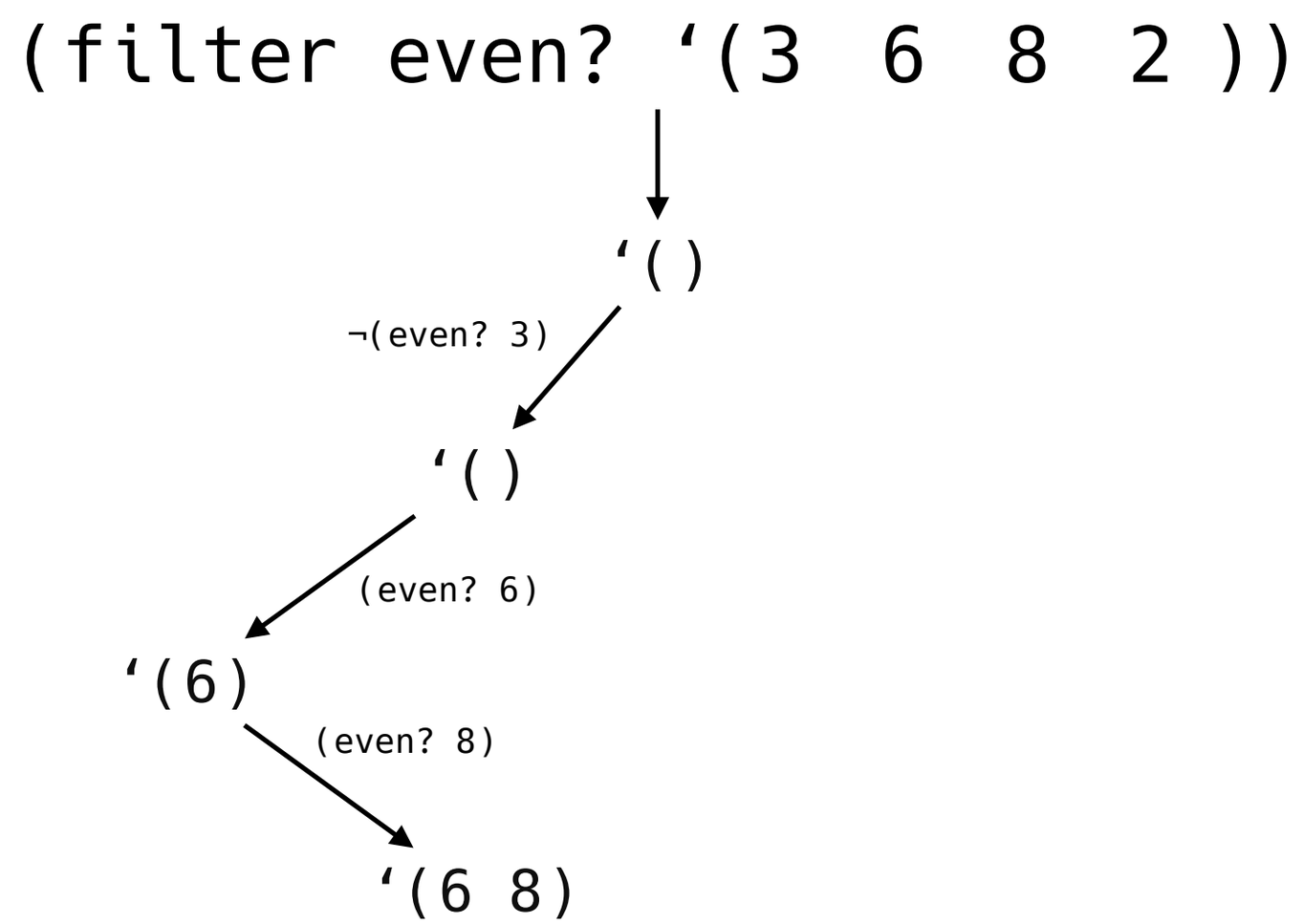
Symbolic evaluation executes *all paths* through a program



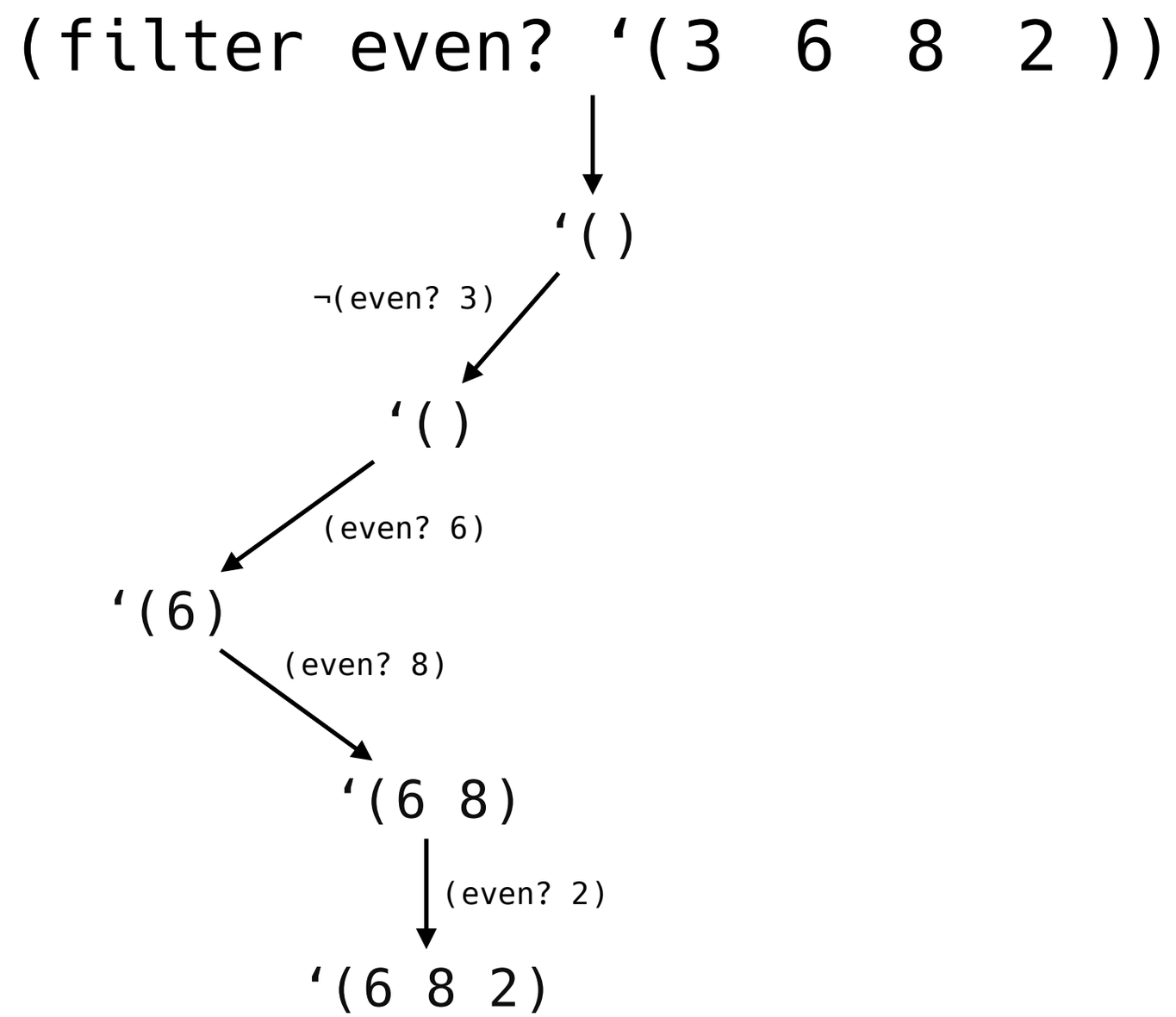
Symbolic evaluation executes *all paths* through a program



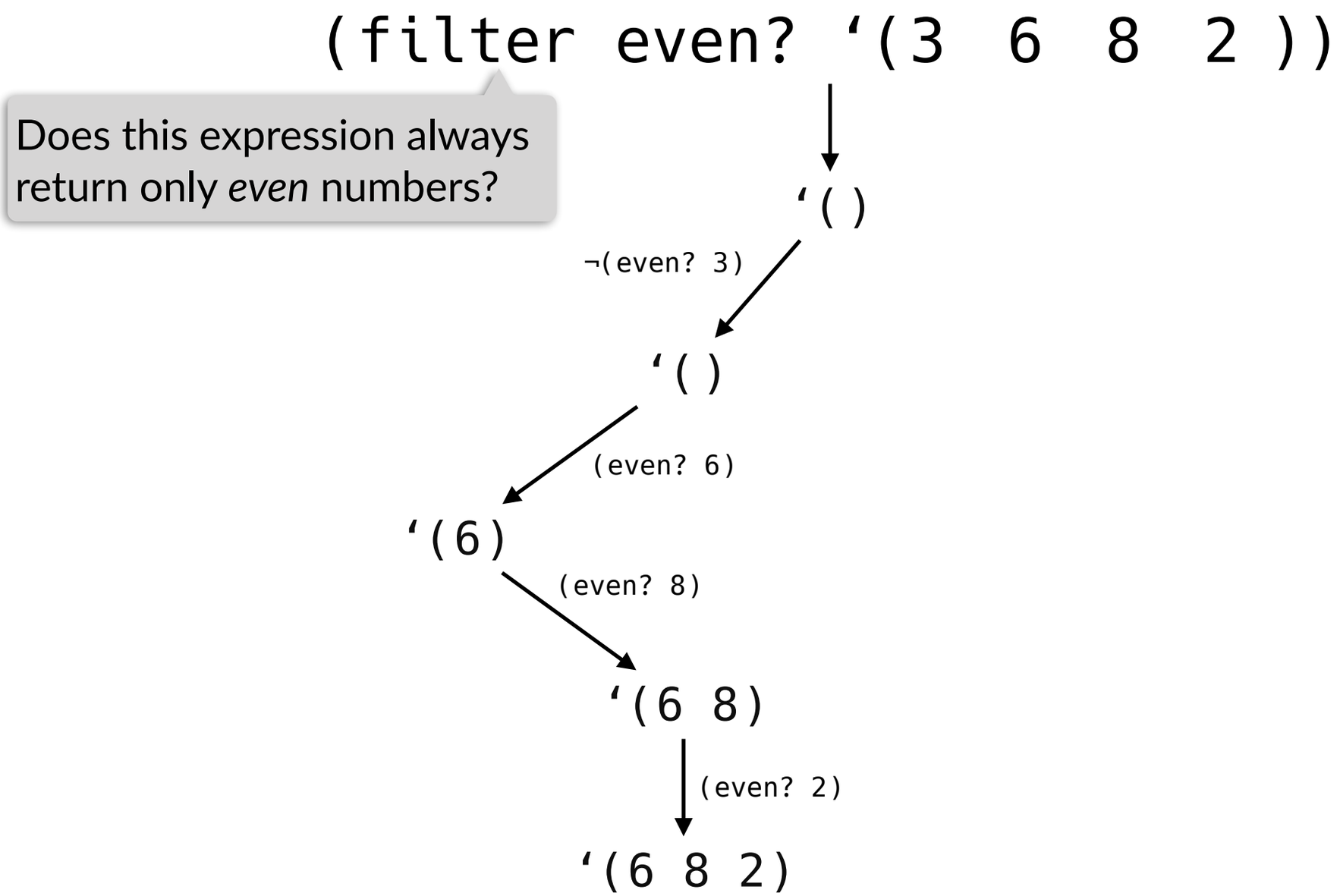
Symbolic evaluation executes *all paths* through a program



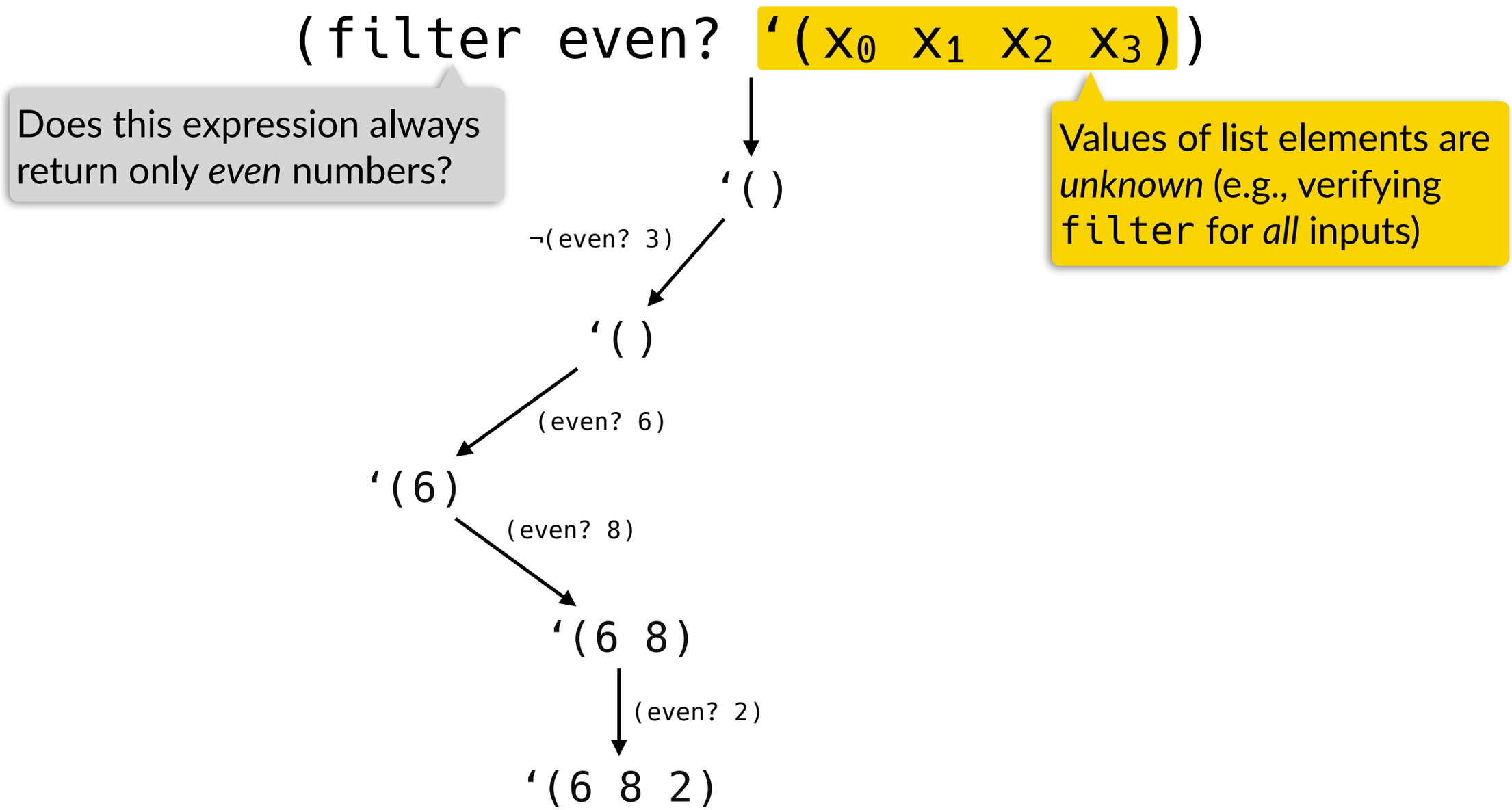
Symbolic evaluation executes *all paths* through a program



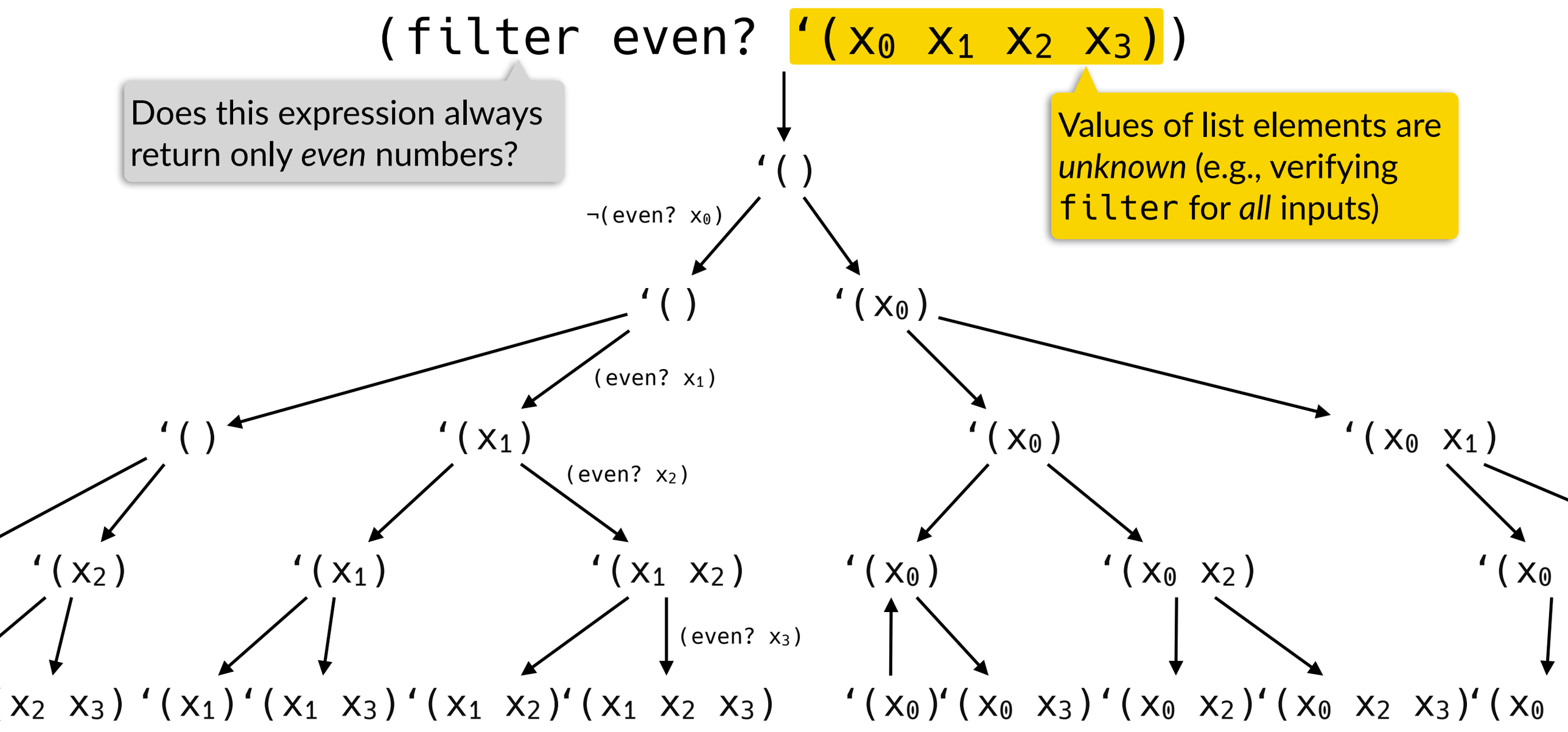
Symbolic evaluation executes *all paths* through a program



Symbolic evaluation executes *all paths* through a program



Symbolic evaluation executes *all paths* through a program

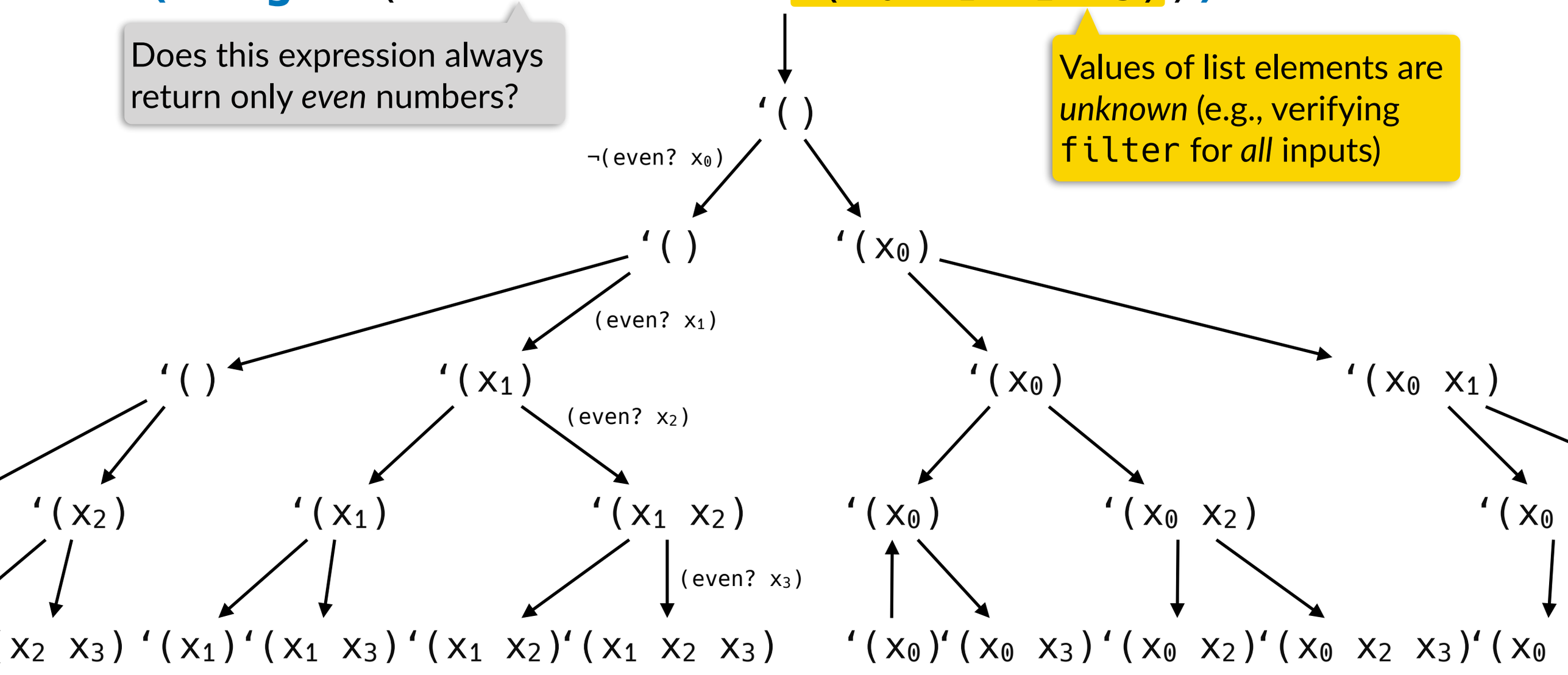


Symbolic evaluation executes *all paths* through a program

```
(length (filter even? '(x0 x1 x2 x3)))
```

Does this expression always return only *even* numbers?

Values of list elements are *unknown* (e.g., verifying `filter` for *all* inputs)

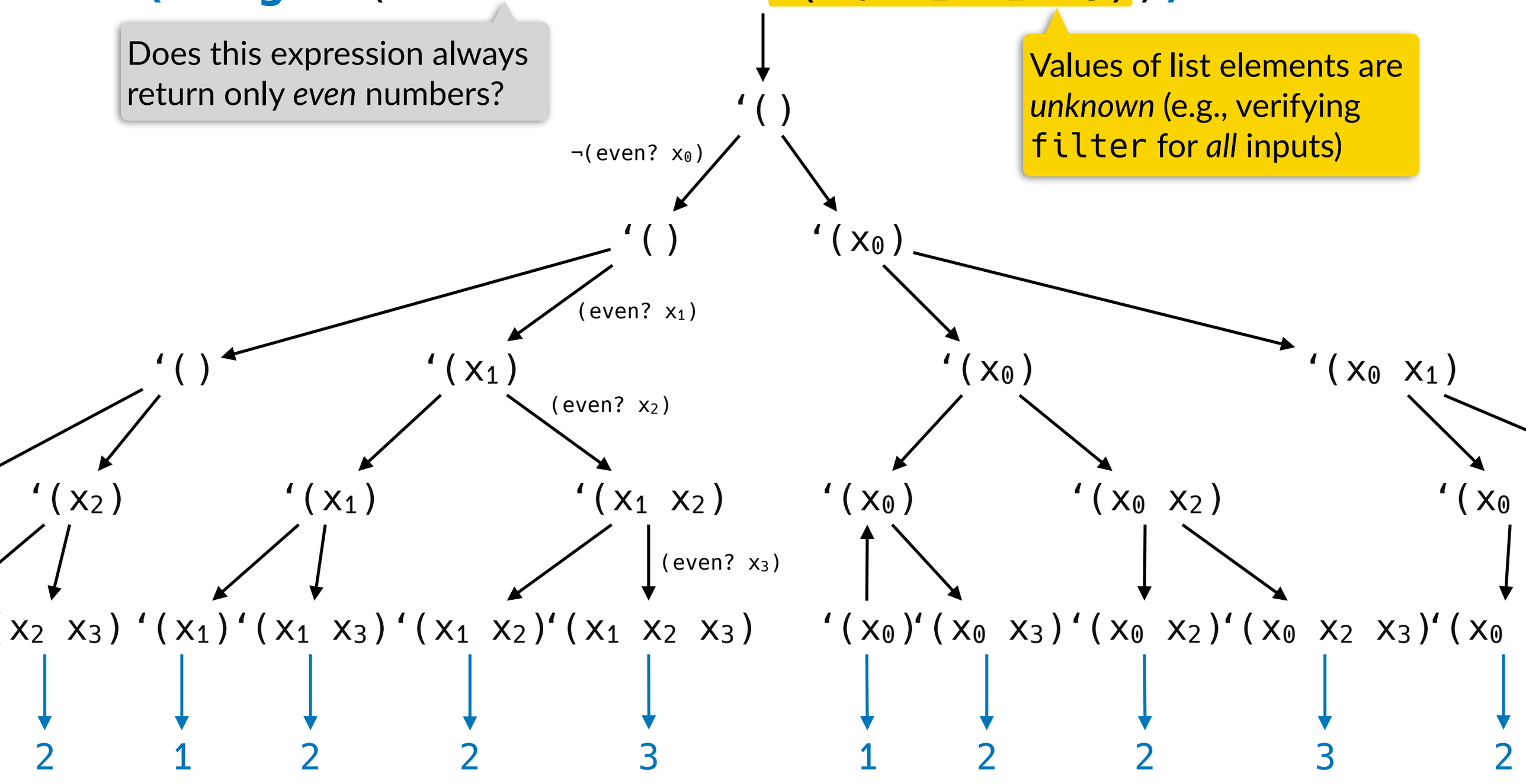


Symbolic evaluation executes *all paths* through a program

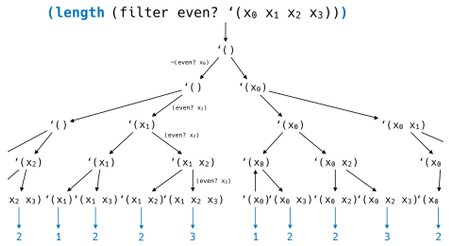
```
(length (filter even? '(x0 x1 x2 x3)))
```

Does this expression always return only *even* numbers?

Values of list elements are *unknown* (e.g., verifying `filter` for *all* inputs)



Symbolic evaluation techniques



Symbolic execution

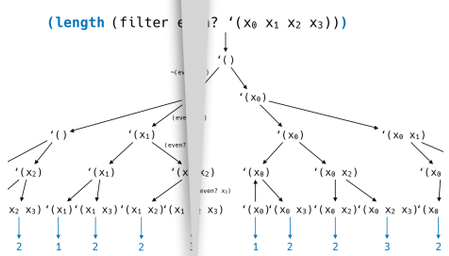
Always fork into independent paths (more paths, but more concrete)

Bounded model checking

Merge after every fork (fewer paths, but less concrete)

Symbolic evaluation techniques

Crucible [Galois, Inc.]



Jalangi [Sen et al 2014]

Rosette [Torlak & Bodik 2014]

Symbolic execution

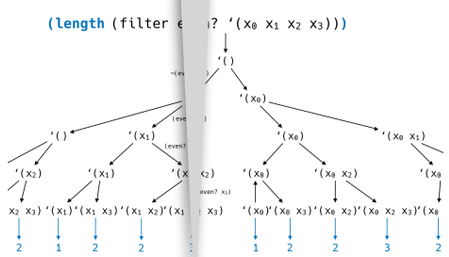
Always fork into independent paths (more paths, but more concrete)

Bounded model checking

Merge after every fork (fewer paths, but less concrete)

Symbolic evaluation techniques

Crucible [Galois, Inc.]



Jalangi [Sen et al 2014]

Rosette [Torlak & Bodik 2014]

Symbolic execution

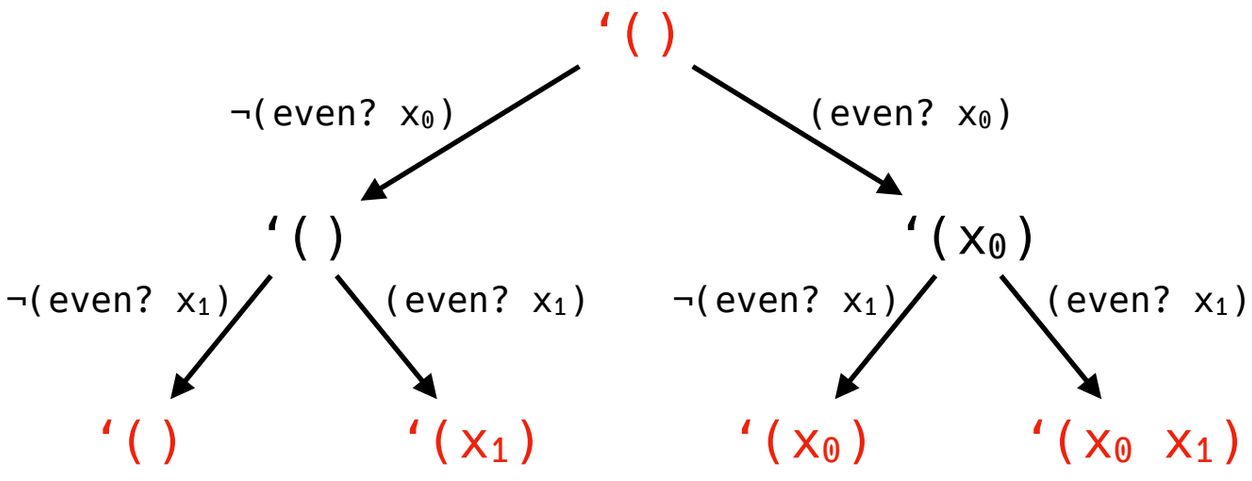
Always fork into independent paths (more paths, but more concrete)

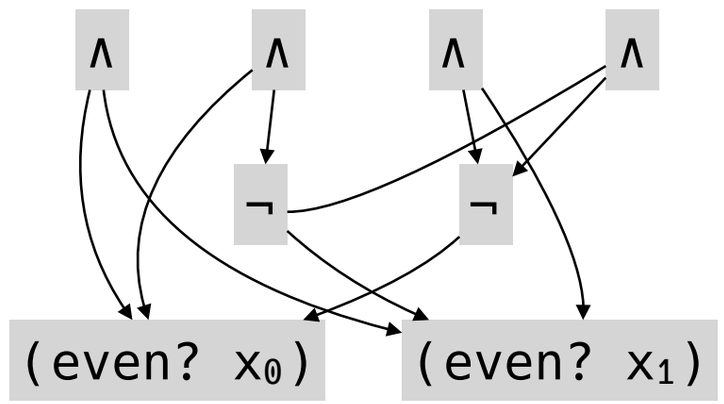
Controlling the trade-off between these strategies is key to good scalability

Bounded model checking

Merge after every fork (fewer paths, but less concrete)

Two data structures to summarize symbolic evaluation

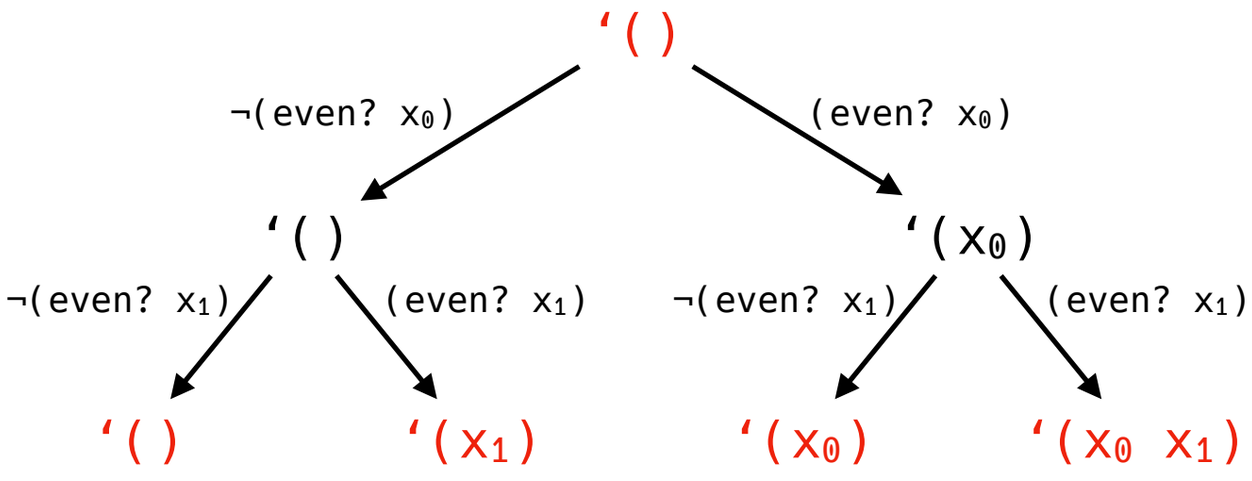


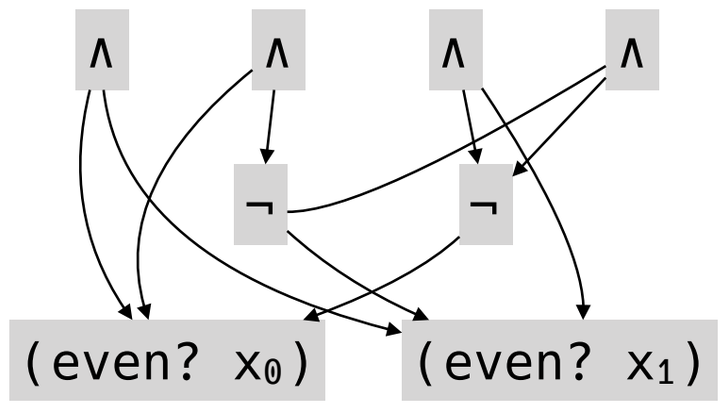


Symbolic evaluation graph
Reflects the evaluator’s strategy
for all-paths execution of the program

Symbolic heap
Shape of all symbolic values
created by the program

Two data structures to summarize symbolic evaluation



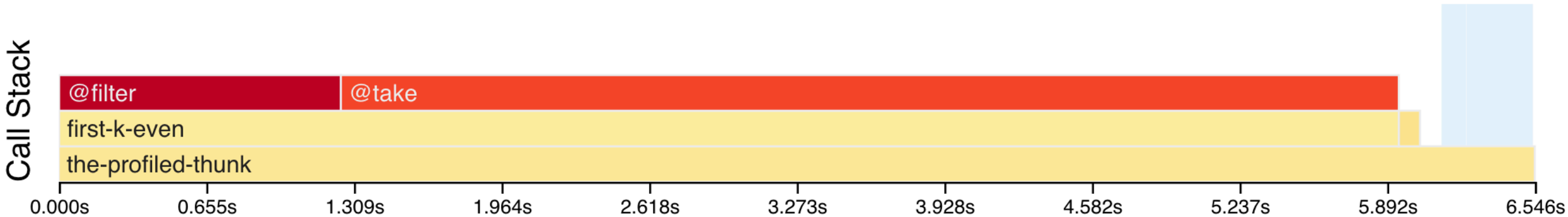


Symbolic evaluation graph
Reflects the evaluator’s strategy for all-paths execution of the program

Symbolic heap
Shape of all symbolic values created by the program

Any symbolic evaluation technique can be summarized by these two data structures

Analyzing symbolic data structures



Function	Score	Time (ms)	Term Count	Unused Terms	Union Size	Merge Cases
filter 1 call	4.3	1249	137408	131164	4288	93664
take 1 call	2.8	4692	50312	49986	2209	49986
andmap 1 call	0.3	94	14180	14180	0	4097
the-profiled-thunk	0.1	511	66	0	0	0

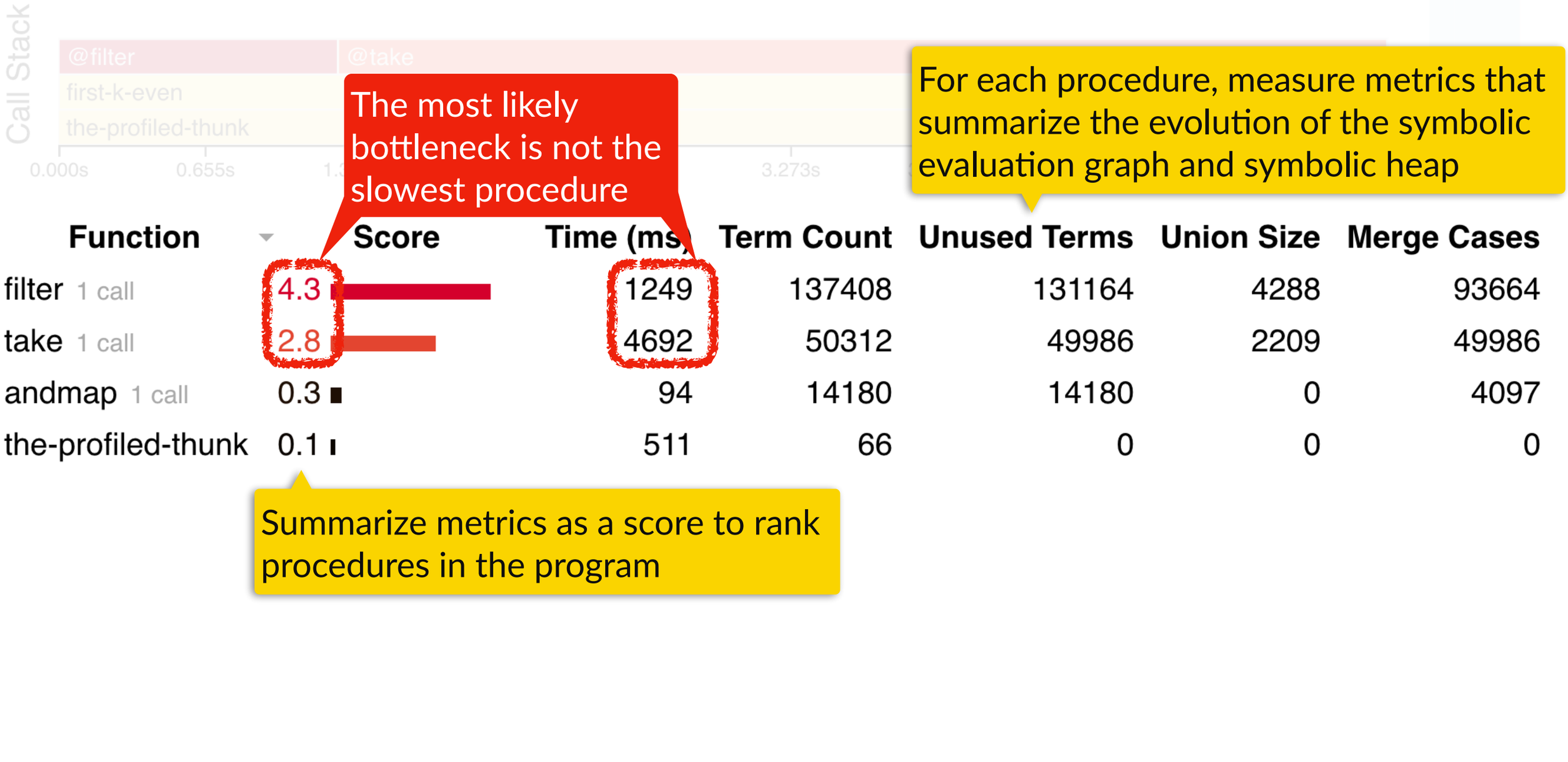
Analyzing symbolic data structures

For each procedure, measure metrics that summarize the evolution of the symbolic evaluation graph and symbolic heap

Function	Score	Time (ms)	Term Count	Unused Terms	Union Size	Merge Cases
filter 1 call	4.3	1249	137408	131164	4288	93664
take 1 call	2.8	4692	50312	49986	2209	49986
andmap 1 call	0.3	94	14180	14180	0	4097
the-profiled-thunk	0.1	511	66	0	0	0

Summarize metrics as a score to rank procedures in the program

Analyzing symbolic data structures



Three symbolic profilers

We developed two implementations:

- The **Rosette** solver-aided language (Racket)
- The **Jalangi** dynamic analysis framework (JavaScript)

Since publication, based on our work:

- The **Crucible** symbolic simulation library (C, Java, …) by Galois

Three symbolic profilers

We developed two implementations:

- The **Rosette** solver-aided language (Racket)

Today

- The **Jalangi** dynamic analysis framework (JavaScript)

Since publication, based on our work:

- The **Crucible** symbolic simulation library (C, Java, …) by Galois

Symbolic profiling in practice

Case studies: fixed 8 performance issues in 15 Rosette tools

Refinement type checker for Ruby [VMCAI'18]	6× speedup
Cryptographic protocol verifier [FM'18]	29× speedup
SQL query verifier [CIDR'17]	75× speedup
Safety-critical radiotherapy system verifier [CAV'16]	290× speedup

Symbolic profiling in practice

Case studies: fixed 8 performance issues in 15 Rosette tools

Refinement type checker for Ruby [VMCAI'18]	6× speedup
Cryptographic protocol verifier [FM'1	29× speedup
SQL query verifier [CIDR'17]	75× speedup
Safety-critical radiotherapy system verifier [CAV'16]	290× speedup

Used in production at the UW Medical Center

Symbolic profiling in practice

Case studies: fixed 8 performance issues in 15 Rosette tools

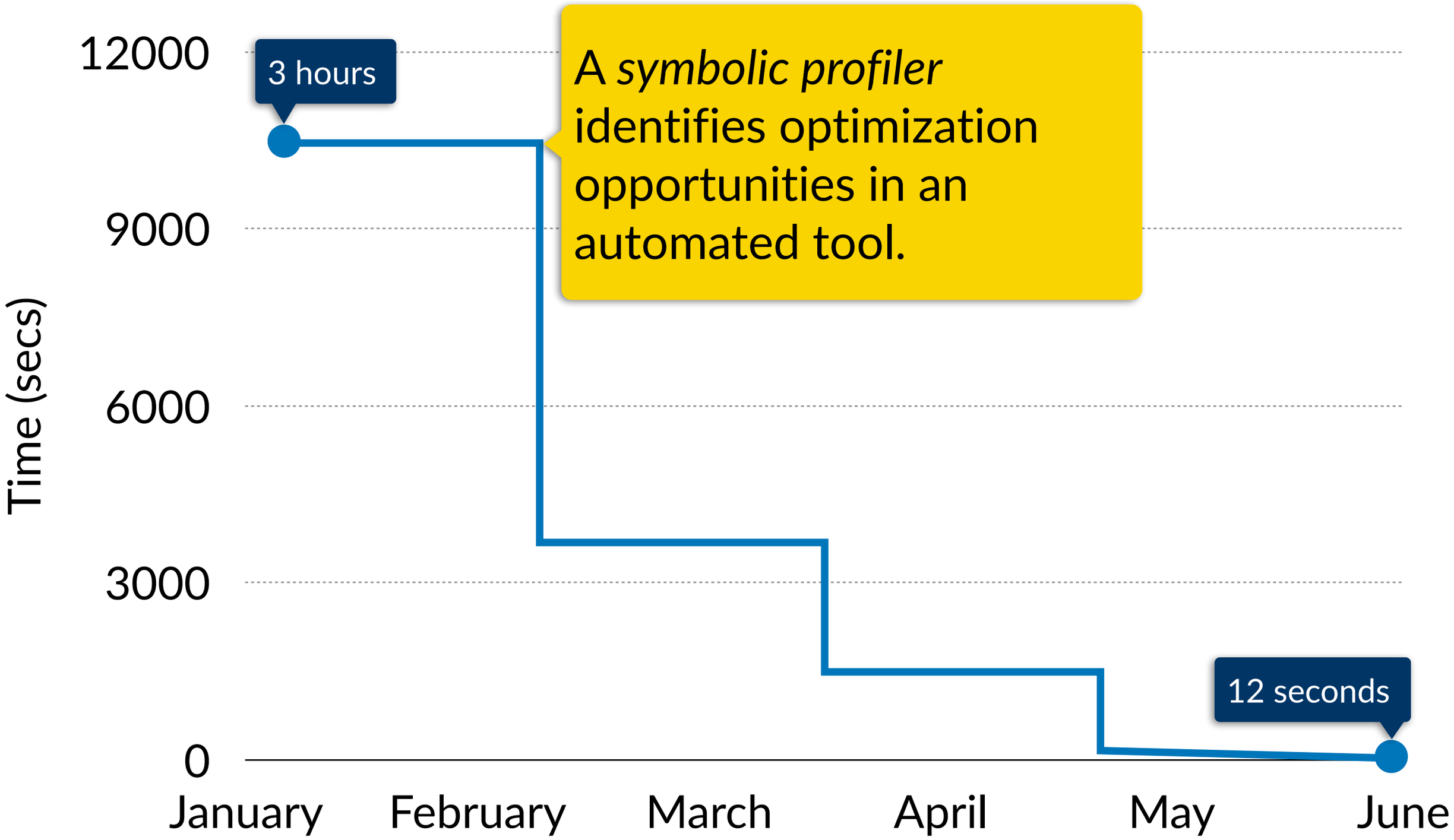
Refinement type checker for Ruby [VMCAI'18]	6× speedup
Cryptographic protocol verifier [FM'18]	29× speedup
SQL query verifier [CIDR'17]	75× speedup
Safety-critical radiotherapy system verifier [CAV'16]	290× speedup

User study: 8 Rosette users tasked with finding known performance issues in 4 programs

Users solved every task more quickly when they had access to symbolic profiling

- 6 failures without symbolic profiling vs. none with it

Symbolic profiling



A *symbolic profiler* identifies optimization opportunities in an automated tool.

Automated tools are ***worth building***

The case of memory models [PLDI'17]

Building them can be ***made systematic***

Symbolic profiling [OOPSLA'18]

The future is ***more automation***

Automating the automated programming stack

Automated tools are ***worth building***

The case of memory models [PLDI'17]

Building them can be ***made systematic***

Symbolic profiling [OOPSLA'18]

The future is ***more automation***

Automating the automated programming stack

Automated programming abstractions

File systems
[ASPLOS'16, OSDI'16]

Operating systems
[SOSP'17, OSDI'18]

Memory models
[PLDI'17]

Solver-aided languages
front-end abstractions for verification/synthesis

Metasketches
[POPL'16]

Symbolic evaluation
algorithms to translate programs to SAT/SMT

Symbolic profiling
[OOPSLA'18]

SAT/SMT solving
improvements in scale and expressiveness

Diagnosing SMT solver behavior

File systems
[ASPLOS'16, OSDI'16]

Operating systems
[SOSP'17, OSDI'18]

Memory models
[PLDI'17]

Solver-aided languages
front-end abstractions for verification/synthesis

Metasketches
[POPL'16]

Symbolic evaluation
algorithms to translate programs to SAT/SMT

Symbolic profiling
[OOPSLA'18]

SAT/SMT solving
improvements in scale and expressiveness

?

Diagnosing SMT solver behavior

Z3 version 4.8.3 slower and unable to solve problem that was solved by Z3 version 4.8.1 `string` — 7
#1979 by pjljvandelaar was closed on Dec 11, 2018

Slow performance on simple query that uses equalities for assignments — 4
#1602 by 4tXJ7f was closed on Nov 25, 2018

The solver slows down of java version when using multi-thread — 1
#1504 by destinyfucker was closed on Feb 24, 2018

bv2int and int2bv slow? — 8
#1481 by kren1 was closed on Feb 14, 2018

Incremental floating point is much slower than one-shot on certain short problems — 8
#1459 opened on Jan 24, 2018 by arotenberg

Suspiciously slow on simple example — 1
#1425 opened on Dec 31, 2017 by DennisYurichev

Performance surprisingly slow: since it can be solved very fast... `string` — 4
#1352 by pjljvandelaar was closed on Dec 12, 2018

(+ (- 1) str.len) instead of (- 1 str.len) make problem very slow to execute — 2
#1140 by jawline was closed on Jul 11, 2017

SAT/SMT solving
improvements in scale and expressiveness

Diagnosing SMT solver behavior

File systems
[ASPLOS'16, OSDI'16]

Operating systems
[SOSP'17, OSDI'18]

Memory models
[PLDI'17]

Solver-aided languages
front-end abstractions for verification/synthesis

Metasketches
[POPL'16]

Symbolic evaluation
algorithms to translate programs to SAT/SMT

Symbolic profiling
[OOPSLA'18]

SAT/SMT solving
improvements in scale and expressiveness

Solver profiling

Self-optimizing automated tools

File systems
[ASPLOS'16, OSDI'16]

Operating systems
[SOSP'17, OSDI'18]

Memory models
[PLDI'17]

Solver-aided languages
front-end abstractions for verification/synthesis

Metasketches
[POPL'16]

Exploit this profiling data for profile-guided optimization

Symbolic evaluation
algorithms to translate programs to SAT/SMT

Symbolic profiling
[OOPSLA'18]

SAT/SMT solving
improvements in scale and expressiveness

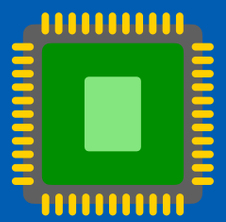
Solver profiling

Application opportunities

File systems
[ASPLOS'16, OSDI'16]

Operating systems
[SOSP'17, OSDI'18]

Memory models
[PLDI'17]



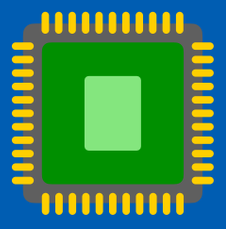
Hardware accelerator design/programming

High-performance low-precision kernels

File systems
[ASPLOS'16, OSDI'16]

Operating systems
[SOSP'17, OSDI'18]

Memory models
[PLDI'17]



Hardware accelerator design/programming

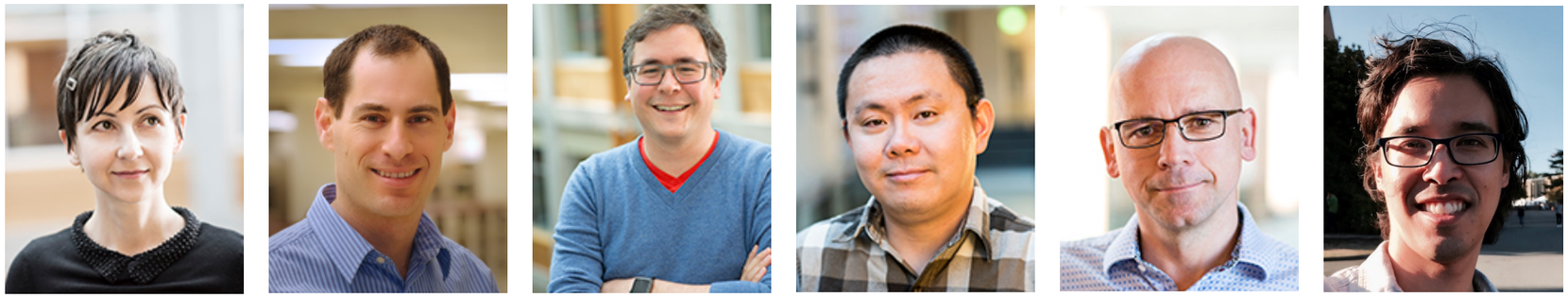
High-performance low-precision kernels

New abstractions and tools can empower programmers to build specialized automated programming tools that improve software reliability.

Metasketches
[POPL'16]

Symbolic profiling
[OOPSLA'18]

Solver profiling



Thanks!

bornholt@uw.edu
https://unsat.org

PLSE sampa UNSAT

New abstractions and tools can empower programmers to build specialized automated programming tools that improve software reliability.