Computer-Aided Reasoning for Software

CSE507

# Solver-Aided Languages

courses.cs.washington.edu/courses/cse507/18sp/

**Emina Torlak**
emina@cs.washington.edu

# Today

topics

### Last lecture

- Program synthesis

### Today

- Solver-aided languages

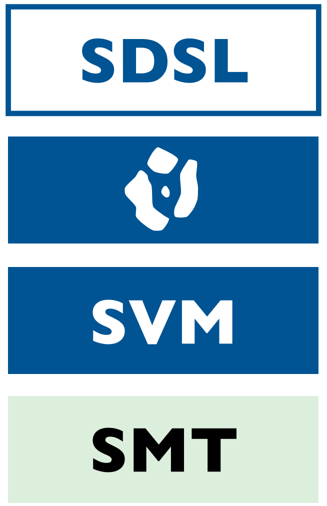
### Announcements

- Next Wednesday: guest lecture by James Bornholt
- Project presentations next Friday in class
  - 13 min per team: 10 min presentation + 3 min questions
- Project reports and prototypes due next Friday at 11:00pm

# How to build your own solver-aided tool

**The classic (hard) way to build a tool**

What is hard about building a solver-aided tool?



SDSL

SVM

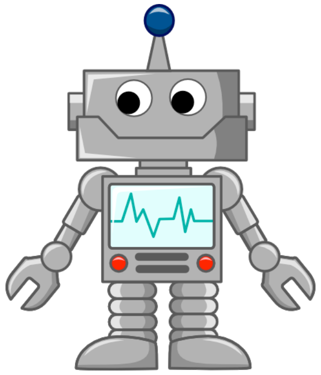
SMT

**An easier way: tools as languages**

How to build tools by stacking layers of languages.

**Behind the scenes: symbolic virtual machine**

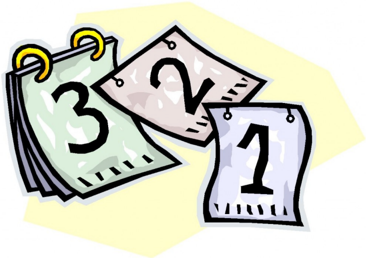
How Rosette works so you don’t have to.



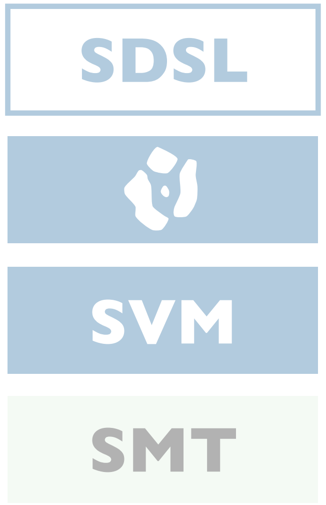
**A last look: a few recent applications**

Cool tools built with Rosette!

# How to build your own solver-aided tool



**The classic (hard) way to build a tool**
What is hard about building a solver-aided tool?



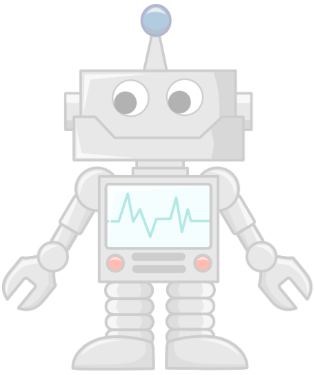
SDSL

SVM

SMT

**An easier way: tools as languages**
How to build tools by stacking layers of languages.

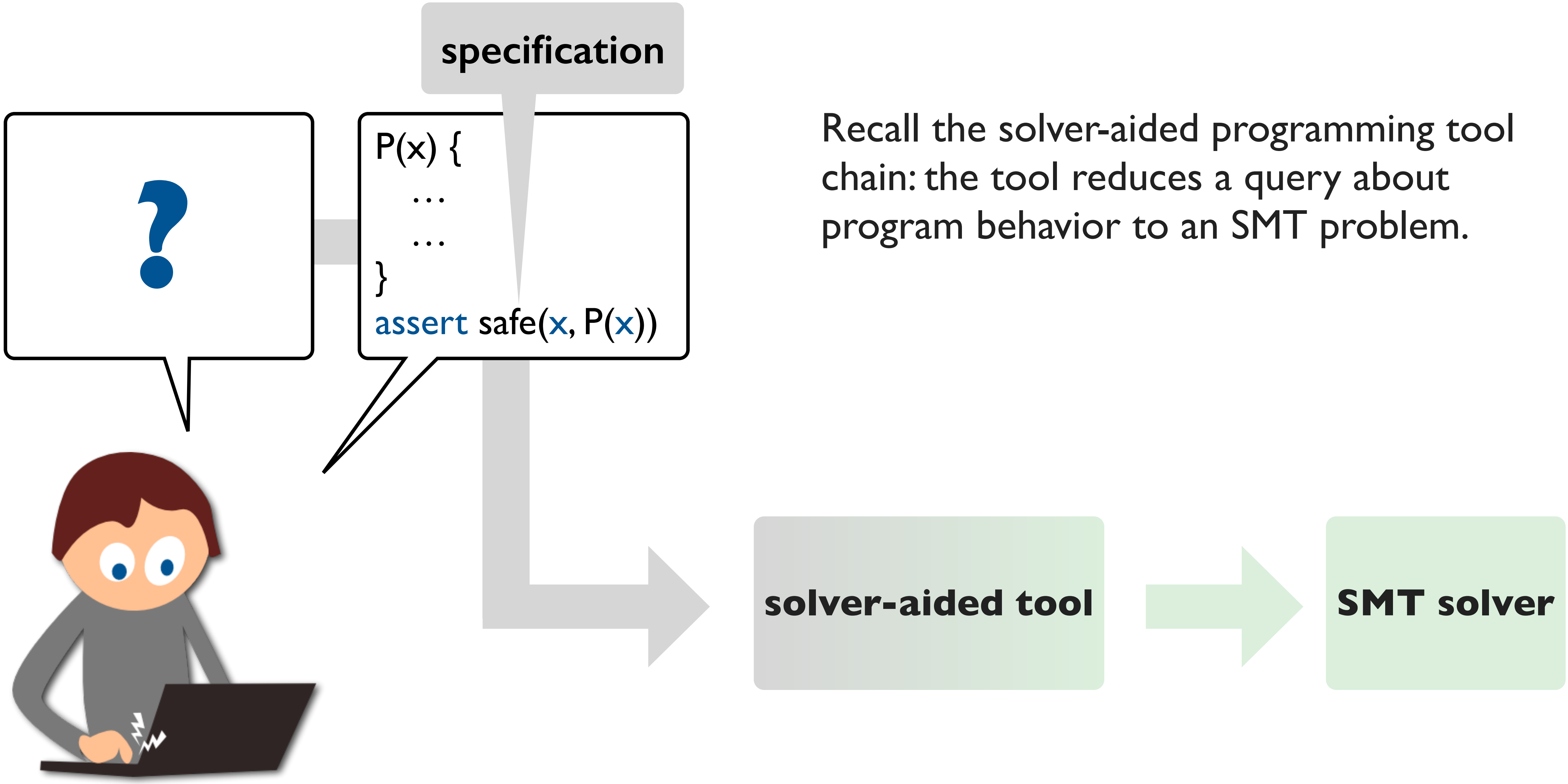
**Behind the scenes: symbolic virtual machine**
How Rosette works so you don’t have to.



**A last look: a few recent applications**
Cool tools built with Rosette!

# The classic (hard) way to build a tool

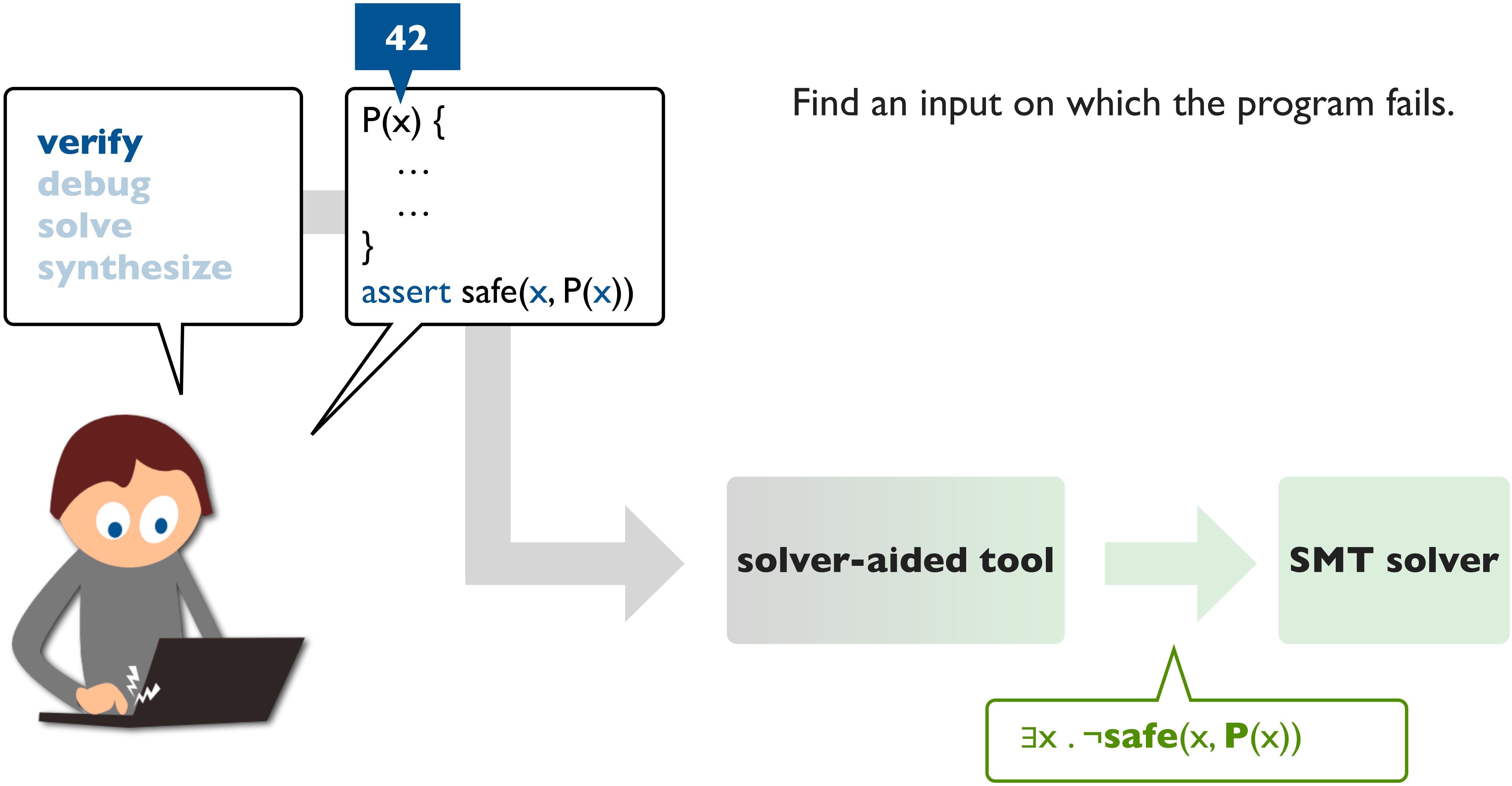
Recall the solver-aided programming tool chain: the tool reduces a query about program behavior to an SMT problem.



```
P(x) {
    …
    …
}
assert safe(x, P(x))
```

# The classic (hard) way to build a tool

Find an input on which the program fails.



verify
debug
solve
synthesize

42

```
P(x) {
    …
    …
}
assert safe(x, P(x))
```

solver-aided tool

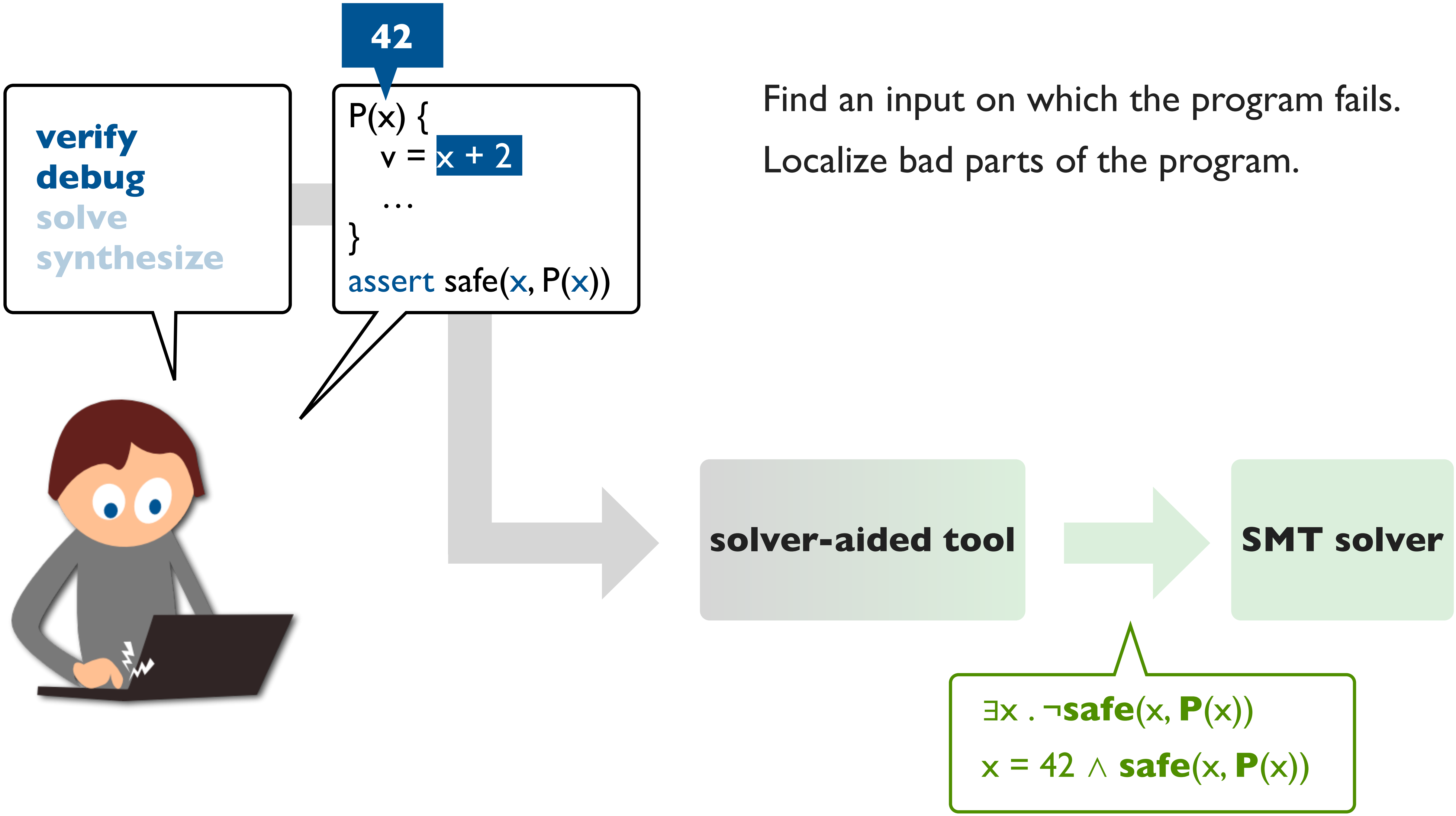
SMT solver

∃x . ¬**safe**(x, **P**(x))

# The classic (hard) way to build a tool

Find an input on which the program fails.

Localize bad parts of the program.



verify
debug
solve
synthesize

42

```
P(x) {
  v = x + 2
  …
}
assert safe(x, P(x))
```

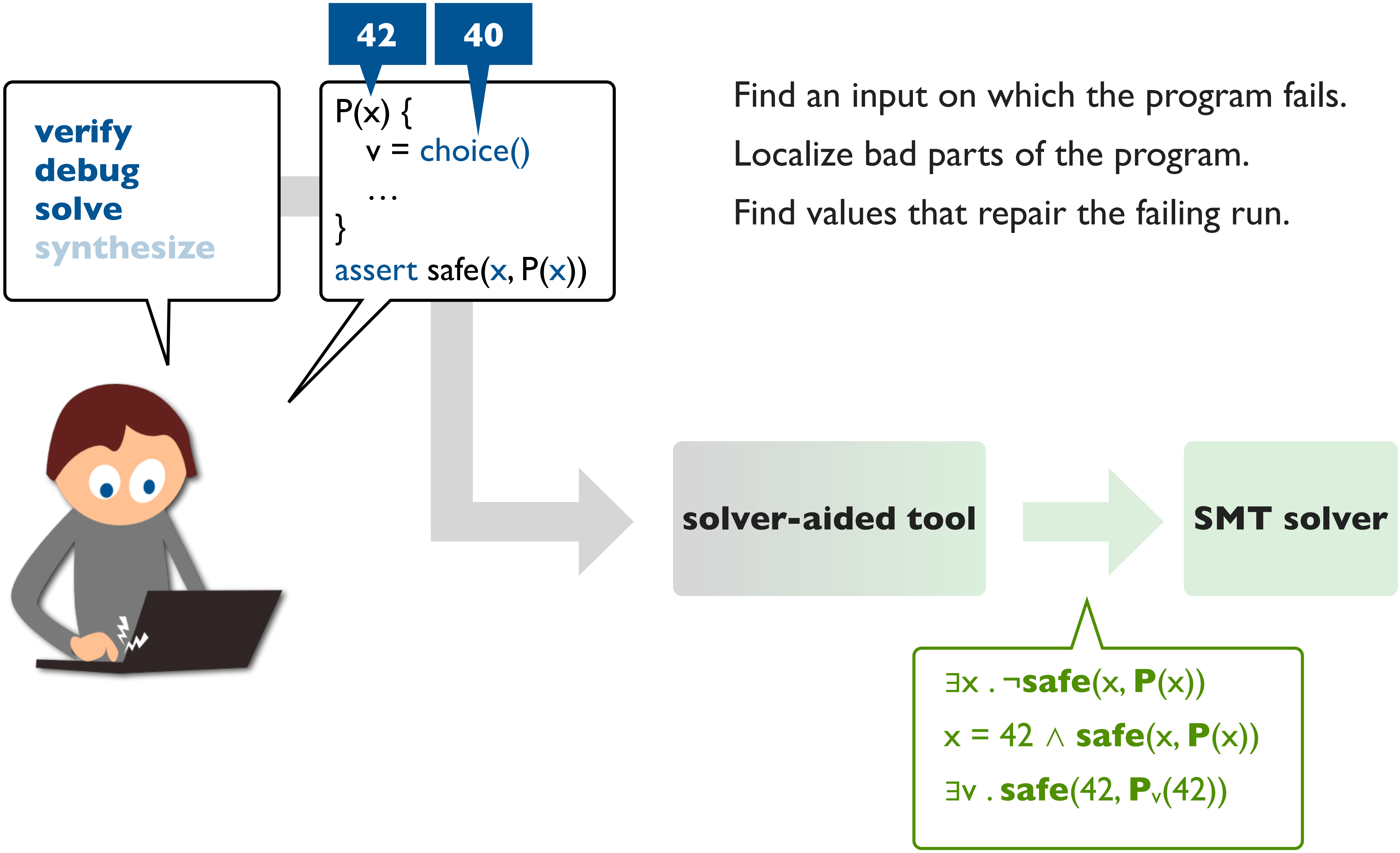
solver-aided tool

SMT solver

$\exists x \,.\, \neg \mathbf{safe}(x, \mathbf{P}(x))$

$x = 42 \wedge \mathbf{safe}(x, \mathbf{P}(x))$

# The classic (hard) way to build a tool



verify
debug
solve
synthesize

42

40

```
P(x) {
   v = choice()
   …
}
assert safe(x, P(x))
```

solver-aided tool

SMT solver

$\exists x \,.\, \neg\mathbf{safe}(x, \mathbf{P}(x))$

$x = 42 \wedge \mathbf{safe}(x, \mathbf{P}(x))$

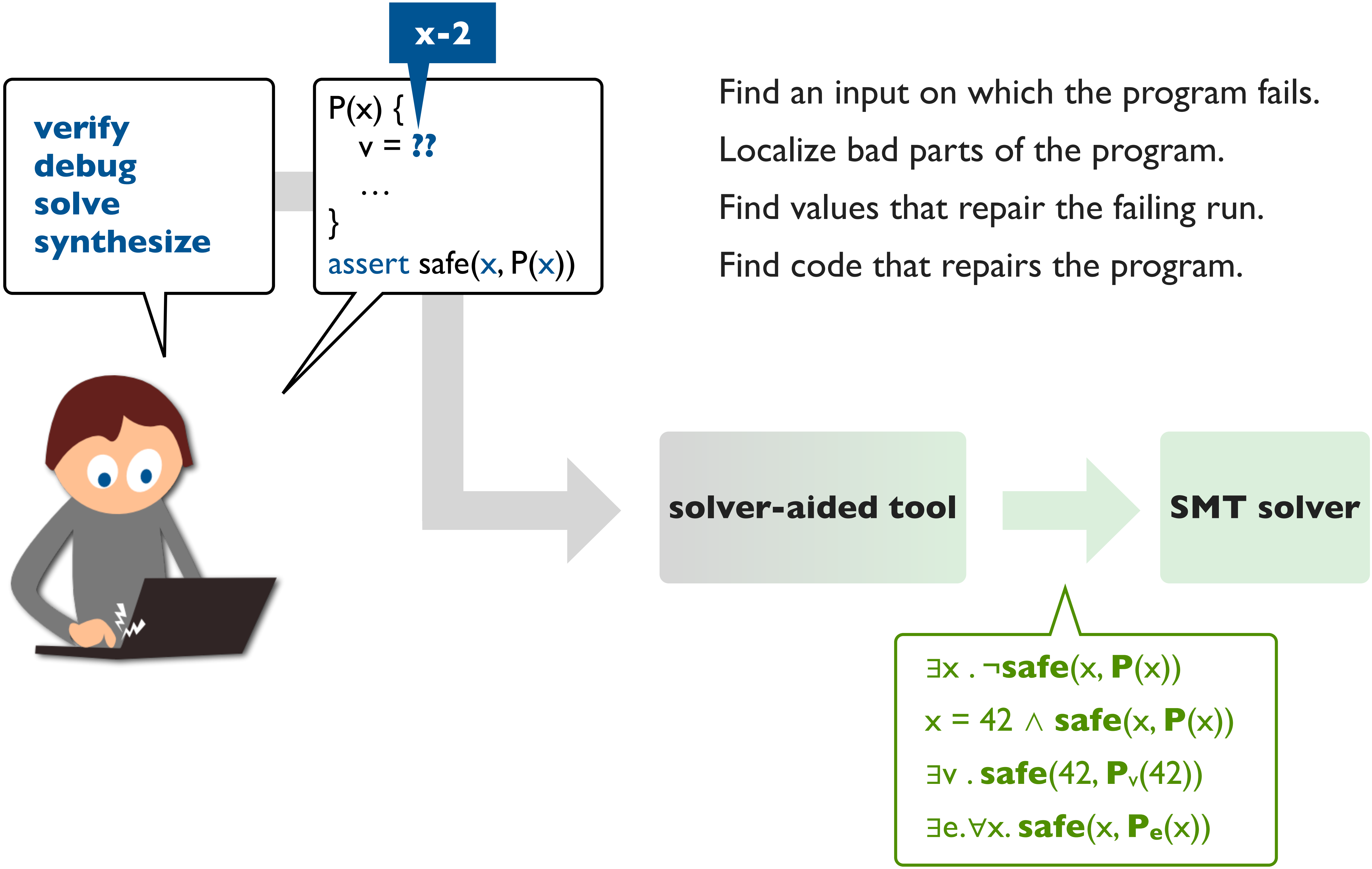
$\exists v \,.\, \mathbf{safe}(42, \mathbf{P}_v(42))$

Find an input on which the program fails.

Localize bad parts of the program.

Find values that repair the failing run.

# The classic (hard) way to build a tool



```
P(x) {
   v = ??
   …
}
assert safe(x, P(x))
```

Find an input on which the program fails.

Localize bad parts of the program.

Find values that repair the failing run.

Find code that repairs the program.

$\exists x\ .\ \neg\mathbf{safe}(x, \mathbf{P}(x))$

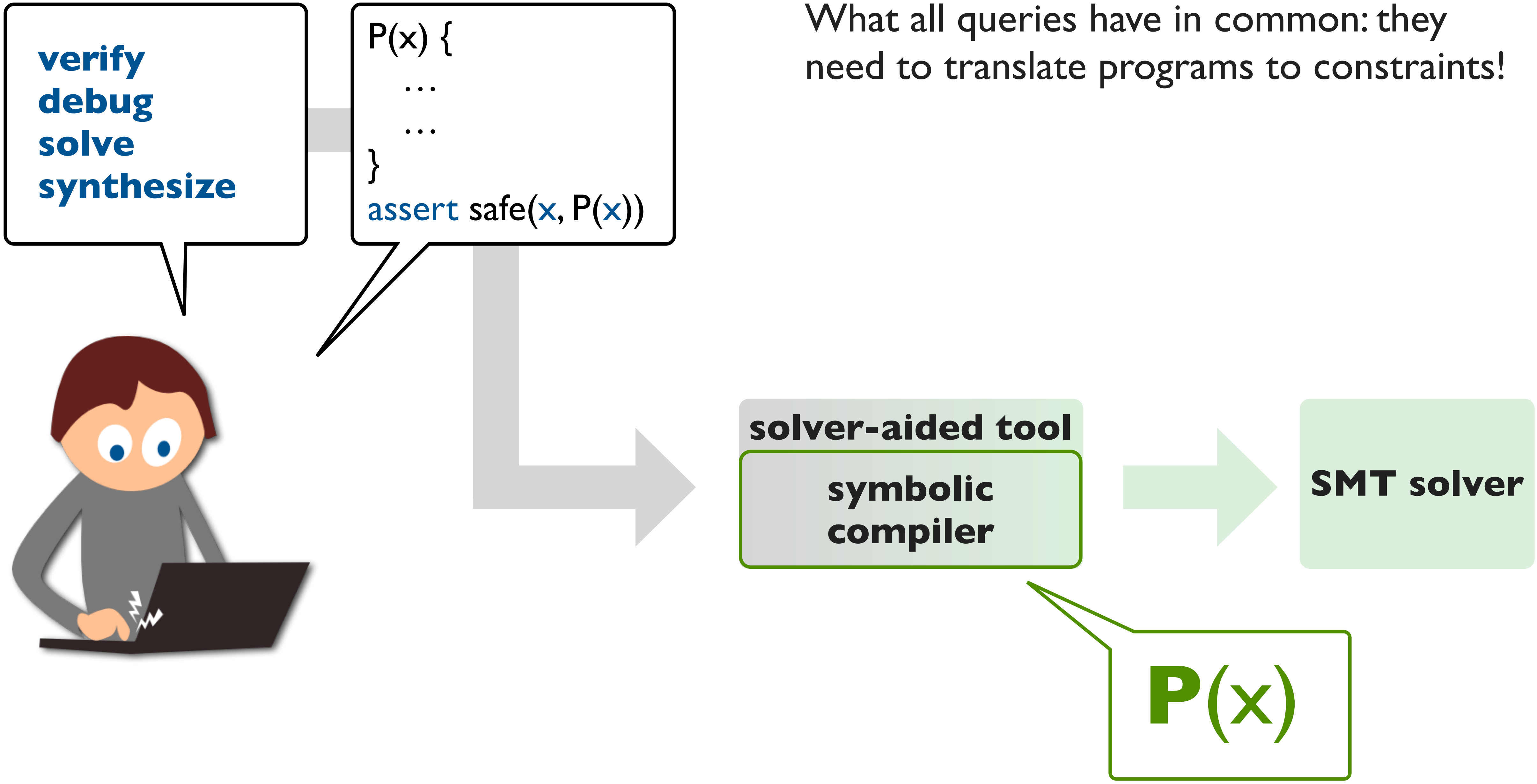
$x = 42\ \wedge\ \mathbf{safe}(x, \mathbf{P}(x))$

$\exists v\ .\ \mathbf{safe}(42, \mathbf{P}_v(42))$

$\exists e.\forall x.\ \mathbf{safe}(x, \mathbf{P}_e(x))$

# The classic (hard) way to build a tool

What all queries have in common: they need to translate programs to constraints!



**verify**
**debug**
**solve**
**synthesize**

```
P(x) {
   …
   …
}
assert safe(x, P(x))
```

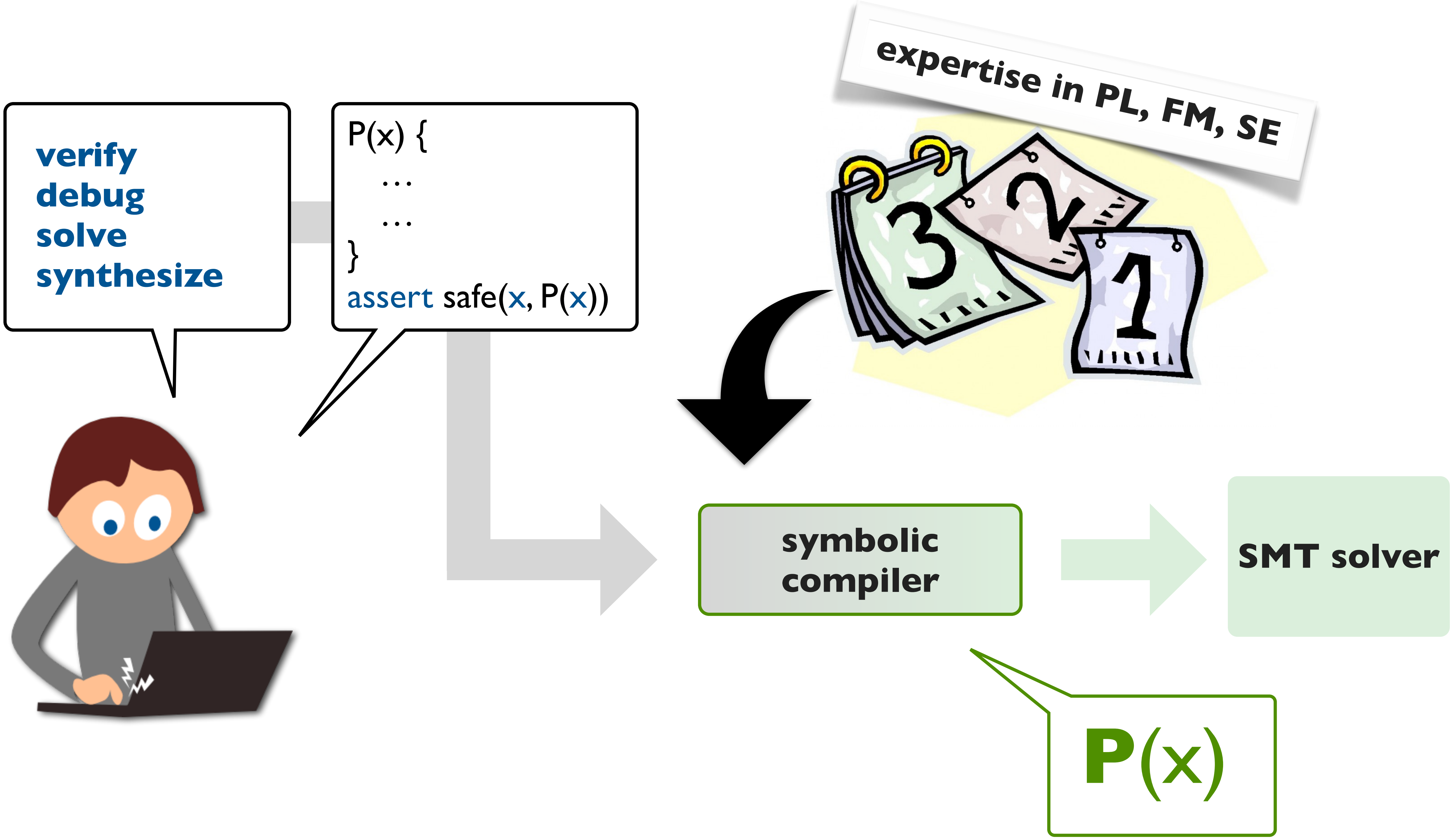
**solver-aided tool**

**symbolic compiler**

**SMT solver**

**P**(x)

# The classic (hard) way to build a tool



verify
debug
solve
synthesize

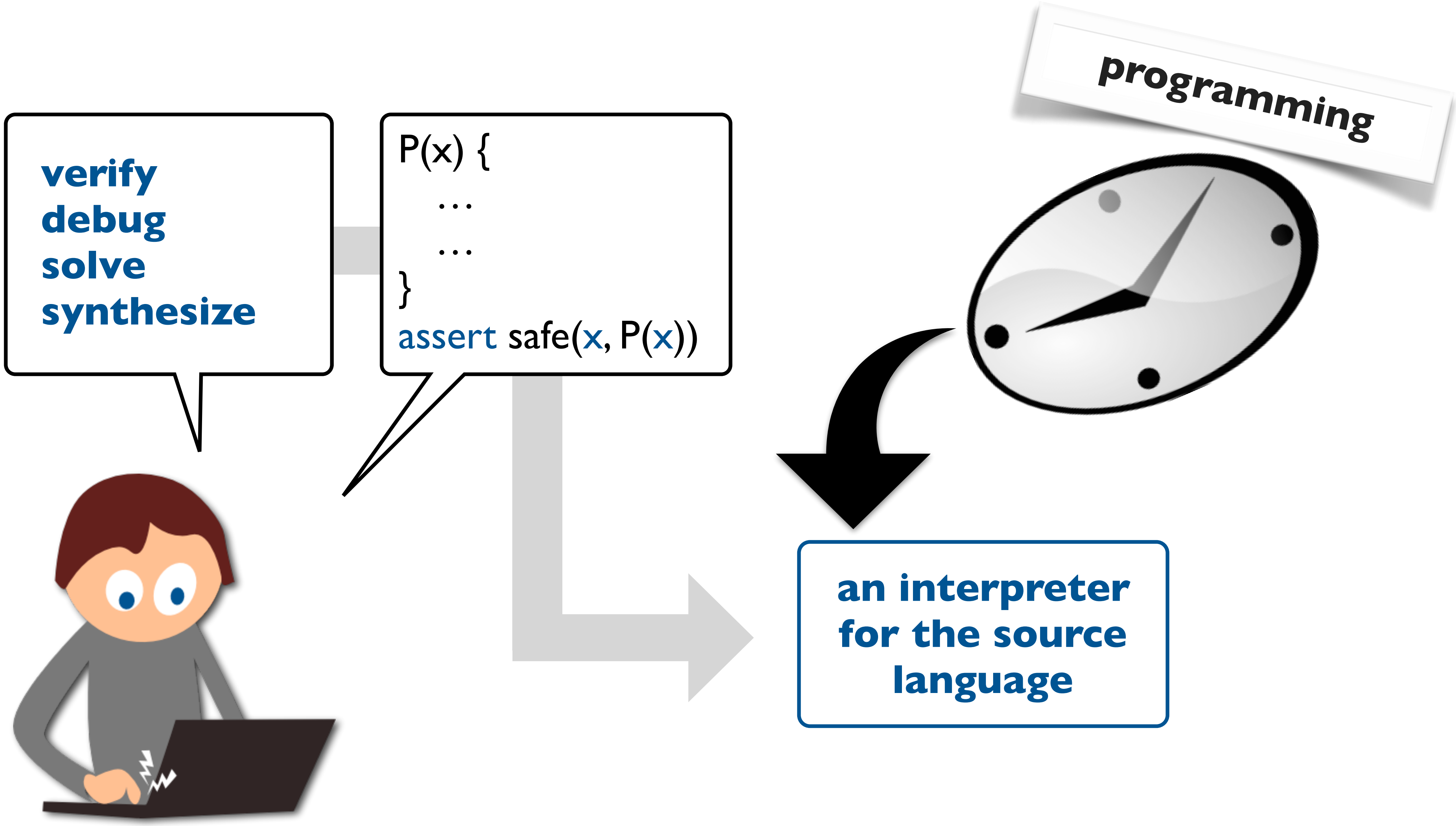
```
P(x) {
    …
    …
}
assert safe(x, P(x))
```

expertise in PL, FM, SE

symbolic compiler

SMT solver

# Wanted: an easier way to build tools



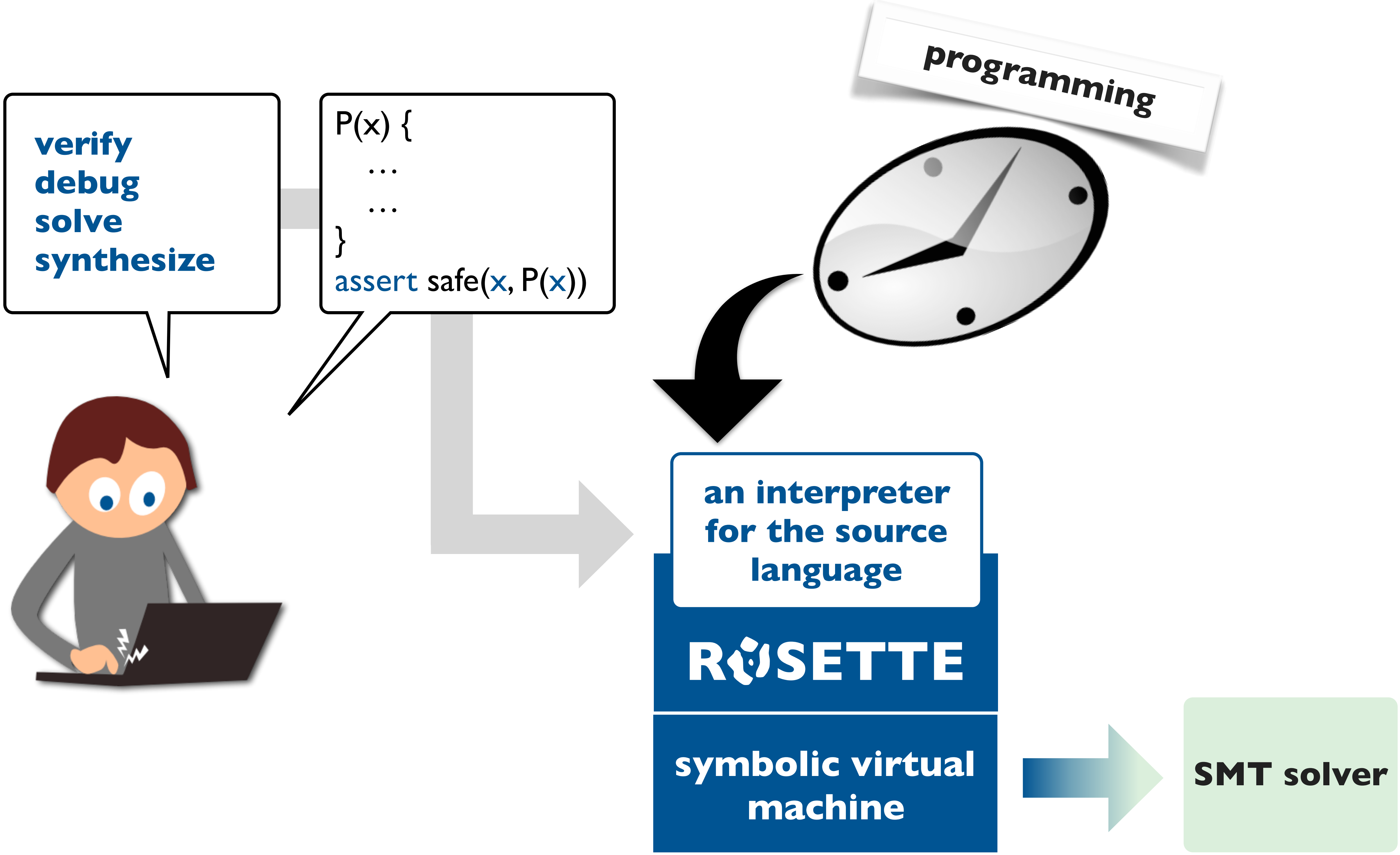
verify
debug
solve
synthesize

```
P(x) {
    …
    …
}
assert safe(x, P(x))
```

programming

an interpreter for the source language

# Wanted: an easier way to build tools



verify
debug
solve
synthesize

```
P(x) {
   …
   …
}
assert safe(x, P(x))
```

programming

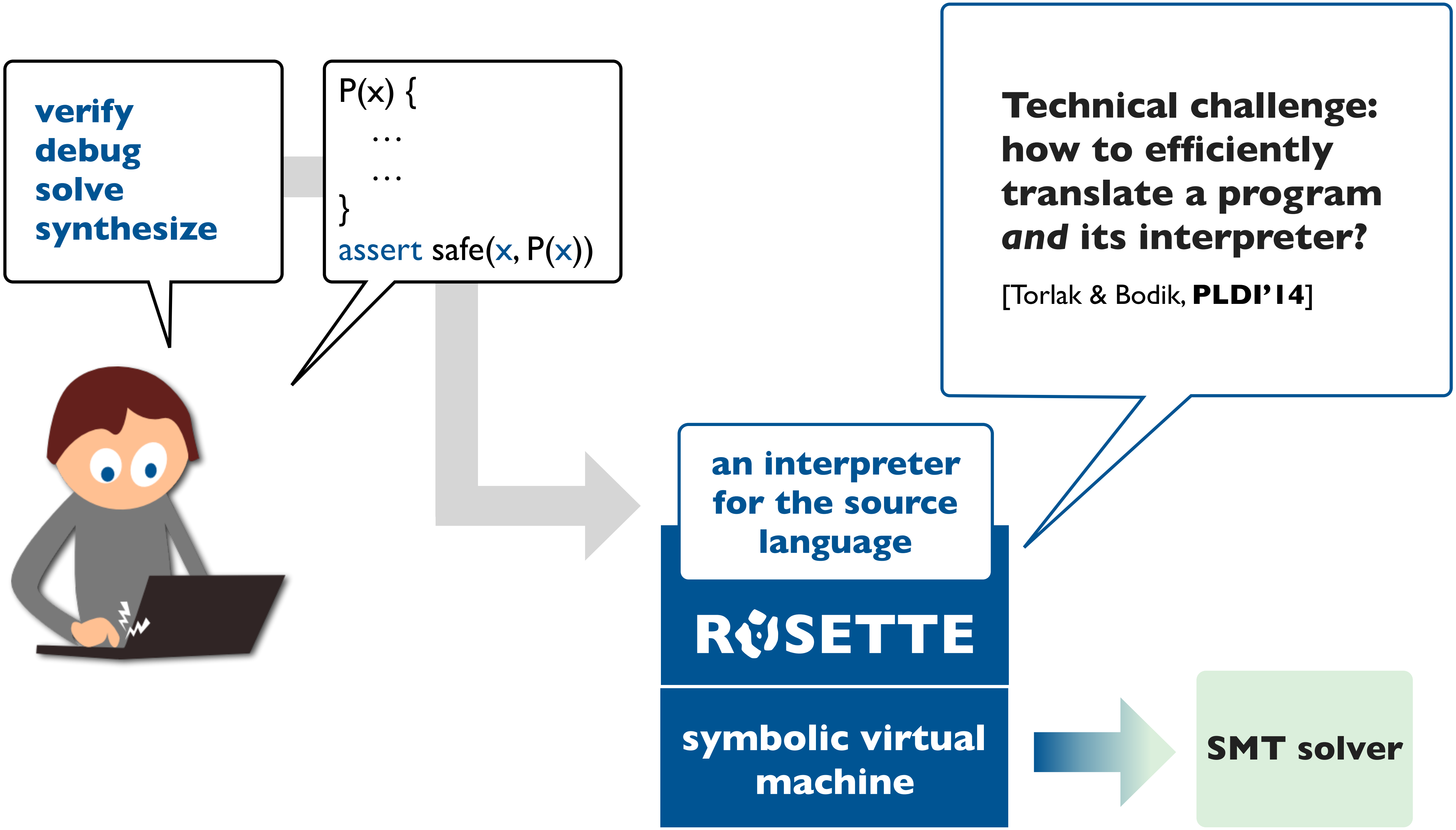
an interpreter for the source language

ROSETTE

symbolic virtual machine

SMT solver

# Wanted: an easier way to build tools



verify
debug
solve
synthesize

```
P(x) {
   …
   …
}
assert safe(x, P(x))
```

**Technical challenge: how to efficiently translate a program *and* its interpreter?**

[Torlak & Bodik, **PLDI'14**]

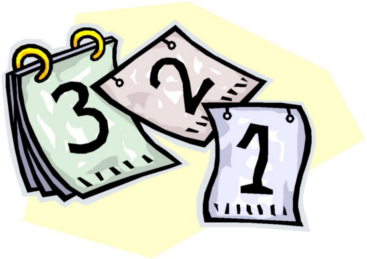
an interpreter for the source language

ROSETTE

symbolic virtual machine

SMT solver

# How to build your own solver-aided tool

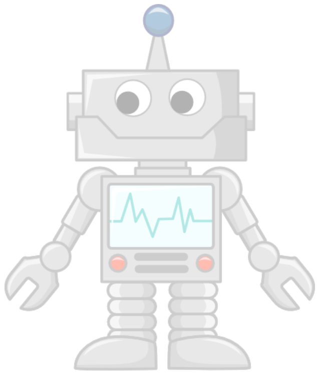


**The classic (hard) way to build a tool**
What is hard about building a solver-aided tool?

SDSL
SVM
SMT

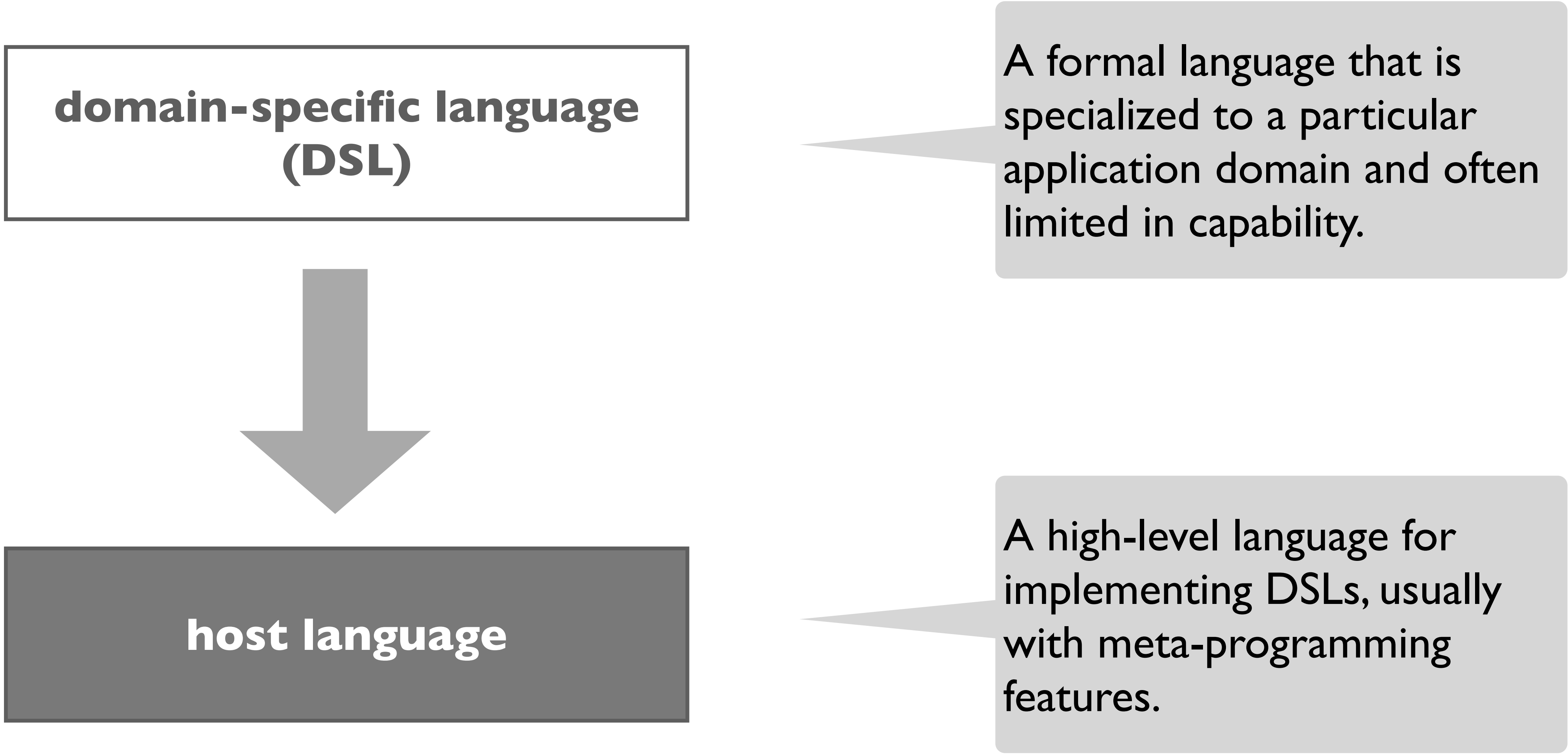
**An easier way: tools as languages**
How to build tools by stacking layers of languages.

**Behind the scenes: symbolic virtual machine**
How Rosette works so you don’t have to.



**A last look: a few recent applications**
Cool tools built with Rosette!

# Layers of classic languages: DSLs and hosts



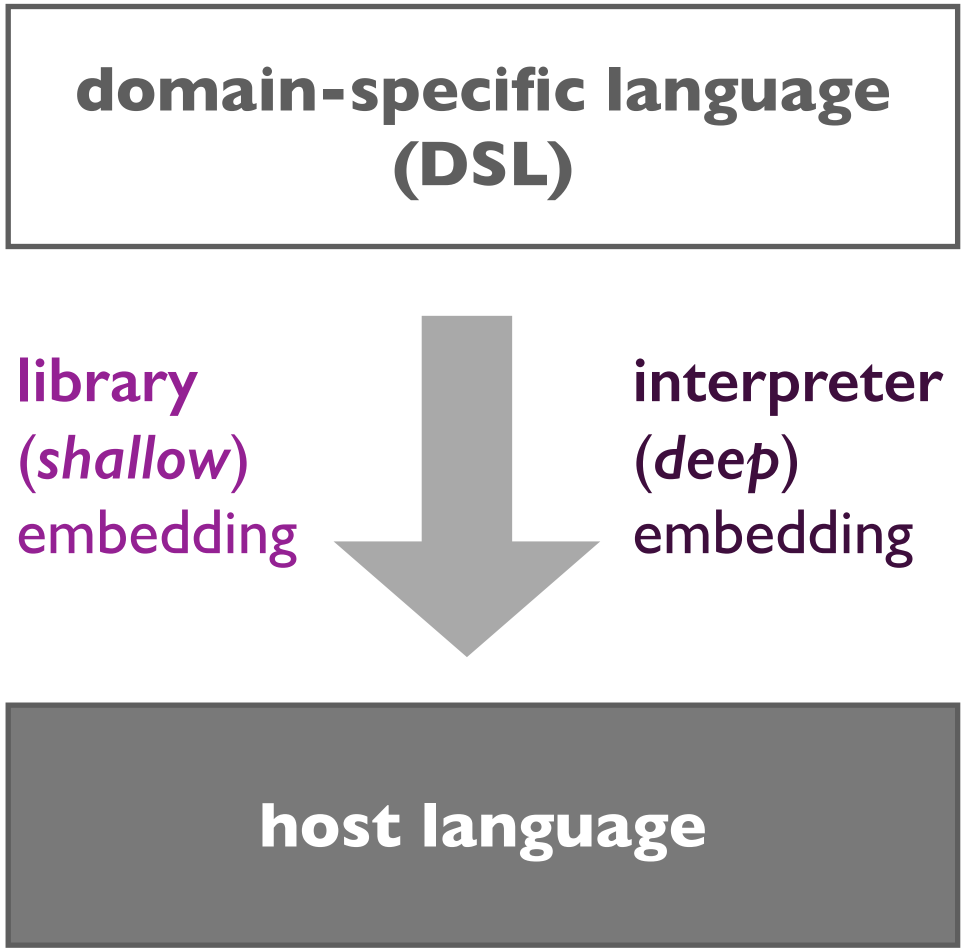
domain-specific language (DSL)

A formal language that is specialized to a particular application domain and often limited in capability.

host language

A high-level language for implementing DSLs, usually with meta-programming features.

# Layers of classic languages: DSLs and hosts



**domain-specific language (DSL)**

library (*shallow*) embedding

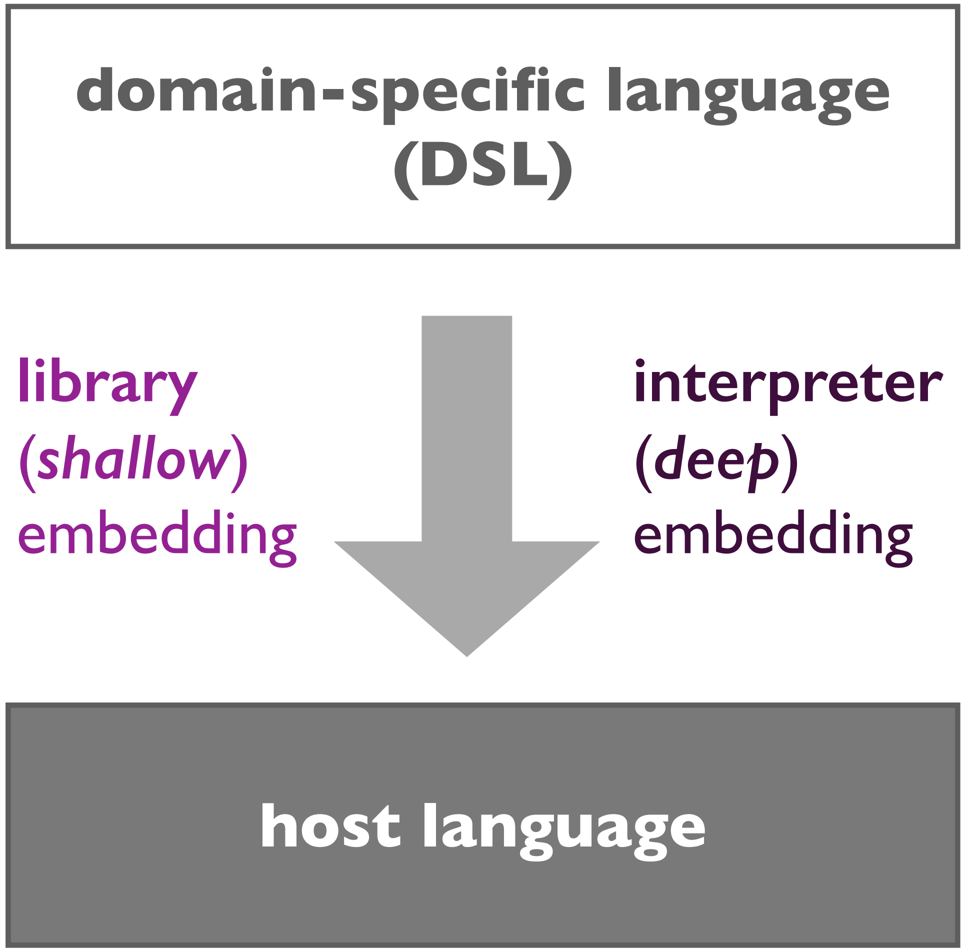
interpreter (*deep*) embedding

**host language**

A formal language that is specialized to a particular application domain and often limited in capability.

A high-level language for implementing DSLs, usually with meta-programming features.

# Layers of classic languages: many DSLs and hosts



domain-specific language (DSL)

library (*shallow*) embedding

interpreter (*deep*) embedding

host language

**artificial intelligence**
Church, BLOG

**databases**
SQL, Datalog

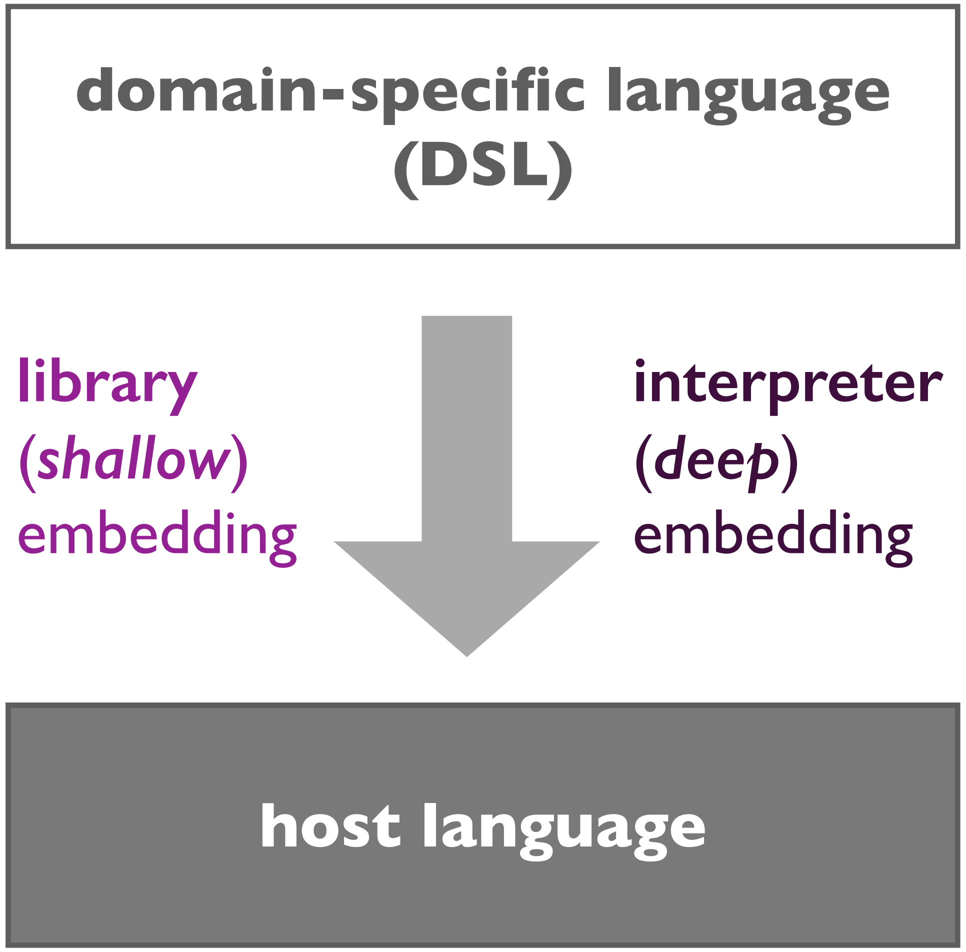
**hardware design**
Bluespec, Chisel, Verilog, VHDL

**math and statistics**
Eigen, Matlab, R

**layout and visualization**
LaTex, dot, dygraphs, D3

Racket, Scala, JavaScript, …

# Layers of classic languages: why DSLs?


domain-specific language (DSL)
library (*shallow*) embedding
interpreter (*deep*) embedding
host language

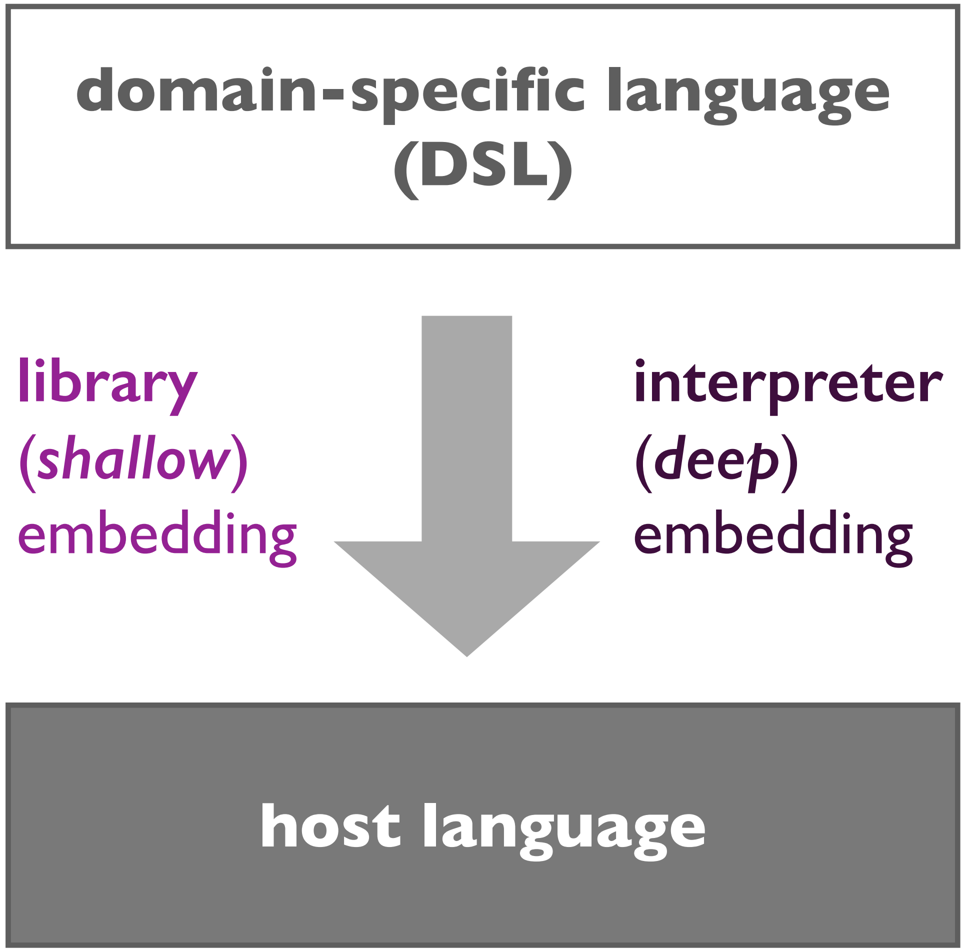
Eigen / Matlab

```
C = A ∗ B
```

C / Java

```
for (i = 0; i < n; i++)
 for (j = 0; j < m; j++)
  for (k = 0; k < p; k++)
   C[i][k] += A[i][j] ∗ B[j][k]
```

# Layers of classic languages: why DSLs?



domain-specific language (DSL)

library (*shallow*) embedding

interpreter (*deep*) embedding

host language

Easier for people to read, write, and get right.

Eigen / Matlab

```
C = A ∗ B
```

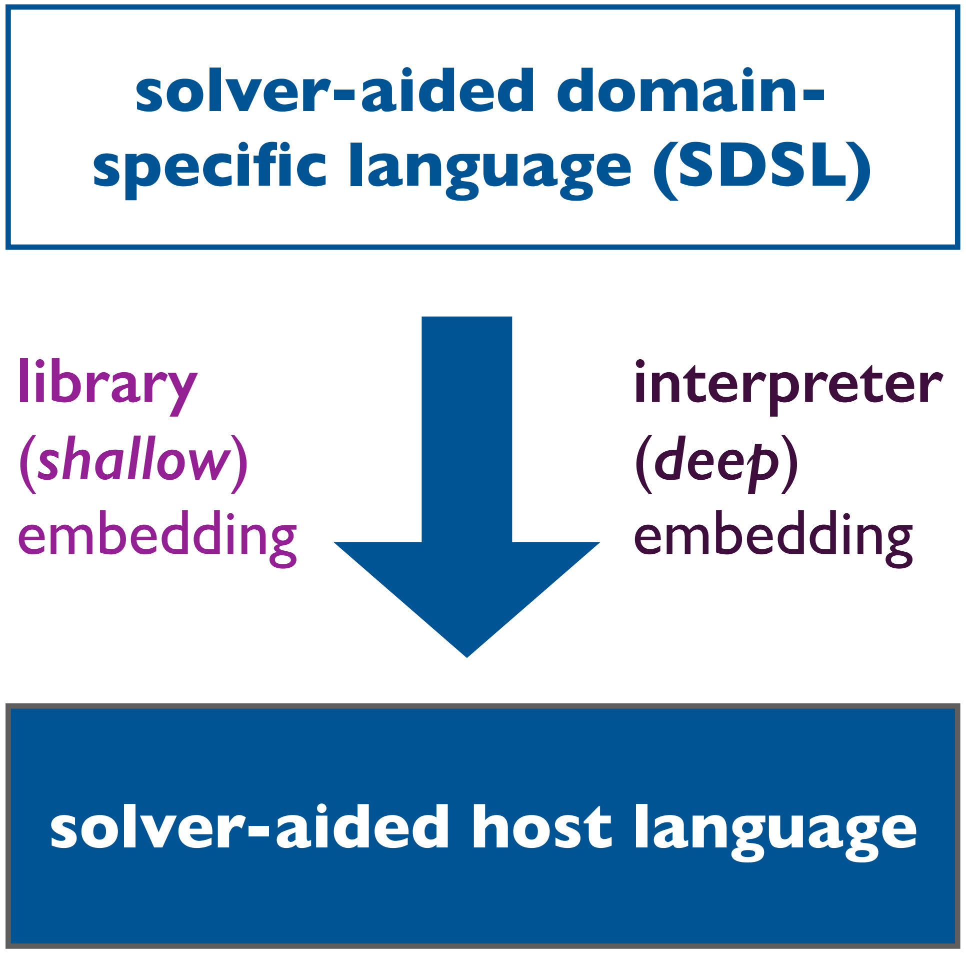
[associativity]

Easier for tools to analyze.

C / Java

```
for (i = 0; i < n; i++)
 for (j = 0; j < m; j++)
  for (k = 0; k < p; k++)
   C[i][k] += A[i][j] ∗ B[j][k]
```

# Layers of solver-aided languages



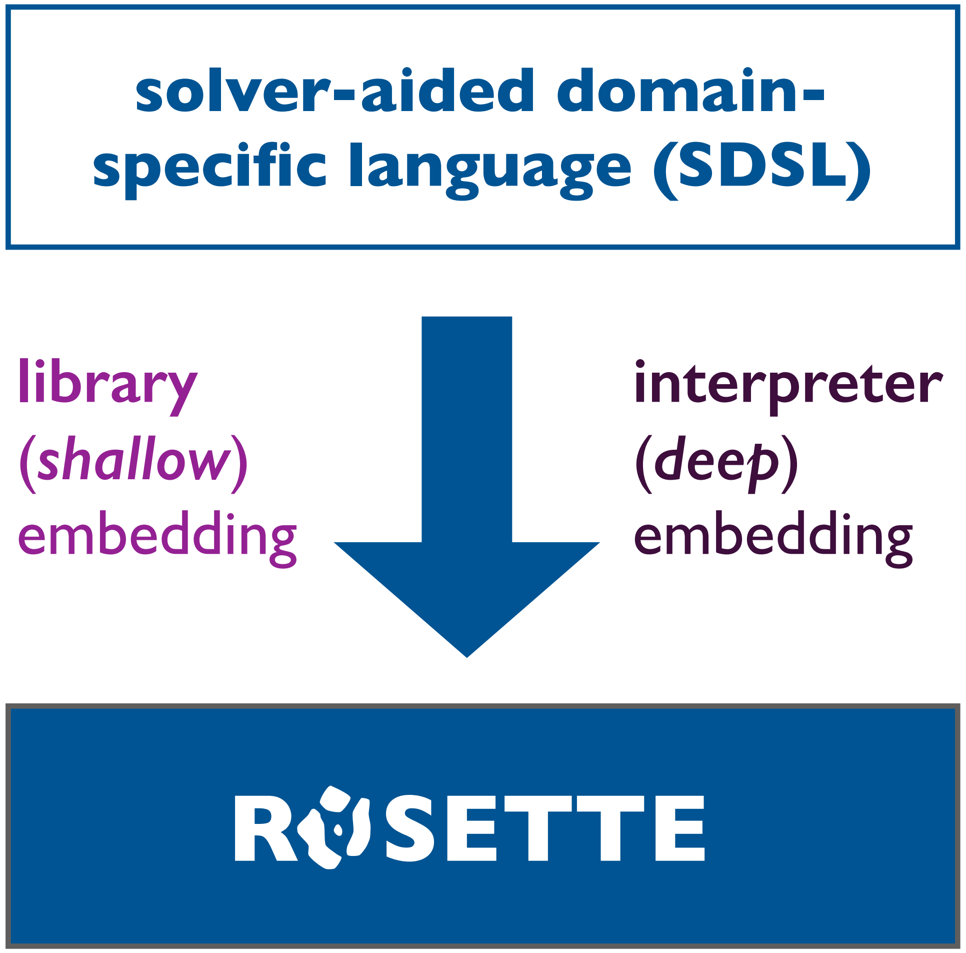
**solver-aided domain-specific language (SDSL)**

library (*shallow*) embedding

interpreter (*deep*) embedding

**solver-aided host language**

# Layers of solver-aided languages: tools as SDSLs



solver-aided domain-specific language (SDSL)

library (*shallow*) embedding

interpreter (*deep*) embedding

ROSETTE

**education and games**
Enlearn, RuleSy (VMCAI'18), Nonograms (FDG'17), UCB feedback generator (ITiCSE'17)

**synthesis-aided compilation**
LinkiTools, Chlorophyll (PLDI'14), GreenThumb (ASPLOS'16)

**type system soundness**
Bonsai (POPL'18)

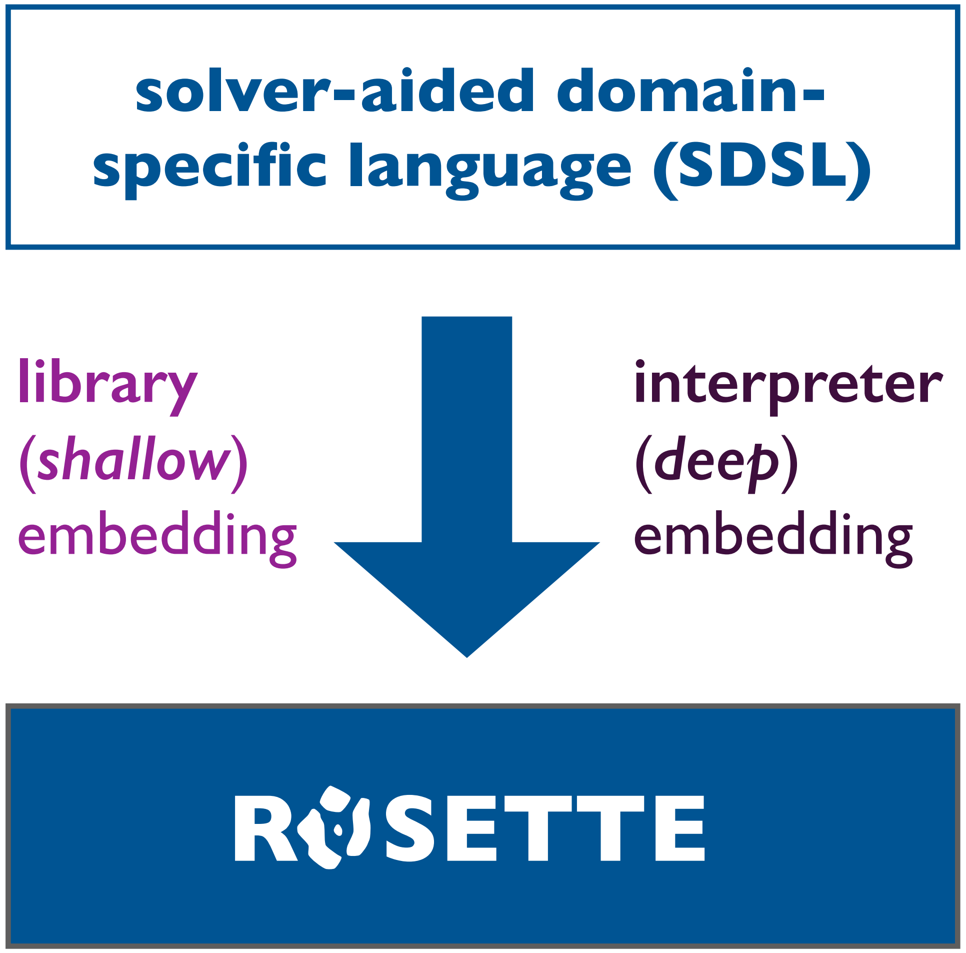
**computer architecture**
MemSynth (PLDI'17)

**databases**
Cosette (CIDR'17)

**radiation therapy control**
Neutrons (CAV'16)

**… and more**

# Layers of solver-aided languages: tools as SDSLs



**solver-aided domain-specific language (SDSL)**

library (*shallow*) embedding

interpreter (*deep*) embedding

ROSETTE

**education and games**
Enlearn, RuleSy (VMCAI'18), Nonograms (FDG'17), UCB feedback generator (ITiCSE'17)

**synthesis-aided compilation**
LinkiTools, Chlorophyll (PLDI'14), GreenThumb (ASPLOS'16)

**type system soundness**
Bonsai (POPL'18)

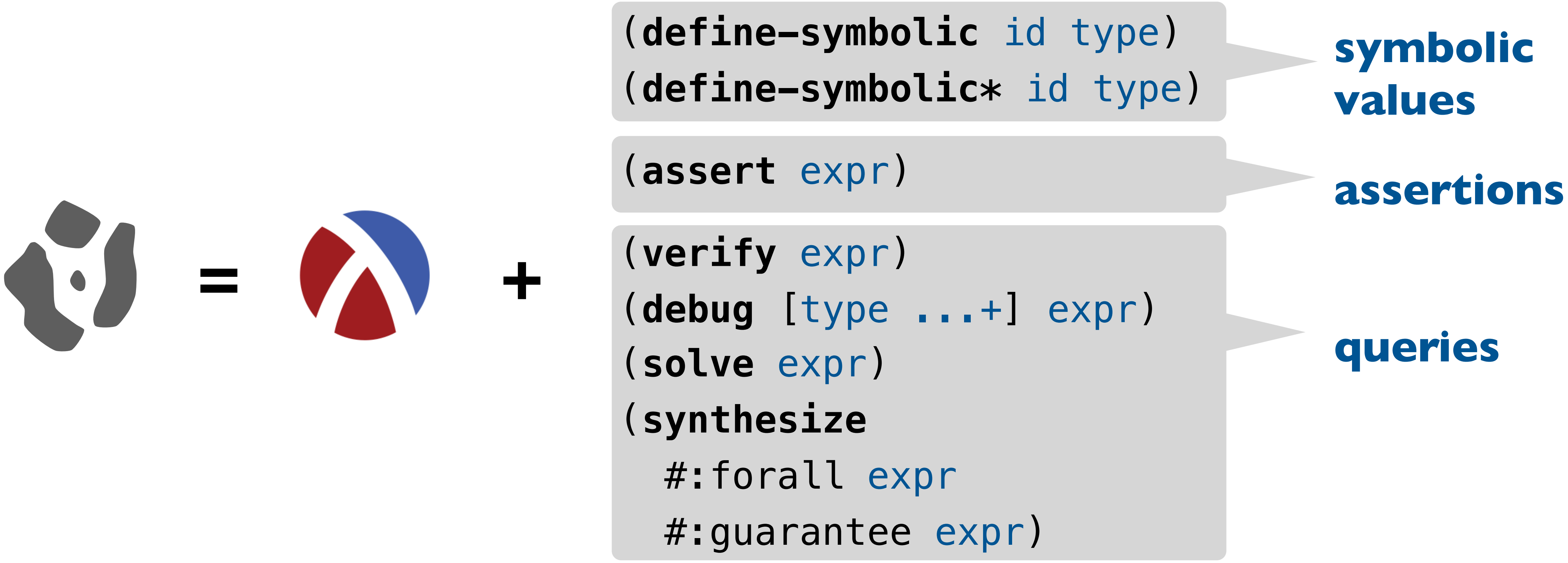
**computer architecture**
MemSynth (PLDI'17)

**databases**
Cosette (CIDR'17)

**radiation therapy control**
Neutrons (CAV'16)

**… and more**

# The anatomy of a solver-aided host language



= +

```
(define-symbolic id type)
(define-symbolic* id type)
```

symbolic values

```
(assert expr)
```

assertions

```
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
  #:forall expr
  #:guarantee expr)
```

queries

# A tiny example SDSL

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6
```

**BV**: A tiny assembly-like language for writing fast, low-level library functions.

# A tiny example SDSL

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6
```

**BV**: A tiny assembly-like language for writing fast, low-level library functions.

We want to **test**, **verify**, **debug**, and **synthesize** programs in the BV SDSL.

# A tiny example SDSL

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6
```

We want to **test**, **verify**, **debug**, and **synthesize** programs in the BV SDSL.

**BV**: A tiny assembly-like language for writing fast, low-level library functions.

| | | |
|---|---|---|
| 1. | interpreter | [10 LOC] |
| 2. | verifier | [free] |
| 3. | debugger | [free] |
| 4. | synthesizer | [free] |

# A tiny example SDSL

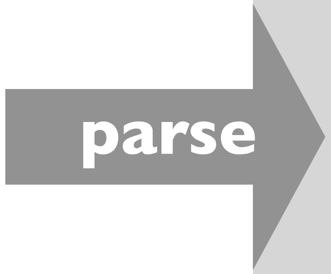
ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> bvmax(-2, -1)
```

# A tiny example SDSL

ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> bvmax(-2, -1)
```



```
(define bvmax
 `((2 bvsge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
```

# A tiny example SDSL

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> bvmax(-2, -1)
```

parse

```
(define bvmax
 `((2 bvsge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
```

```
(out opcode in ...)
```

# A tiny example SDSL

ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> bvmax(-2, -1)
```

interpret

```
(define bvmax
 `((2 bvsge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
```

```
`(-2 -1)
```

```
(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)))
```

# A tiny example SDSL

ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> bvmax(-2, -1)
```

```
(define bvmax
 `((2 bvsge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
```

| | |
|---|---|
| 0 | -2 |
| 1 | -1 |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |

```
(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)))
```

# A tiny example SDSL

# ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> bvmax(-2, -1)
```

```
(define bvmax
 `((2 bvsge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
```

| | |
|---|---|
| 0 | -2 |
| 1 | -1 |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |

```
(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)))
```

# A tiny example SDSL

ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> bvmax(-2, -1)
```

interpret

```
(define bvmax
 `((2 bvsge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
```

| | |
|---|---|
| 0 | -2 |
| 1 | -1 |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |

```
(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)))
```

# A tiny example SDSL

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> bvmax(-2, -1)
```

interpret

## ROSETTE

```
(define bvmax
 `((2 bvsge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
```

| | |
|---|---|
| 0 | -2 |
| 1 | -1 |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |

```
(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)))
```

# A tiny example SDSL

ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> bvmax(-2, -1)
```

interpret

```
(define bvmax
 `((2 bvsge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
```

| | |
|---|---|
| 0 | -2 |
| 1 | -1 |
| 2 | 0 |
| 3 | |
| 4 | |
| 5 | |
| 6 | |

```
(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)))
```

# A tiny example SDSL

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6
```

```
> bvmax(-2, -1)
```

interpret

ROSETTE

```
(define bvmax
 `((2 bvsge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
```

| | |
|---|---|
| 0 | -2 |
| 1 | -1 |
| 2 | 0 |
| 3 | 0 |
| 4 | -2 |
| 5 | 0 |
| 6 | -1 |

```
(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)))
```

# A tiny example SDSL

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6
```

```
> bvmax(-2, -1)
-1
```

interpret

ROSETTE

```
(define bvmax
 `((2 bvsge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
```

| | |
|---|---|
| 0 | -2 |
| 1 | -1 |
| 2 | 0 |
| 3 | 0 |
| 4 | -2 |
| 5 | 0 |
| 6 | -1 |

```
(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)))
```

# A tiny example SDSL

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> bvmax(-2, -1)
-1
```

# ROSETTE

```
(define bvmax
 `((2 bvsge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
```

- pattern matching
- dynamic evaluation
- first-class & higher-order procedures
- side effects

```
(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)))
```

# A tiny example SDSL

ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> verify(bvmax, max)
```

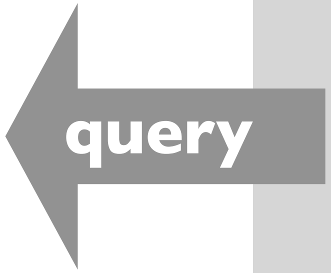
query

```
(define-symbolic* in (bitvector 32) [2])
(verify
  (assert (equal? (interpret bvmax in)
                  (interpret max in))))
```

# A tiny example SDSL

ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> verify(bvmax, max)
(0, -2)
```


query

```
(define-symbolic* in (bitvector 32) [2])
(verify
  (assert (equal? (interpret bvmax in)
                  (interpret max in))))
```

# A tiny example SDSL

ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> verify(bvmax, max)
(0, -2)

> bvmax(0, -2)
-1
```

query

```
(define-symbolic* in (bitvector 32) [2])
(verify
  (assert (equal? (interpret bvmax in)
                  (interpret max in))))
```

# A tiny example SDSL

ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6
```

```
> debug(bvmax, max,‘(0, -2))
```

query

```
(define in (list (bv 0 32) (bv -2 32)))
(debug [integer?]
  (assert (equal? (interpret bvmax in)
                  (interpret max in))))
```

# A tiny example SDSL

ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6
```

```
> debug(bvmax, max,‘(0, -2))
```

query

```
(define in (list (bv 0 32) (bv -2 32)))
(debug [integer?]
  (assert (equal? (interpret bvmax in)
                  (interpret max in))))
```

# A tiny example SDSL

ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(??, ??)
  r5 = bvand(r3, ??)
  r6 = bvxor(??, ??)
  return r6

> synthesize(bvmax, max)
```

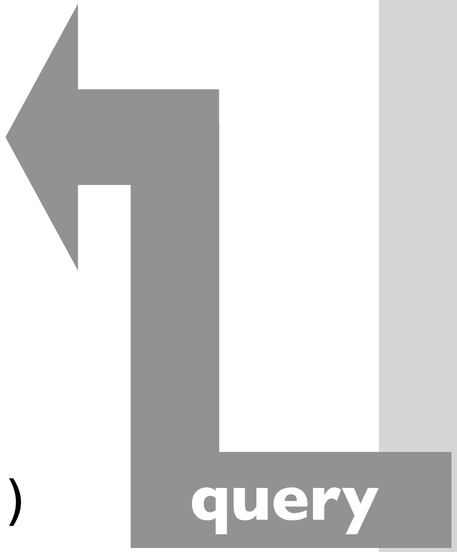
query

```
(define-symbolic* in (bitvector 32) [2])
(synthesize
  #:forall in
  #:guarantee
  (assert (equal? (interpret bvmax in)
                  (interpret max in)))))
```

# A tiny example SDSL

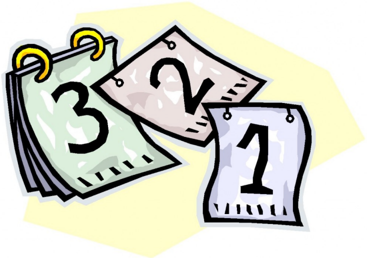
ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvsge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r1)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

> synthesize(bvmax, max)
```



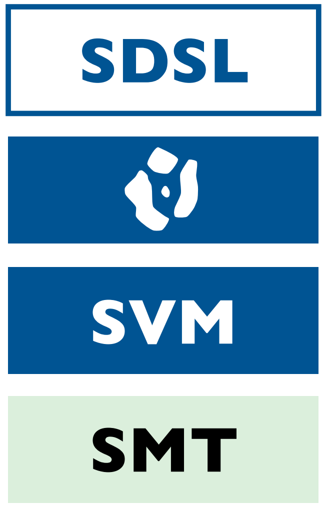
```
(define-symbolic* in (bitvector 32) [2])
(synthesize
  #:forall in
  #:guarantee
  (assert (equal? (interpret bvmax in)
                  (interpret max in)))))
```

# How to build your own solver-aided tool



**The classic (hard) way to build a tool**

What is hard about building a solver-aided tool?

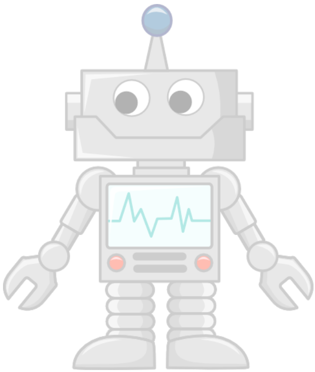


**An easier way: tools as languages**

How to build tools by stacking layers of languages.

**Behind the scenes: symbolic virtual machine**

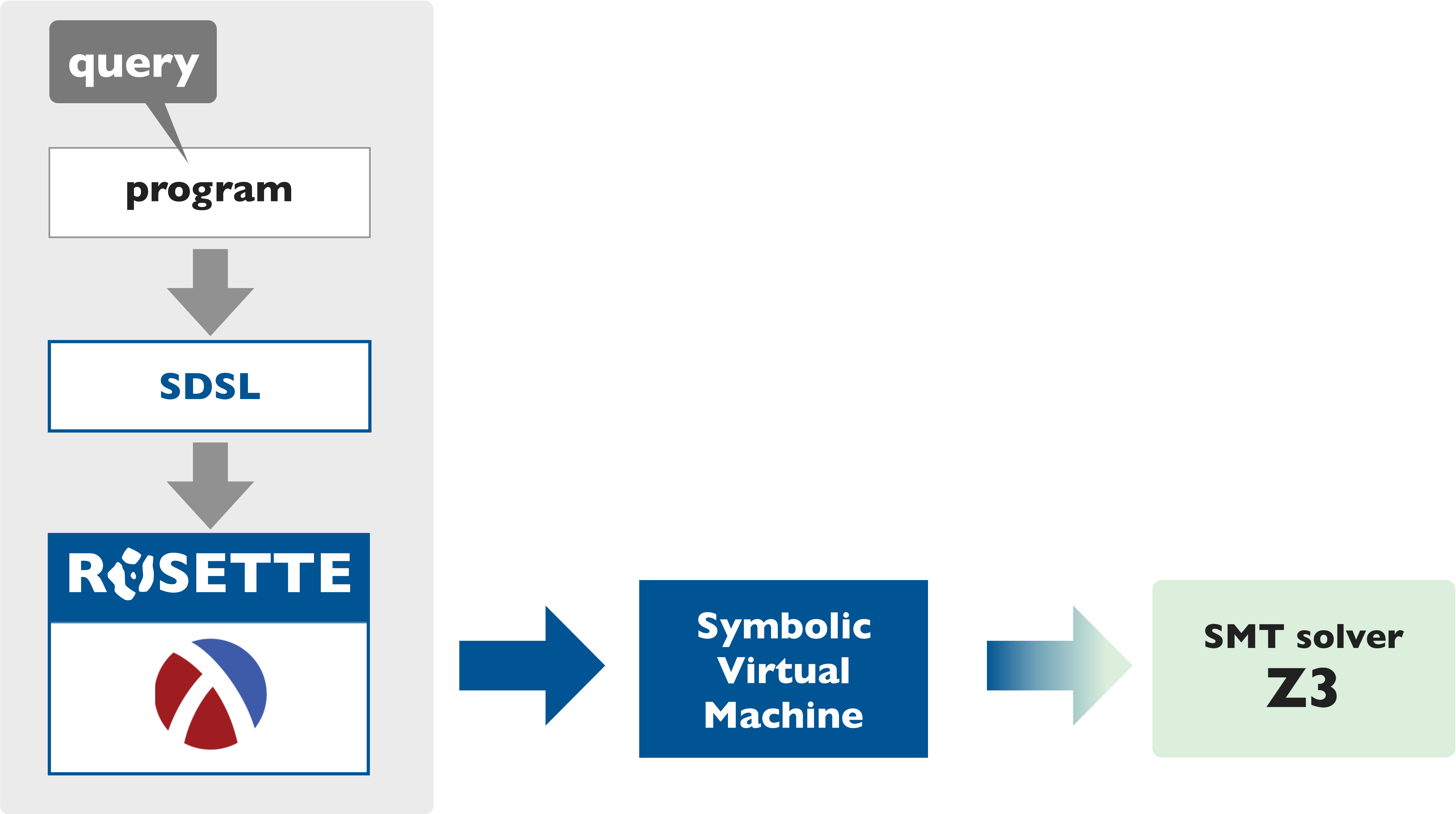
How Rosette works so you don't have to.



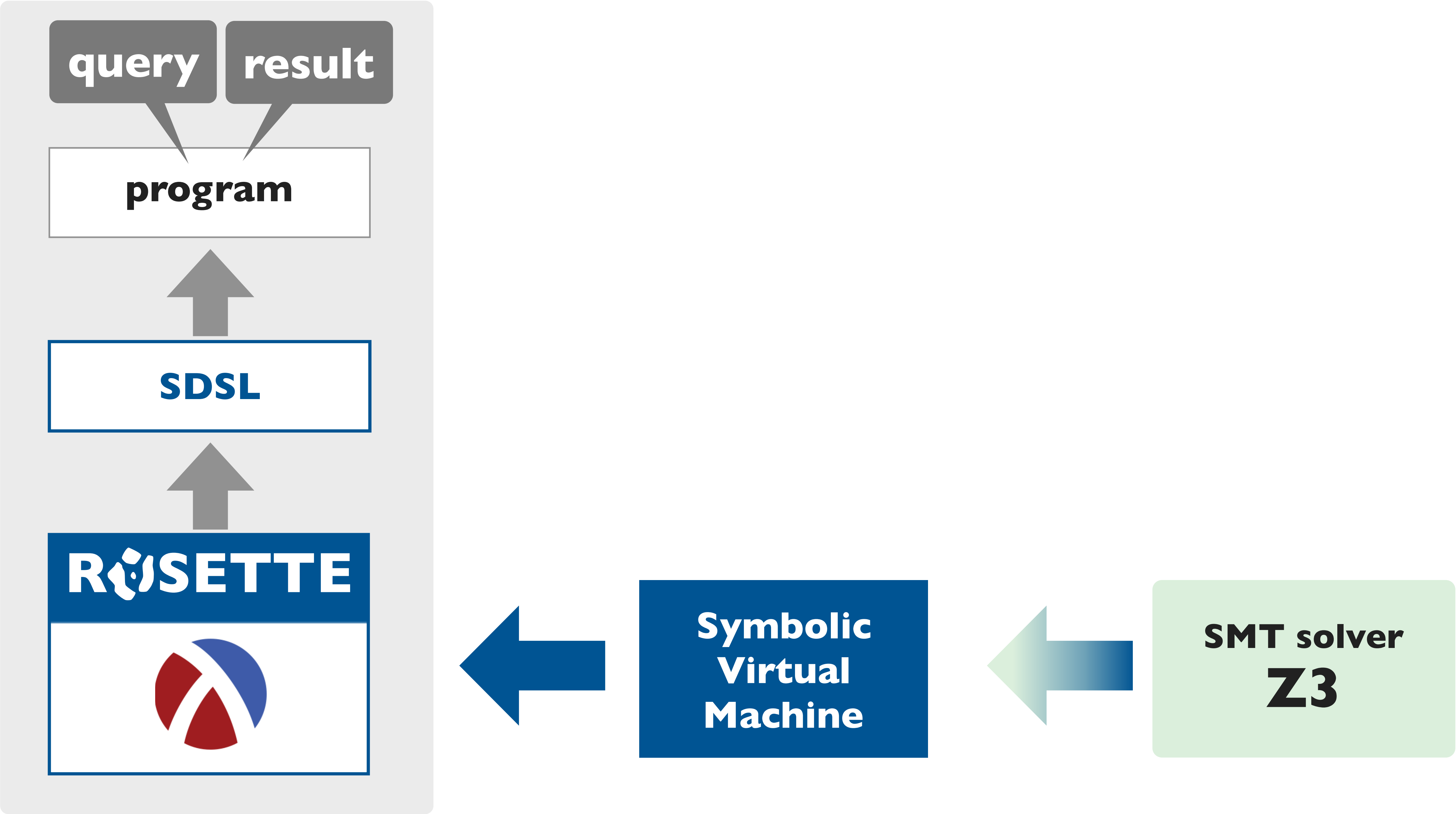
**A last look: a few recent applications**

Cool tools built with Rosette!

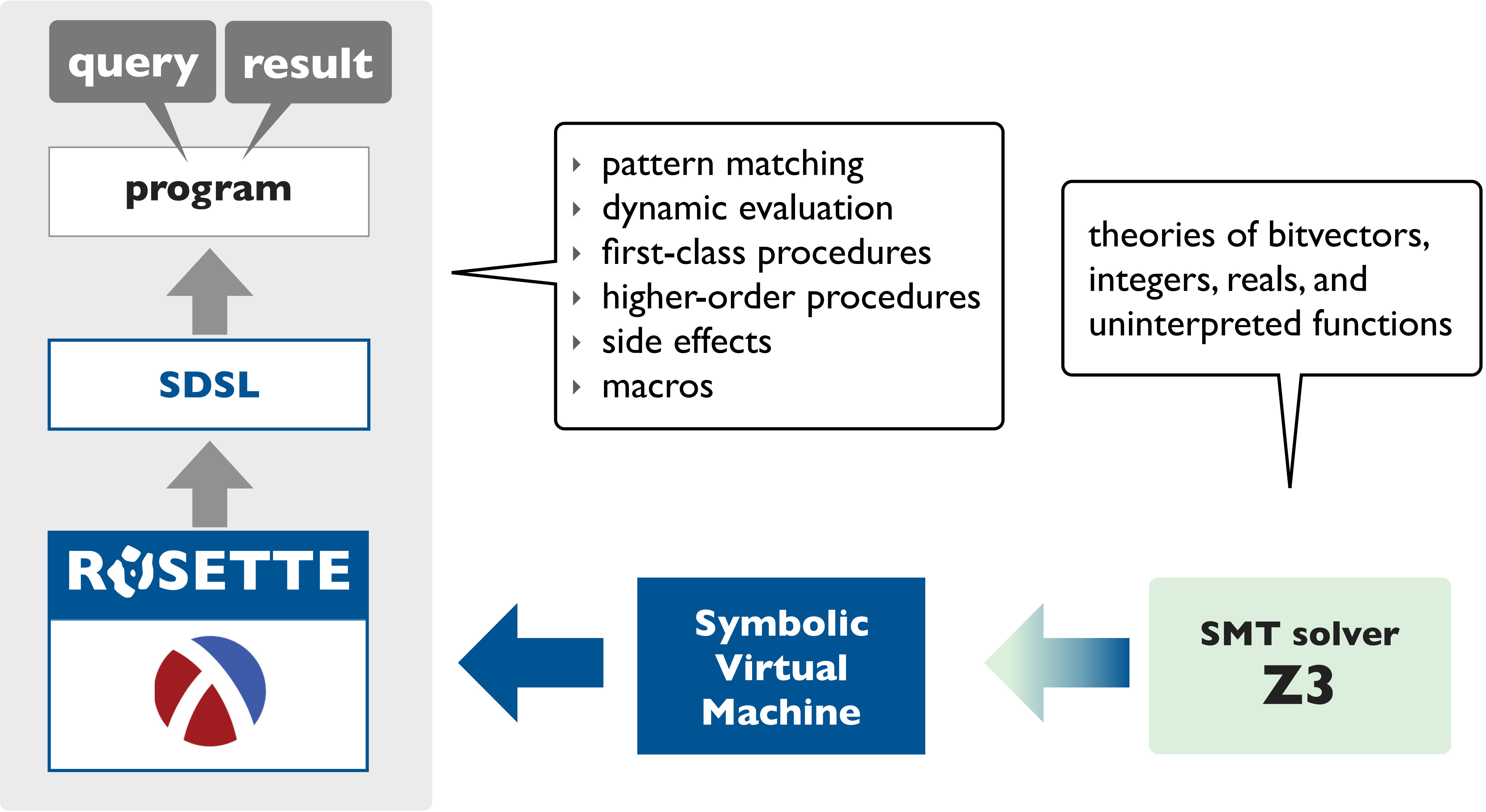
# How it all works:  a big picture view



# How it all works: a big picture view



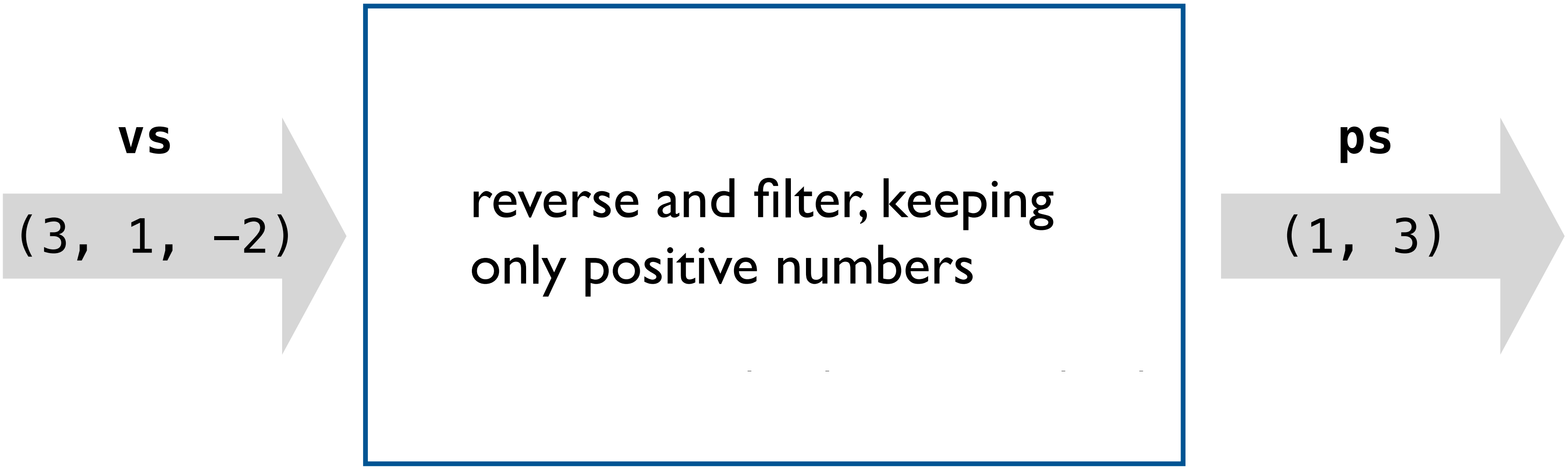
# How it all works:  a big picture view



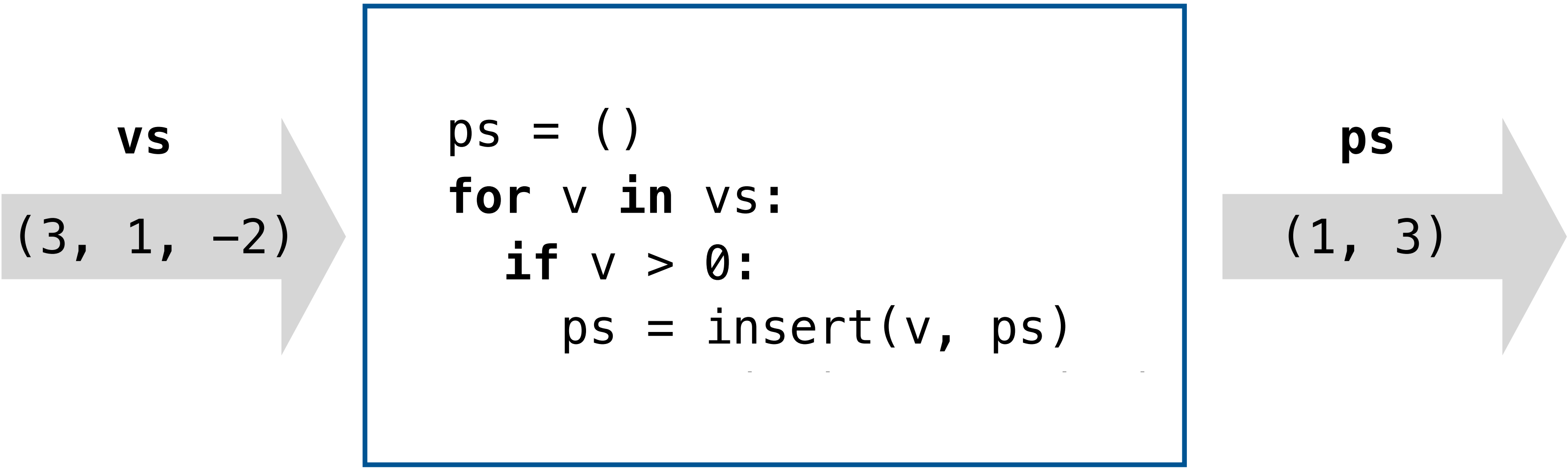
- pattern matching
- dynamic evaluation
- first-class procedures
- higher-order procedures
- side effects
- macros

theories of bitvectors, integers, reals, and uninterpreted functions

# Translation to constraints by example



# Translation to constraints by example



**vs**

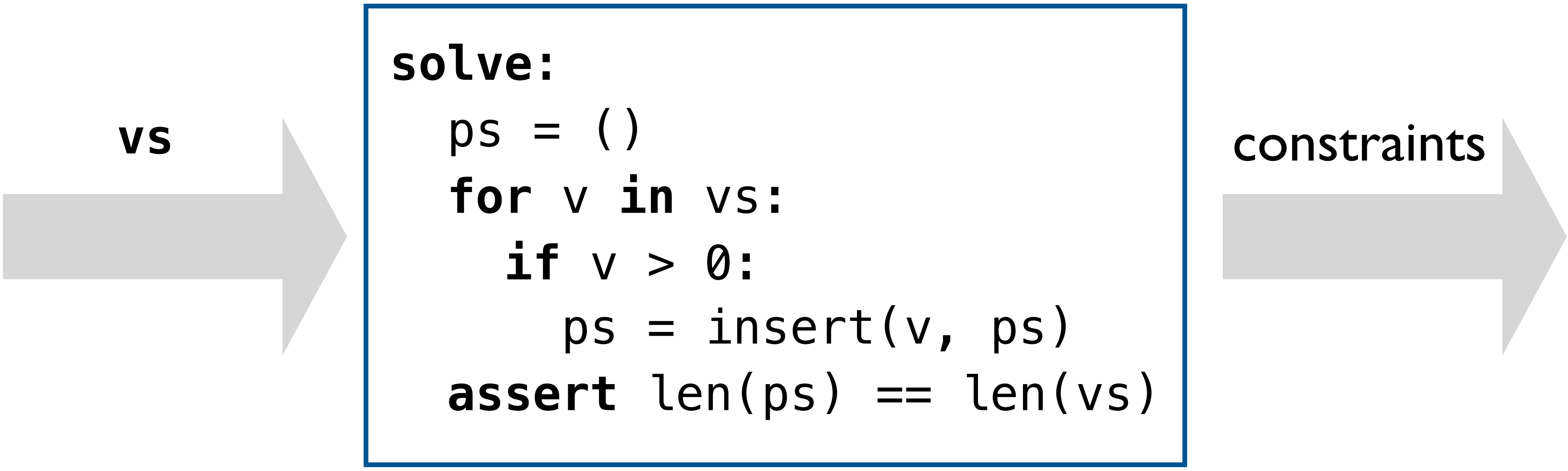
(3, 1, −2)

```
ps = ()
for v in vs:
  if v > 0:
    ps = insert(v, ps)
```

**ps**

(1, 3)

# Translation to constraints by example

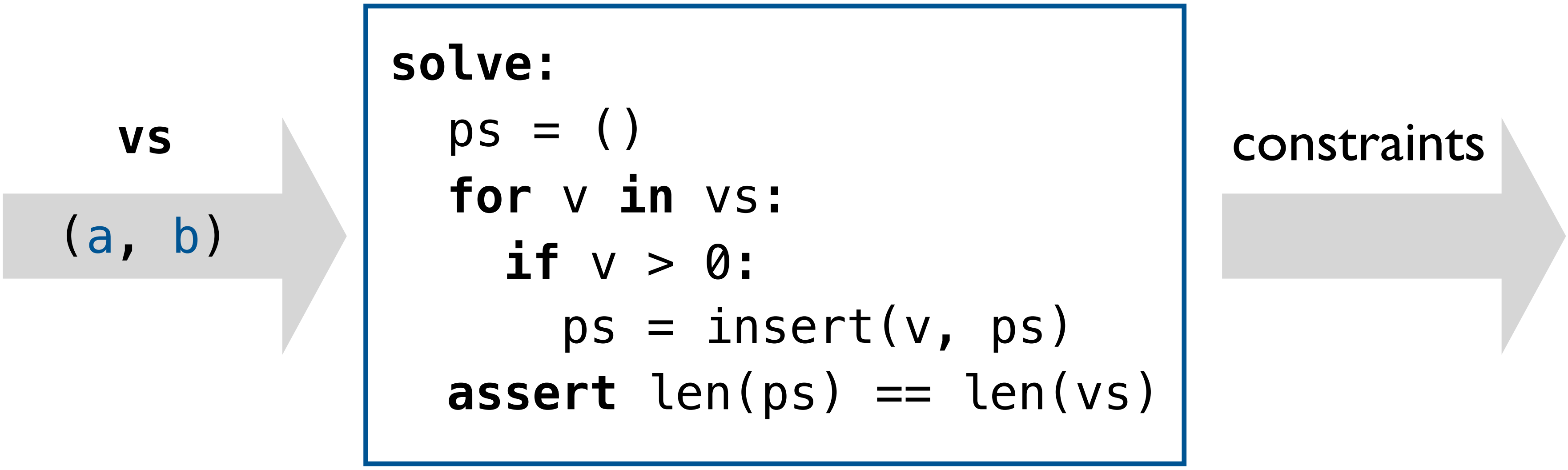


vs

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

constraints

# Translation to constraints by example



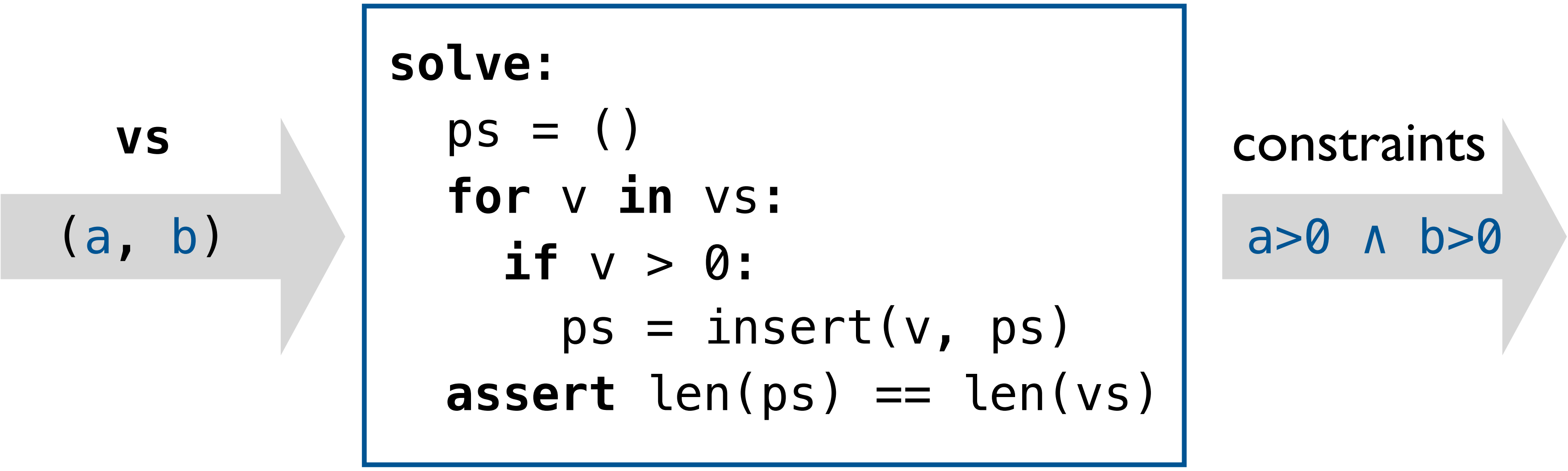
**vs**

(a, b)

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

constraints

# Translation to constraints by example



vs

(a, b)

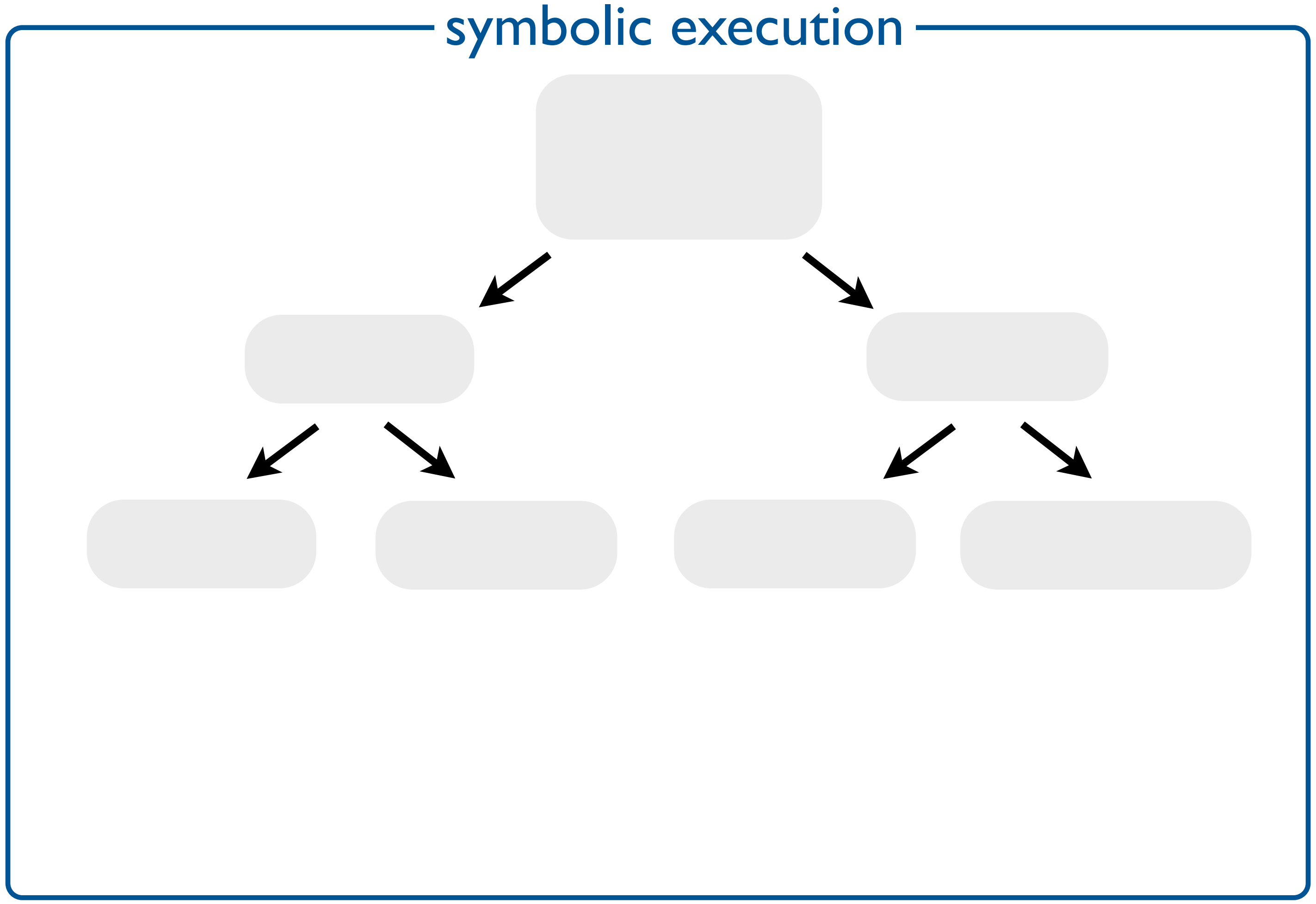
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

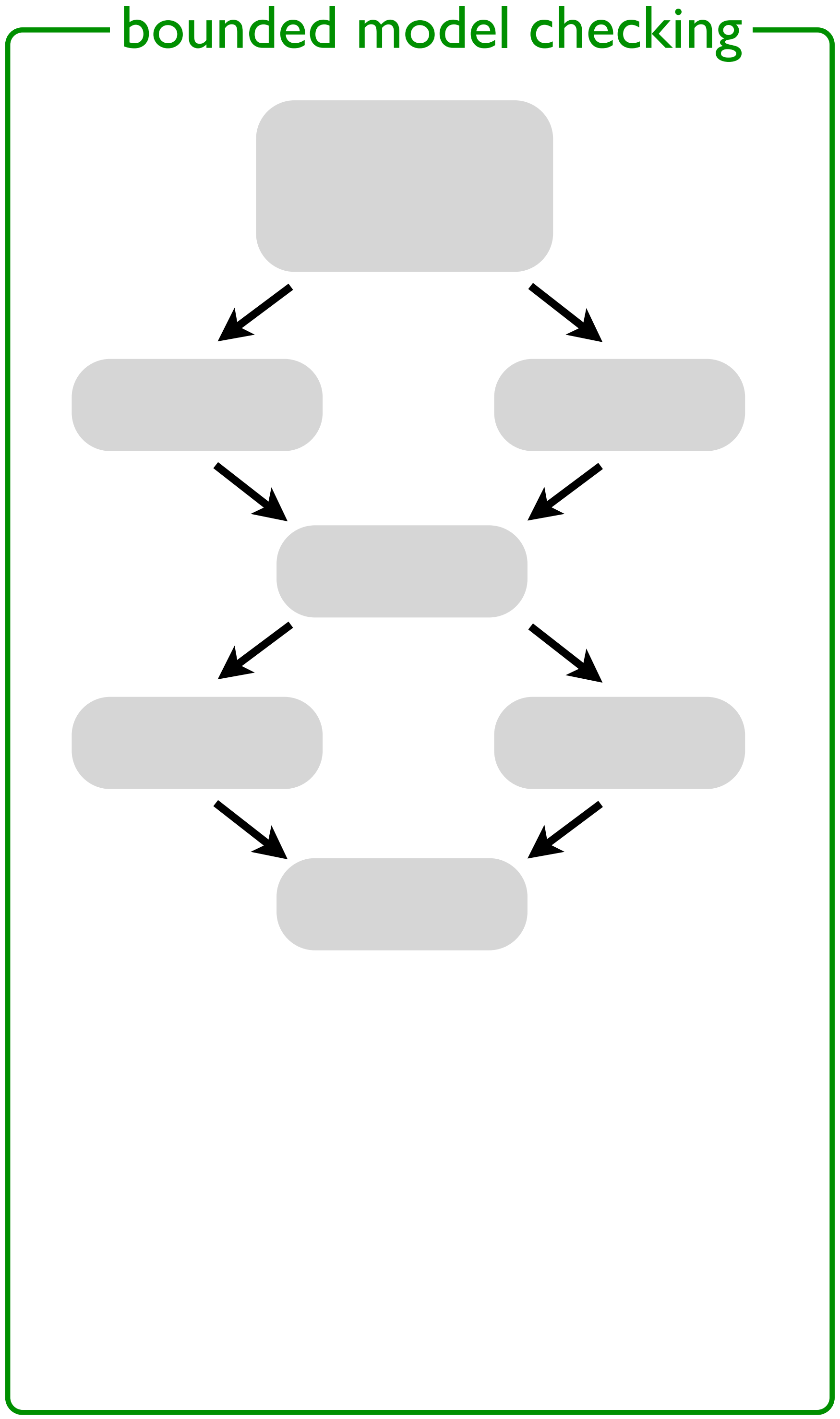
constraints

a>0 ∧ b>0

# Design space of precise symbolic encodings

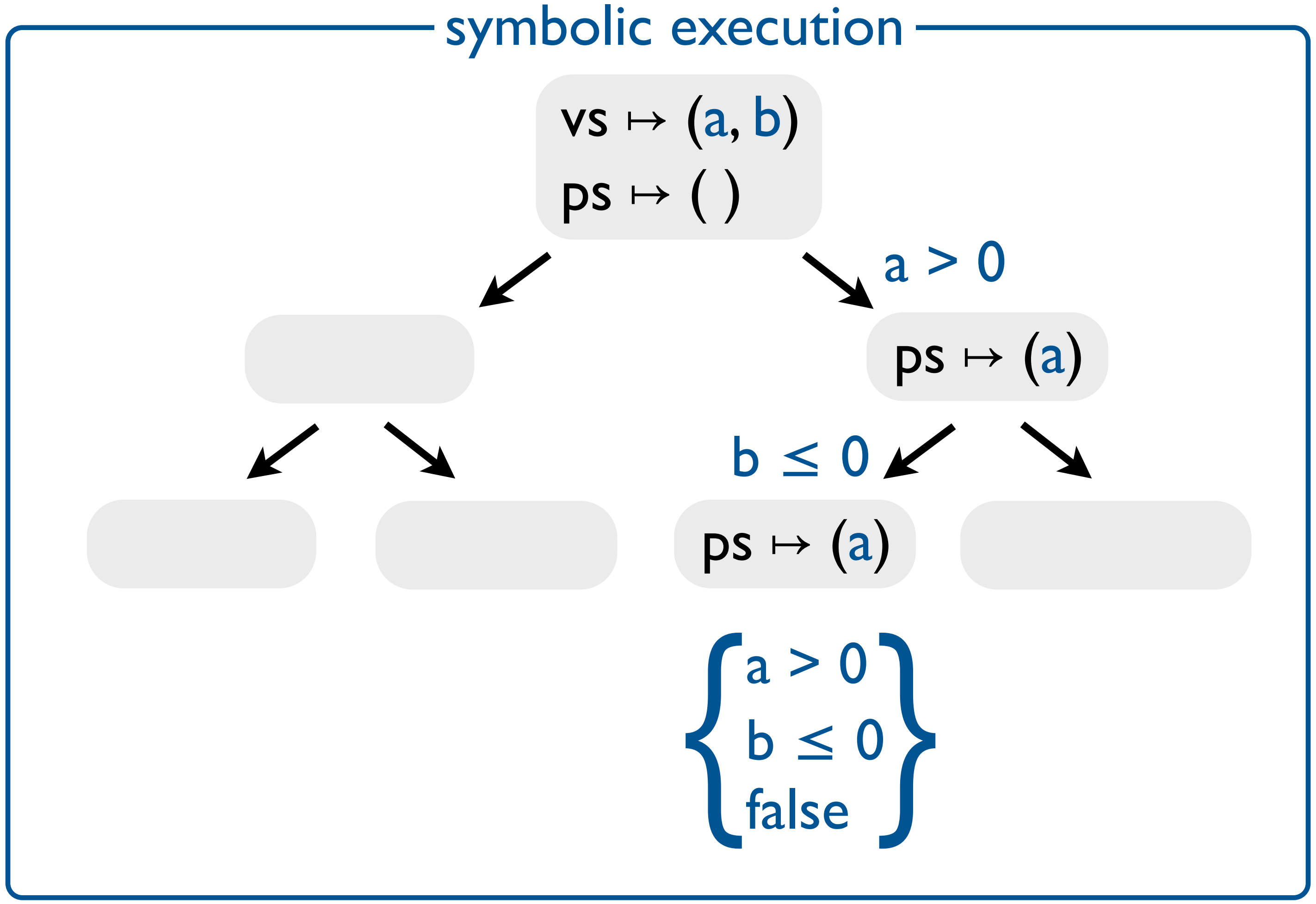
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

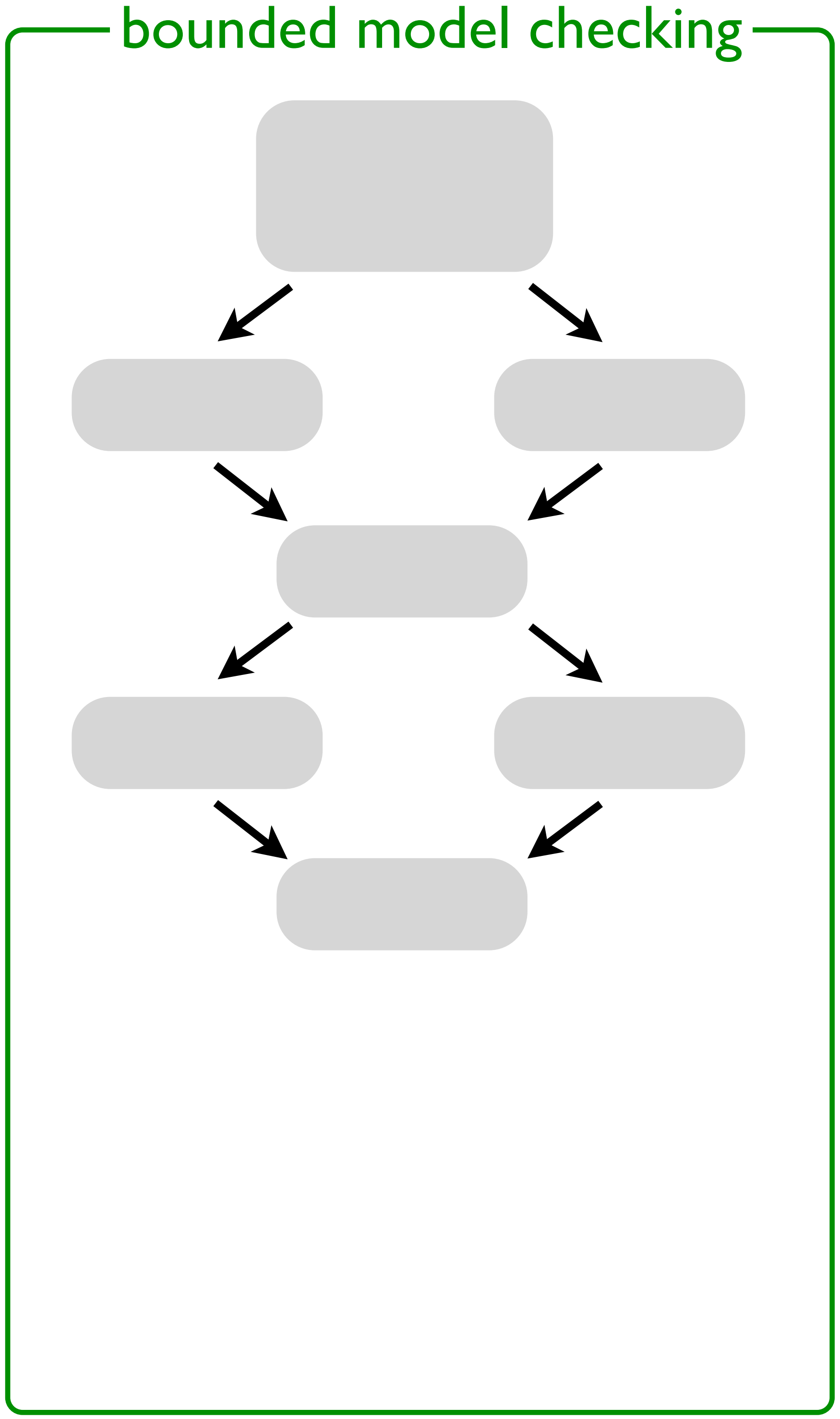




# Design space of precise symbolic encodings

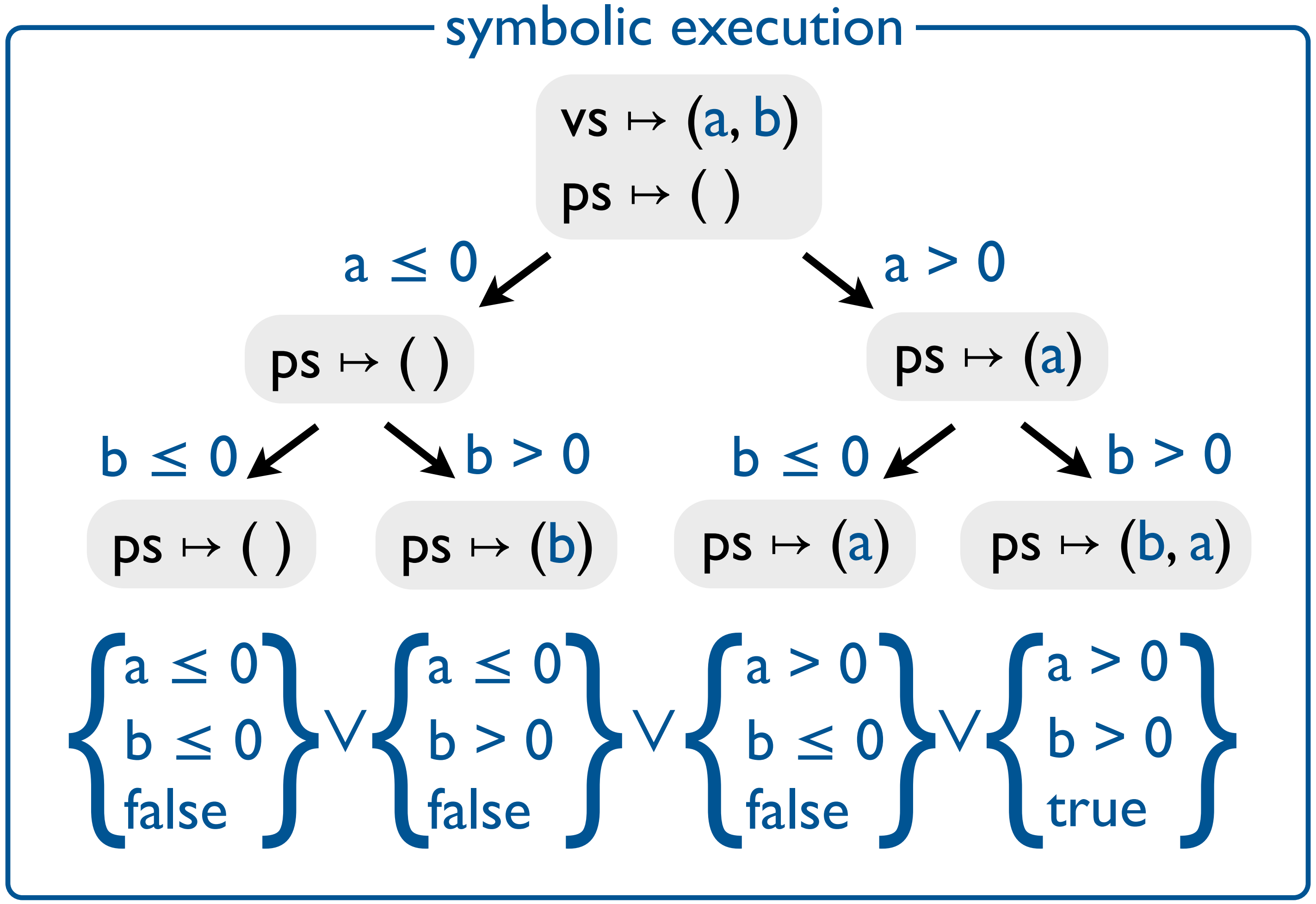
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```





# Design space of precise symbolic encodings

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```



symbolic execution

vs ↦ (a, b)
ps ↦ ( )

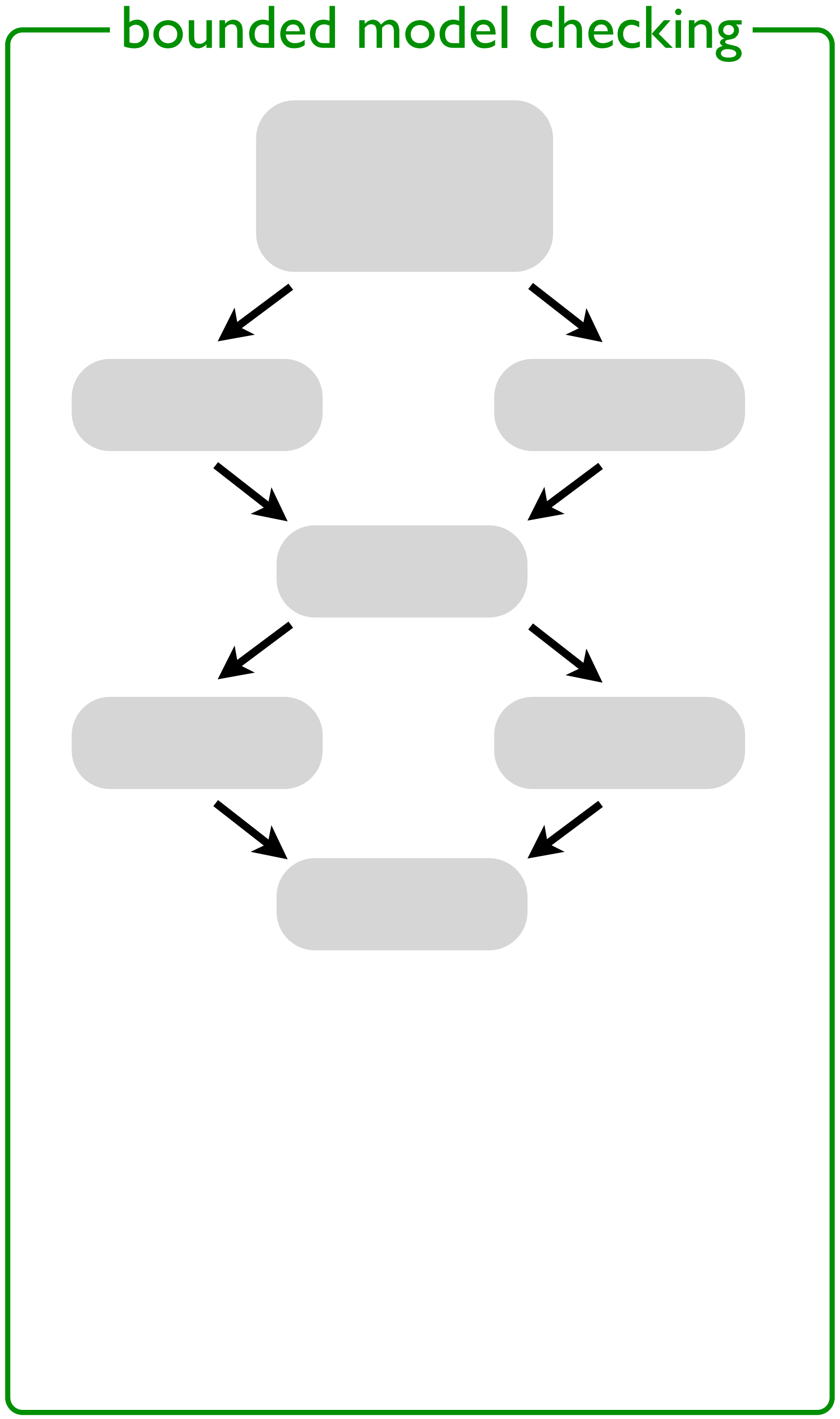
$a \leq 0$ | $a > 0$

ps ↦ ( ) | ps ↦ (a)

$b \leq 0$ | $b > 0$ | $b \leq 0$ | $b > 0$

ps ↦ ( ) | ps ↦ (b) | ps ↦ (a) | ps ↦ (b, a)

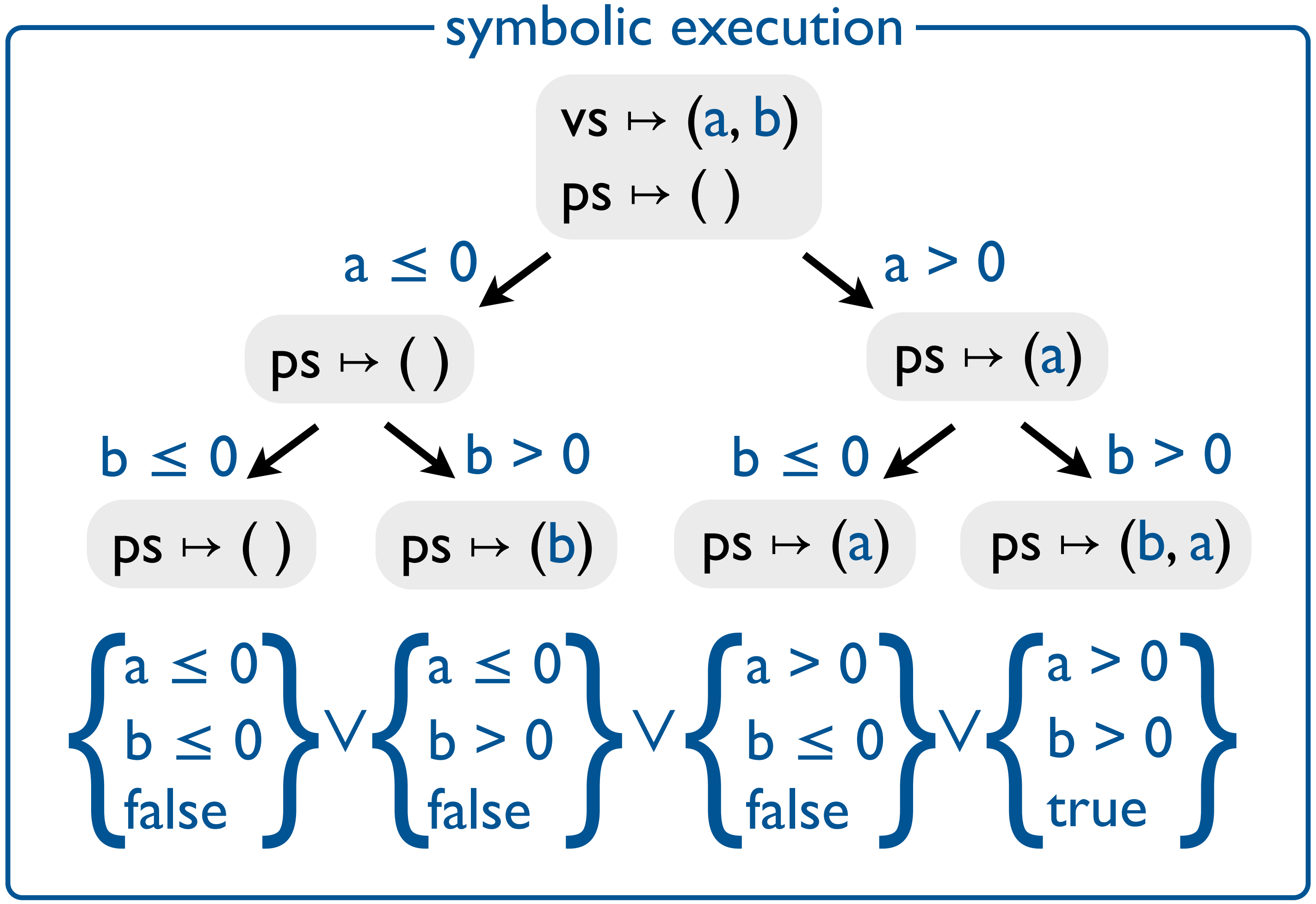
$$\left\{\begin{array}{l} a \leq 0 \\ b \leq 0 \\ \text{false} \end{array}\right\} \lor \left\{\begin{array}{l} a \leq 0 \\ b > 0 \\ \text{false} \end{array}\right\} \lor \left\{\begin{array}{l} a > 0 \\ b \leq 0 \\ \text{false} \end{array}\right\} \lor \left\{\begin{array}{l} a > 0 \\ b > 0 \\ \text{true} \end{array}\right\}$$



bounded model checking

# Design space of precise symbolic encodings

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```



symbolic execution

$vs \mapsto (a, b)$

$ps \mapsto (\,)$

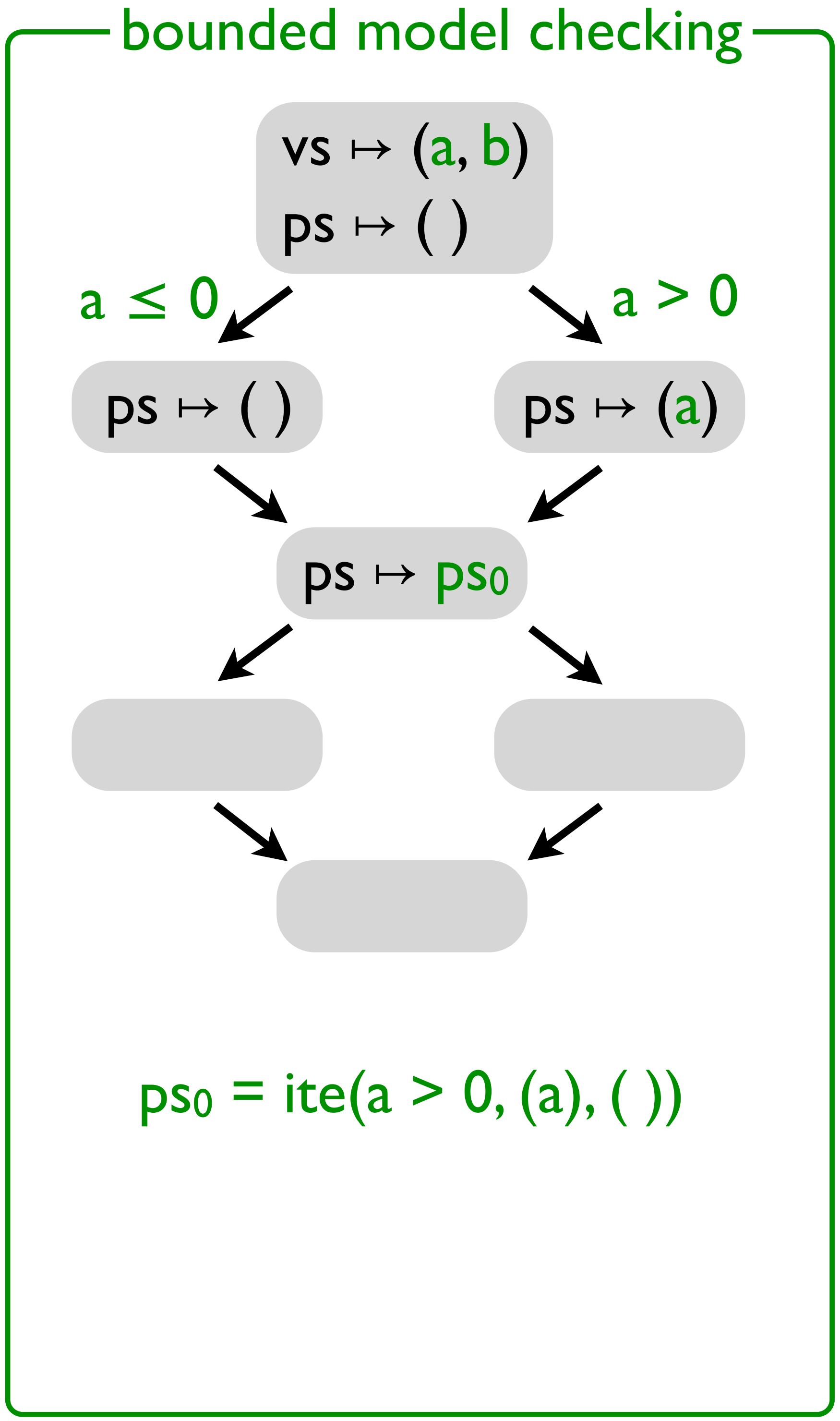
$a \le 0$ $a > 0$

$ps \mapsto (\,)$ $ps \mapsto (a)$

$b \le 0$ $b > 0$ $b \le 0$ $b > 0$

$ps \mapsto (\,)$ $ps \mapsto (b)$ $ps \mapsto (a)$ $ps \mapsto (b, a)$

$$\begin{Bmatrix} a \le 0 \\ b \le 0 \\ \text{false} \end{Bmatrix} \vee \begin{Bmatrix} a \le 0 \\ b > 0 \\ \text{false} \end{Bmatrix} \vee \begin{Bmatrix} a > 0 \\ b \le 0 \\ \text{false} \end{Bmatrix} \vee \begin{Bmatrix} a > 0 \\ b > 0 \\ \text{true} \end{Bmatrix}$$



bounded model checking

$vs \mapsto (a, b)$

$ps \mapsto (\,)$

$a \le 0$ $a > 0$

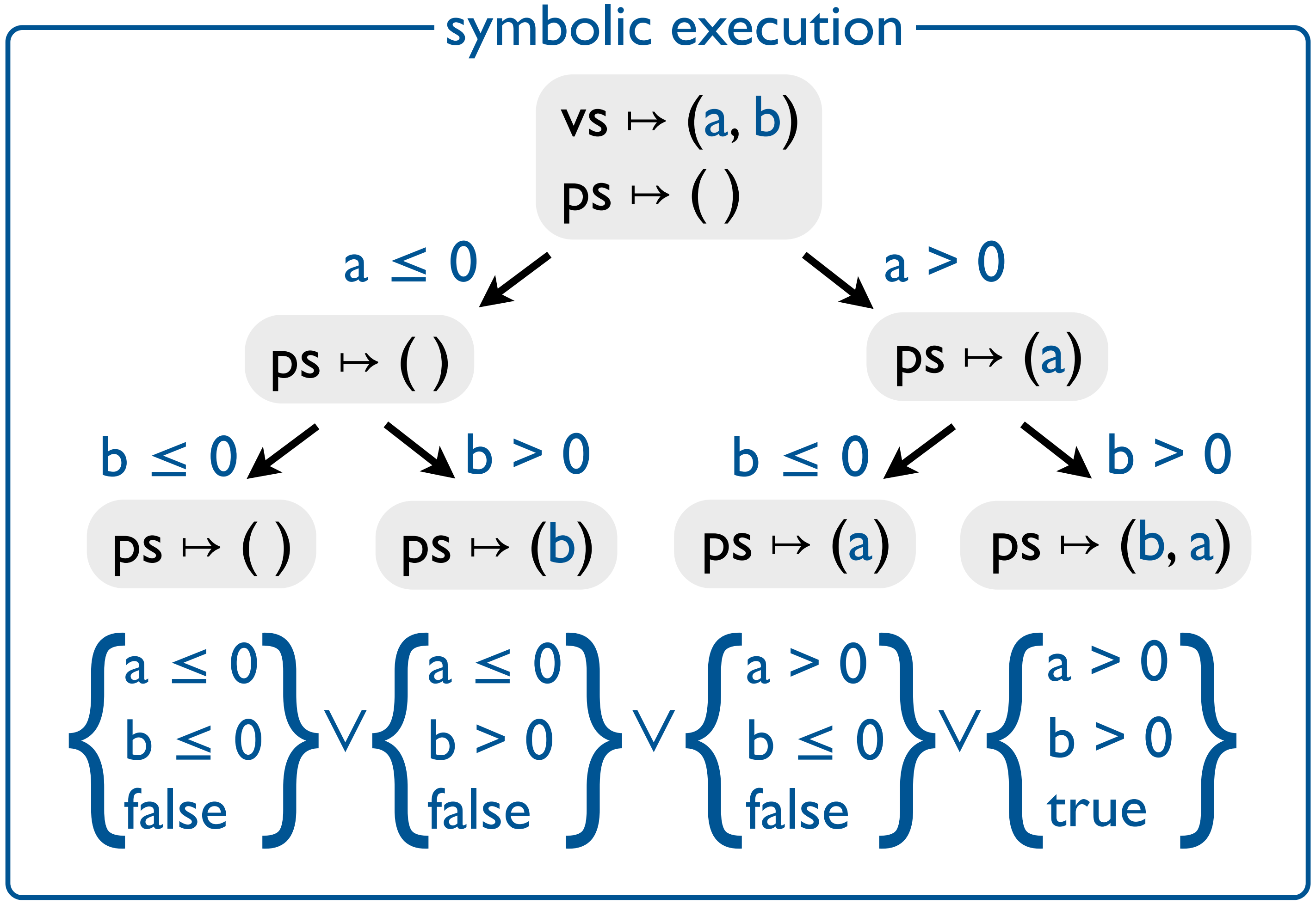
$ps \mapsto (\,)$ $ps \mapsto (a)$

$ps \mapsto ps_0$

$ps_0 = ite(a > 0, (a), (\,))$

# Design space of precise symbolic encodings

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```



symbolic execution

vs ↦ (a, b)
ps ↦ ( )

a ≤ 0 → ps ↦ ( )

a > 0 → ps ↦ (a)

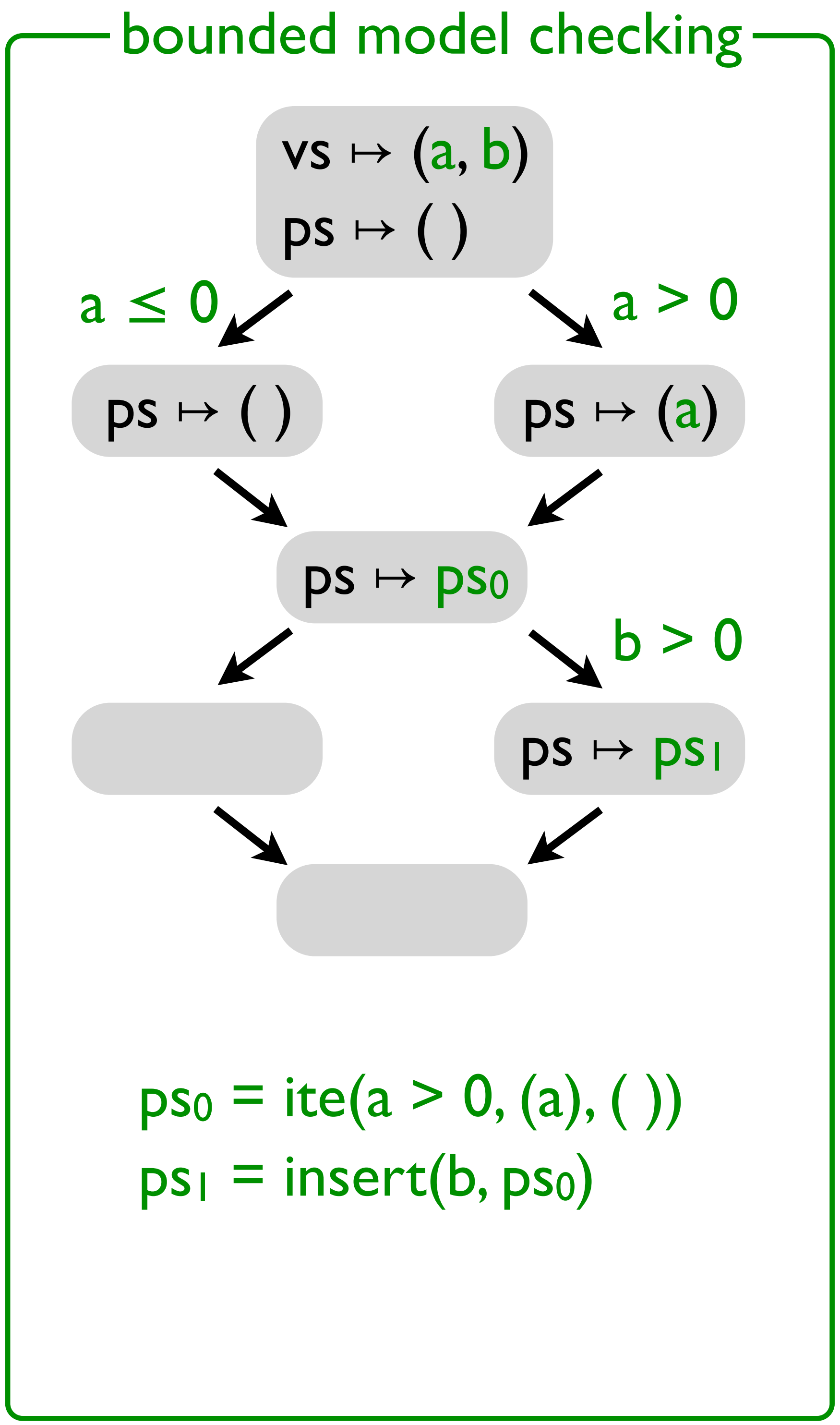
b ≤ 0 → ps ↦ ( )

b > 0 → ps ↦ (b)

b ≤ 0 → ps ↦ (a)

b > 0 → ps ↦ (b, a)

$$\left\{\begin{matrix} a \le 0 \\ b \le 0 \\ \text{false} \end{matrix}\right\} \vee \left\{\begin{matrix} a \le 0 \\ b > 0 \\ \text{false} \end{matrix}\right\} \vee \left\{\begin{matrix} a > 0 \\ b \le 0 \\ \text{false} \end{matrix}\right\} \vee \left\{\begin{matrix} a > 0 \\ b > 0 \\ \text{true} \end{matrix}\right\}$$



bounded model checking

vs ↦ (a, b)
ps ↦ ( )

a ≤ 0 → ps ↦ ( )

a > 0 → ps ↦ (a)

ps ↦ $ps_0$

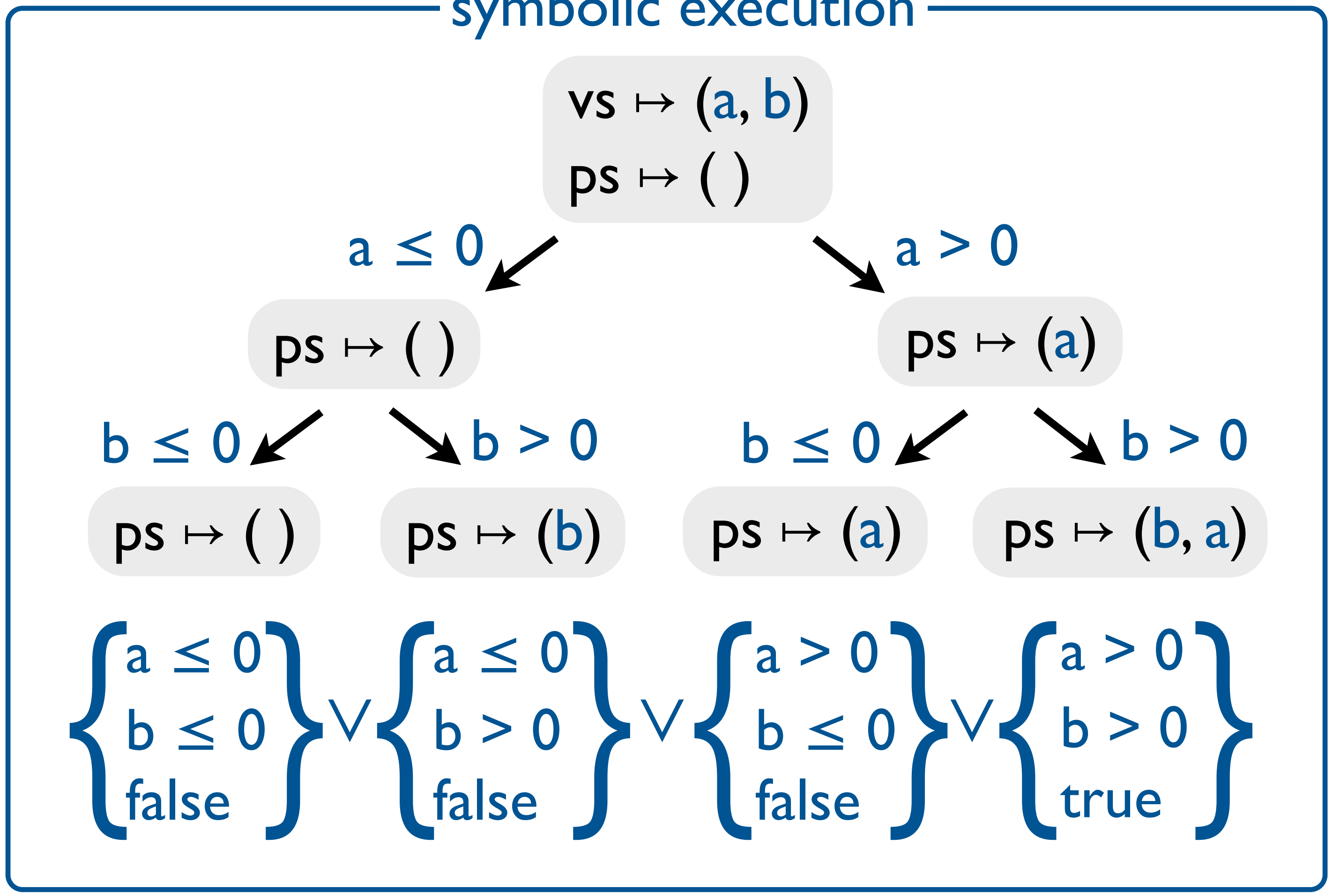
b > 0 → ps ↦ $ps_1$

$ps_0$ = ite(a > 0, (a), ( ))

$ps_1$ = insert(b, $ps_0$)

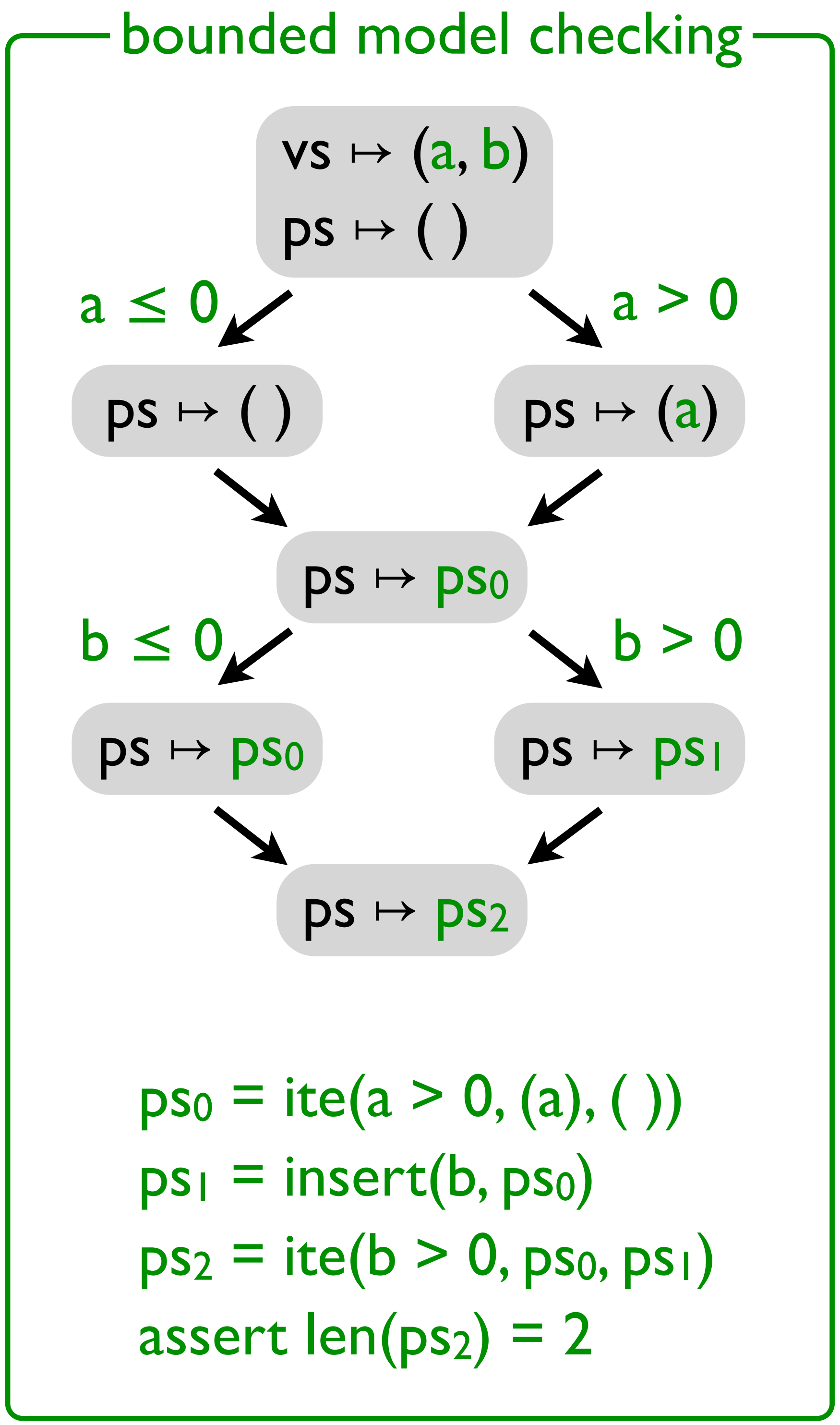
# Design space of precise symbolic encodings

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```



symbolic execution

$$\left\{\begin{array}{l} a \leq 0 \\ b \leq 0 \\ \text{false} \end{array}\right\} \lor \left\{\begin{array}{l} a \leq 0 \\ b > 0 \\ \text{false} \end{array}\right\} \lor \left\{\begin{array}{l} a > 0 \\ b \leq 0 \\ \text{false} \end{array}\right\} \lor \left\{\begin{array}{l} a > 0 \\ b > 0 \\ \text{true} \end{array}\right\}$$



bounded model checking

$ps_0$ = ite(a > 0, (a), ( ))

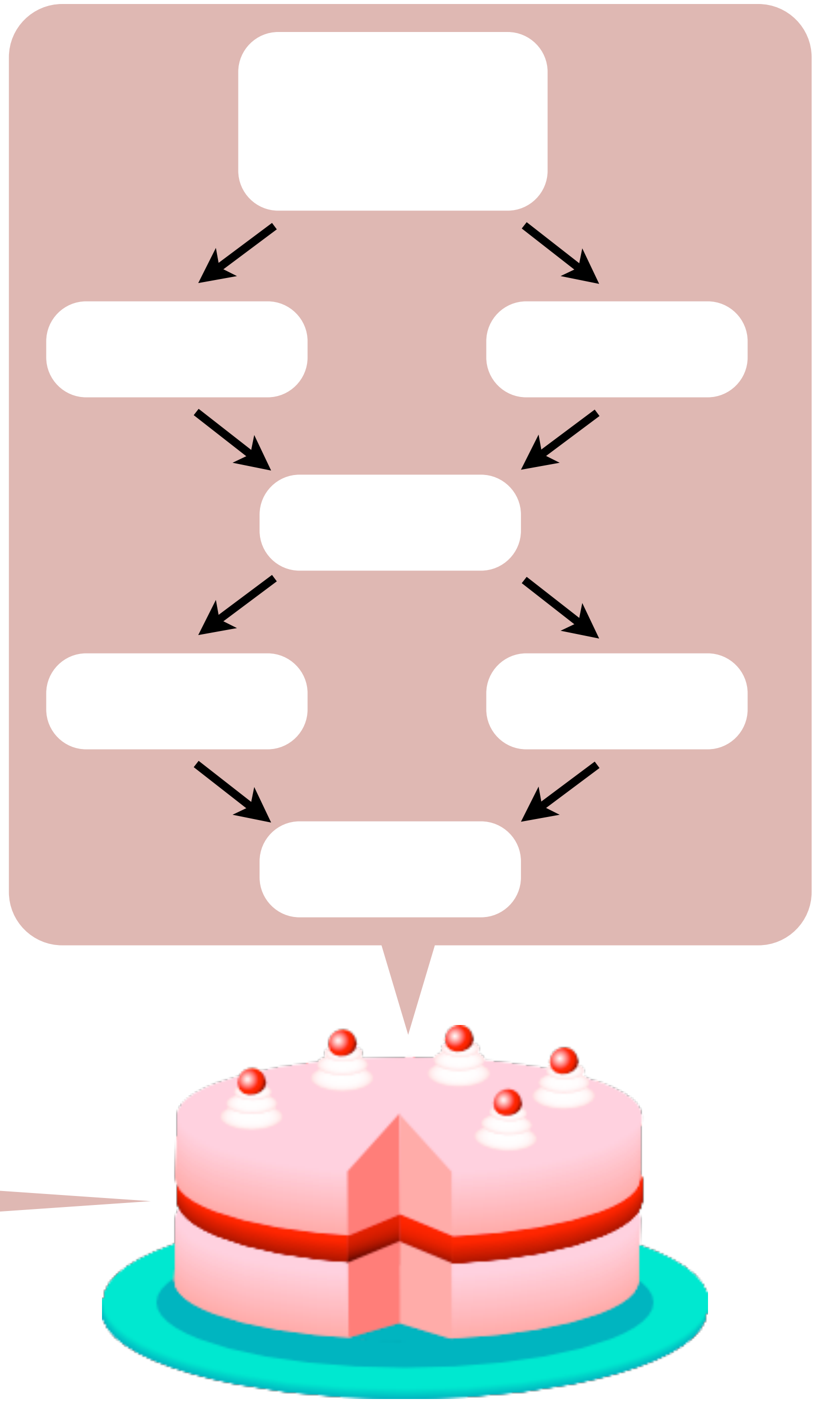
$ps_1$ = insert(b, $ps_0$)

$ps_2$ = ite(b > 0, $ps_0$, $ps_1$)

assert len($ps_2$) = 2

# A new design: type-driven state merging

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```



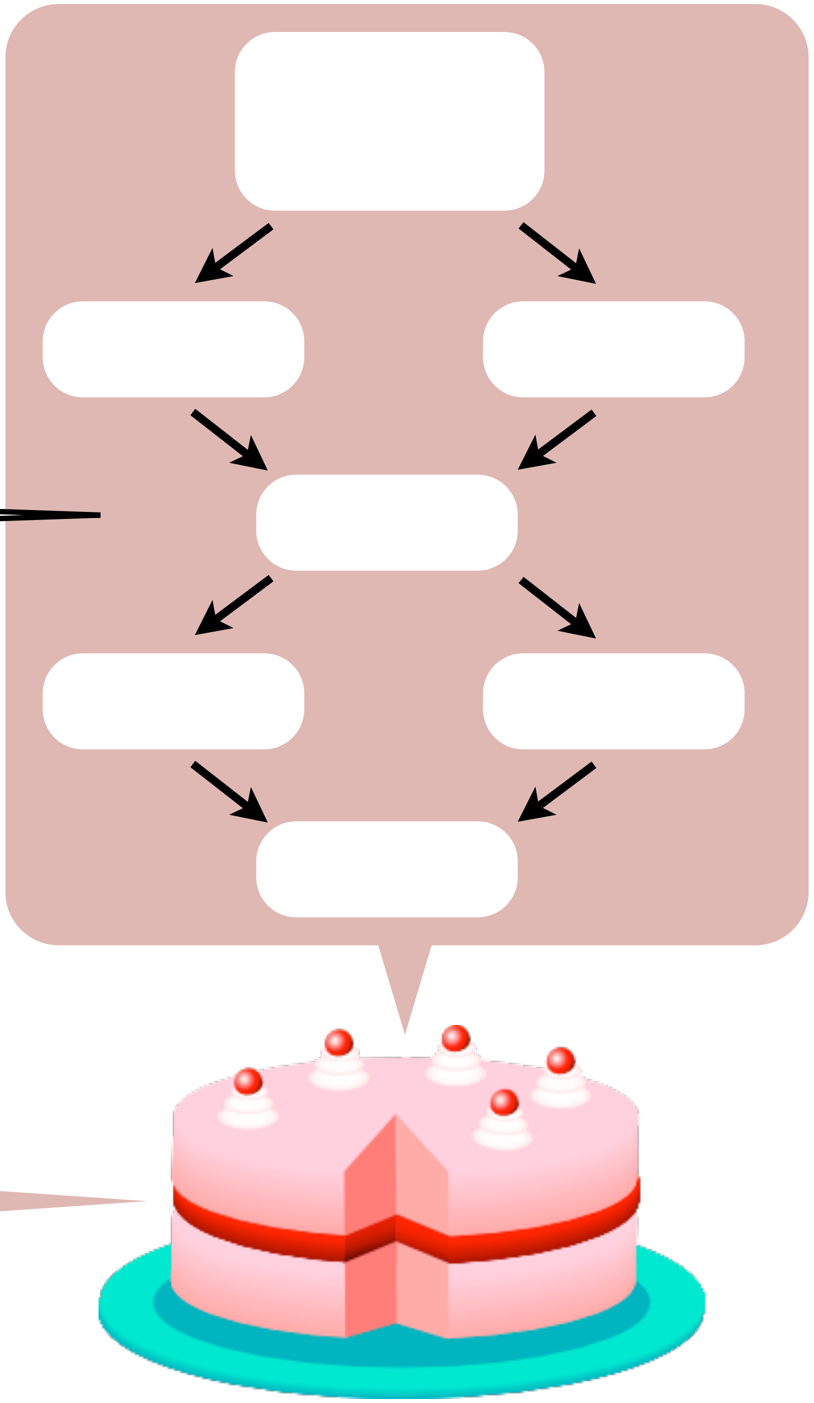
$$\left\{\begin{array}{l} a > 0 \\ b > 0 \\ \text{true} \end{array}\right\}$$

# A new design:  type-driven state merging

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

**Merge values of**

- primitive types:    symbolically
- immutable types:  structurally
- all other types:     via unions



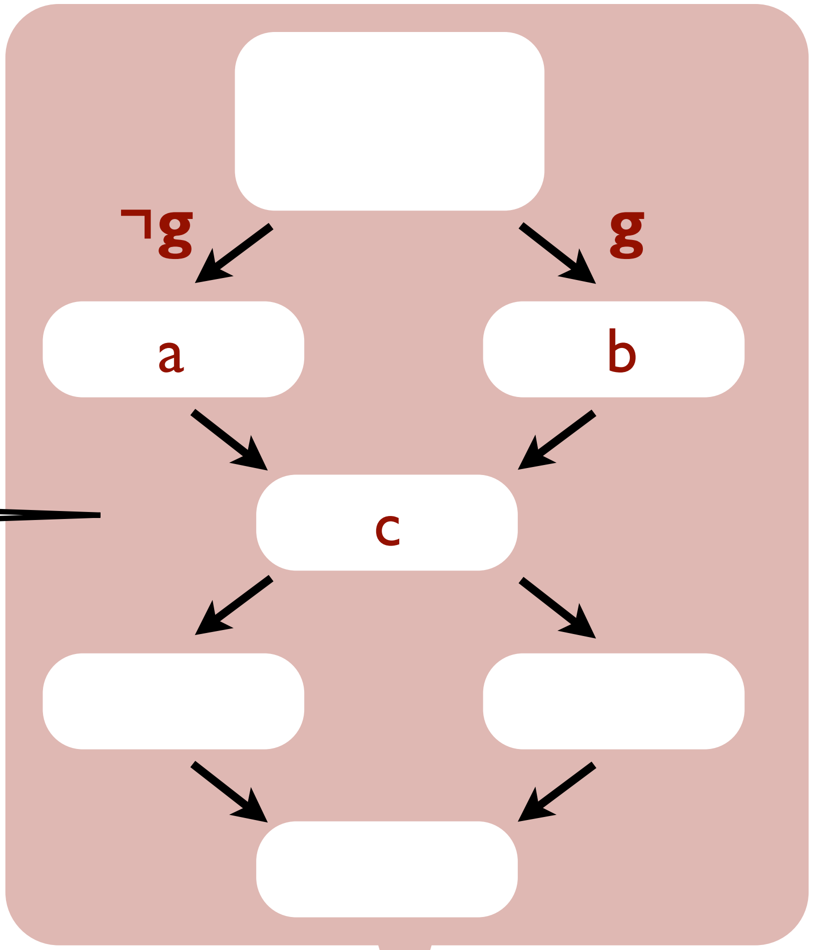
$$\left\{ \begin{array}{l} a > 0 \\ b > 0 \\ \text{true} \end{array} \right\}$$

# A new design:  type-driven state merging

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

**Merge values of**

- primitive types: symbolically
- immutable types: structurally
- all other types: via unions



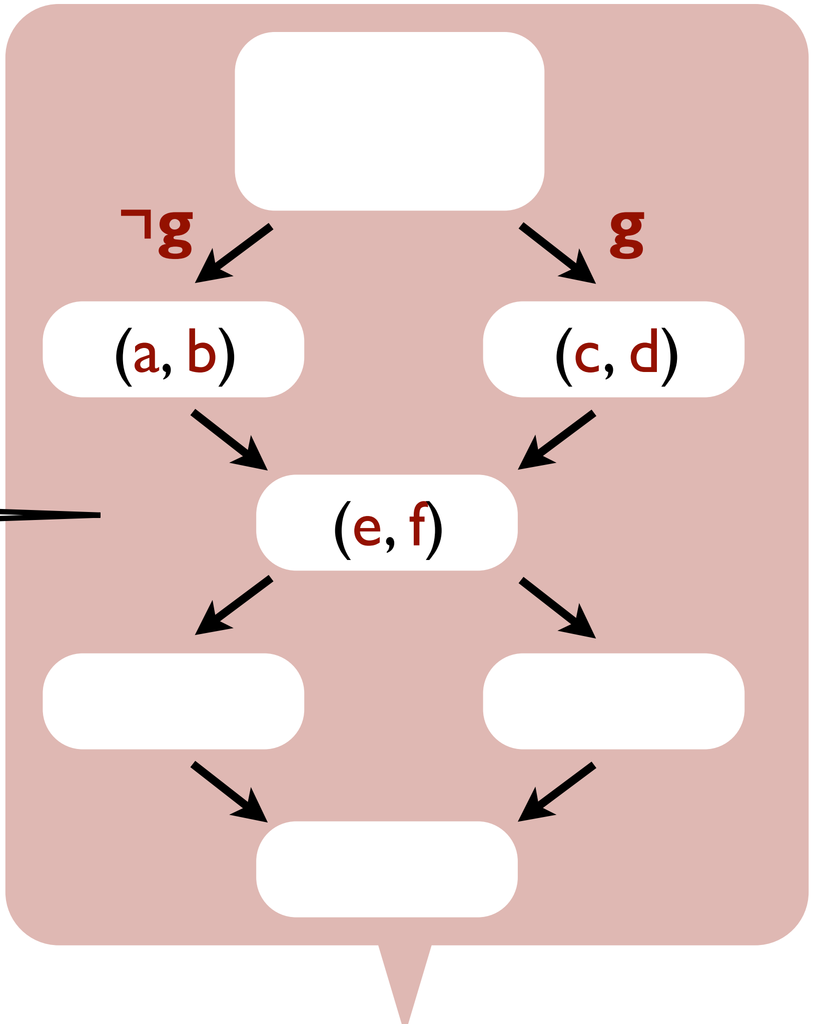
$$\left\{\begin{matrix} a > 0 \\ b > 0 \\ \text{true} \end{matrix}\right\}$$

# A new design: type-driven state merging

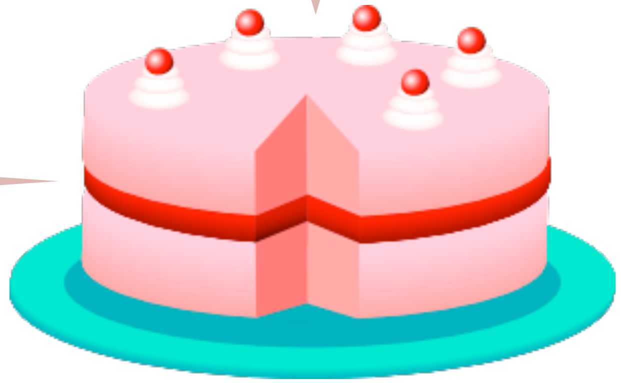
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

**Merge values of**

- primitive types: symbolically
- immutable types: structurally
- all other types: via unions



$$\left\{\begin{array}{l} a > 0 \\ b > 0 \\ \text{true} \end{array}\right\}$$

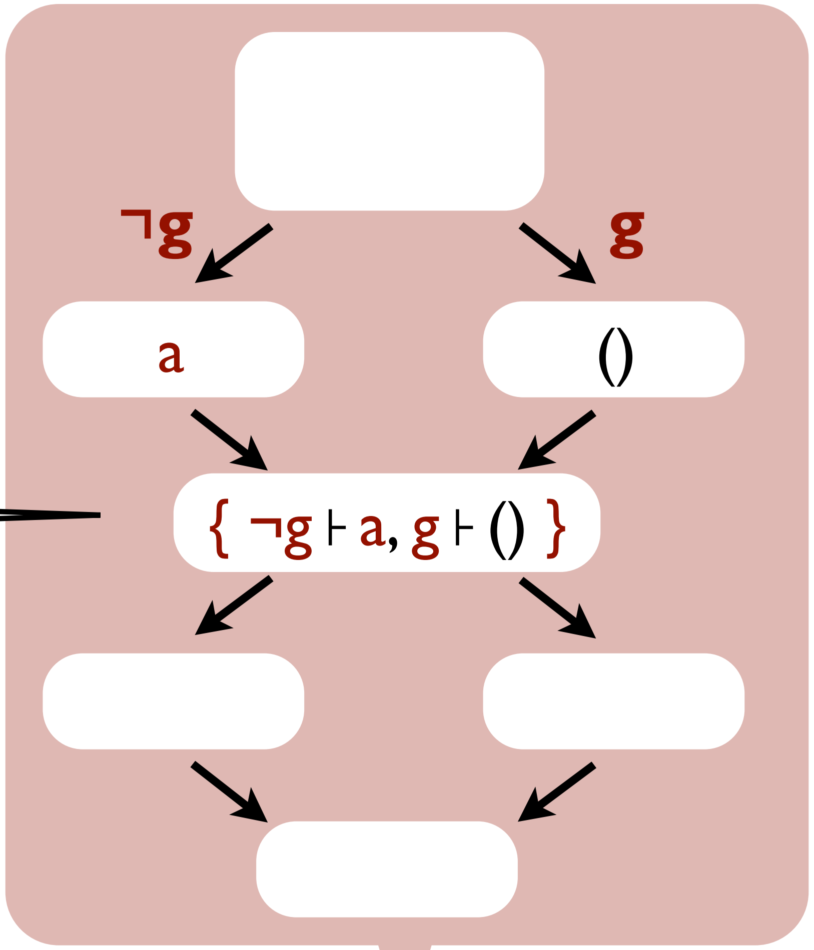


# A new design: type-driven state merging

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

**Merge values of**

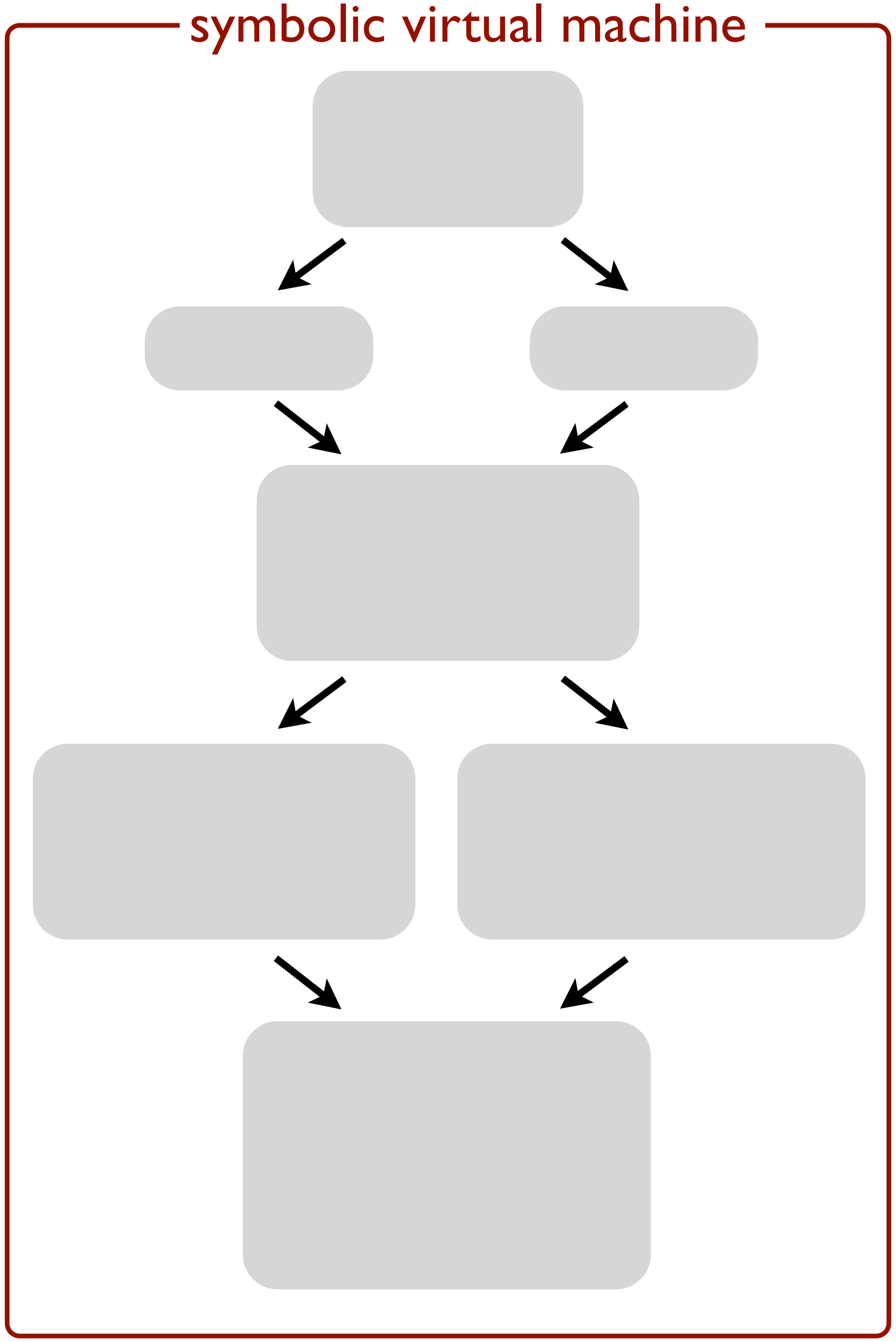
- primitive types: symbolically
- immutable types: structurally
- all other types: via unions



$$\left\{\begin{array}{l} a > 0 \\ b > 0 \\ \text{true} \end{array}\right\}$$

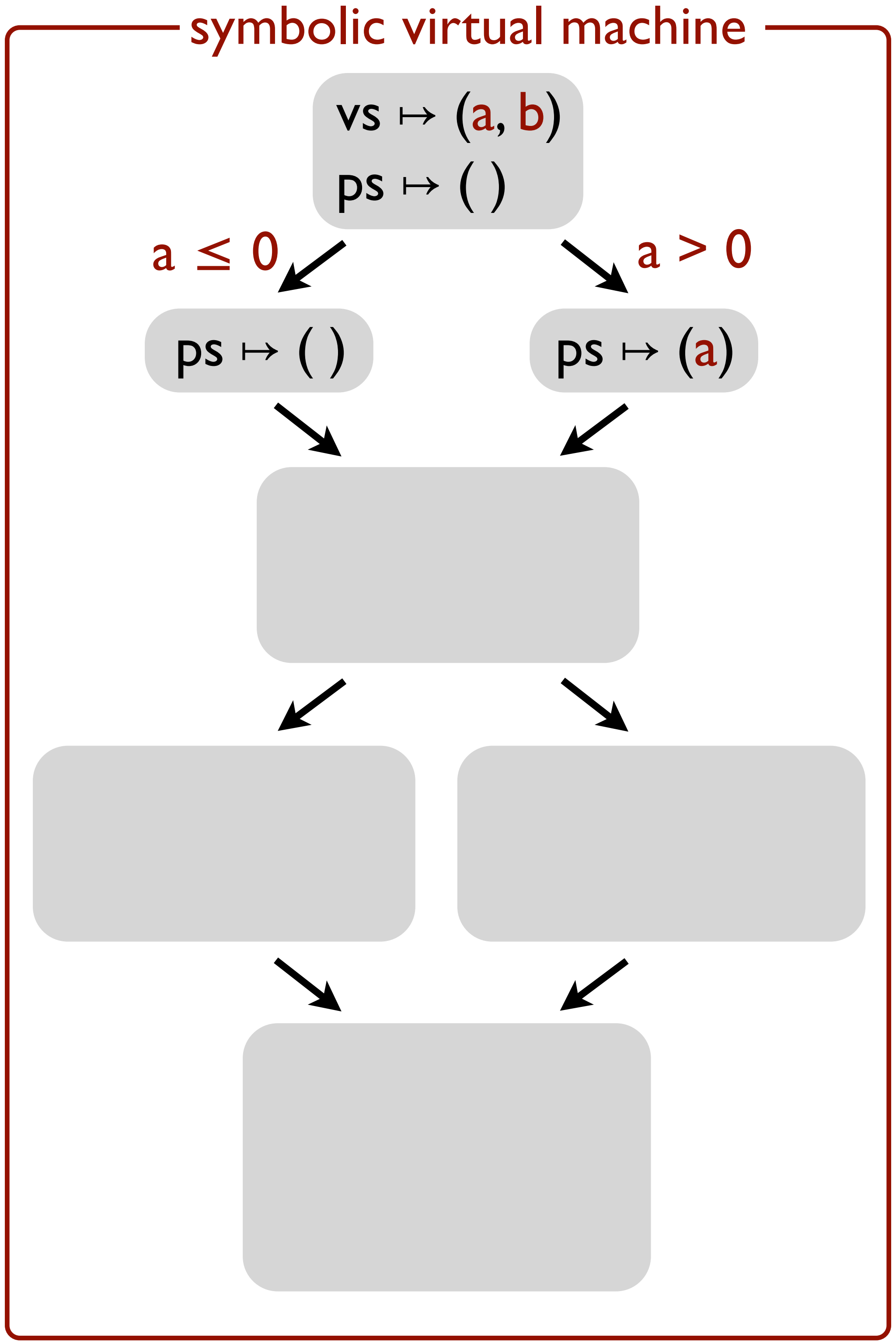
# A new design:  type-driven state merging

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```



# A new design:  type-driven state merging

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

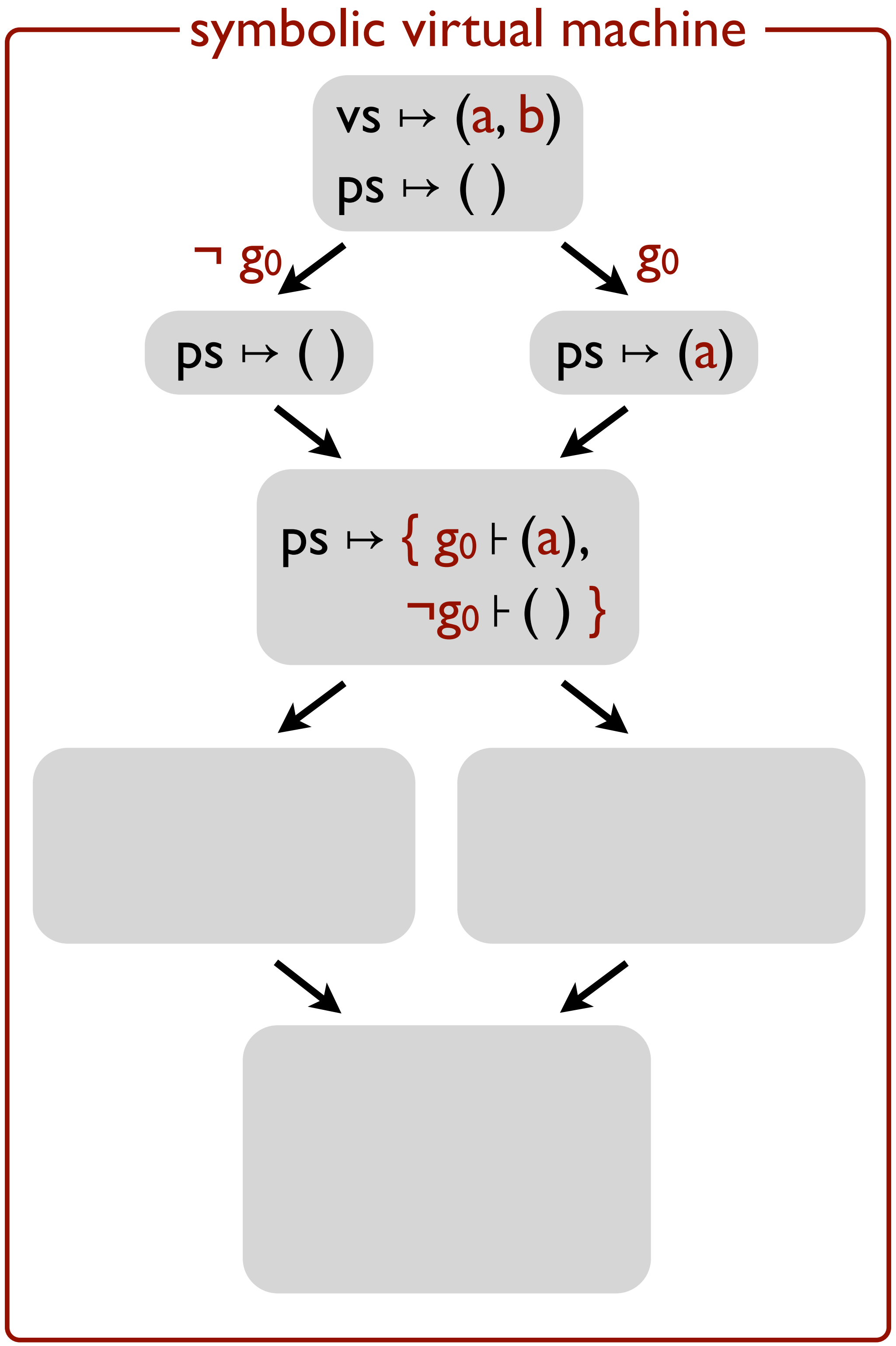


# A new design: type-driven state merging

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

Symbolic union: a set of guarded values, with disjoint guards.

$g_0 = a > 0$



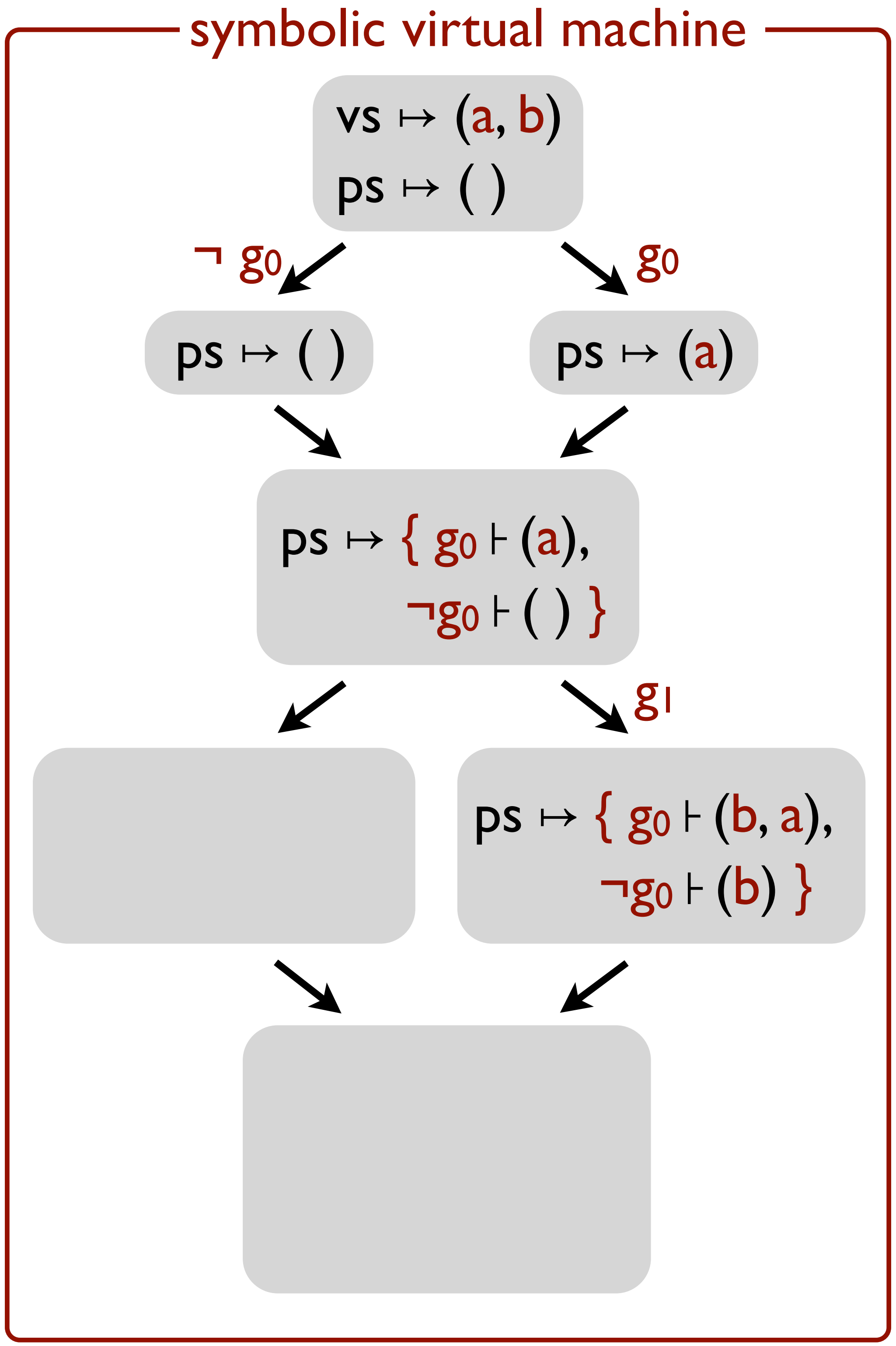
# A new design: type-driven state merging

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

Execute `insert` concretely on all lists in the union.

$g_0 = a > 0$

$g_1 = b > 0$



symbolic virtual machine

$vs \mapsto (a, b)$
$ps \mapsto ()$

$\neg g_0$

$g_0$

$ps \mapsto ()$

$ps \mapsto (a)$

$ps \mapsto \{ g_0 \vdash (a), \neg g_0 \vdash () \}$

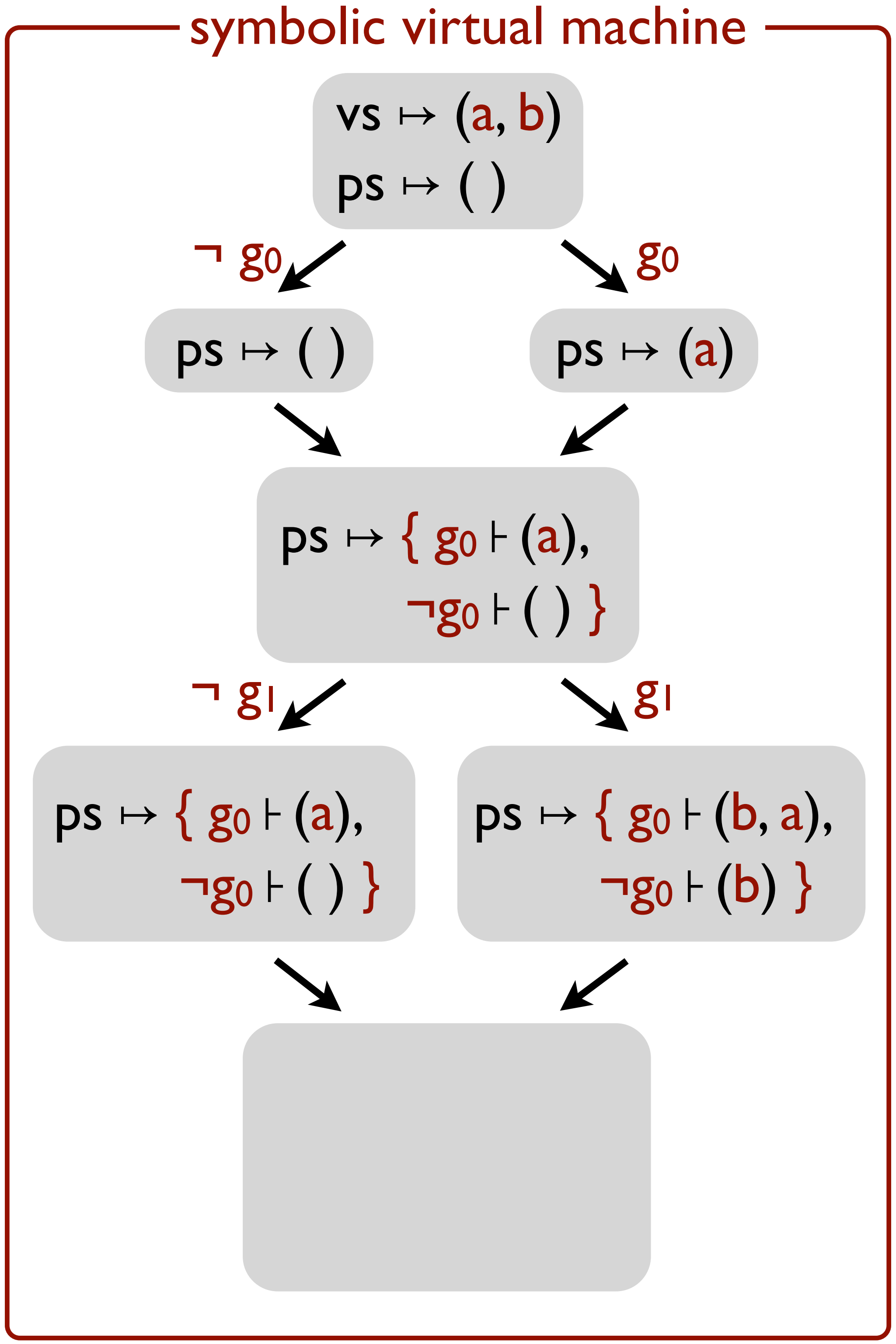
$g_1$

$ps \mapsto \{ g_0 \vdash (b, a), \neg g_0 \vdash (b) \}$

# A new design: type-driven state merging

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

$g_0 = a > 0$
$g_1 = b > 0$



symbolic virtual machine

$vs \mapsto (a, b)$
$ps \mapsto (\,)$

$\neg g_0$

$g_0$

$ps \mapsto (\,)$

$ps \mapsto (a)$

$ps \mapsto \{ g_0 \vdash (a), \neg g_0 \vdash (\,) \}$

$\neg g_1$

$g_1$

$ps \mapsto \{ g_0 \vdash (a), \neg g_0 \vdash (\,) \}$

$ps \mapsto \{ g_0 \vdash (b, a), \neg g_0 \vdash (b) \}$

# A new design: type-driven state merging

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

Evaluate `len` concretely on all lists in the union; assertion true only on the list guarded by $g_2$.

$g_0 = a > 0$

$g_1 = b > 0$

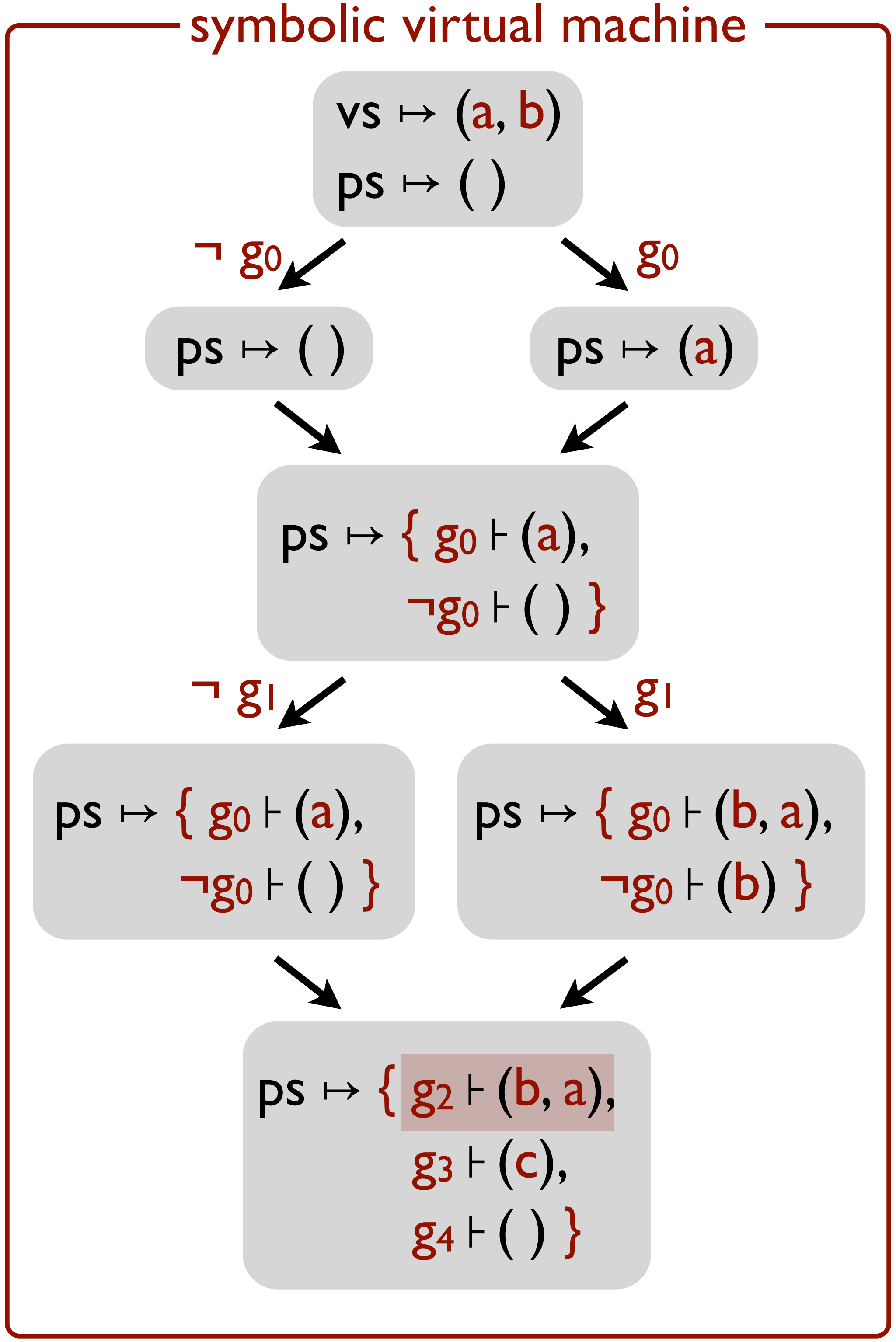
$g_2 = g_0 \wedge g_1$

$g_3 = \neg(g_0 \Leftrightarrow g_1)$

$g_4 = \neg g_0 \wedge \neg g_1$

$c = \mathrm{ite}(g_1, b, a)$

**assert $g_2$**



symbolic virtual machine

$vs \mapsto (a, b)$
$ps \mapsto (\,)$

$\neg\, g_0$ ; $g_0$

$ps \mapsto (\,)$ ; $ps \mapsto (a)$

$ps \mapsto \{\, g_0 \vdash (a),\ \neg g_0 \vdash (\,) \,\}$

$\neg\, g_1$ ; $g_1$

$ps \mapsto \{\, g_0 \vdash (a),\ \neg g_0 \vdash (\,) \,\}$ ; $ps \mapsto \{\, g_0 \vdash (b, a),\ \neg g_0 \vdash (b) \,\}$

$ps \mapsto \{\, g_2 \vdash (b, a),\ g_3 \vdash (c),\ g_4 \vdash (\,) \,\}$

# A new design: type-driven state merging

```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

**polynomial encoding**

**concrete evaluation**

$g_0 = a > 0$

$g_1 = b > 0$

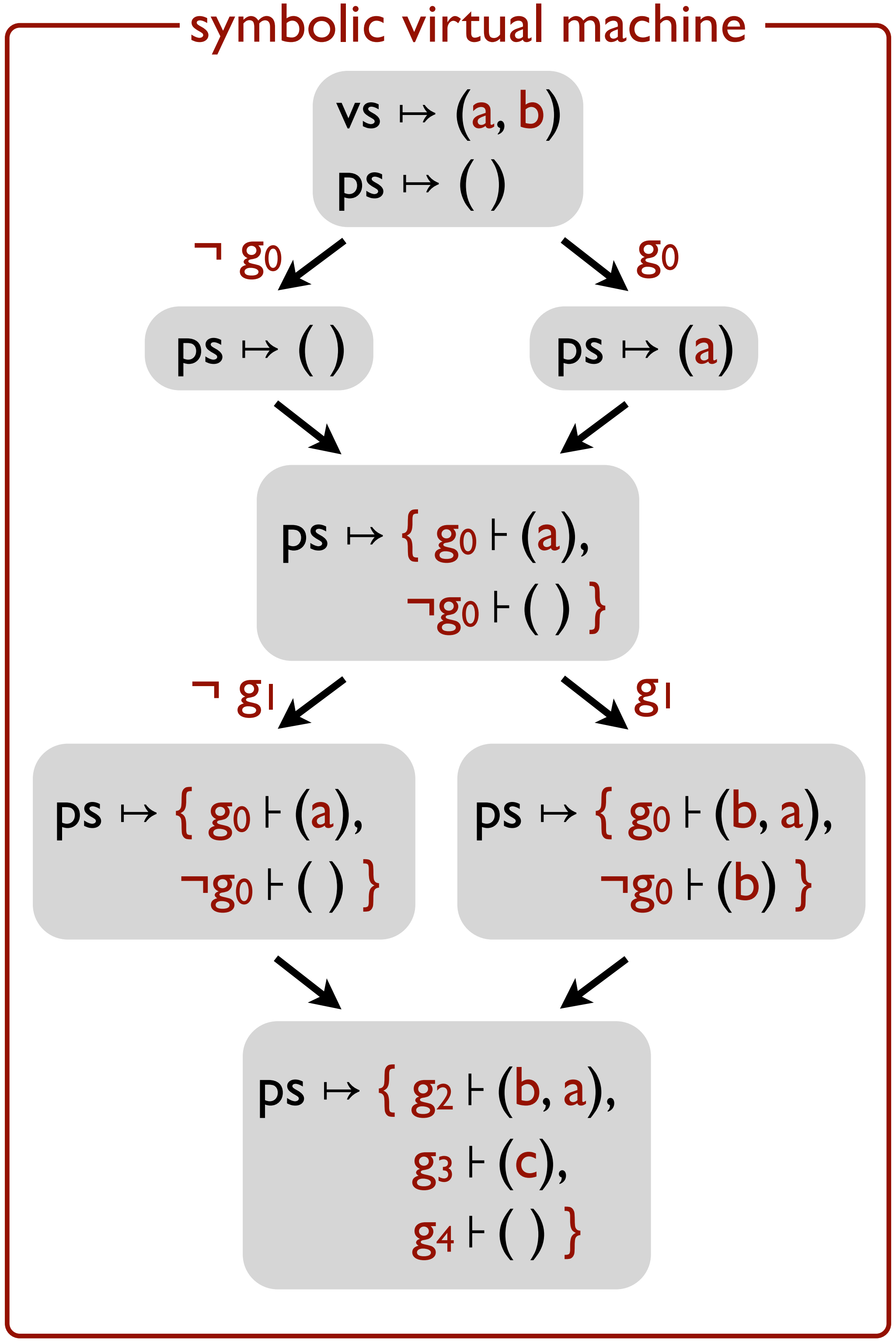
$g_2 = g_0 \wedge g_1$

$g_3 = \neg(g_0 \Leftrightarrow g_1)$

$g_4 = \neg g_0 \wedge \neg g_1$

$c = \mathrm{ite}(g_1, b, a)$

**assert $g_2$**



symbolic virtual machine

$vs \mapsto (a, b)$

$ps \mapsto (\,)$

$\neg\, g_0$

$g_0$

$ps \mapsto (\,)$

$ps \mapsto (a)$

$ps \mapsto \{\, g_0 \vdash (a), \neg g_0 \vdash (\,) \,\}$

$\neg\, g_1$

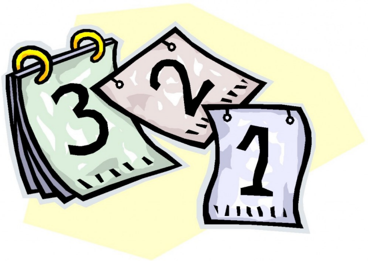
$g_1$

$ps \mapsto \{\, g_0 \vdash (a), \neg g_0 \vdash (\,) \,\}$

$ps \mapsto \{\, g_0 \vdash (b, a), \neg g_0 \vdash (b) \,\}$

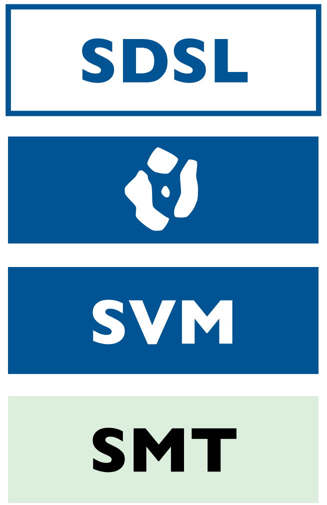
$ps \mapsto \{\, g_2 \vdash (b, a), g_3 \vdash (c), g_4 \vdash (\,) \,\}$

# How to build your own solver-aided tool



### The classic (hard) way to build a tool

What is hard about building a solver-aided tool?



SDSL

SVM

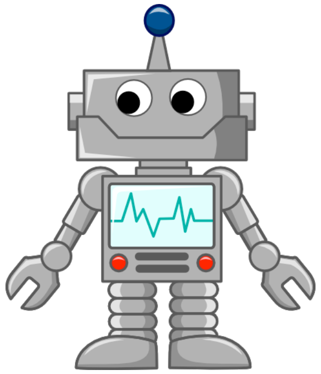
SMT

### An easier way: tools as languages

How to build tools by stacking layers of languages.

### Behind the scenes: symbolic virtual machine

How Rosette works so you don’t have to.



### A last look: a few recent applications

Cool tools built with Rosette!

# Chlorophyll:  ultra low-power computing

## Instructions/Second vs Power

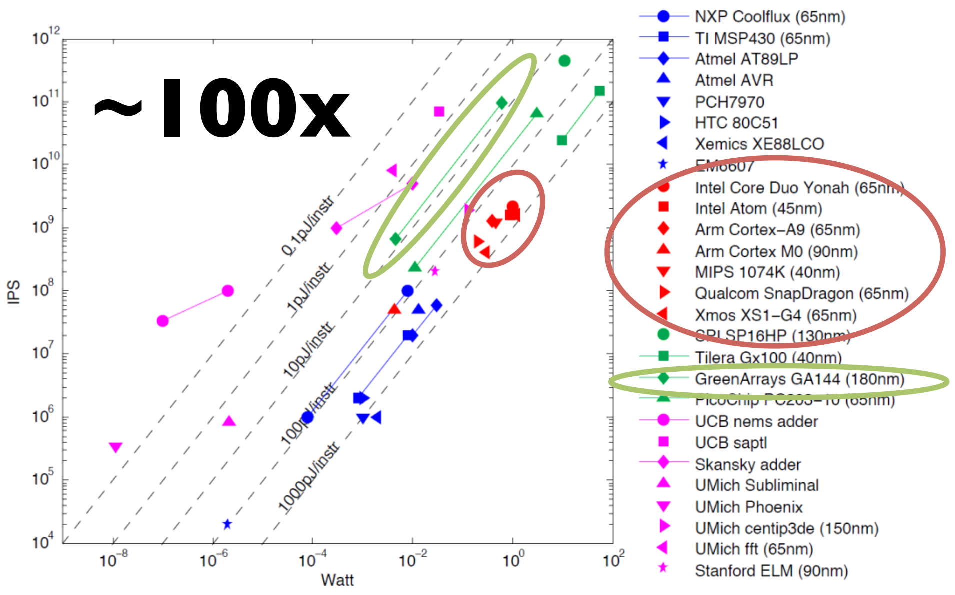
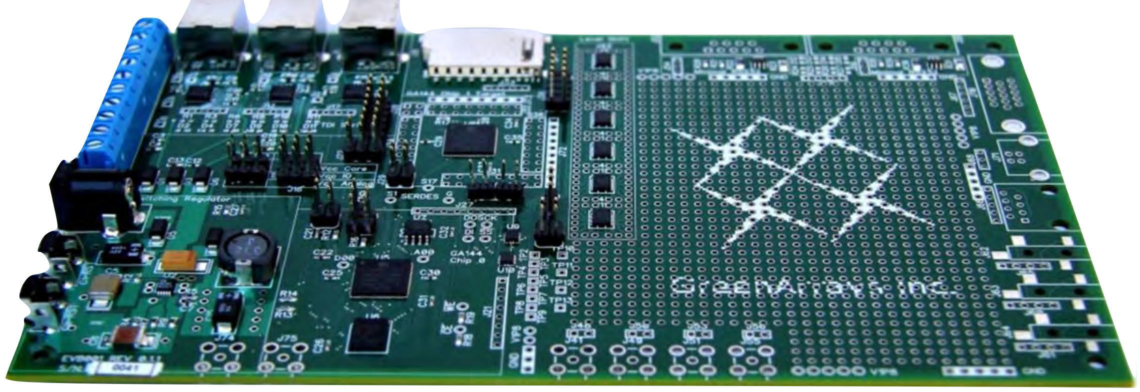


Figure by Per Ljung

## GreenArrays GA144 Processor



# Chlorophyll: ultra low-power computing

## GreenArrays GA144 Processor

- Stack-based 18-bit architecture
- 32 instructions
- 8 x 18 array of asynchronous cores
- No shared resources (cache, memory)
- Limited communication, neighbors only
- < 300 byte memory per core

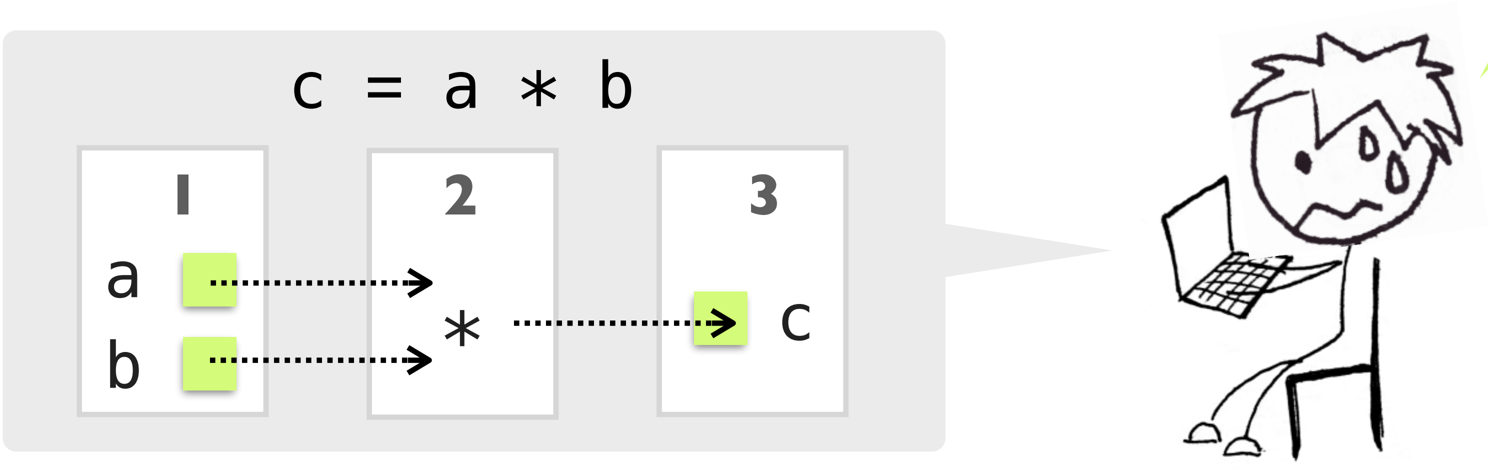
Manual program partitioning: break programs up into a pipeline with a few operations per core.

Drawing by Mangpo Phothilimthana

# Chlorophyll: ultra low-power computing

## GreenArrays GA144 Processor

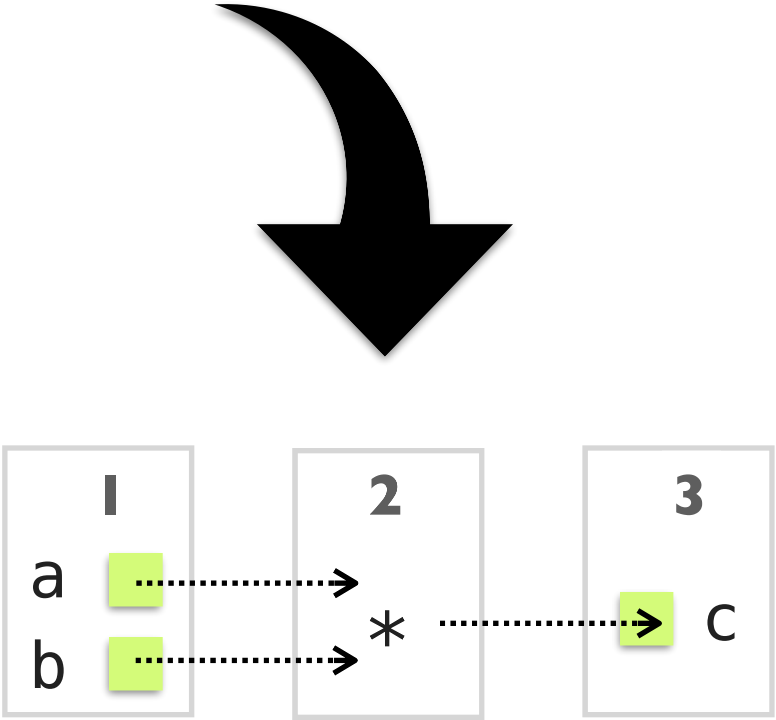
- Stack-based 18-bit architecture
- 32 instructions
- 8 x 18 array of asynchronous cores
- No shared resources (cache, memory)
- Limited communication, neighbors only
- < 300 byte memory per core



Drawing by Mangpo Phothilimthana

# Chlorophyll:  ultra low-power computing

```
int a, b;
int c = a * b;
```

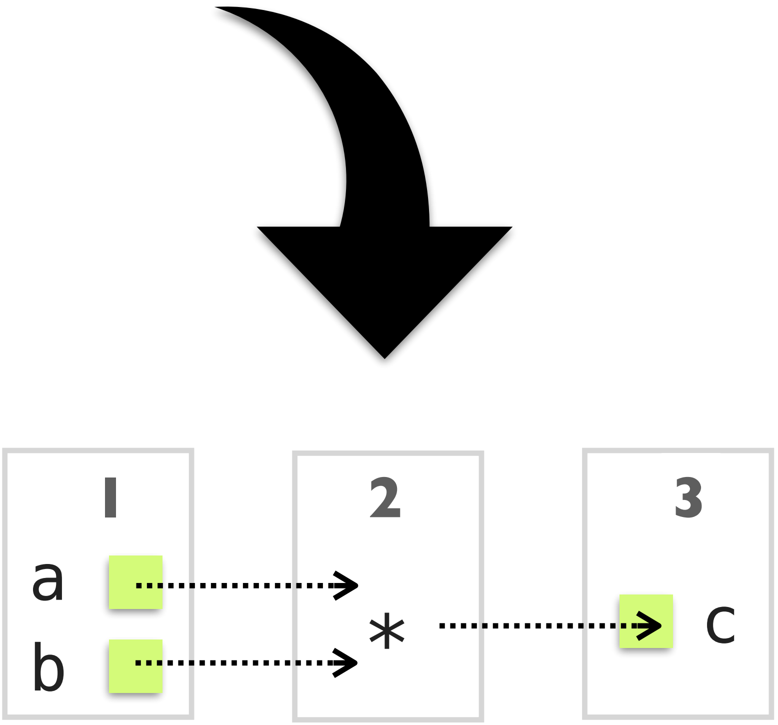


Synthesizes placement of code and data onto cores, by type-checking a program sketch in a C-like DSL.

# Chlorophyll: ultra low-power computing

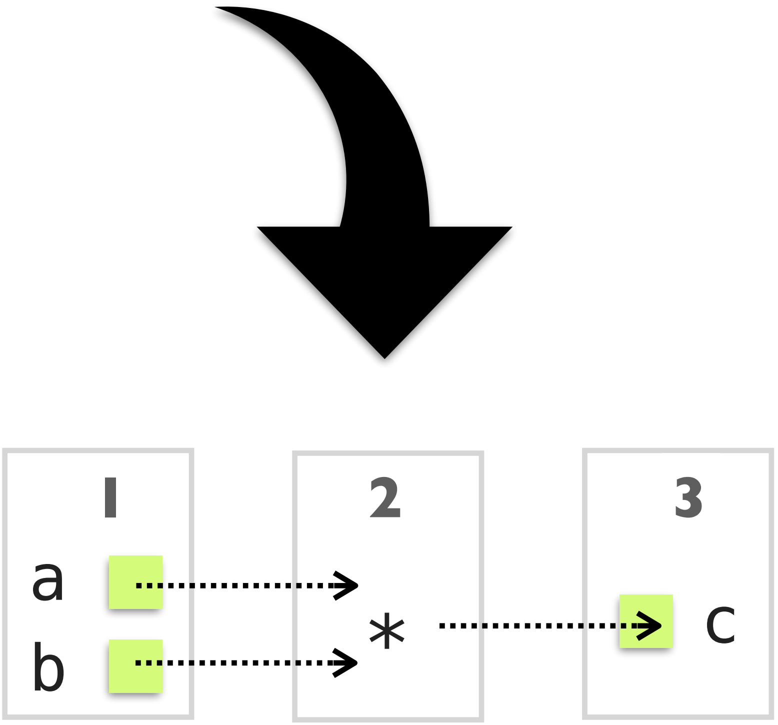
```
int@1 a, b;
int@3 c = a *@2 b;
```

Synthesizes placement of code and data onto cores, by **type-checking a program** sketch in a C-like DSL.



# Chlorophyll:  ultra low-power computing

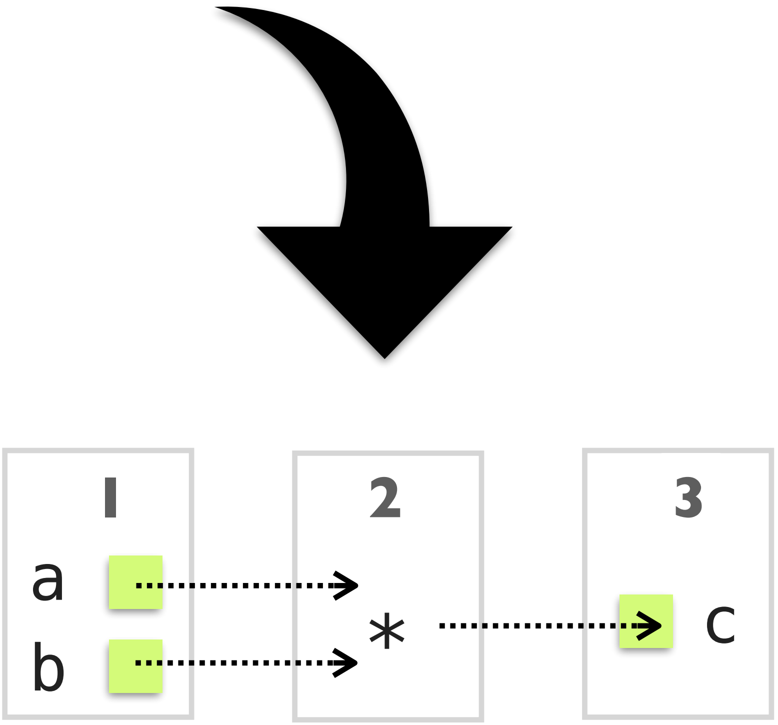
```
int@?? a, b;
int@?? c = a *@?? b;
```



Synthesizes placement of code and data onto cores, by type-checking a program **sketch** in a C-like DSL.

# Chlorophyll: ultra low-power computing

```
int@?? a, b;
int@?? c = a *@?? b;
```

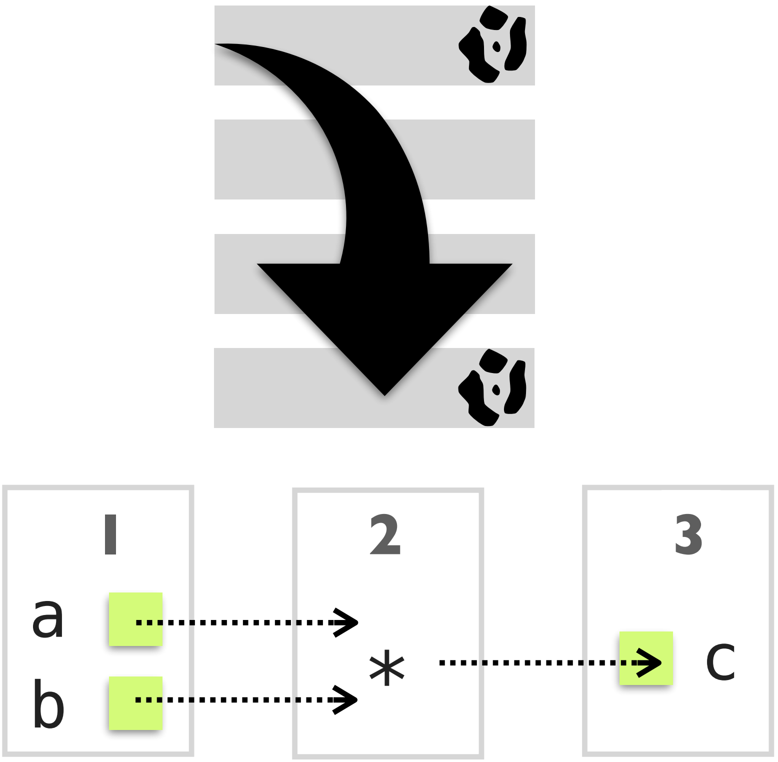


**Built by a first-year grad in a few weeks**

**Phitchaya Mangpo Phothilimthana**

# Chlorophyll: ultra low-power computing

```
int@?? a, b;
int@?? c = a *@?? b;
```

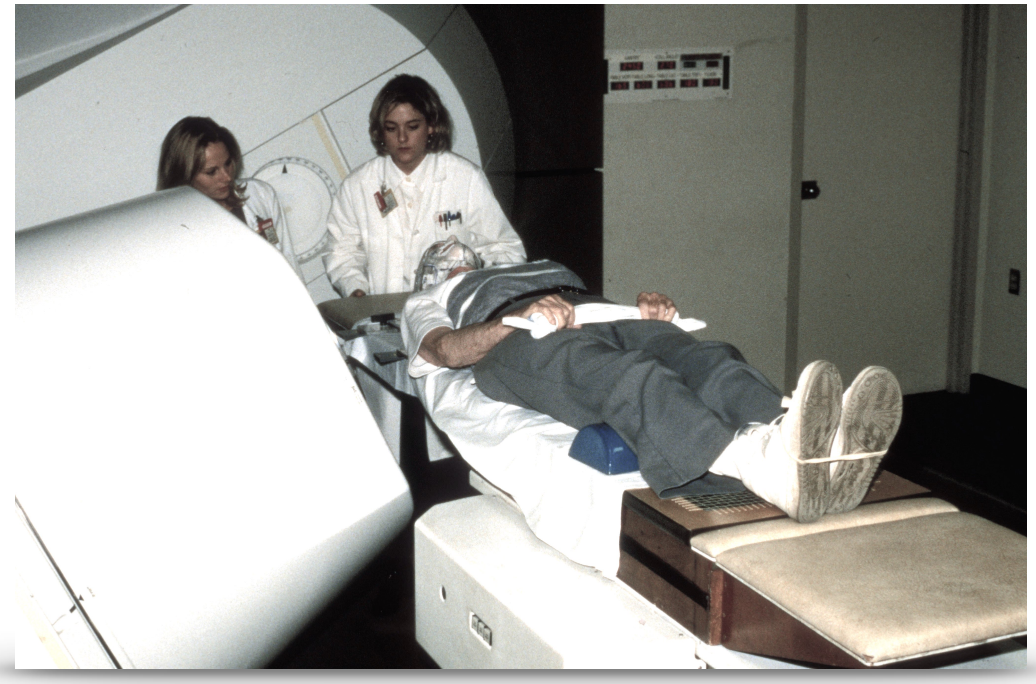


With Chlorophyll, it took one afternoon to build a set of apps that took 3 months to build manually.

[Phothilimthana et al., **PLDI'14**]

# Neutrons: verifying a radiotherapy system

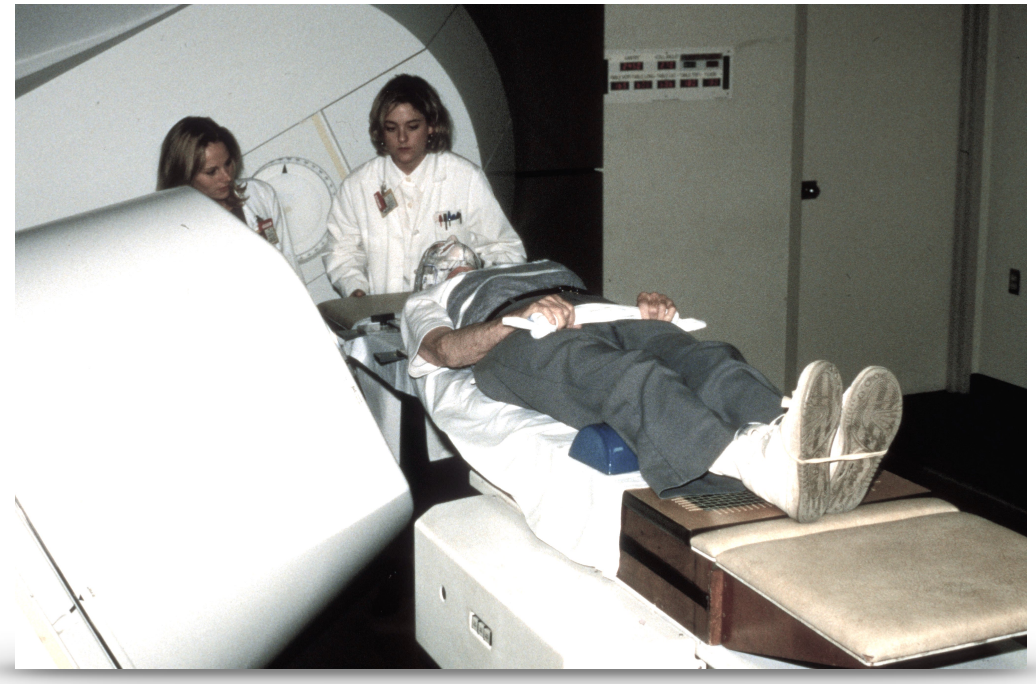
**Clinical Neutron Therapy System (CNTS) at UW**

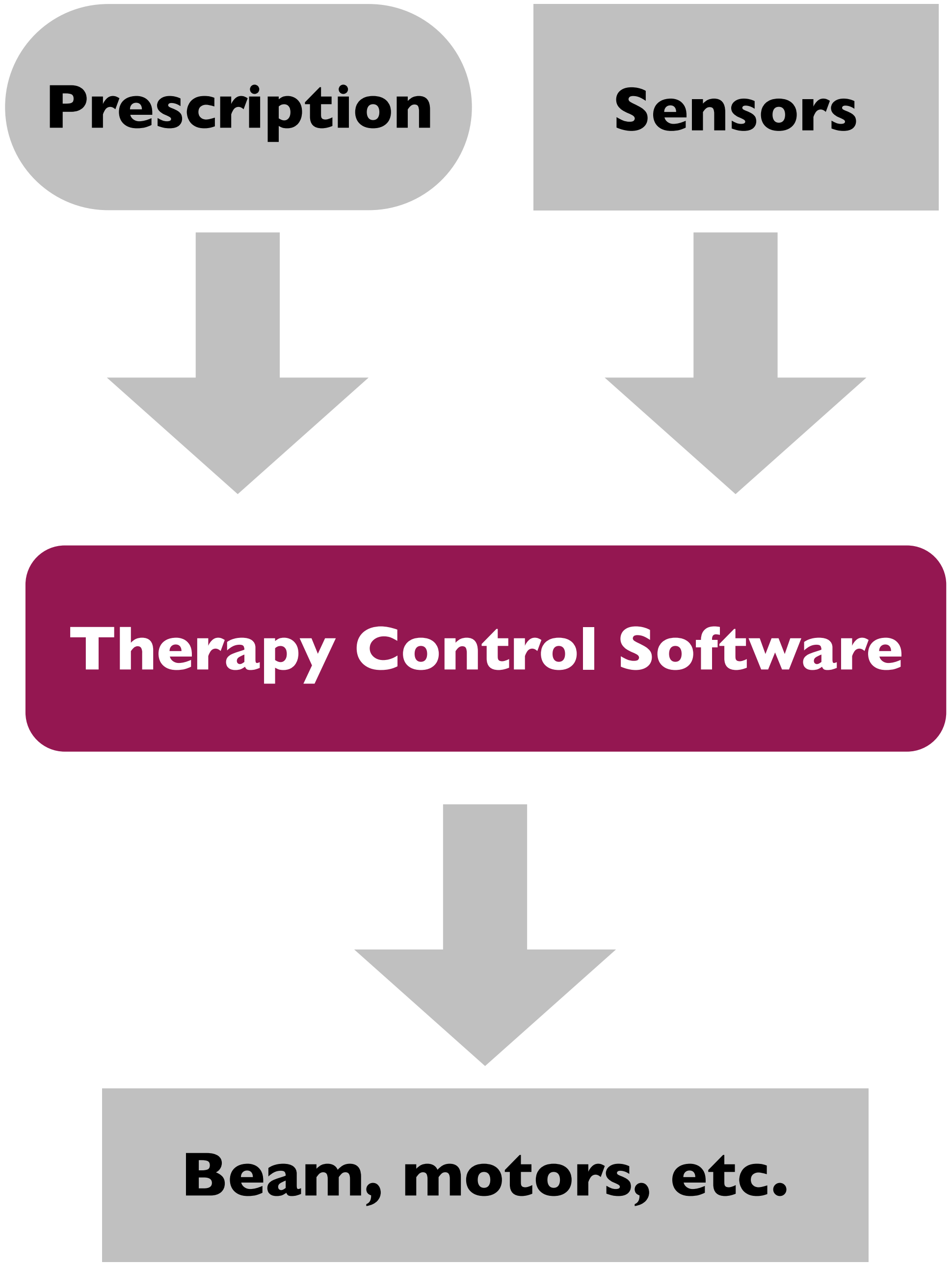


- 30 years of incident-free service.
- Controlled by custom software, built by CNTS engineering staff.
- Third generation of Therapy Control software built recently.

# Neutrons: verifying a radiotherapy system

Clinical Neutron Therapy System (CNTS) at UW





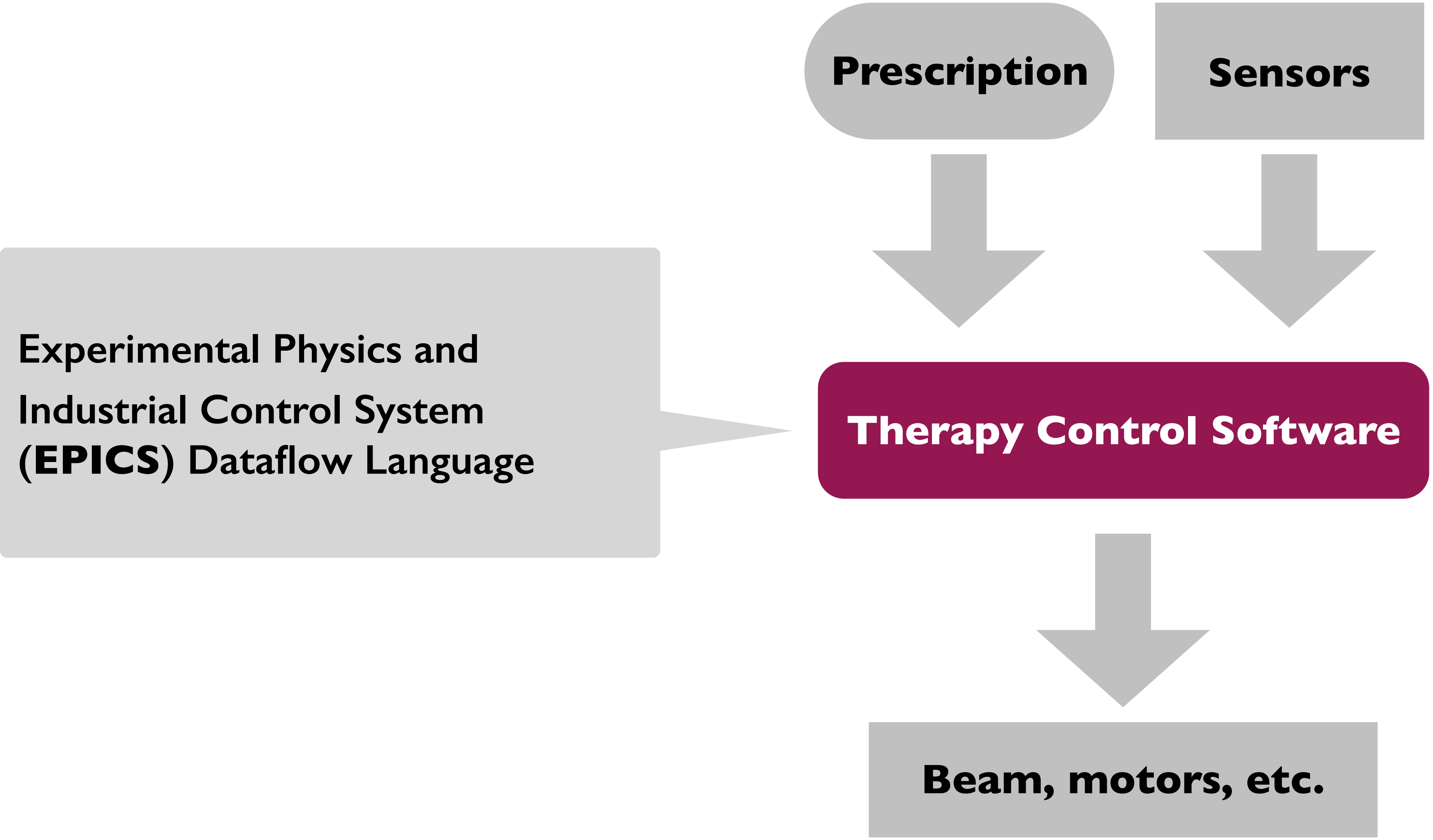
Prescription

Sensors

Therapy Control Software

Beam, motors, etc.

# Neutrons: verifying a radiotherapy system



Prescription

Sensors

Therapy Control Software

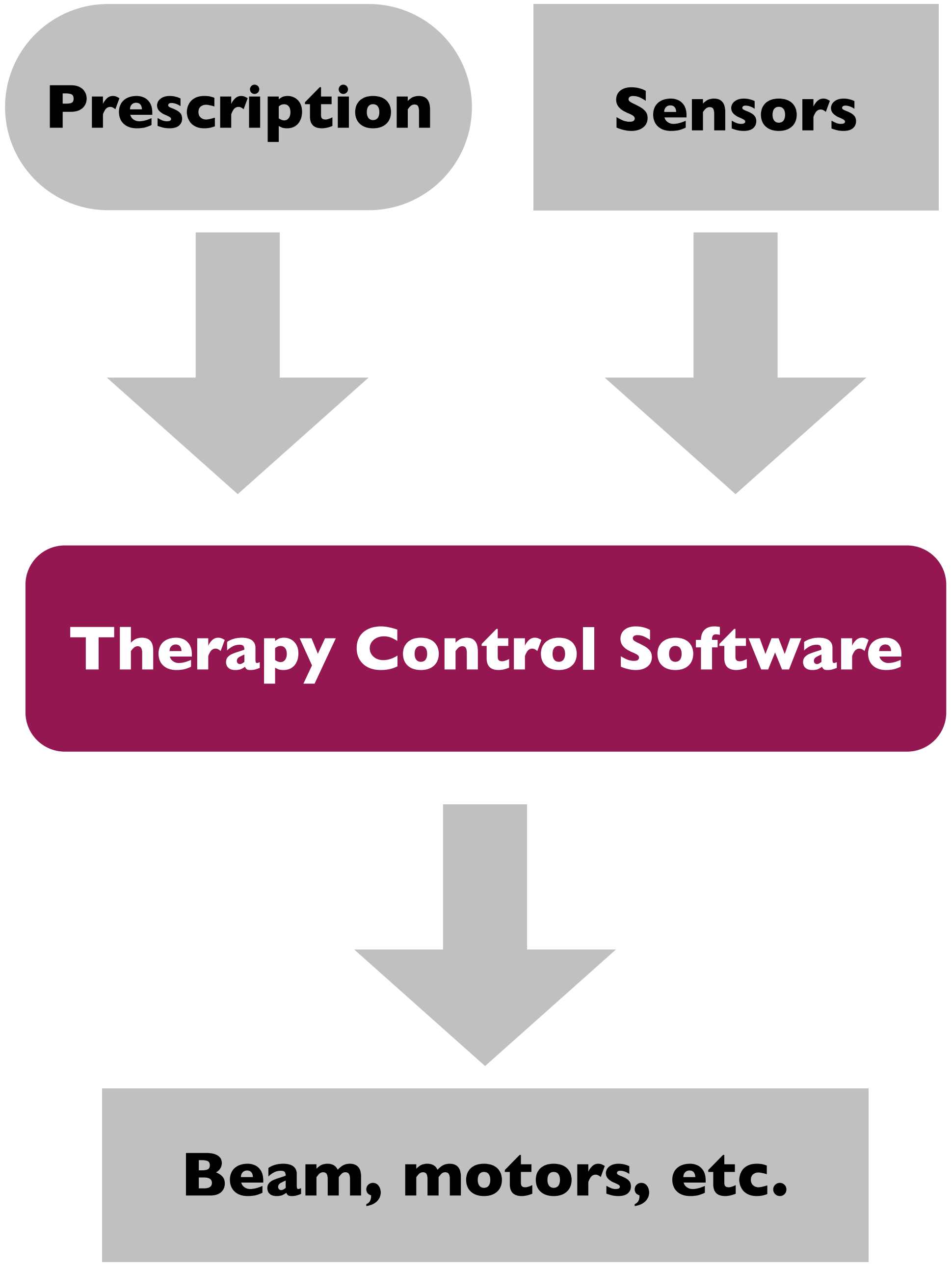
Beam, motors, etc.

Experimental Physics and Industrial Control System (**EPICS**) Dataflow Language

# Neutrons: verifying a radiotherapy system

## EPICS documentation / semantics

The Maximize Severity attribute is one of NMS (Non-Maximize Severity), MS (Maximize Severity), MSS (Maximize Status and Severity) or MSI (Maximize Severity if Invalid). It determines whether alarm severity is propagated across links. If the attribute is MSI only a severity of INVALID_ALARM is propagated; settings of MS or MSS propagate all alarms that are more severe than the record's current severity. For input links the alarm severity of the record referred to by the link is propagated to the record containing the link. For output links the alarm severity of the record containing the link is propagated to the record referred to by the link. If the severity is changed the associated alarm status is set to LINK_ALARM, except if the attribute is MSS when the alarm status will be copied along with the severity.



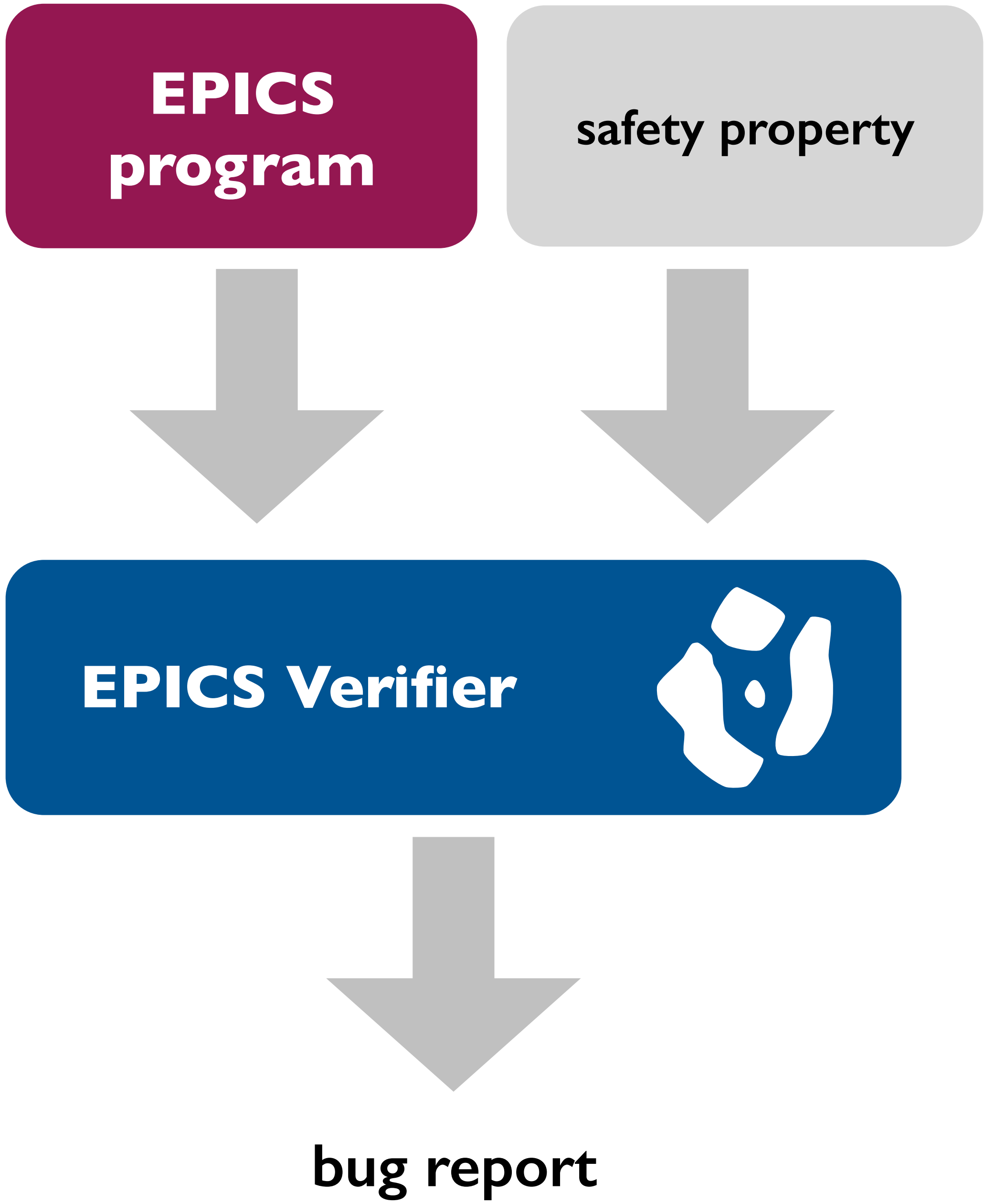
Prescription

Sensors

Therapy Control Software

Beam, motors, etc.

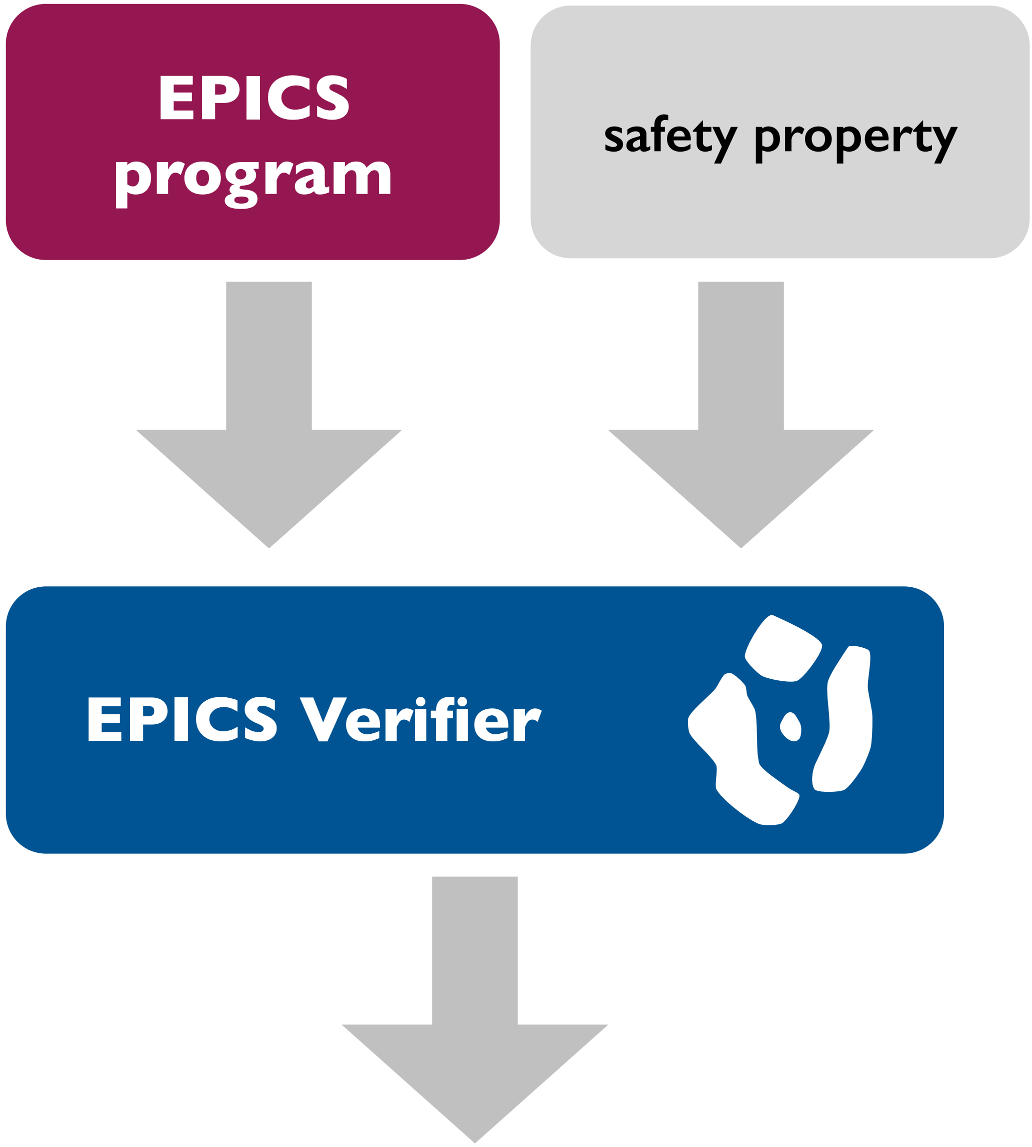
# Neutrons: verifying a radiotherapy system



**Built by a 2nd year grad in a few days**

**Calvin Loncaric**

# Neutrons: verifying a radiotherapy system



EPICS program

safety property

EPICS Verifier

Found a bug in the EPICS runtime! Therapy Control depended on this bug for correct operation.

[Pernsteiner et al., CAV'16]

# MemSynth: synthesizing memory models

**Memory consistency models define memory reordering behaviors on multiprocessors.**

```
  x = y = 0
a = x | b = y
y = 1 | x = 1
  a ≡ b ≡ 1
```

# MemSynth: synthesizing memory models

**Memory consistency models define memory reordering behaviors on multiprocessors.**

```
  x = y = 0
-------------
a = x | b = y
y = 1 | x = 1
-------------
  a ≡ b ≡ 1
```

Forbidden by sequential consistency.

Allowed by x86 and other hardware memory models.

# MemSynth: synthesizing memory models

**Memory consistency models define memory reordering behaviors on multiprocessors.**

Formalizing memory models is hard: e.g., PowerPC formalized over 7 publications in 2009-2015.

```
   x = y = 0
--------------
a = x | b = y
y = 1 | x = 1
--------------
   a ≡ b ≡ 1
```

Forbidden by sequential consistency.

Allowed by x86 and other hardware memory models.

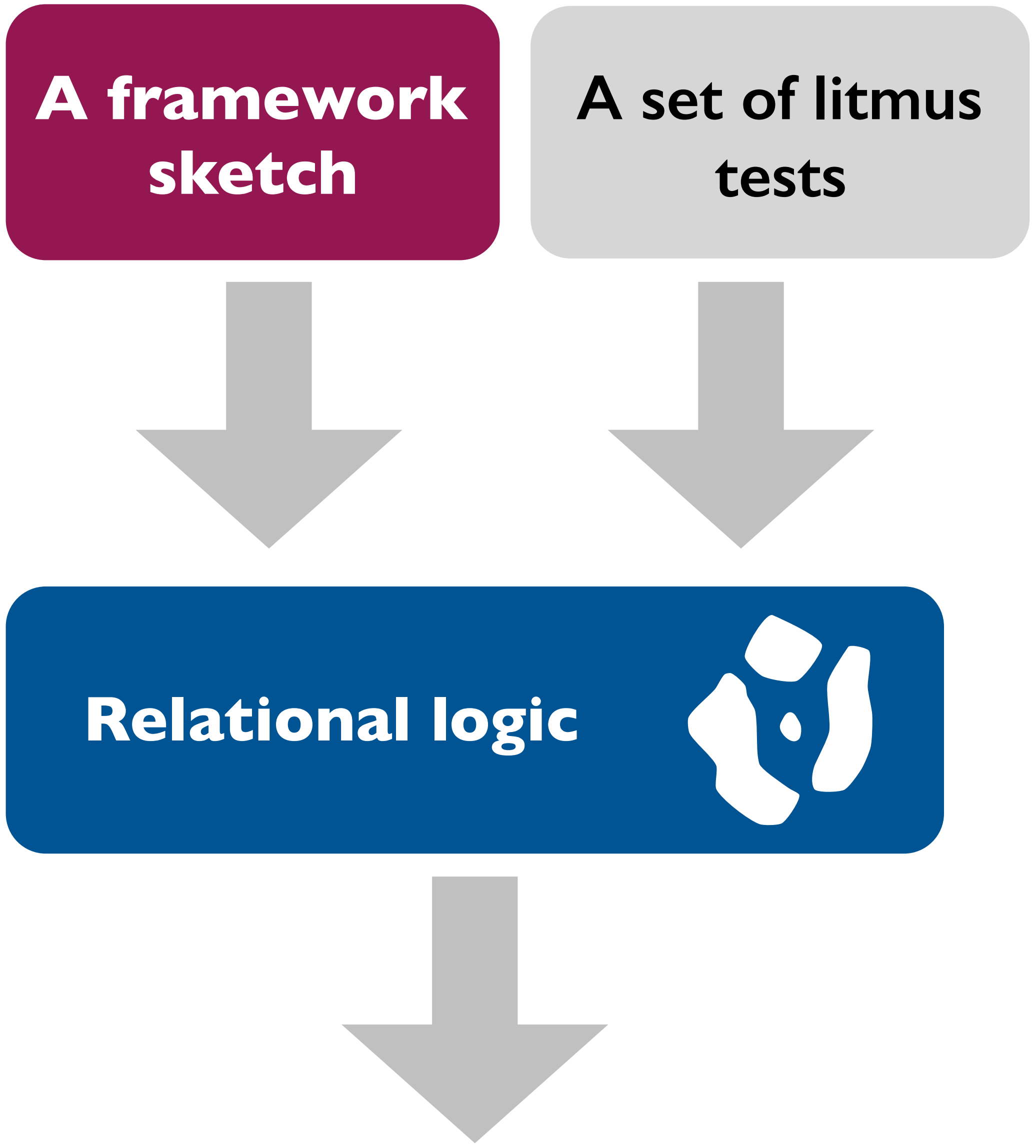
# MemSynth: synthesizing memory models

Memory consistency models define memory reordering behaviors on multiprocessors.

```
   x = y = 0
-------------
a = x | b = y
y = 1 | x = 1
-------------
   a ≡ b ≡ 1
```

Forbidden by sequential consistency.

Allowed by x86 and other hardware memory models.



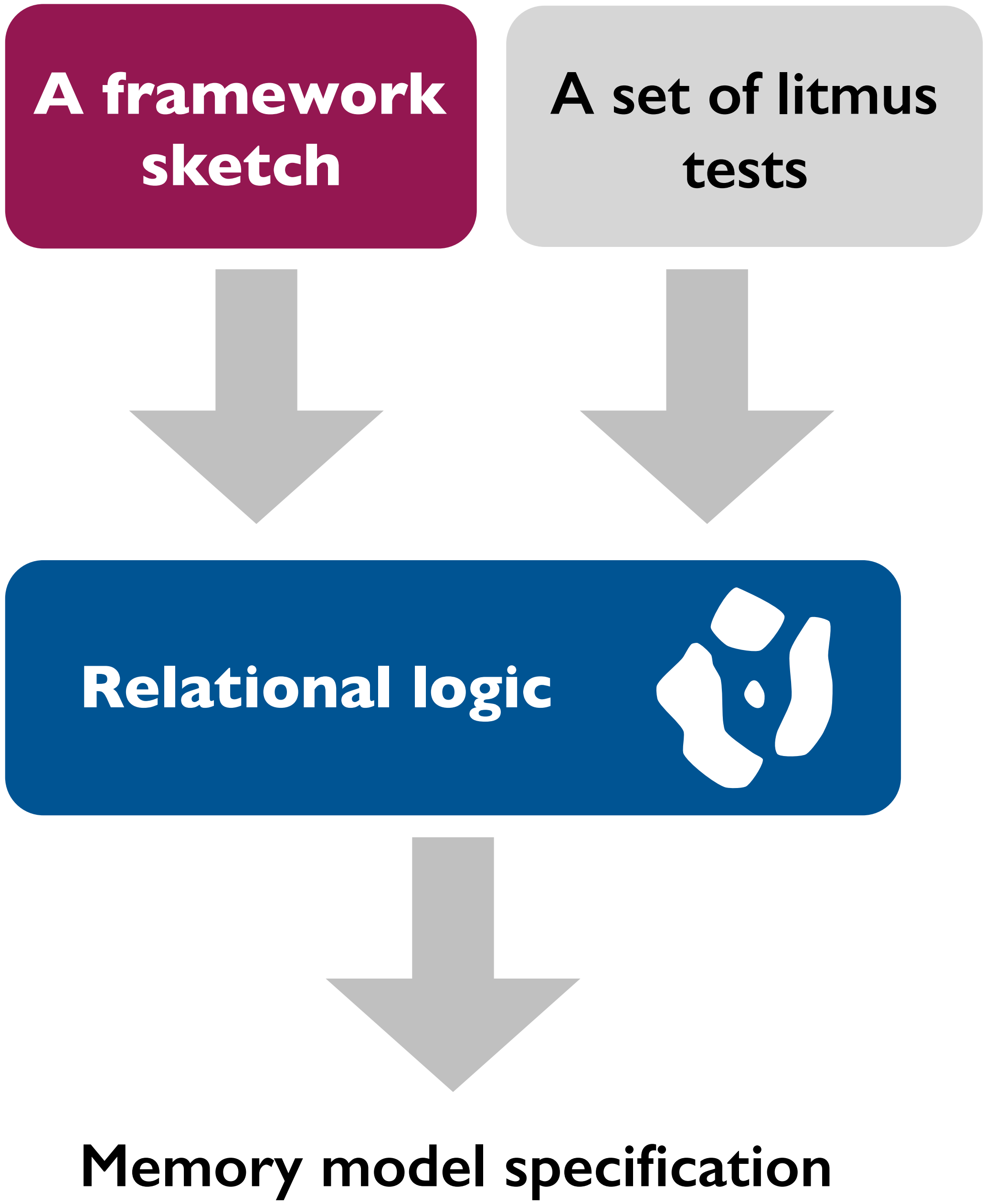
A framework sketch

A set of litmus tests

Relational logic

Memory model specification

# MemSynth: synthesizing memory models



A framework sketch

A set of litmus tests

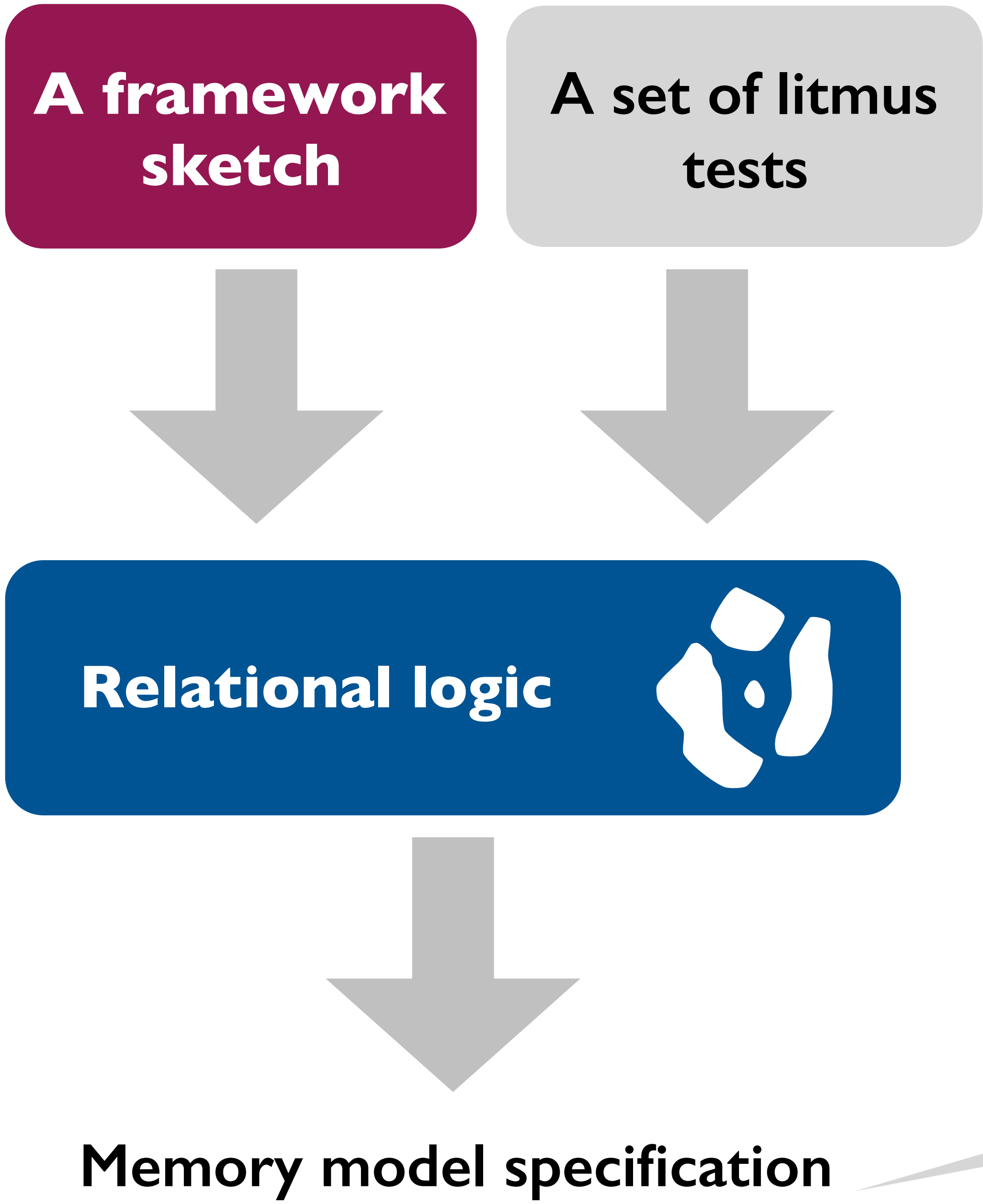
Relational logic

Memory model specification

**Built by a 2nd year grad in a few weeks**

**James Bornholt**

# MemSynth: synthesizing memory models



A framework sketch

A set of litmus tests

Relational logic

Memory model specification

[Bornholt and Torlak, **PLDI'17**]

Synthesized PowerPC in 12 seconds from 768 previously published tests.

Synthesized x86 in 2 seconds from Intel's litmus tests. Discovered 4 tests are missing from the Intel manual.

Thanks for a great quarter!