Computer-Aided Reasoning for Software

CSE507

Solver-Aided Languages

courses.cs.washington.edu/courses/cse507/17wi/

Emina Torlak
emina@cs.washington.edu

Today

topics

Last lecture

- Program synthesis

Today

- The next N years: solver-aided languages (?)

Reminders

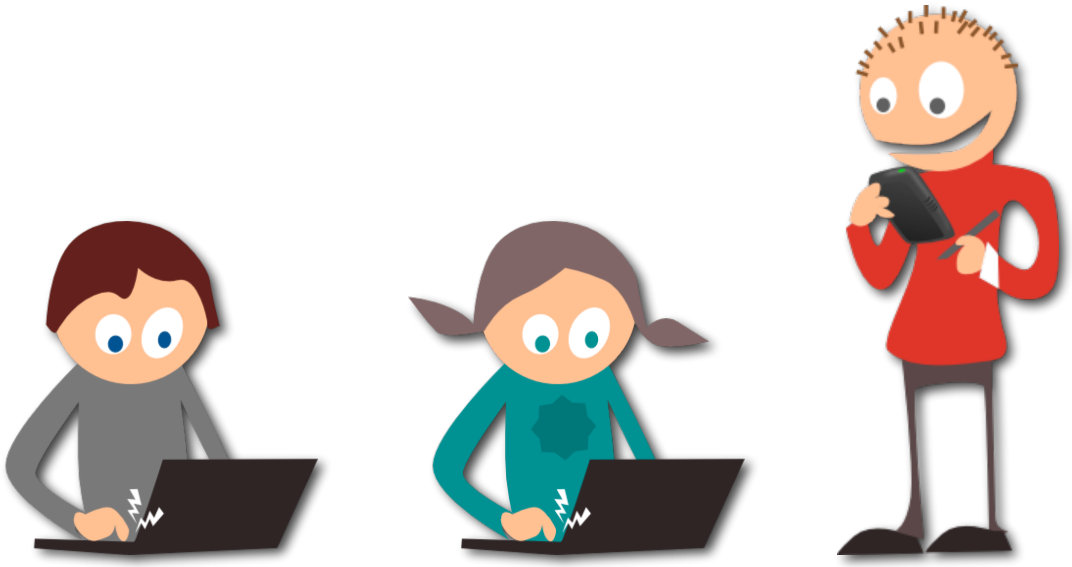
- Next lecture: metasketches!
- Project presentations next Friday in class
  - 11 min per team: 8 min presentation + 3 min questions
- Project reports and prototypes due next Friday at 11:00pm

vision

a little programming for everyone

A little programming for **everyone**

Every knowledge worker wants to program …



A little programming for everyone

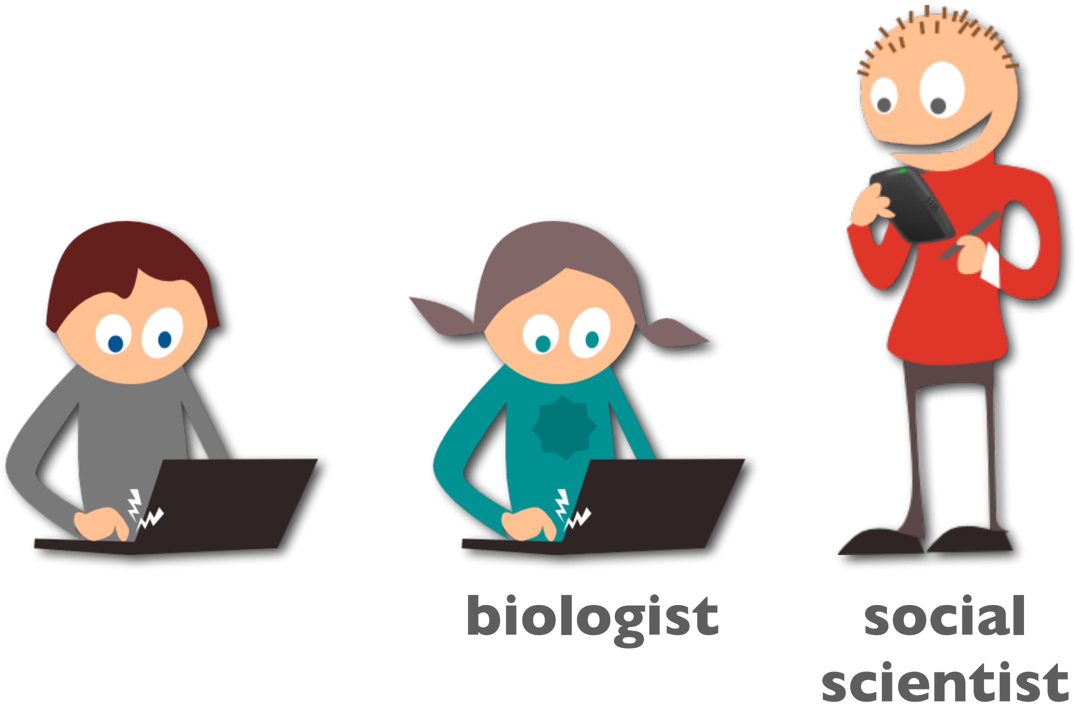
Every knowledge worker wants to program …

- spreadsheet data manipulation

A little programming for everyone

Every knowledge worker wants to program …

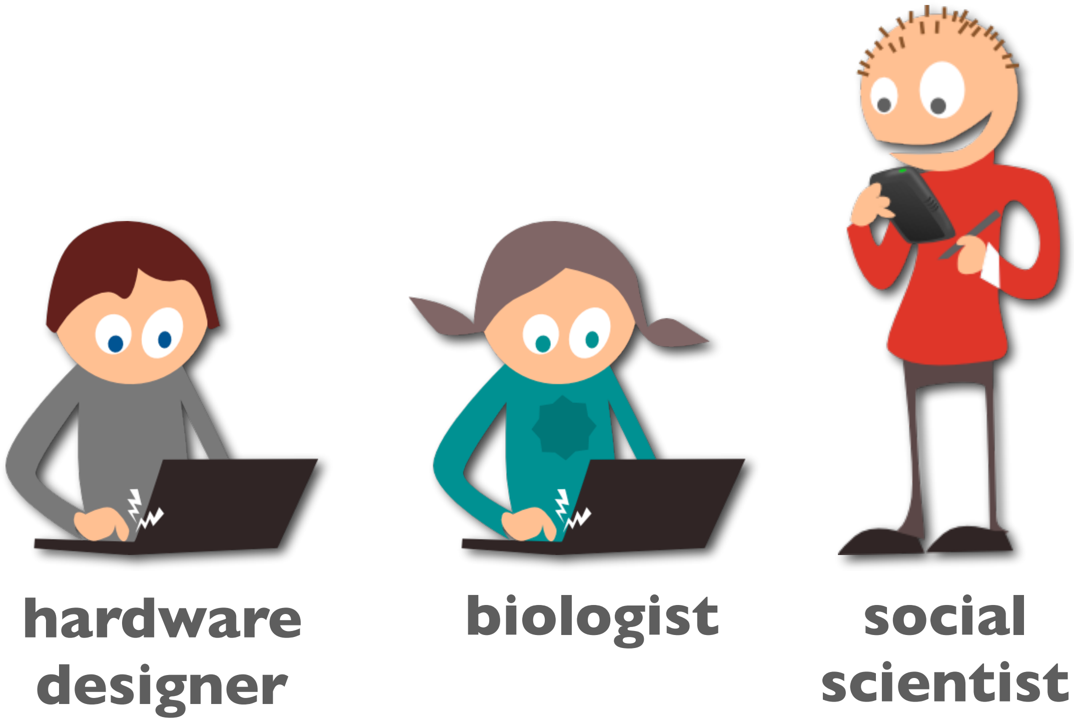
- spreadsheet data manipulation
- models of cell fates



A little programming for everyone

Every knowledge worker wants to program …

- spreadsheet data manipulation
- models of cell fates
- cache coherence protocols
- memory models



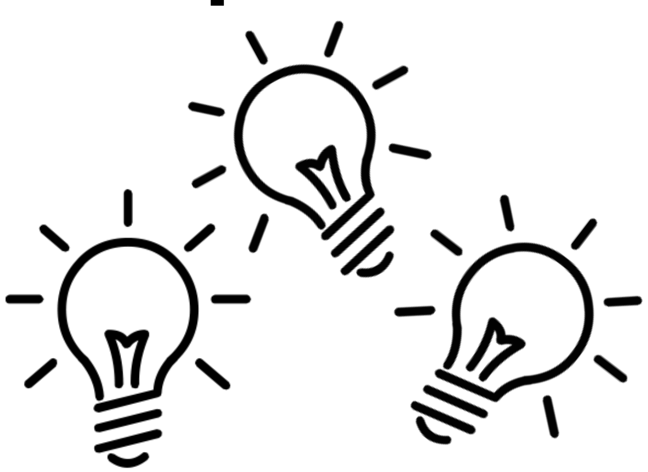
A little programming for **everyone**

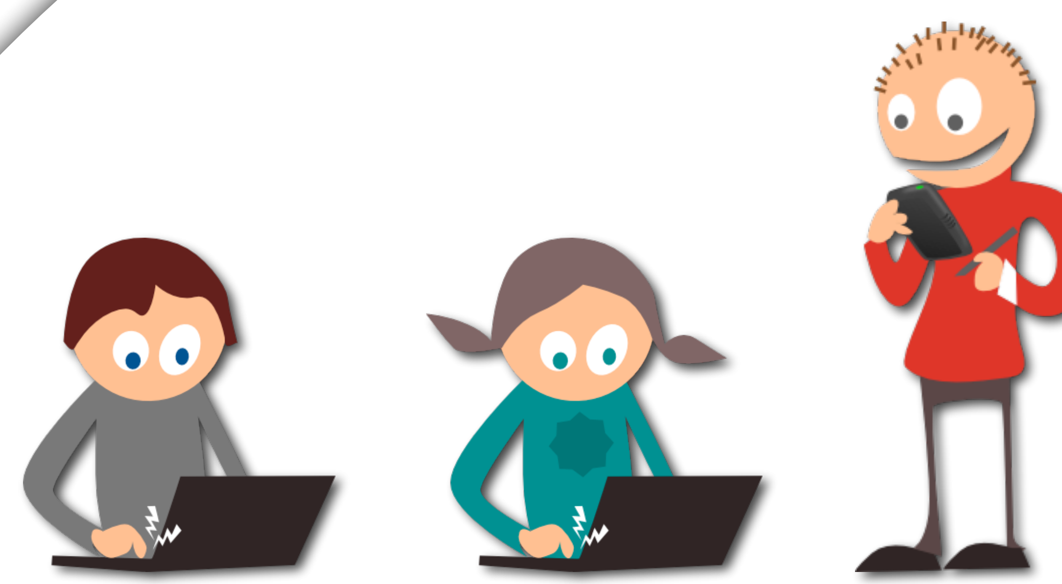
Every knowledge worker wants to program …

- spreadsheet data manipulation [Flashfill, **POPL'11**]
- models of cell fates [SBL, **POPL'13**]
- cache coherence protocols [Transit, **PLDI'13**]
- memory models [MemSAT, **PLDI'10**]

time

expertise





hardware designer

biologist

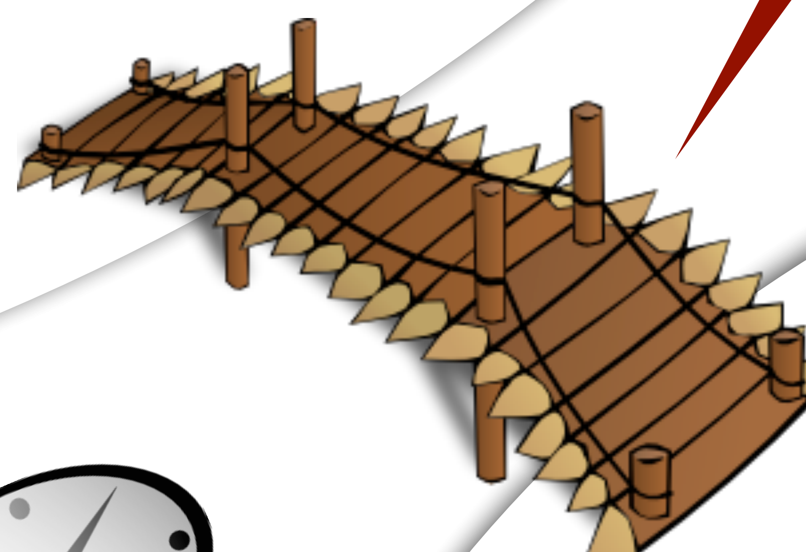
social scientist

A **little** programming for everyone

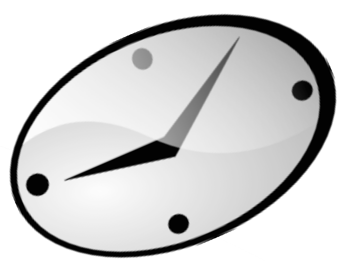
We all want to build programs …

- spreadsheet data manipulation
- models of cell fates
- cache coherence protocols
- memory models

solver-aided languages



less time

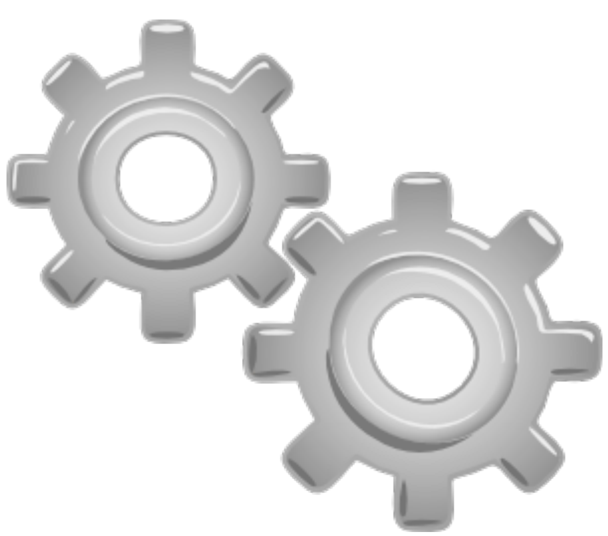


less expertise

outline

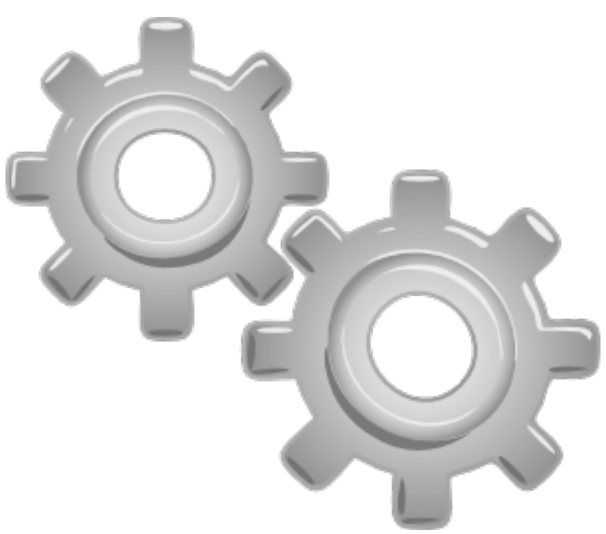
outline

solver-aided tools



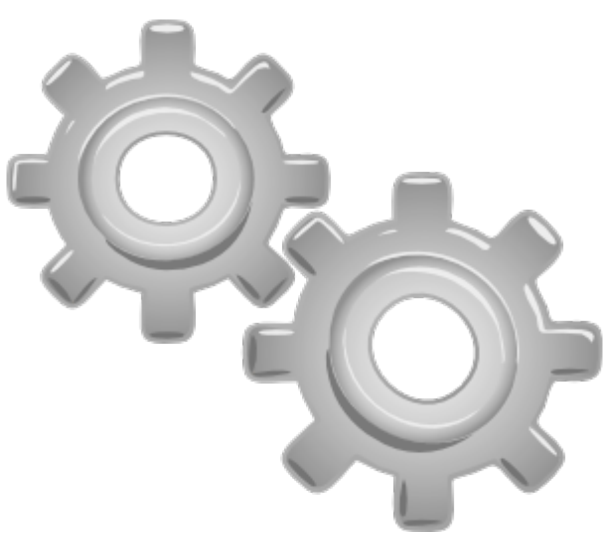
outline

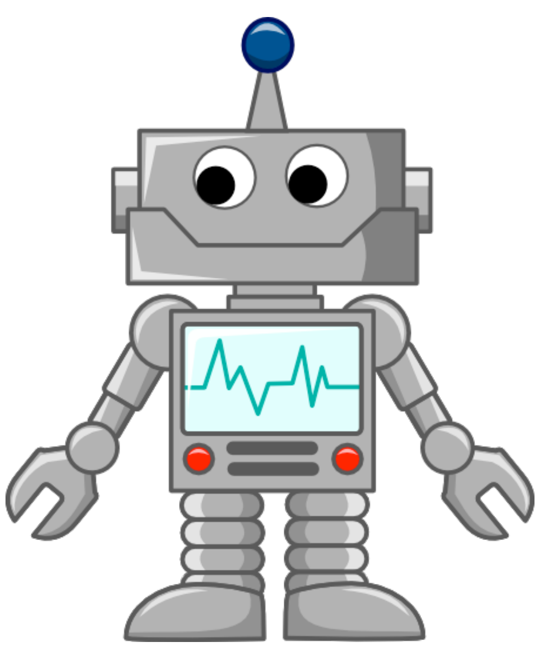
solver-aided tools, languages



outline

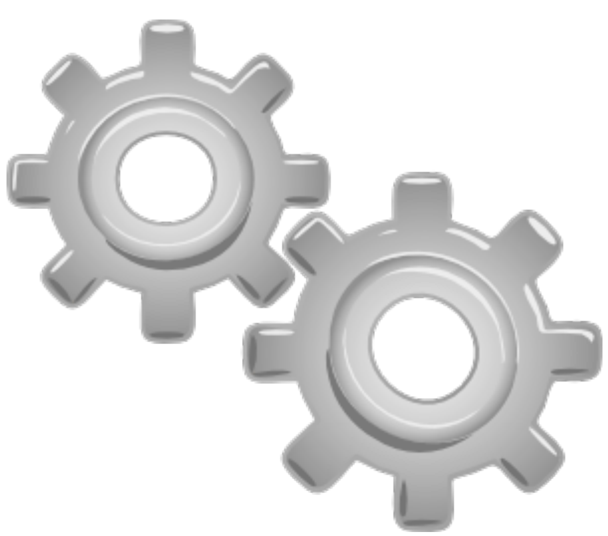
solver-aided tools, languages, and applications



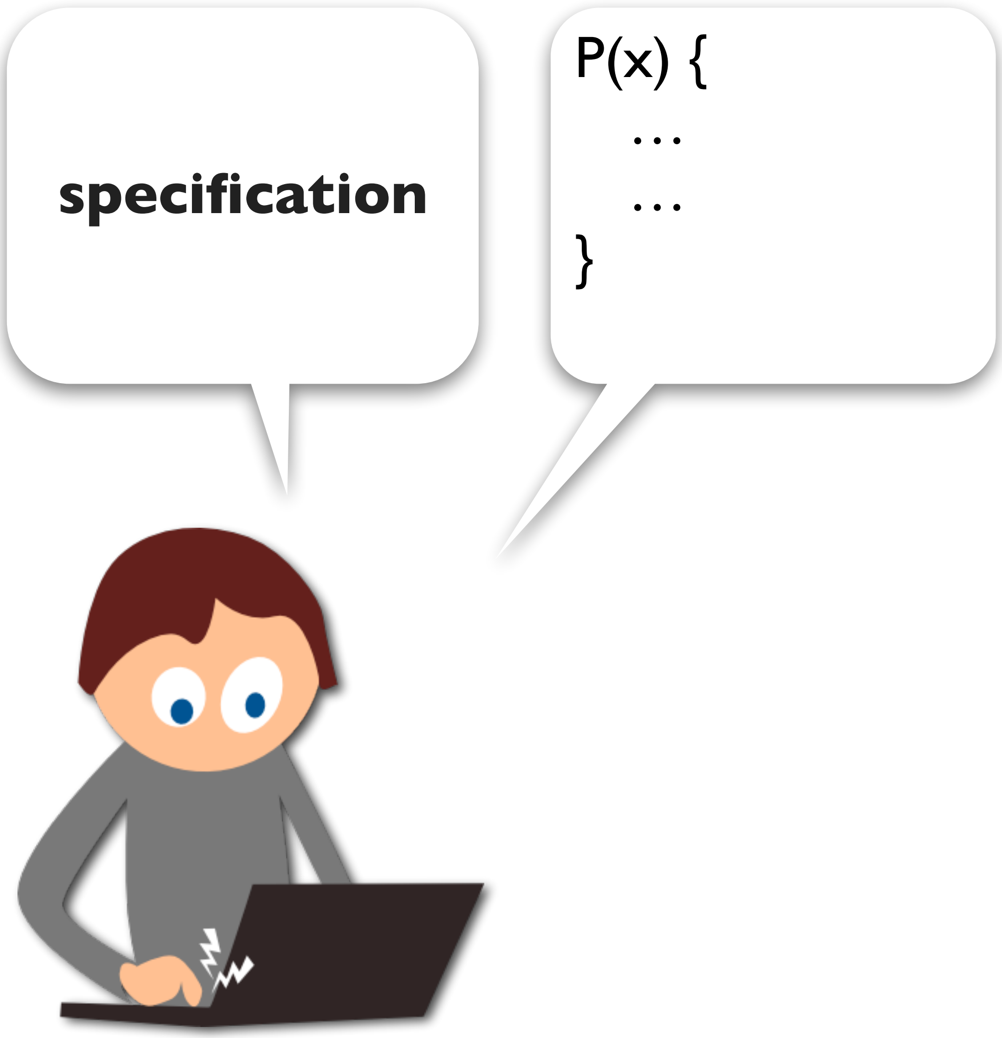


tools

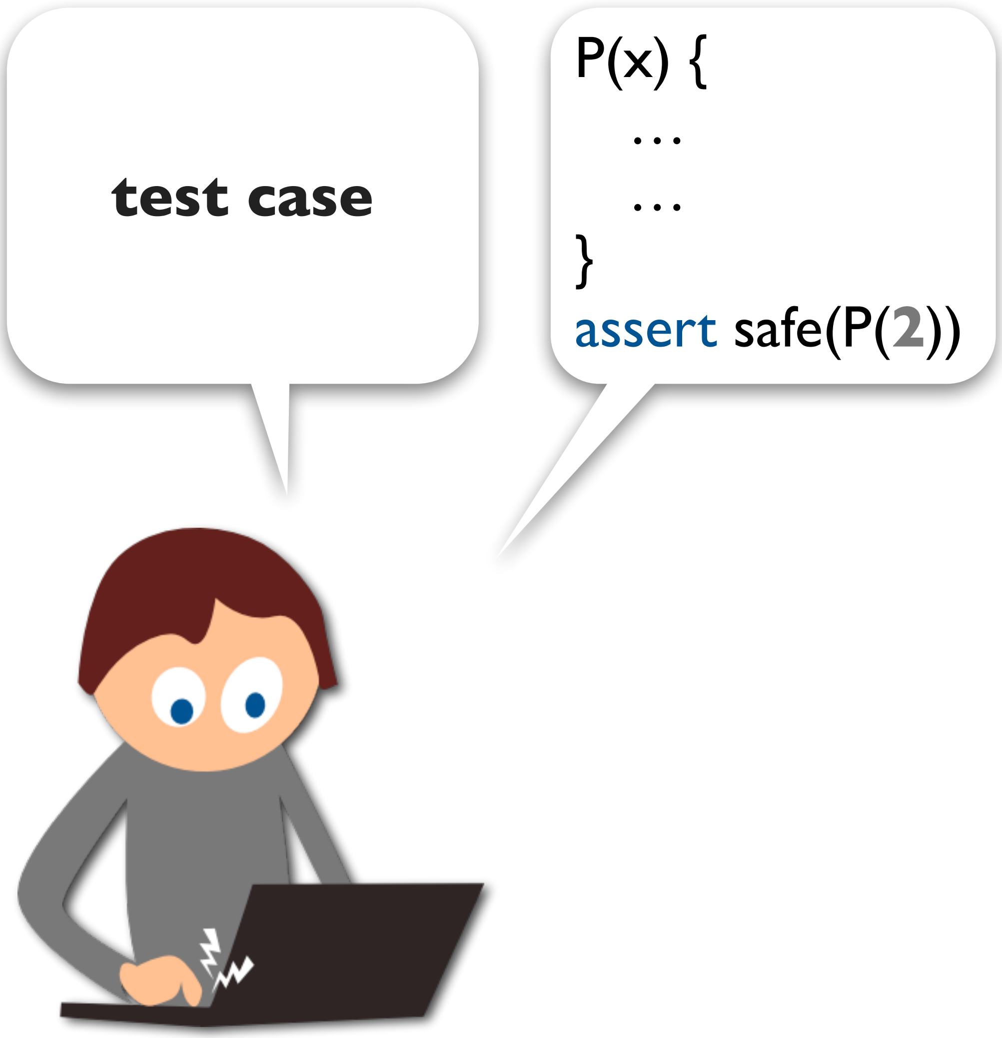
solver-aided tools



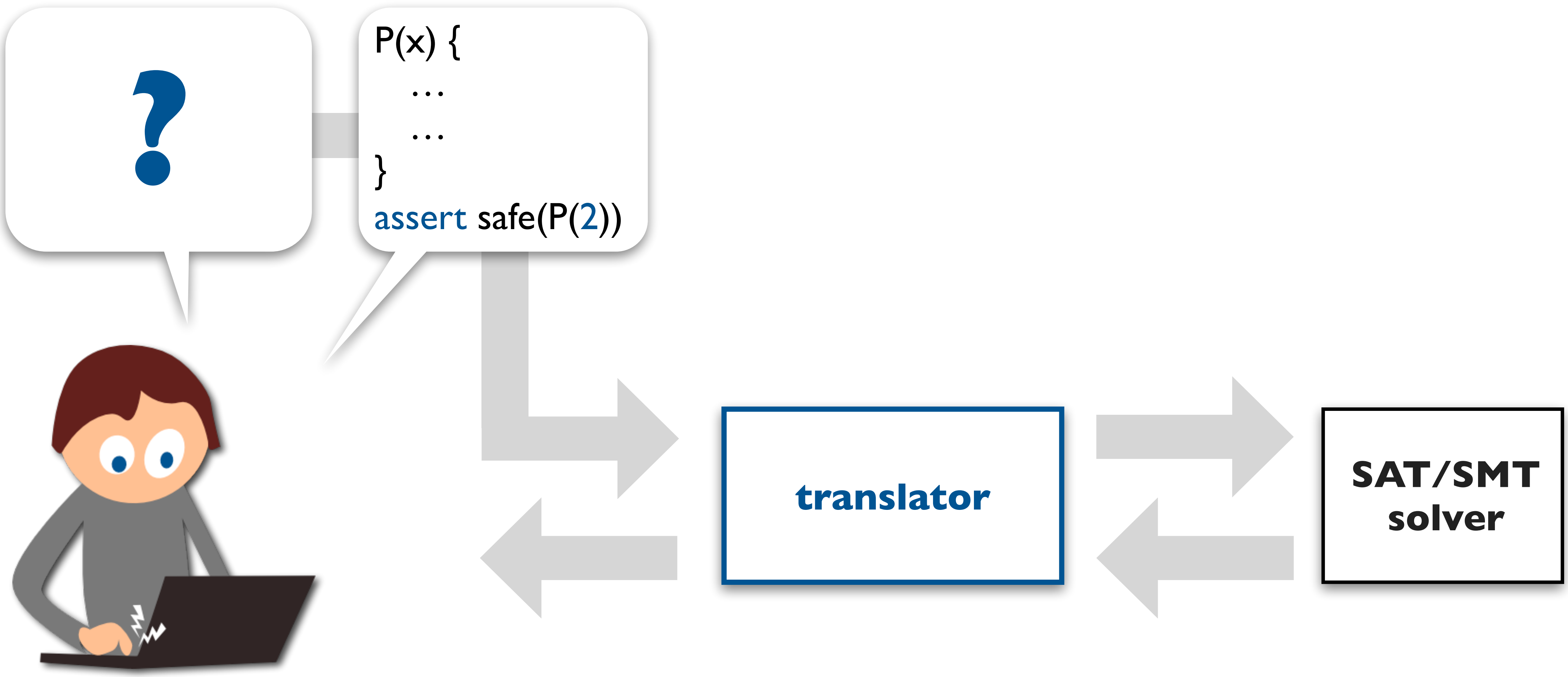
Programming ...



Programming …

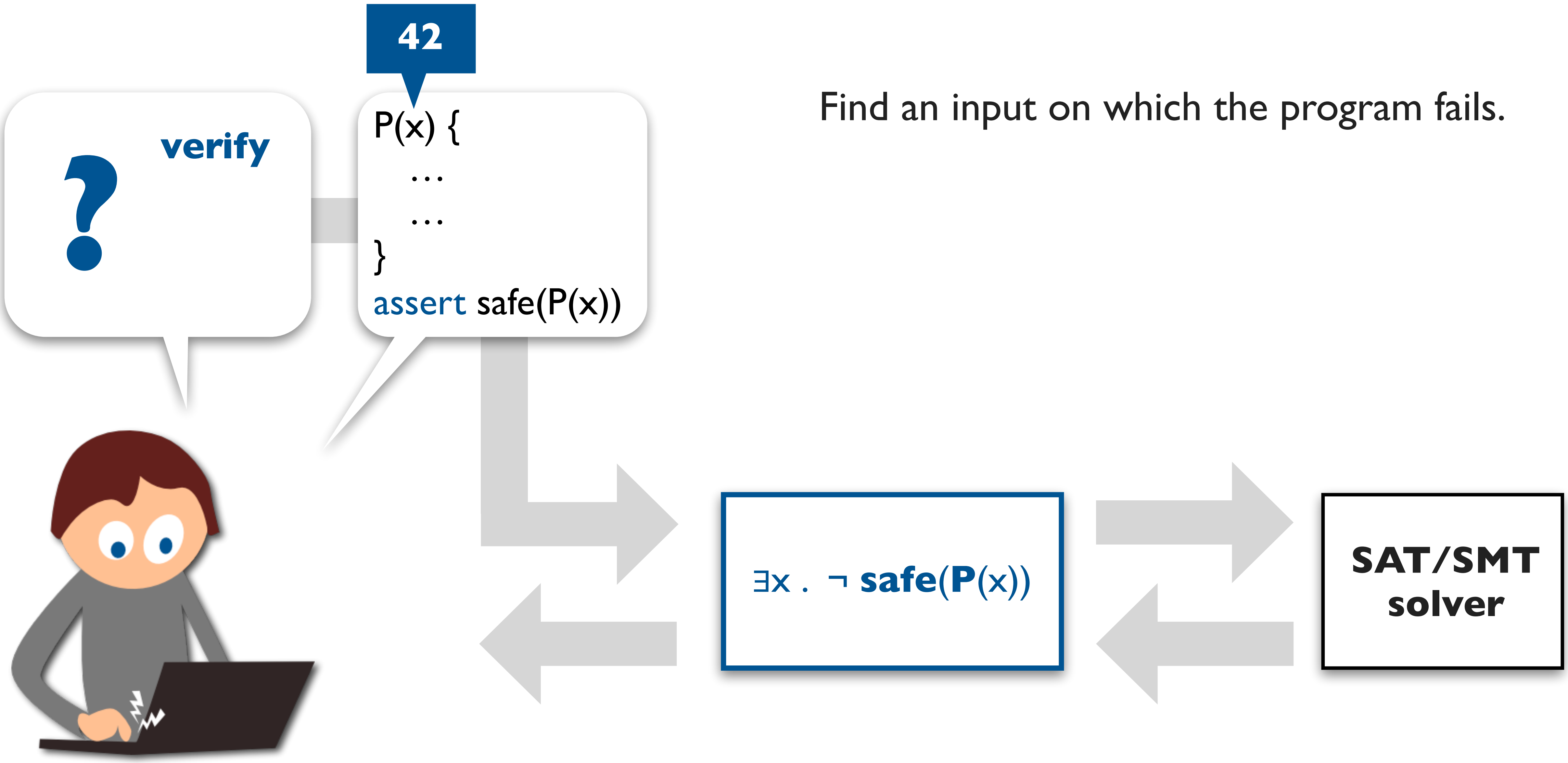


Programming with a solver-aided tool



Programming with a solver-aided tool

Find an input on which the program fails.

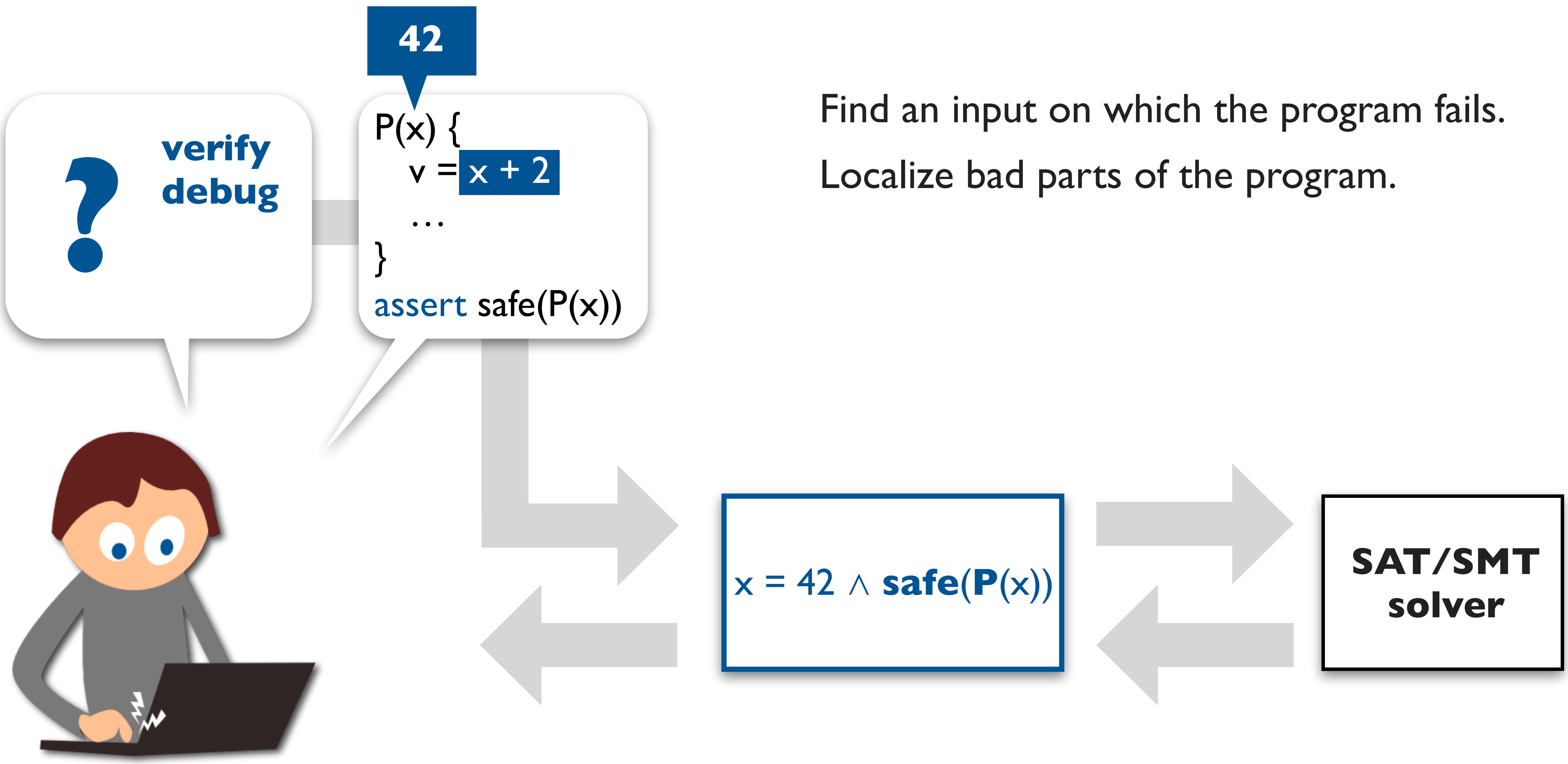


CBMC [Kroening et al., DAC'03]
Dafny [Leino, LPAR'10]
Miniatur [Vaziri et al., FSE'07]
Klee [Cadar et al., OSDI'08]

Programming with a solver-aided tool

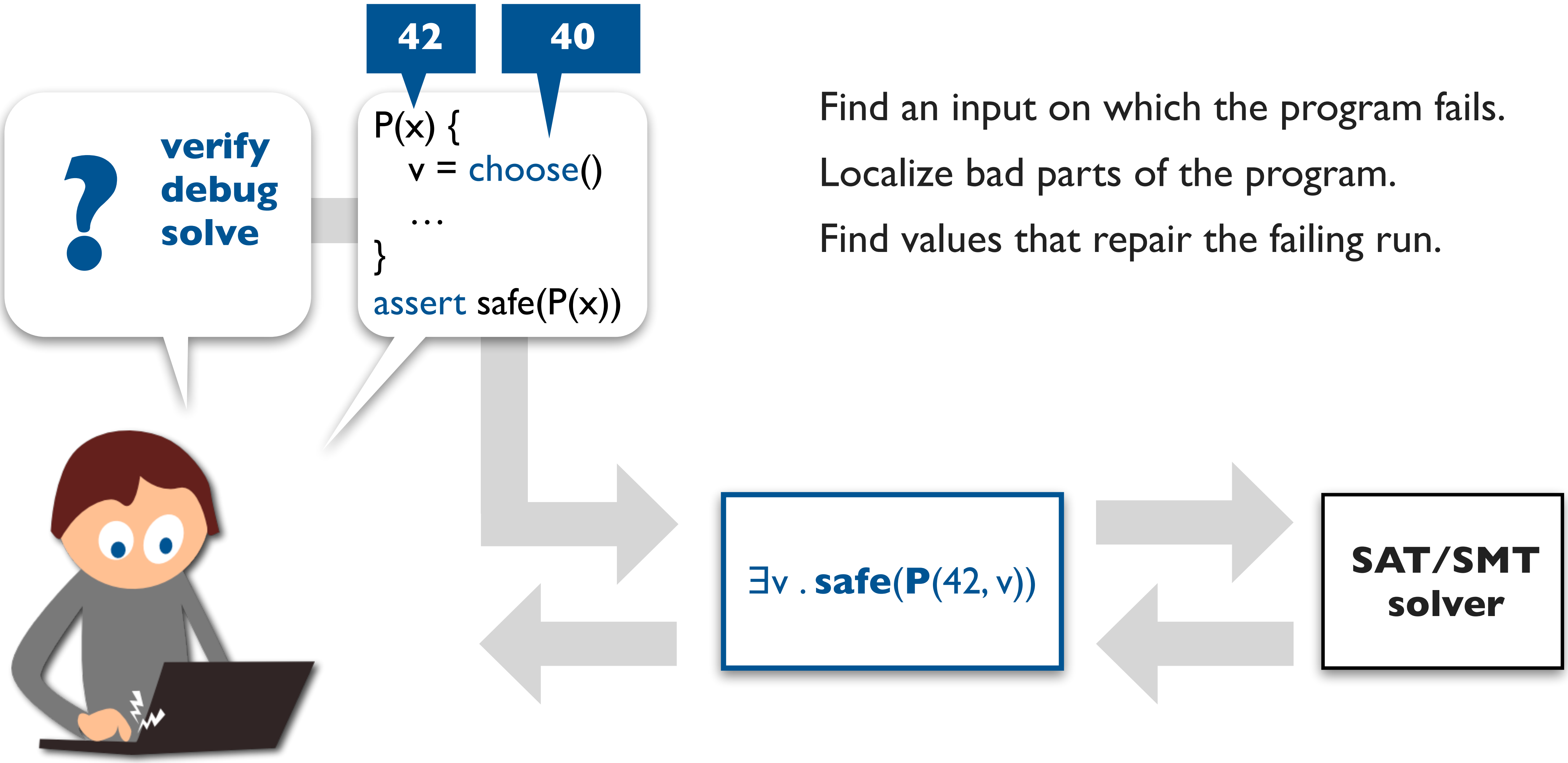
Find an input on which the program fails.

Localize bad parts of the program.



BugAssist [Jose & Majumdar, PLDI'11]

Programming with a solver-aided tool



Find an input on which the program fails.

Localize bad parts of the program.

Find values that repair the failing run.

Kaplan [Koksal et al, POPL'12]
PBnJ [Samimi et al., ECOOP'10]
Squander [Milicevic et al., ICSE'11]

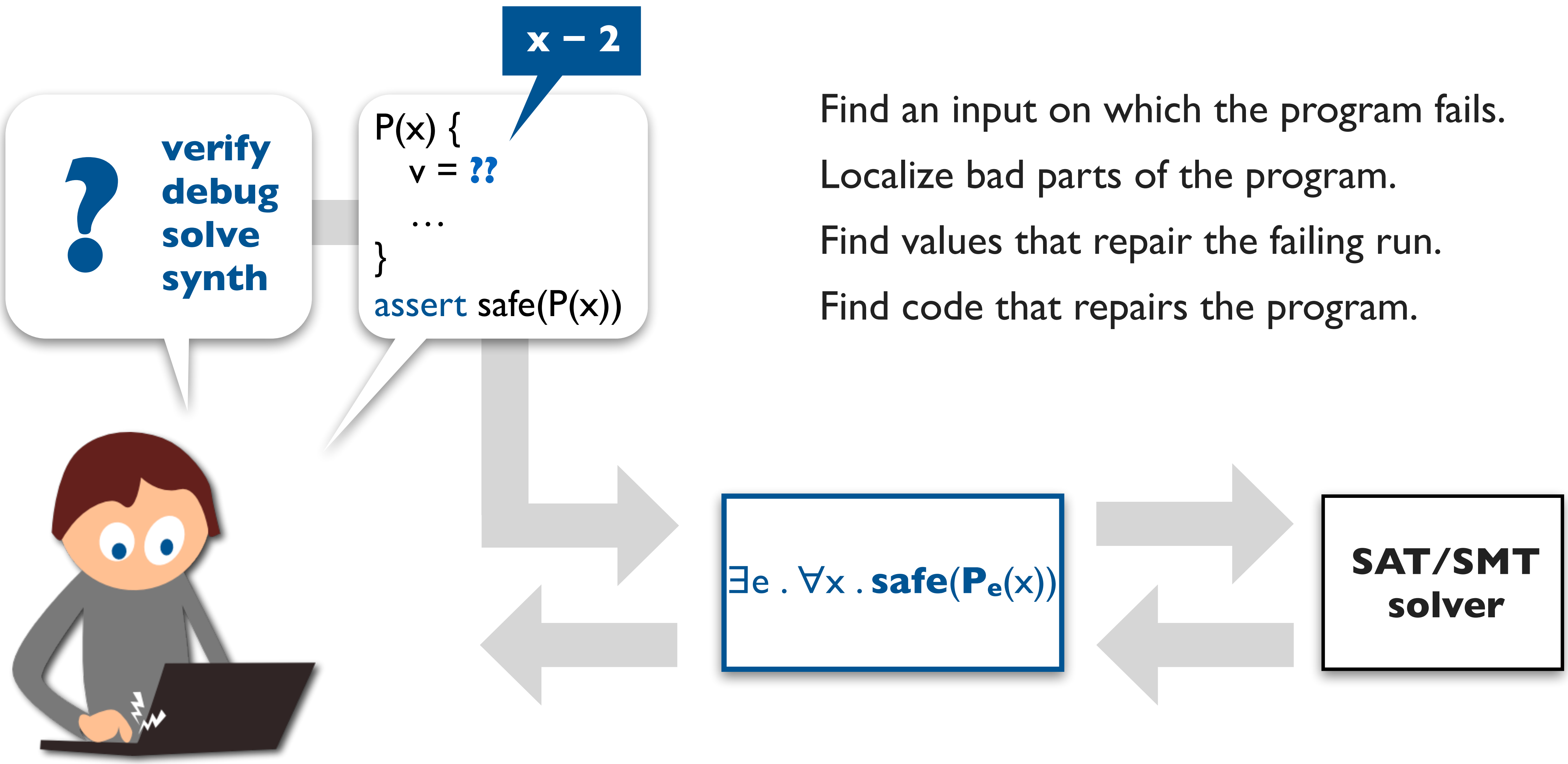
Programming with a solver-aided tool

Find an input on which the program fails.

Localize bad parts of the program.

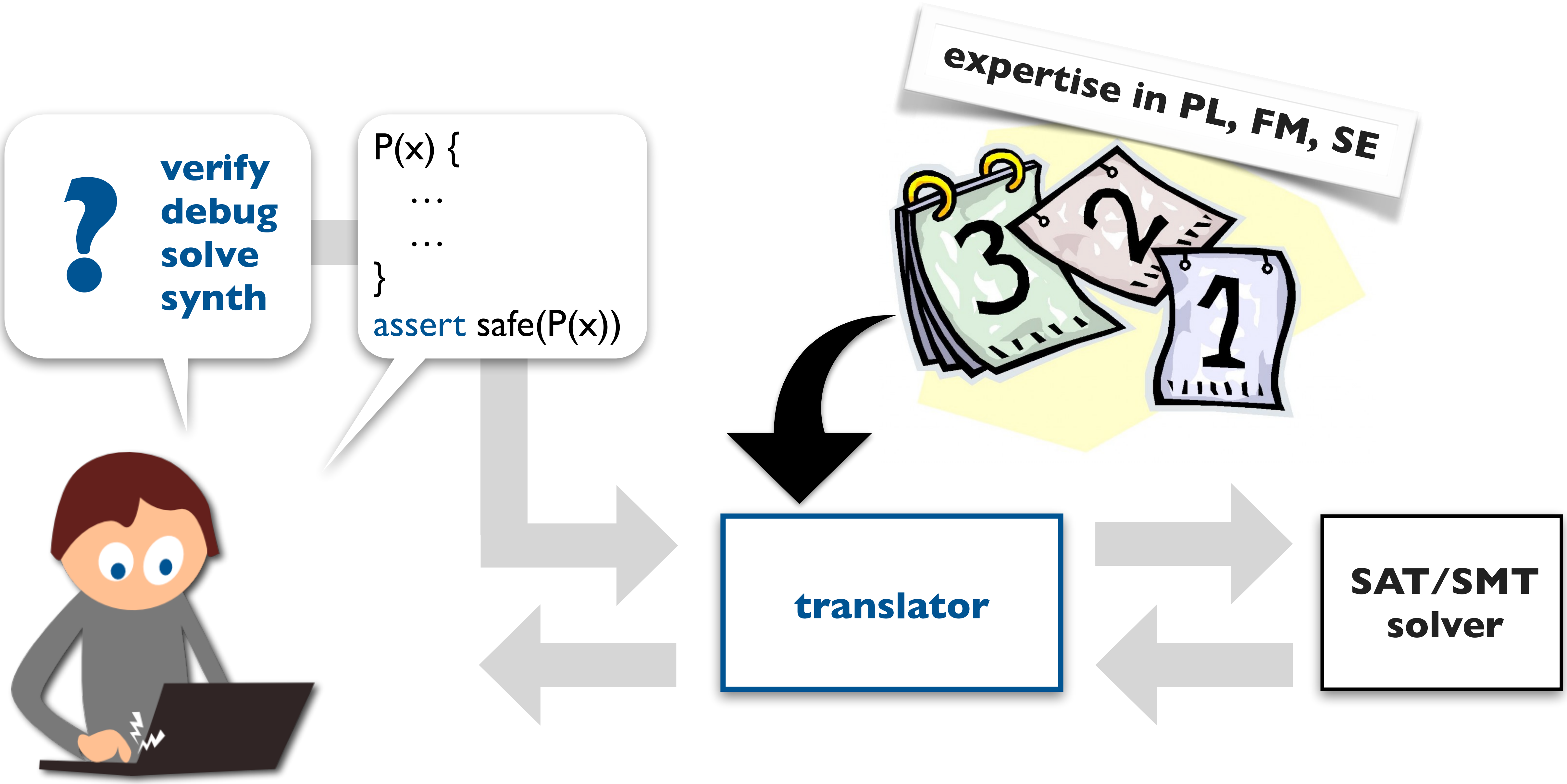
Find values that repair the failing run.

Find code that repairs the program.

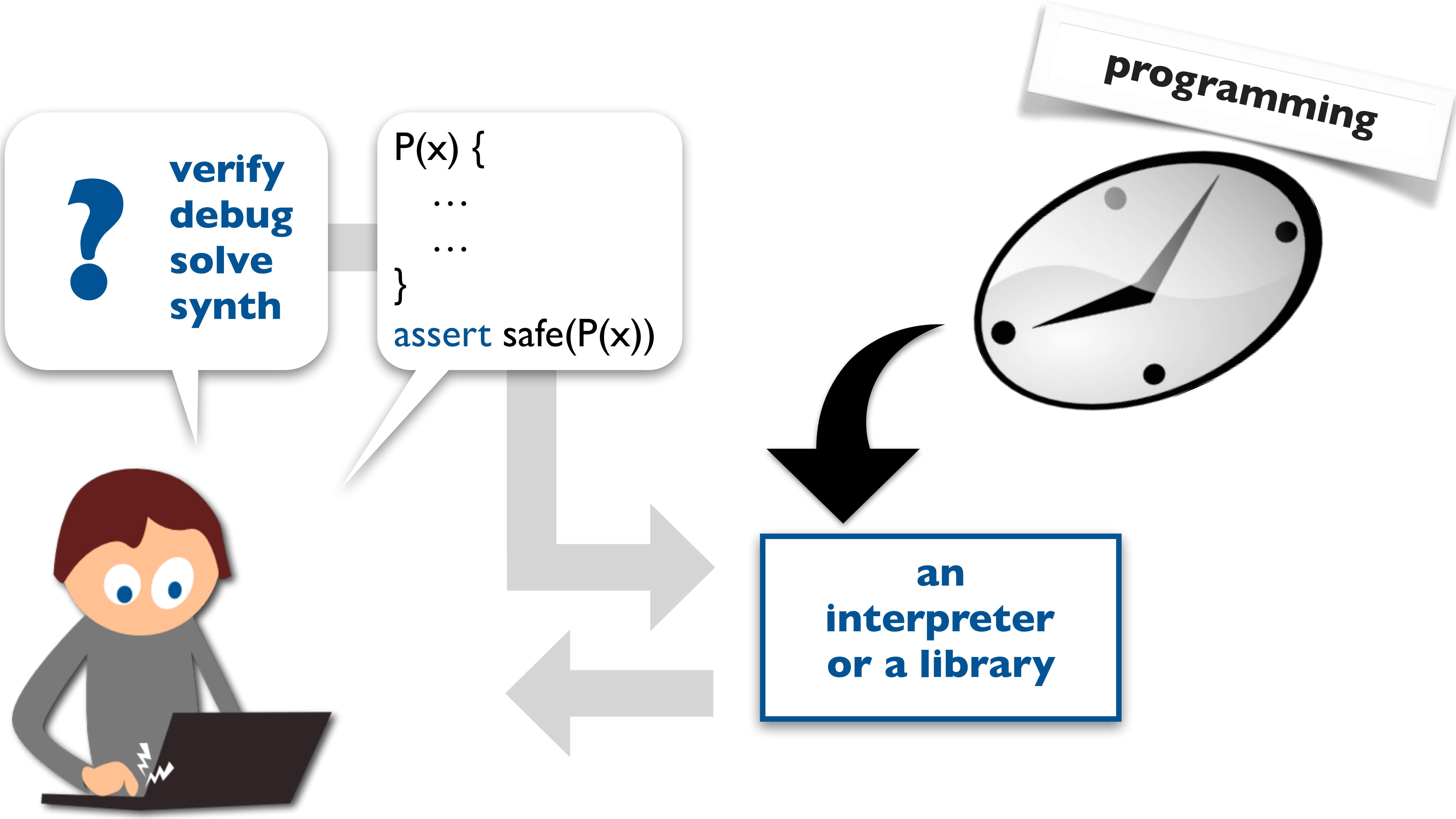


Sketch [Solar-Lezama et al., ASPLOS'06]
Comfusy [Kuncak et al., CAV'10]

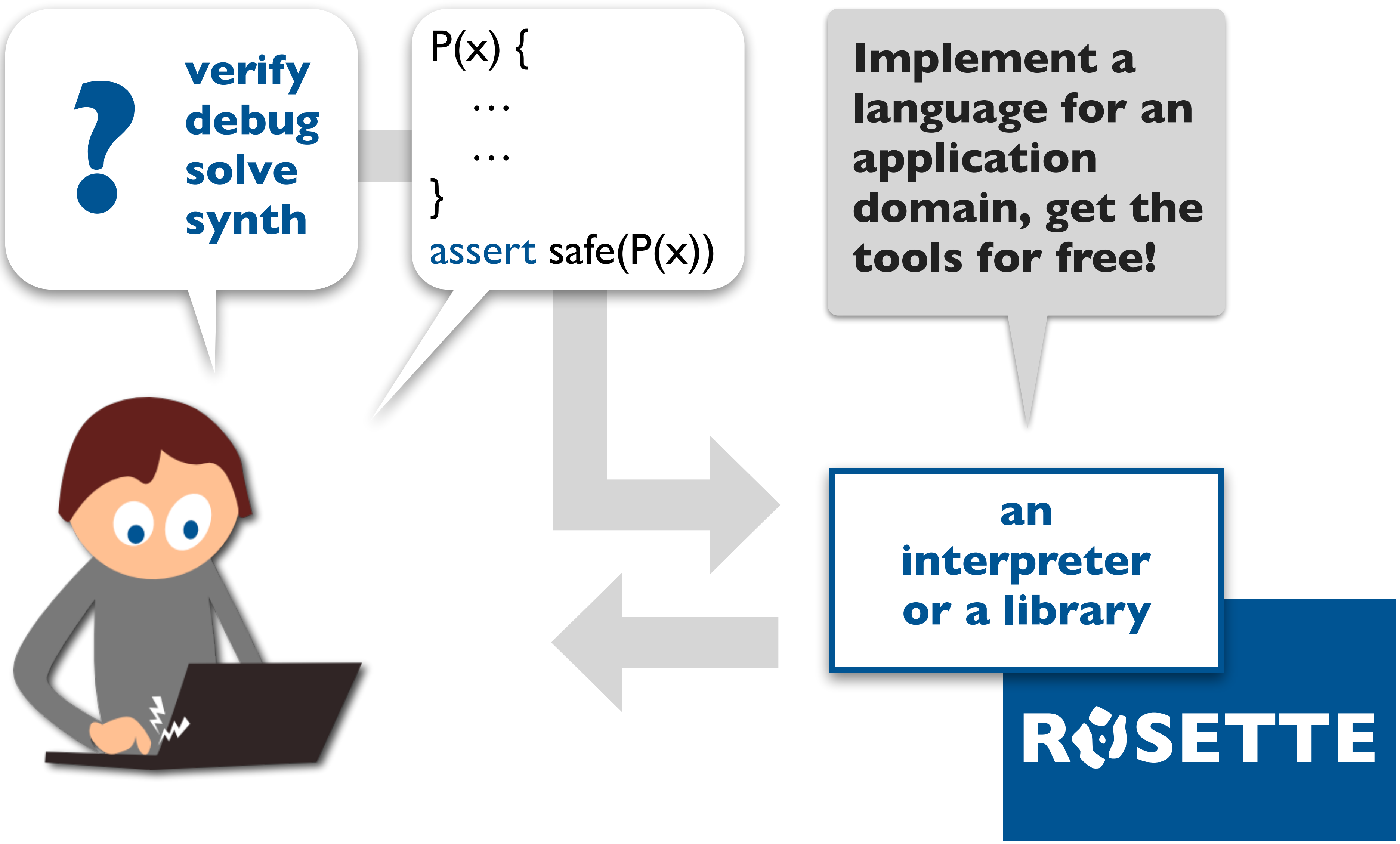
The standard (hard) way to build a tool



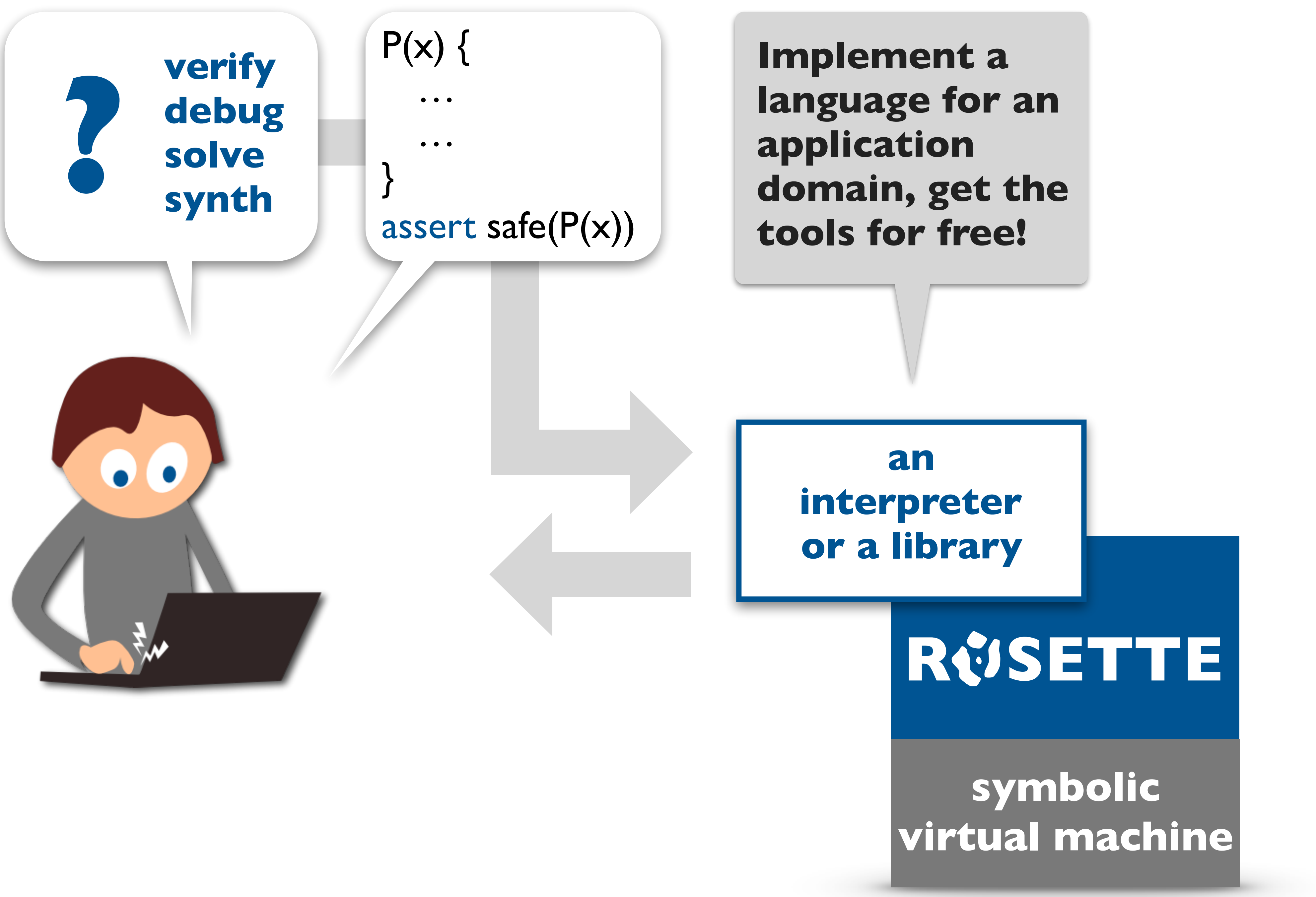
A new, easy way to build tools



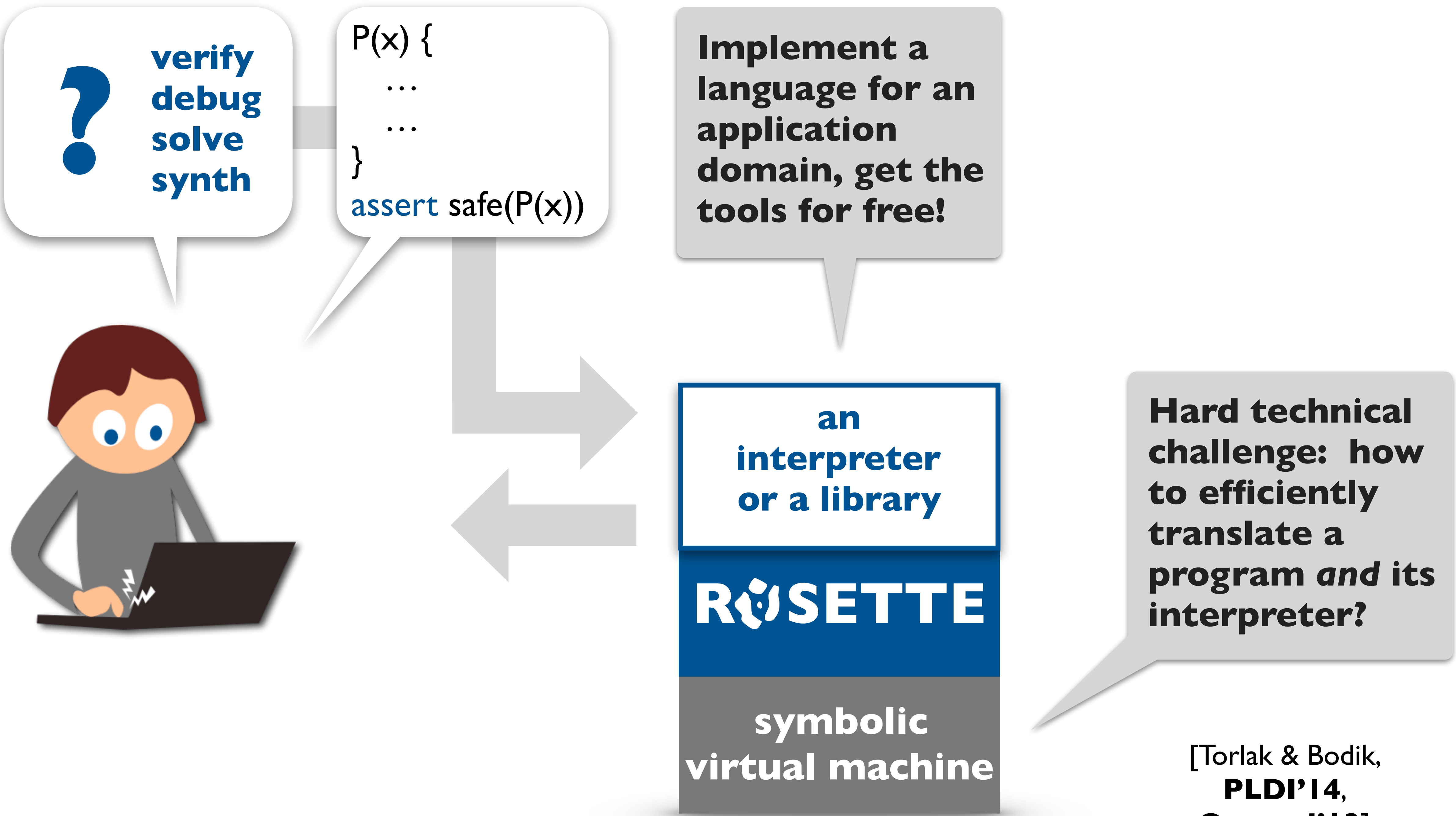
A new, easy way to build tools



A new, easy way to build tools



A new, easy way to build tools



[Torlak & Bodik, **PLDI'14**, **Onward'13**] design

solver-aided languages

Layers of languages

domain-specific language (DSL)

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

library ↓ interpreter

host language

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

Layers of languages

domain-specific language (DSL)

library | interpreter

host language

artificial intelligence
Church, BLOG

databases
SQL, Datalog

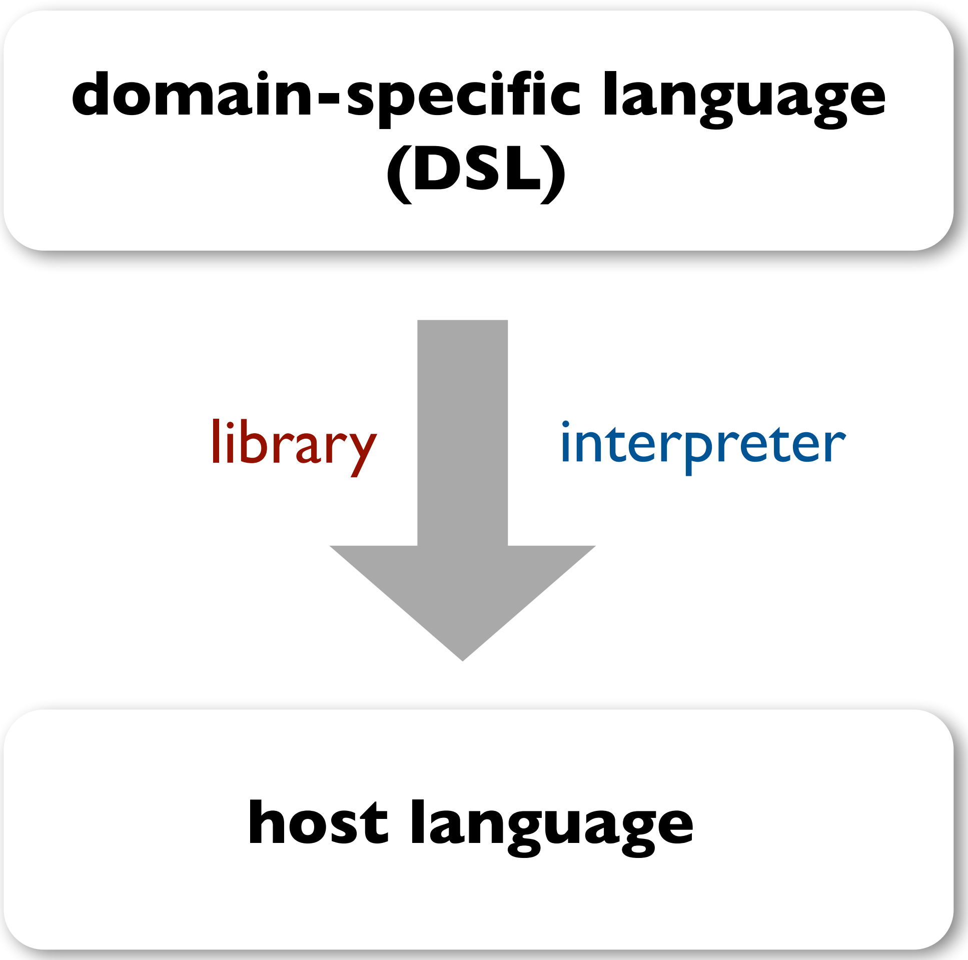
hardware design
Bluespec, Chisel, Verilog, VHDL

math and statistics
Eigen, Matlab, R

layout and visualization
LaTex, dot, dygraphs, D3

Scala, Racket, JavaScript

Layers of languages



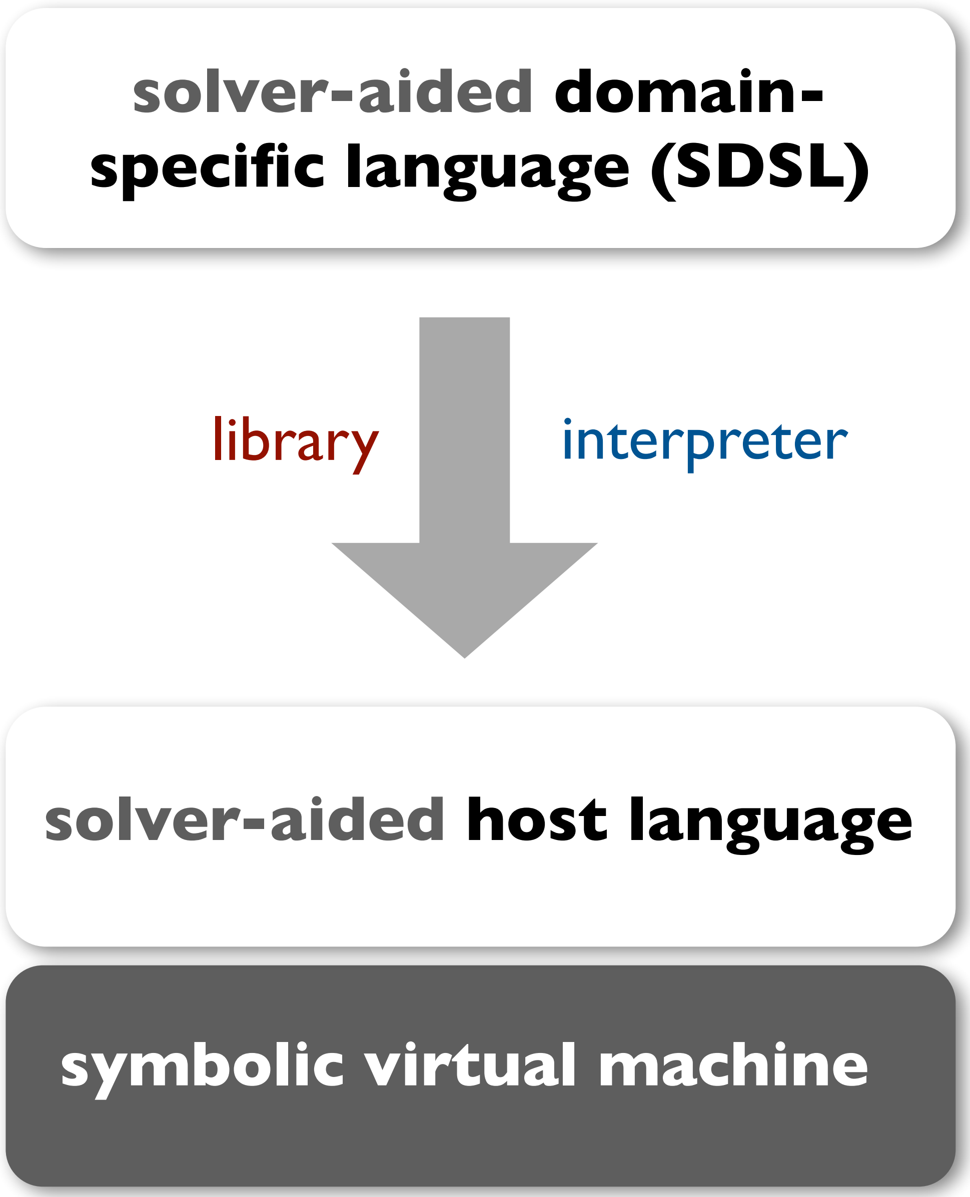
Eigen / Matlab

```
C = A * B        [associativity]
```

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



Layers of solver-aided languages

solver-aided domain-specific language (SDSL)

library | interpreter

solver-aided host language

symbolic virtual machine

ROSETTE

[Torlak & Bodik, **Onward'13**, **PLDI'14**]

Layers of solver-aided languages

solver-aided domain-specific language (SDSL)

library | interpreter

solver-aided host language

symbolic virtual machine

spatial programming
Chlorophyll

intelligent tutoring
RuleSynth

memory models
MemSynth

optimal synthesis
Synapse

radiotherapy controllers
Neutrons

BGP router configurations
BagPipe

ROSETTE

[Torlak & Bodik, **Onward'13**, **PLDI'14**]

Layers of solver-aided languages

solver-aided domain-specific language (SDSL)

library | interpreter

solver-aided host language

symbolic virtual machine

spatial programming
Chlorophyll

intelligent tutoring
RuleSynth

memory models
MemSynth

optimal synthesis
Synapse

radiotherapy controllers
Neutrons

BGP router configurations
BagPipe

ROSETTE

[Torlak & Bodik, **Onward'13**, **PLDI'14**]

Anatomy of a solver-aided host language

Modern descendent of Scheme with macro-based metaprogramming.

Racket

Anatomy of a solver-aided host language

```
(define-symbolic id type)

(assert expr)

(verify expr)
(debug [expr] expr)
(solve expr)
(synthesize [expr] expr)
```

ROSETTE

A tiny example SDSL

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

test debug
verify synth

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

A tiny example SDSL

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

test debug
verify synth

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 = bvge(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 = bvge(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 bvge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))
```

A tiny example SDSL: ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvge(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 bvge 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 = bvge(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 bvge 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 = bvge(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 bvge 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	

interpret

```
(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 = bvge(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 bvge 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 = bvge(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 bvge 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 = bvge(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 bvge 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	

interpret

```
(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 = bvge(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 bvge 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: ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvge(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 bvge 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

interpret

```
(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 = bvge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6
```

```
(define bvmax
 `((2 bvge 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

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

interpret

```
(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 = bvge(r0, r1)
  r3 = bvneg(r2)
  r4 = bvxor(r0, r2)
  r5 = bvand(r3, r4)
  r6 = bvxor(r1, r5)
  return r6

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

```
(define bvmax
 `((2 bvge 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 = bvge(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 n0 n1 integer?)
(define inputs (list n0 n1))
(verify
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

A tiny example SDSL: ROSETTE

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

> verify(bvmax, max)
```

query

Creates two fresh symbolic constants of type number and binds them to variables n0 and n1.

```
(define-symbolic n0 n1 integer?)
(define inputs (list n0 n1))
(verify
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

A tiny example SDSL: ROSETTE

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

> verify(bvmax, max)
```

query

Symbolic values can be used just like concrete values of the same type.

```
(define-symbolic n0 n1 integer?)
(define inputs (list n0 n1))
(verify
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

A tiny example SDSL: ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvge(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 n0 n1 integer?)
(define inputs (list n0 n1))
(verify
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

(verify *expr*) searches for a concrete interpretation of symbolic constants that causes *expr* to fail.

A tiny example SDSL: ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvge(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 n0 n1 integer?)
(define inputs (list n0 n1))
(verify
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

A tiny example SDSL: ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvge(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 inputs (list 0 -2))
(debug [input-register?]
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

A tiny example SDSL: ROSETTE

```
def bvmax(r0, r1) :
  r2 = bvge(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 inputs (list 0 -2))
(debug [input-register?]
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

A tiny example SDSL: ROSETTE

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

> synthesize(bvmax, max)
```

query

```
(define-symbolic n0 n1 integer?)
(define inputs (list n0 n1))
(synthesize [inputs]
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

A tiny example SDSL: ROSETTE

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

> synthesize(bvmax, max)
```

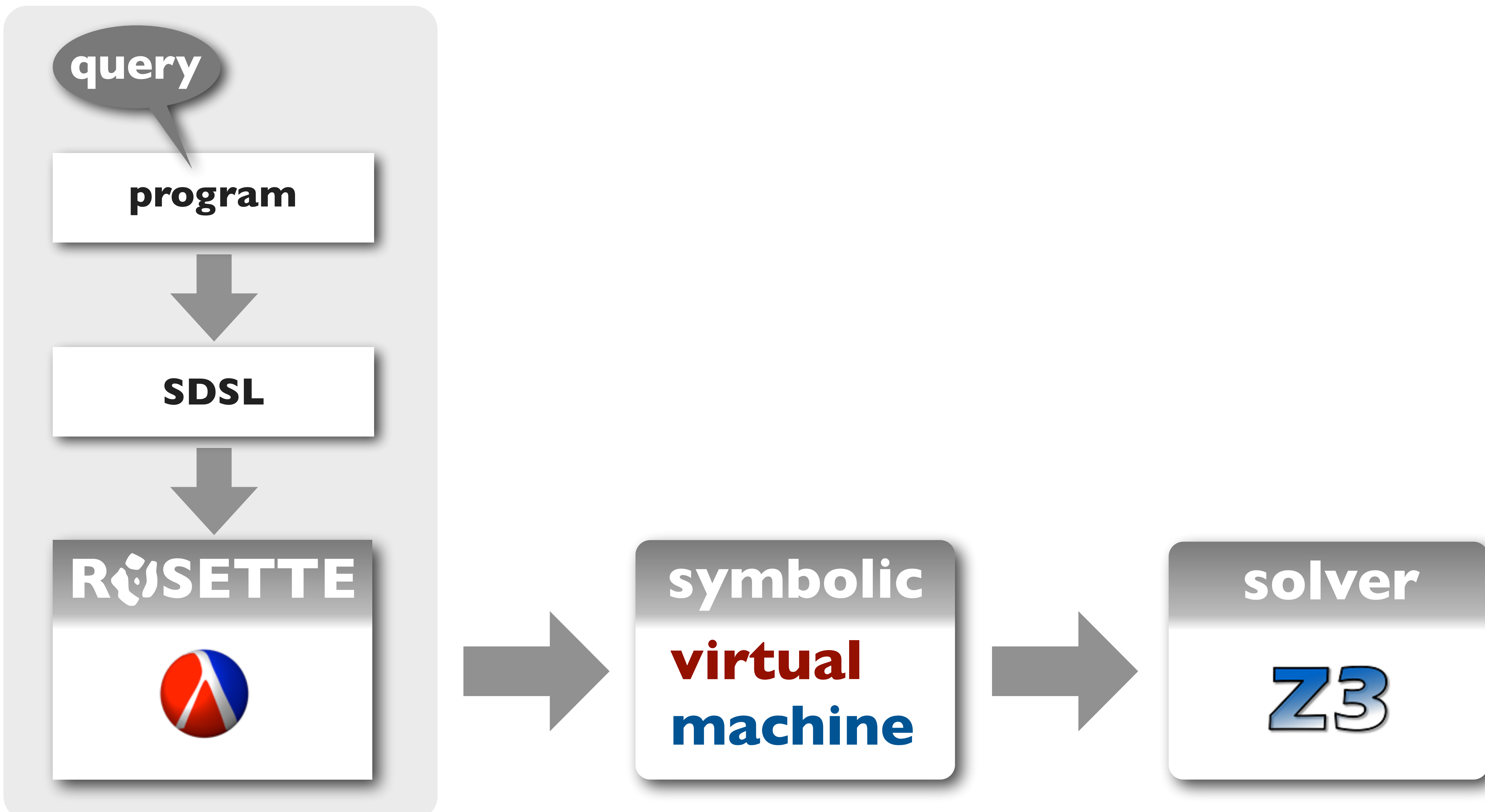
query

```
(define-symbolic n0 n1 integer?)
(define inputs (list n0 n1))
(synthesize [inputs]
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```

tech

symbolic virtual machine (SVM)

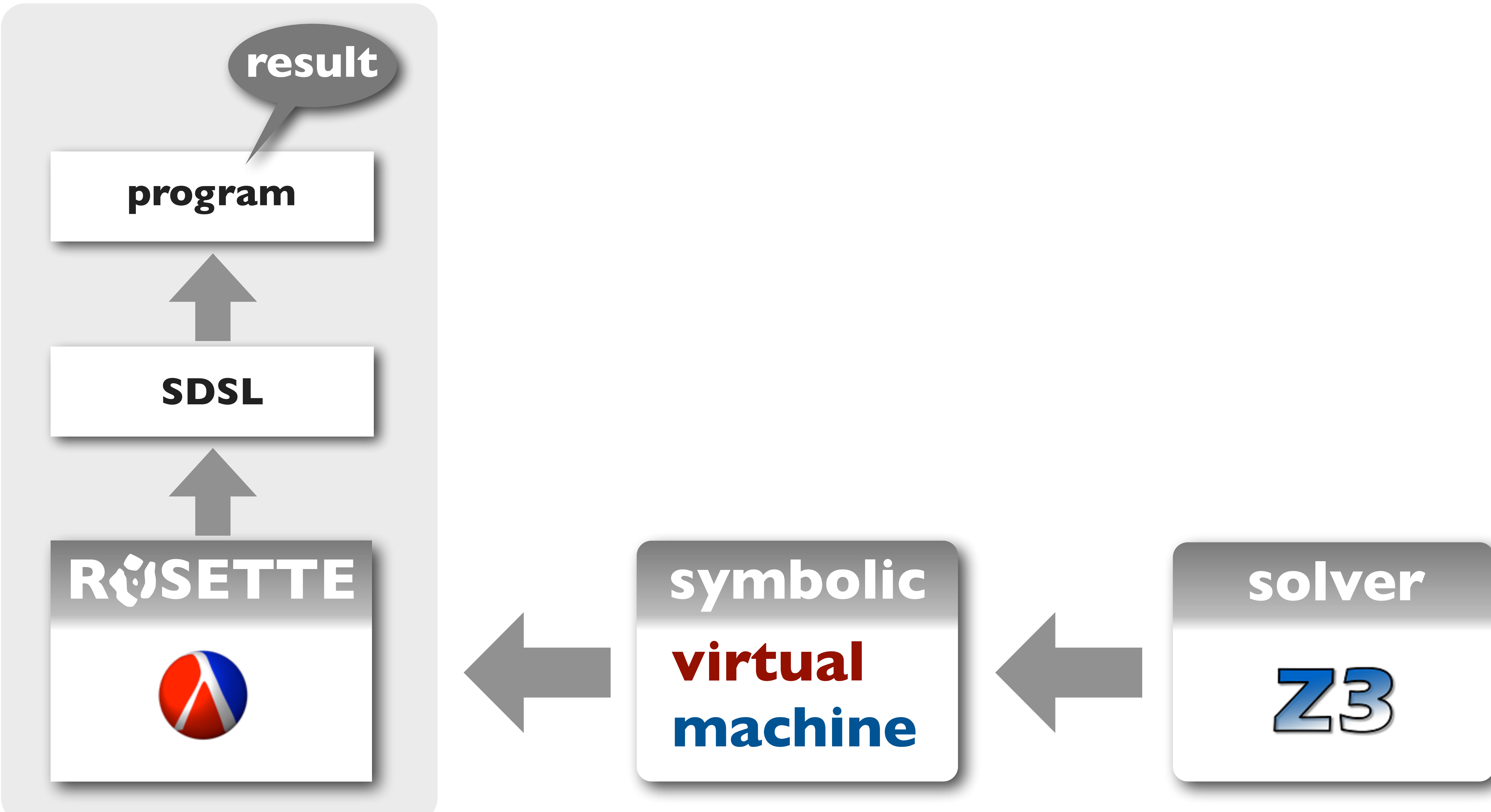
How it all works: a big picture view



[Torlak & Bodik, **Onward'13**]

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

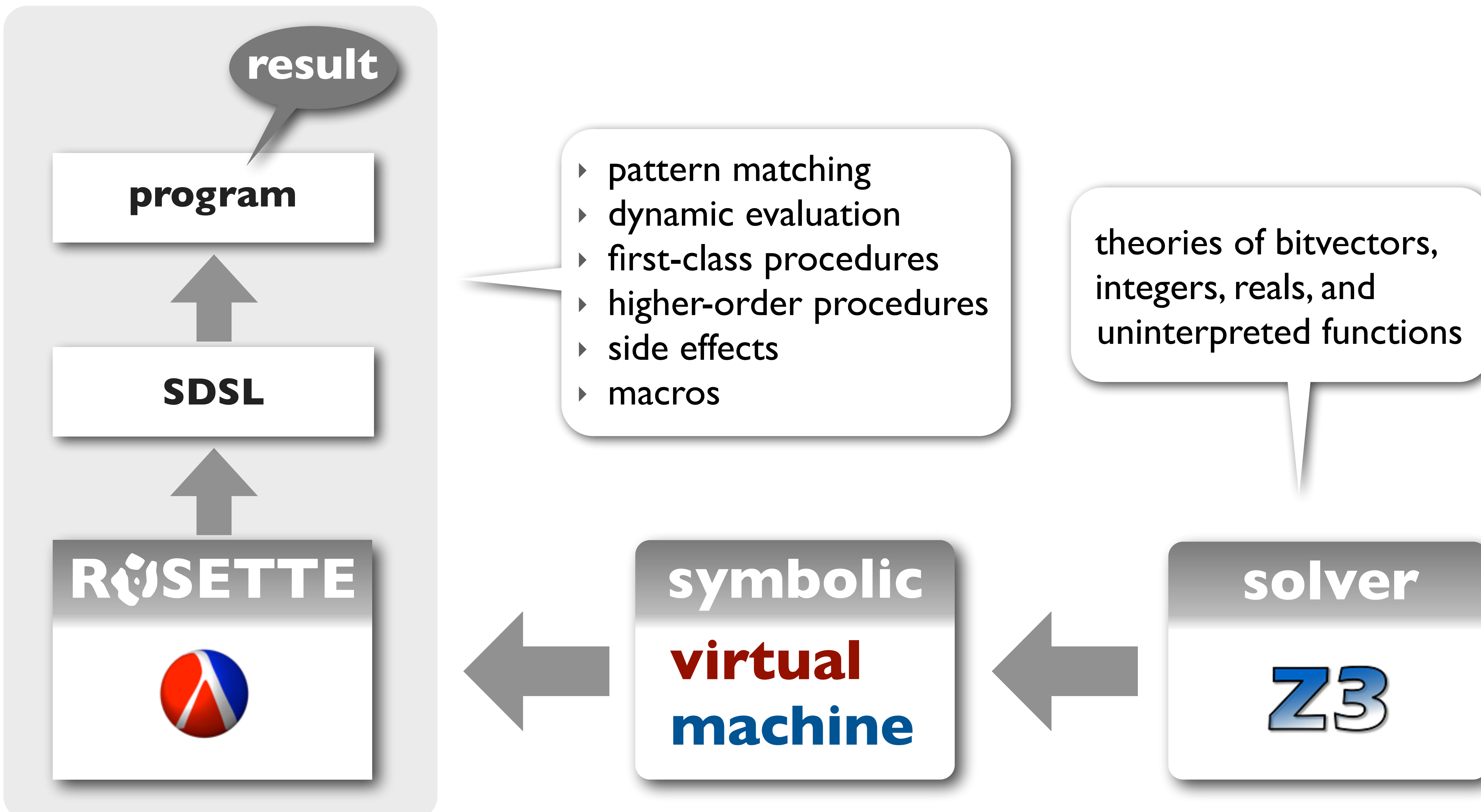
How it all works: a big picture view



[Torlak & Bodik, **Onward'13**]

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

How it all works: a big picture view



[Torlak & Bodik, **Onward'13**]

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

Translation to constraints by example

vs

`(3, 1, -2)`

reverse and filter, keeping only positive numbers

ps

`(1, 3)`

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)

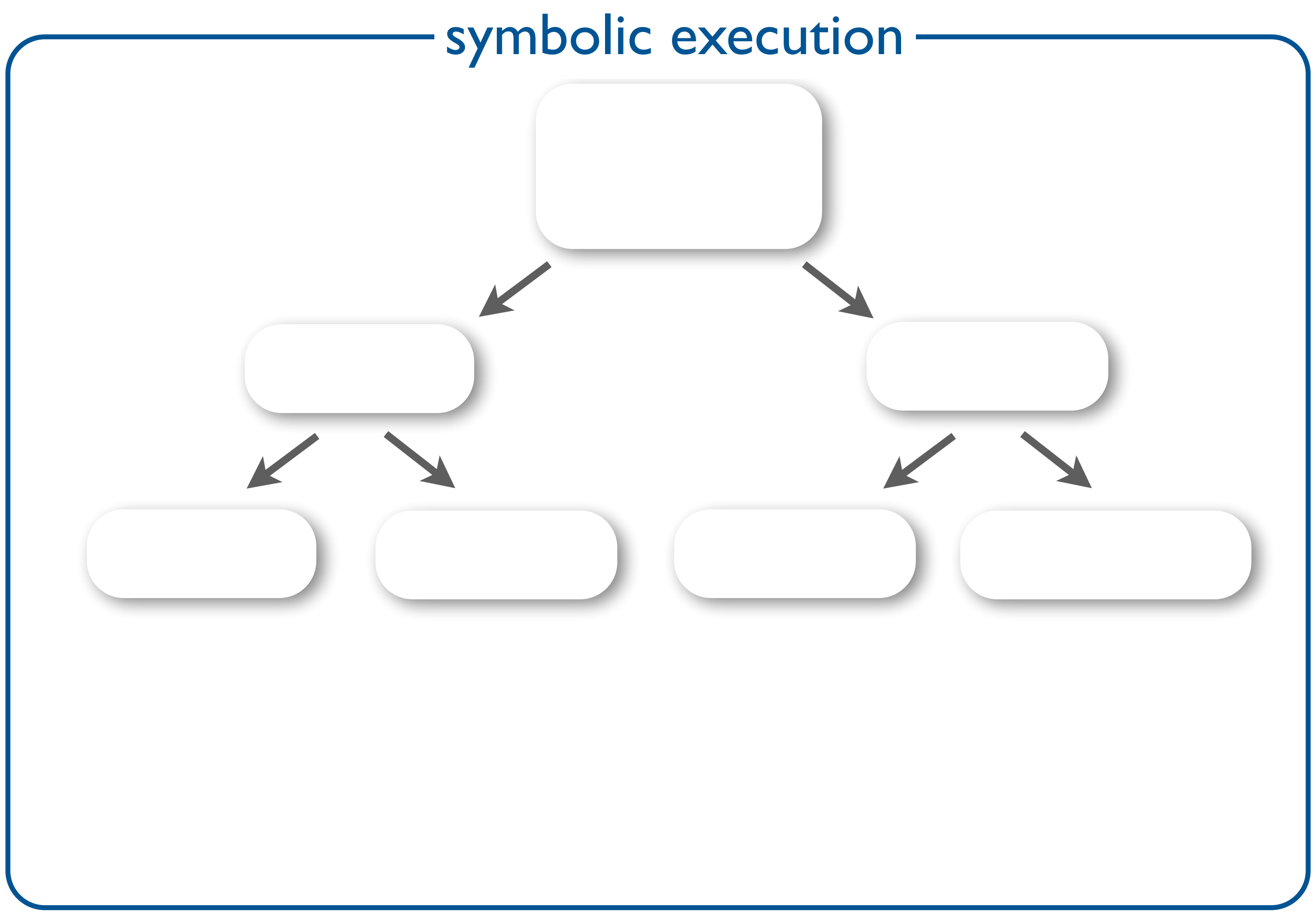
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

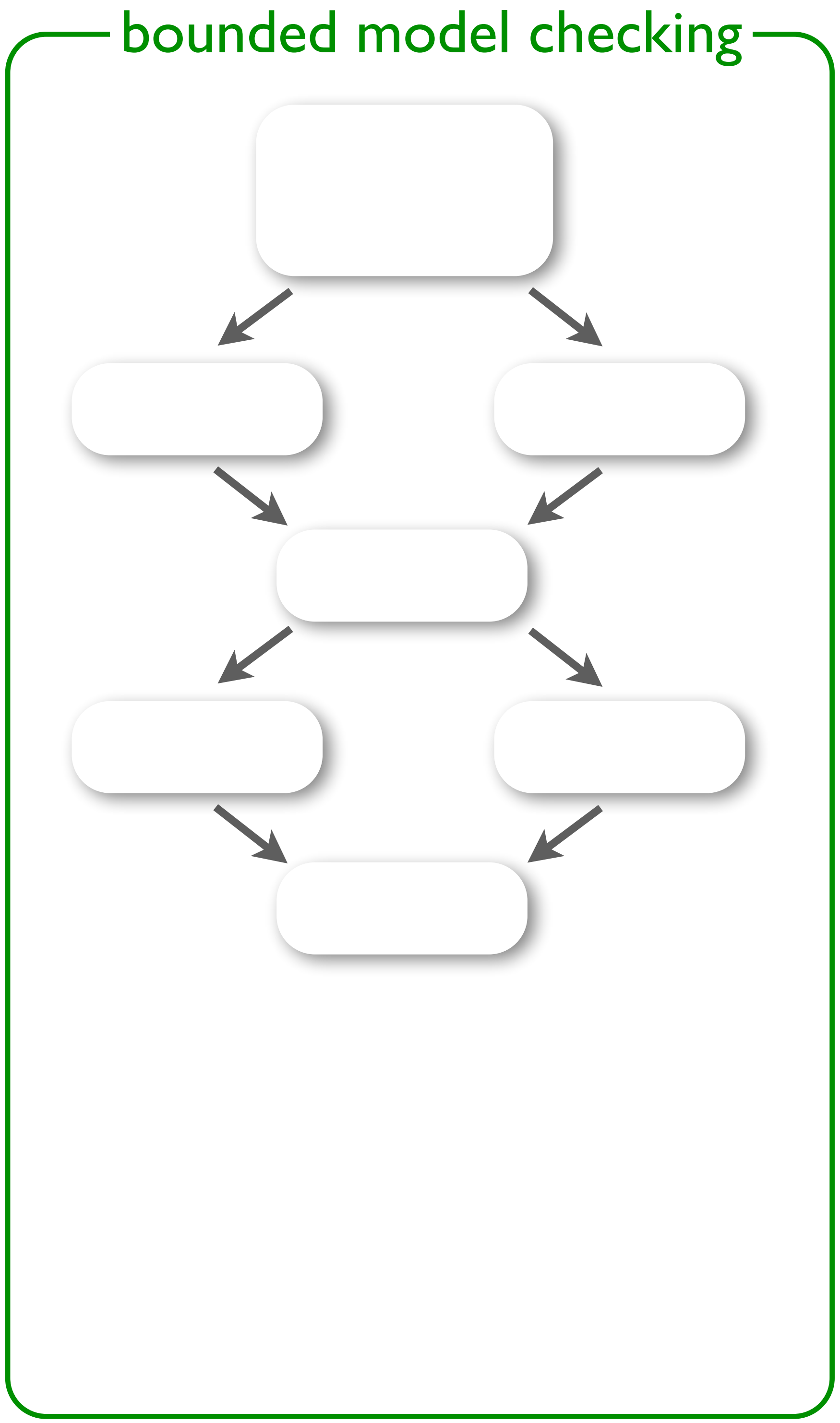
constraints

a>0 ∧ b>0

Design space of precise symbolic encodings

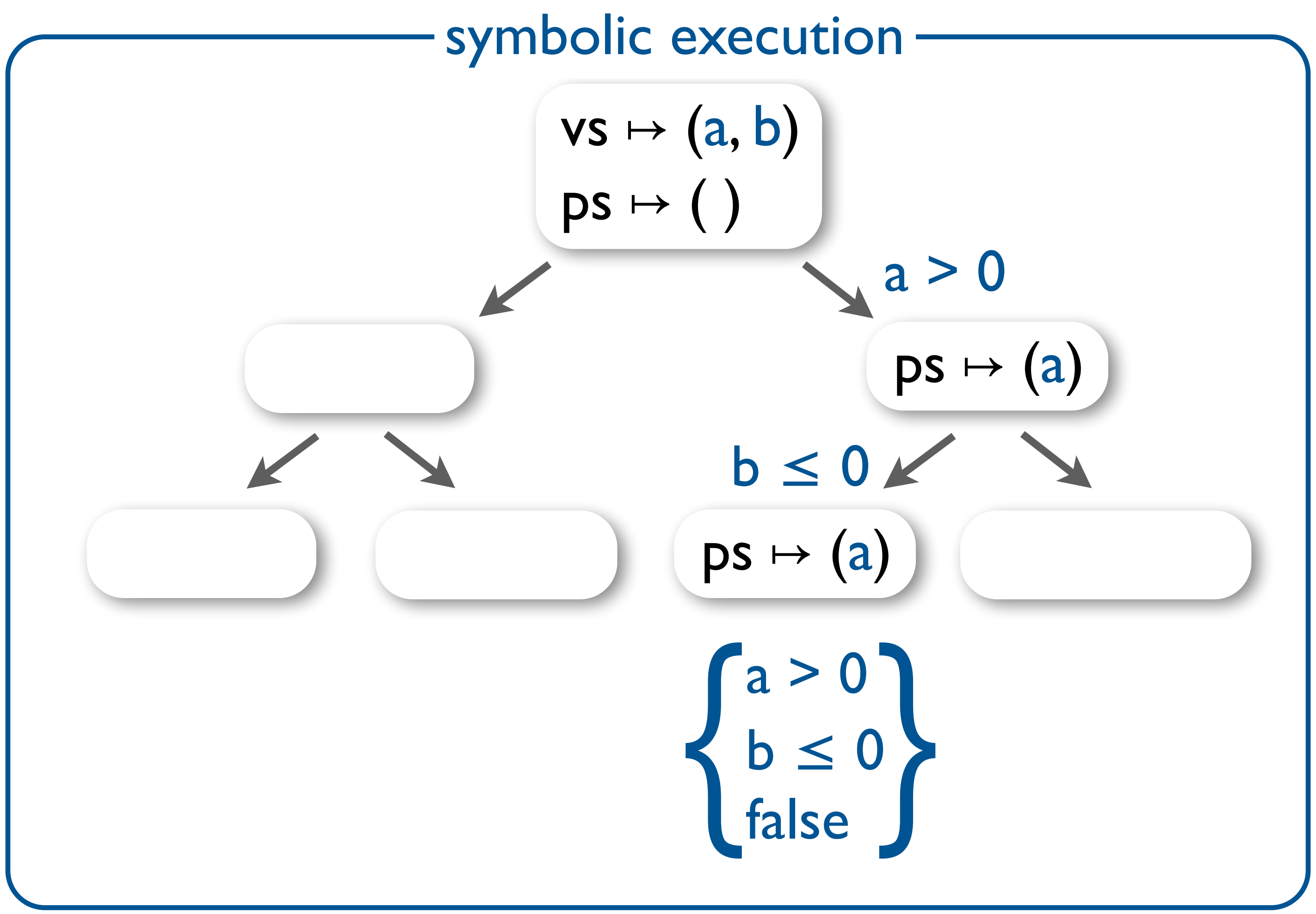
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

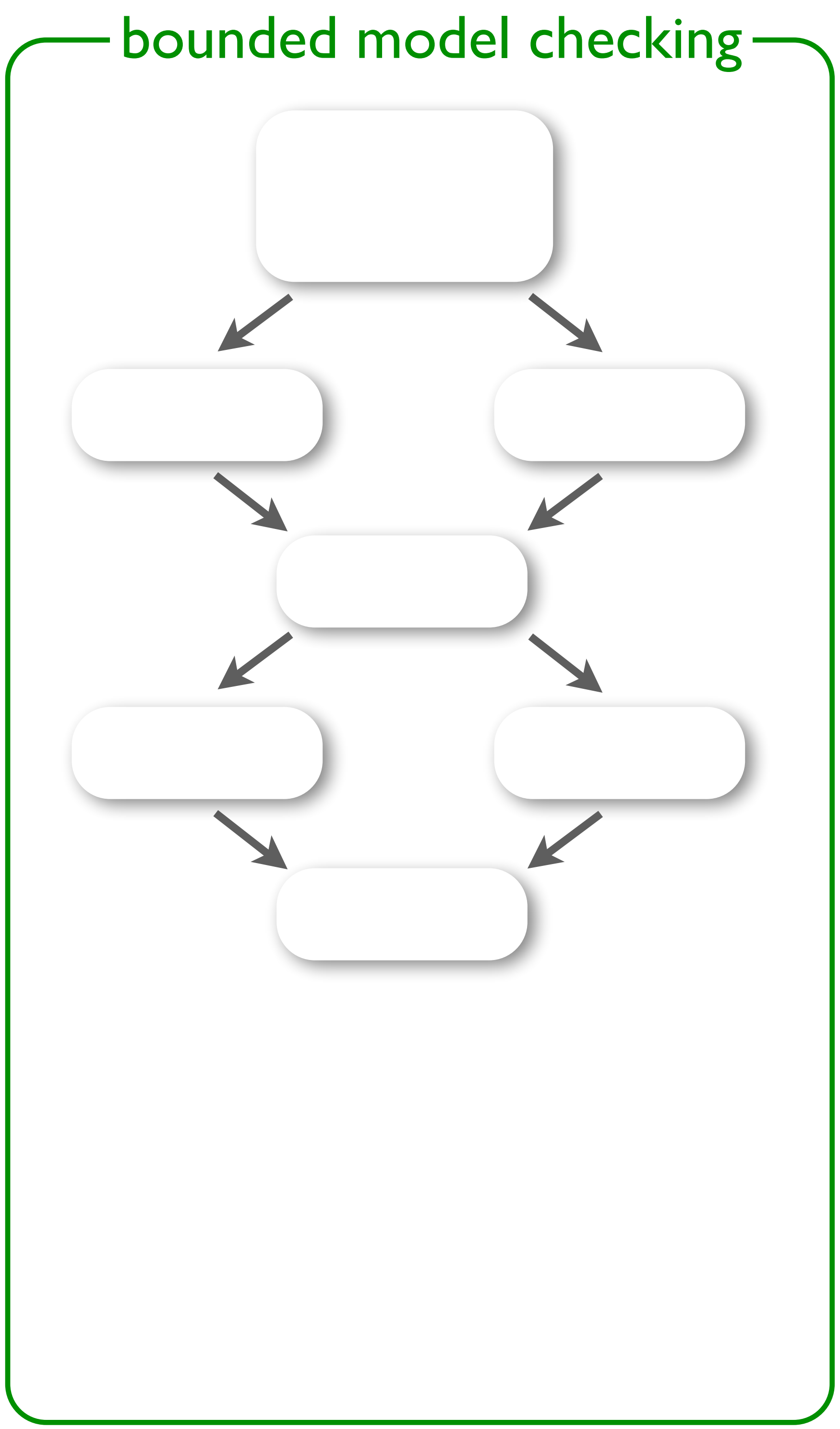




Design space of precise symbolic encodings

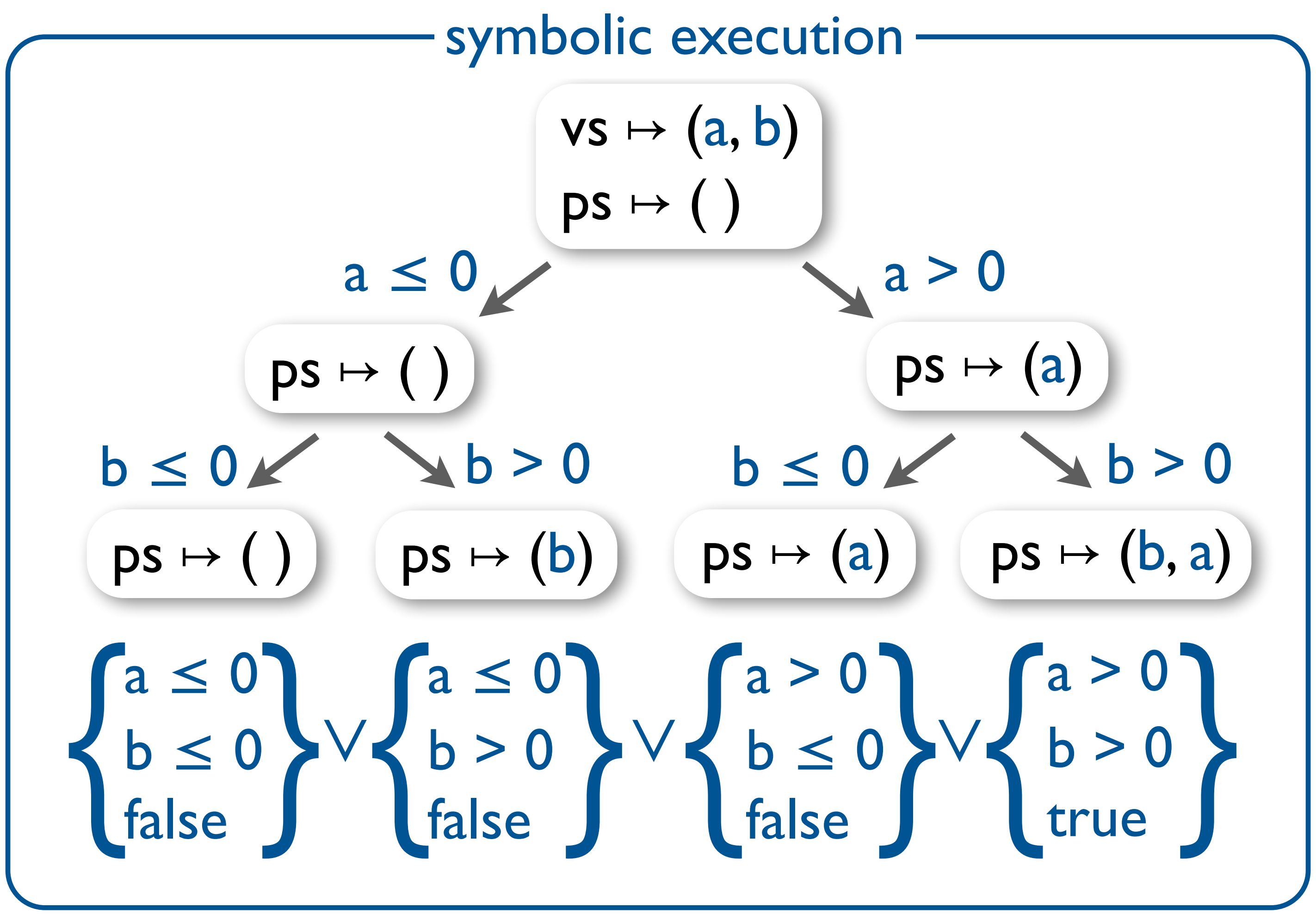
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

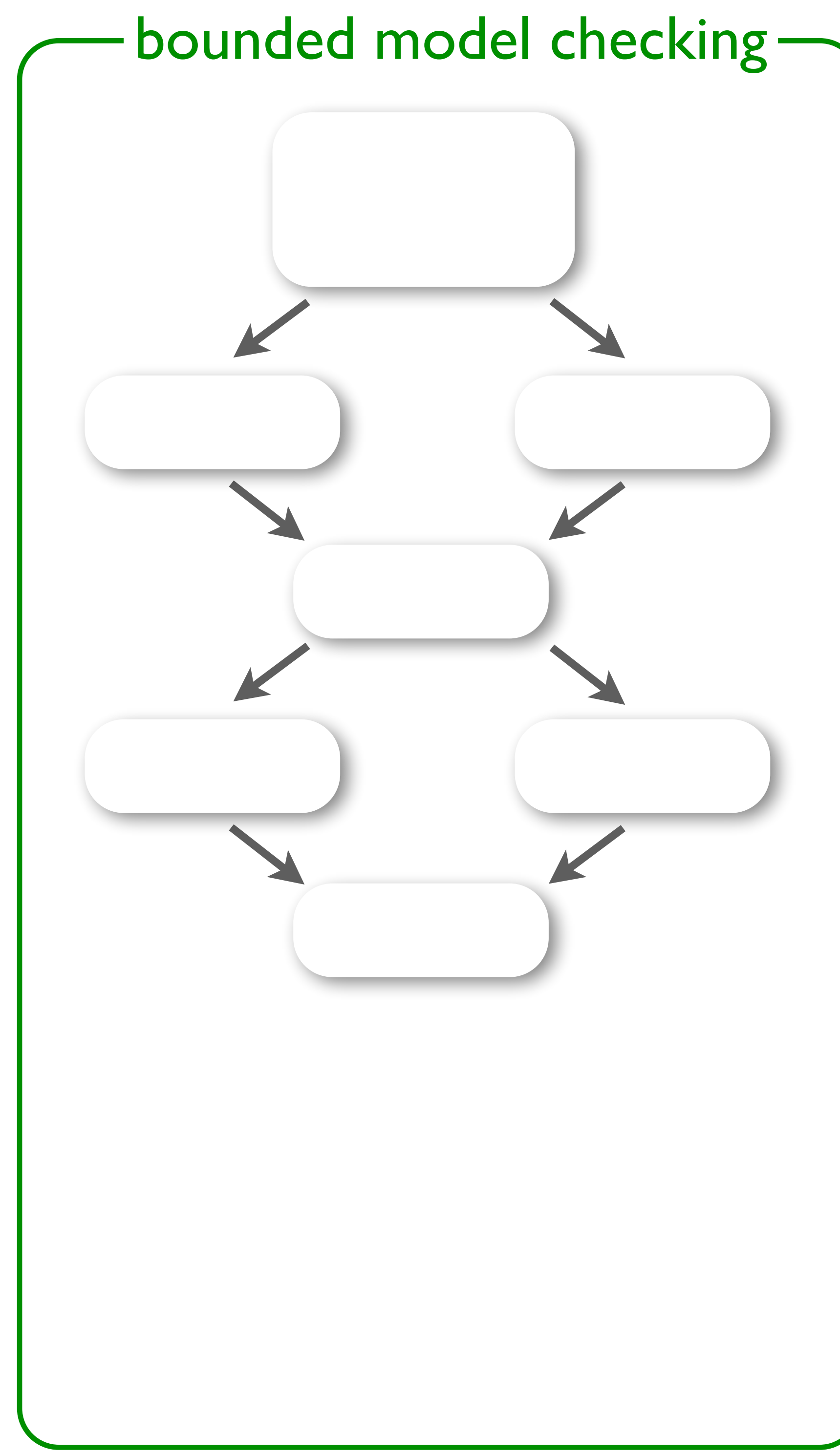




Design space of precise symbolic encodings

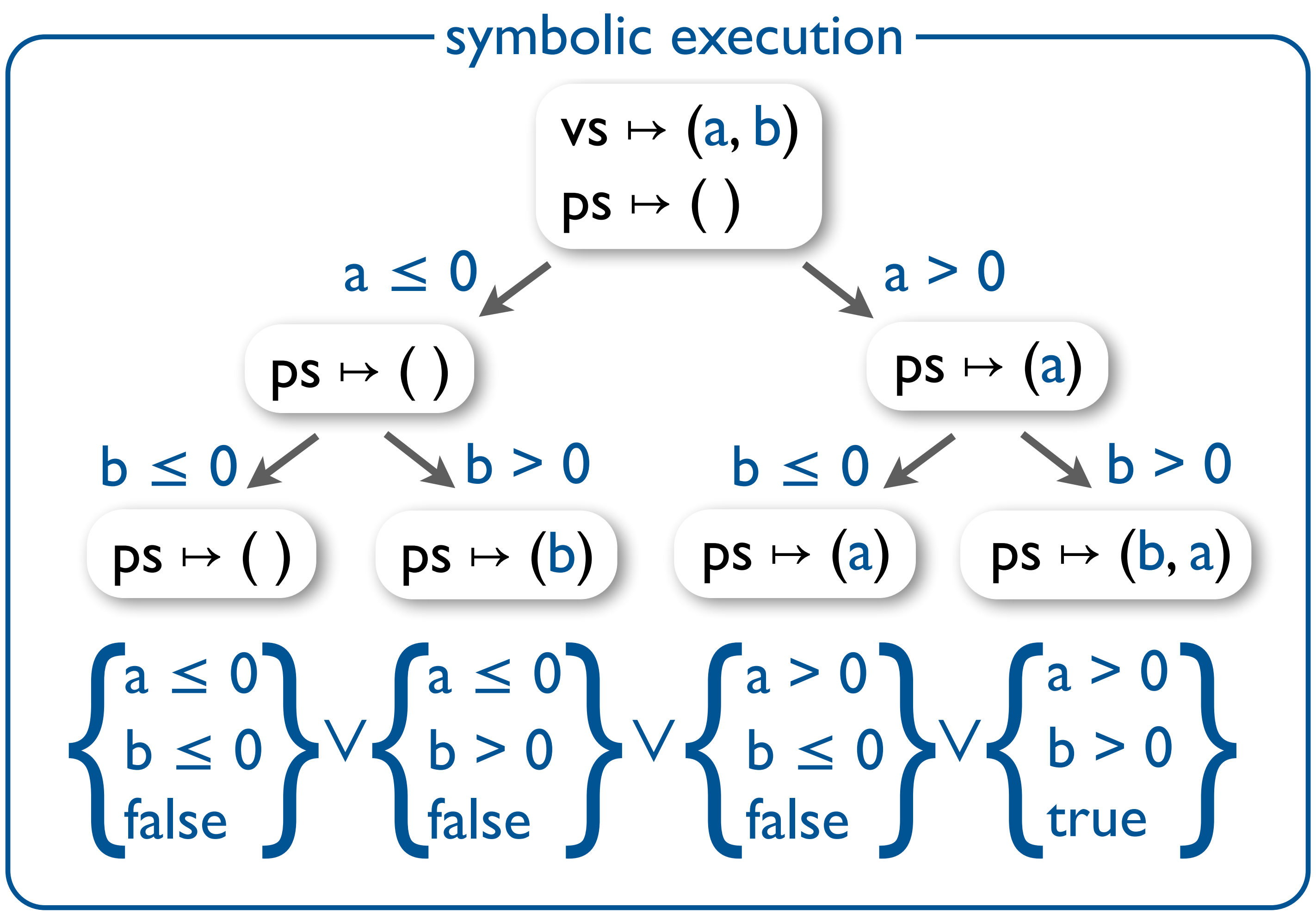
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

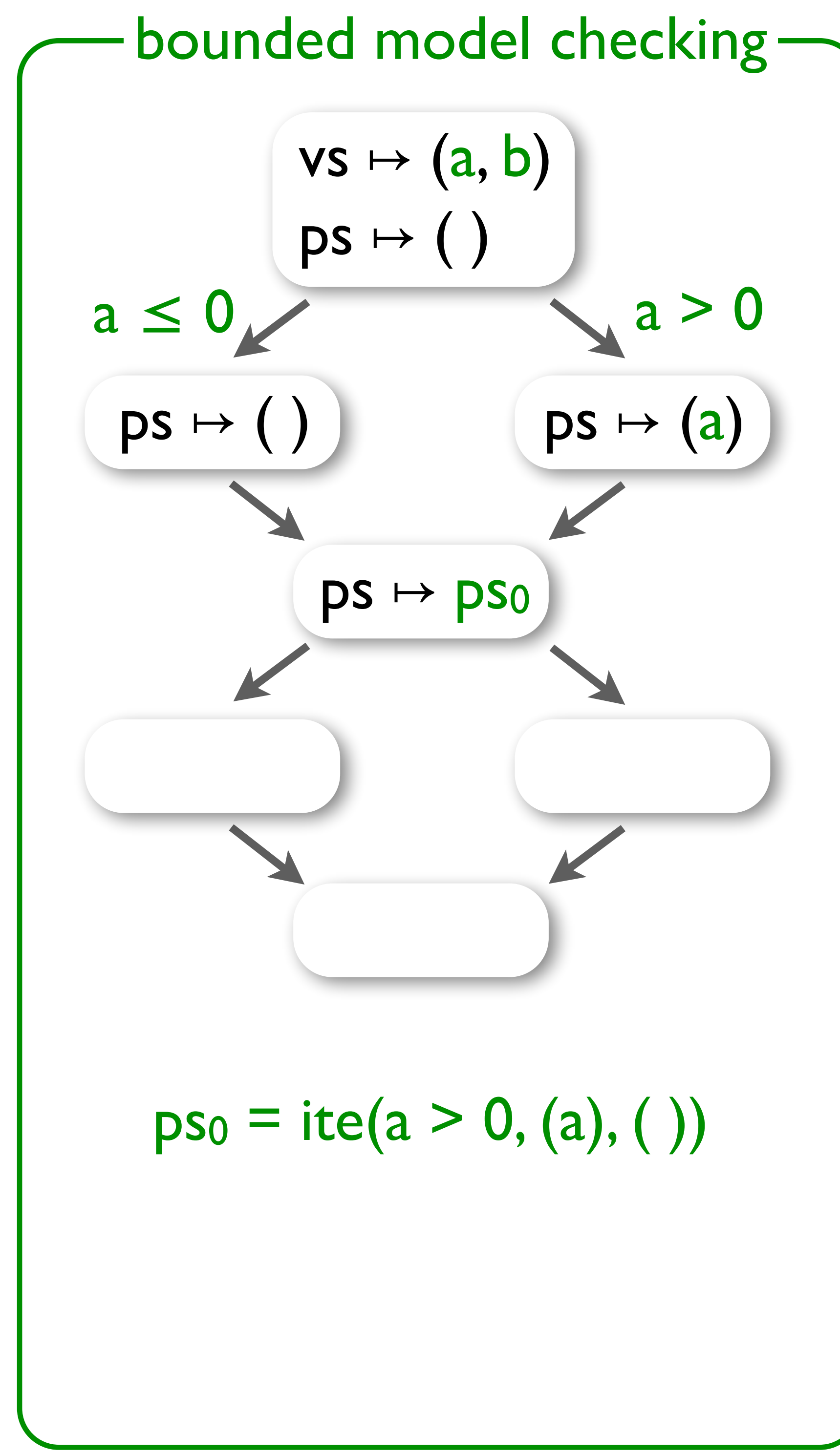




Design space of precise symbolic encodings

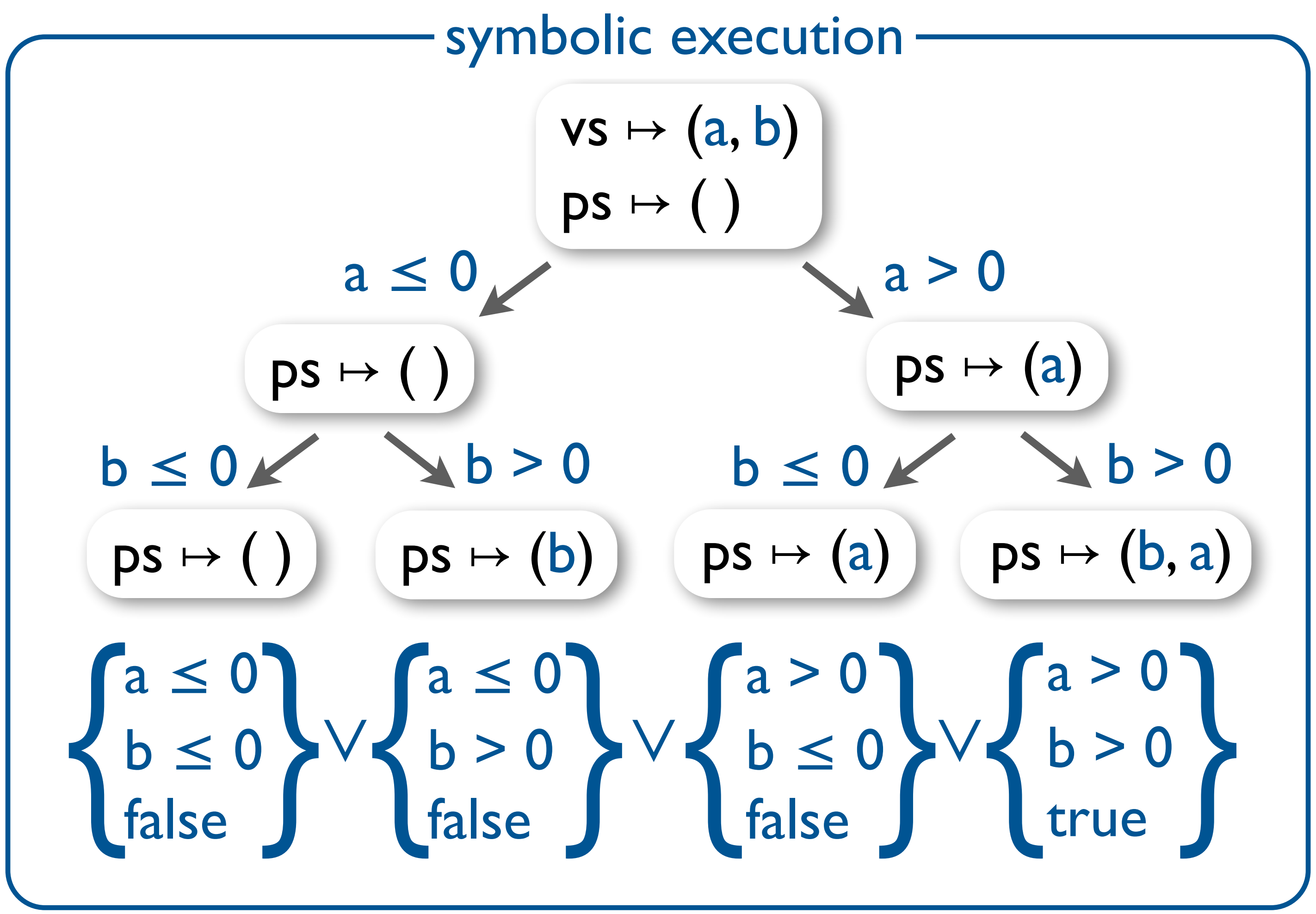
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

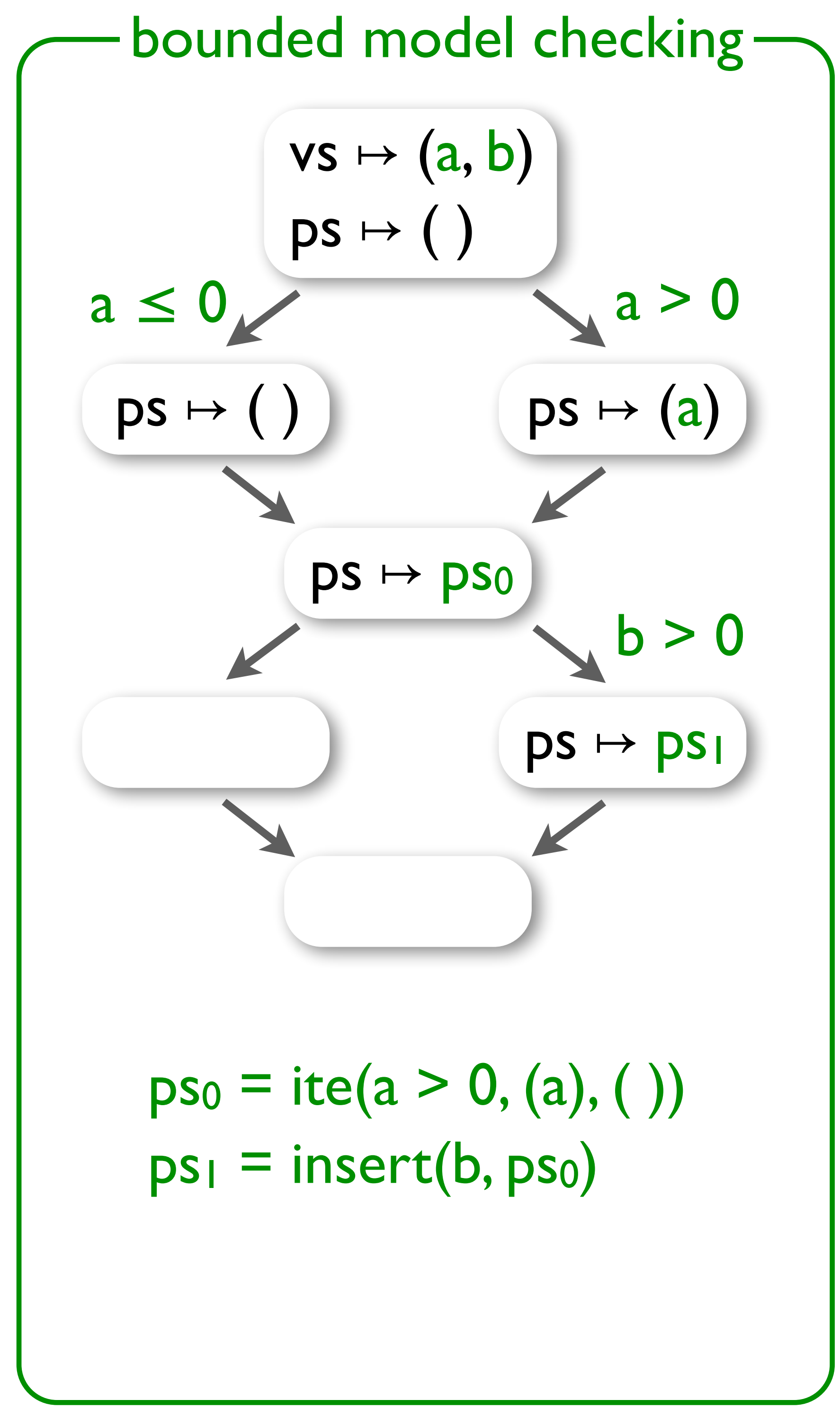




Design space of precise symbolic encodings

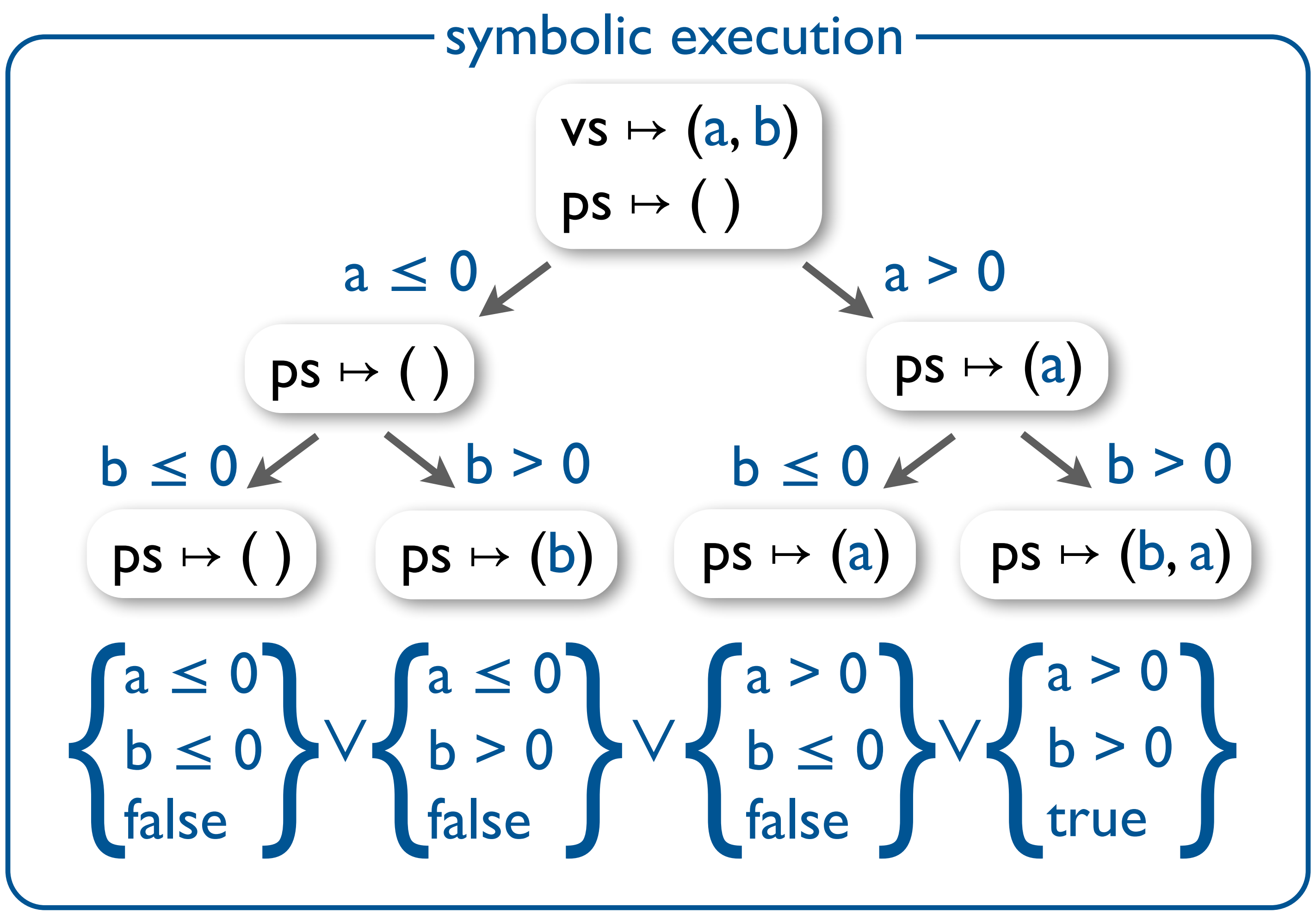
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

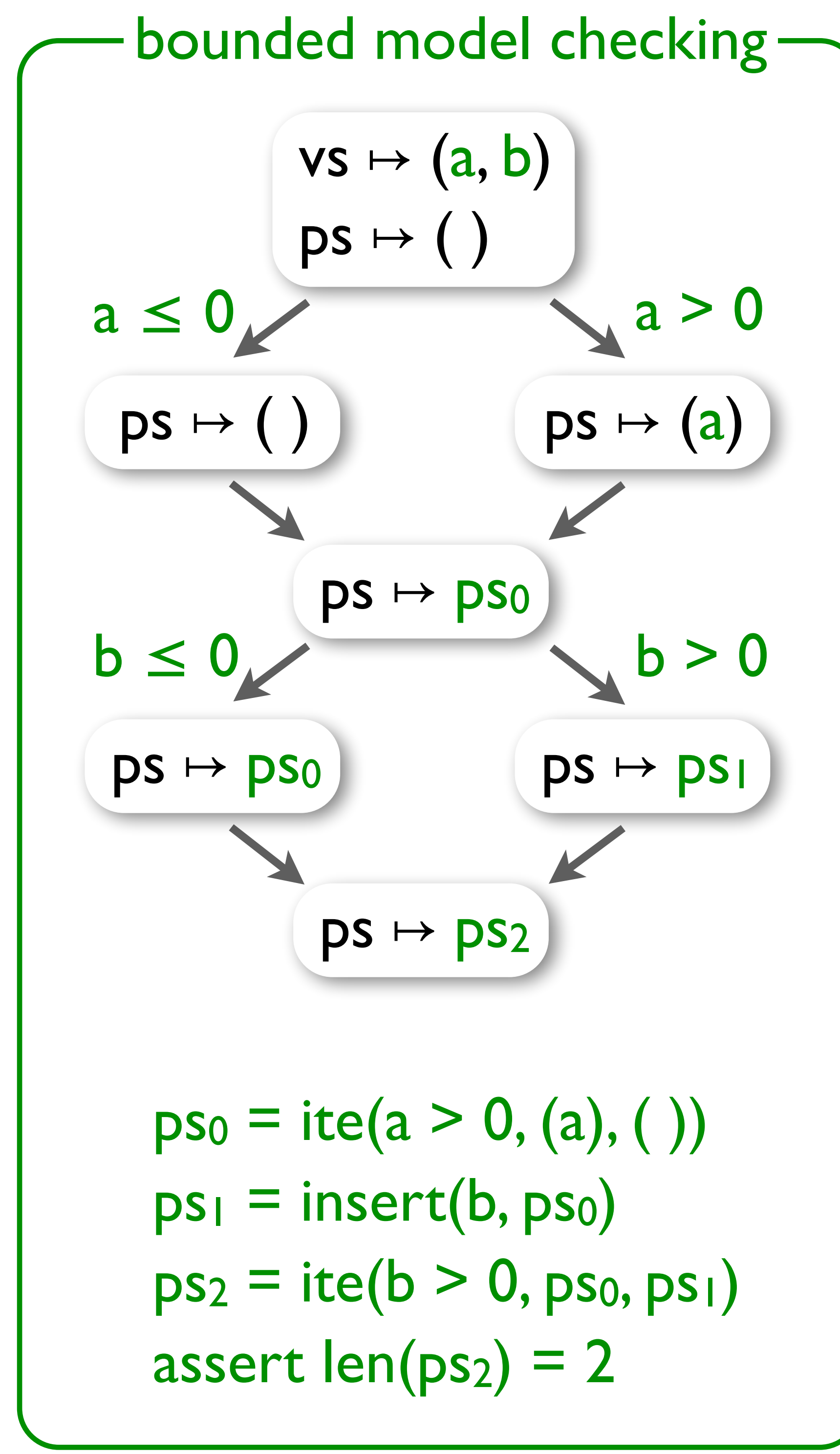




Design space of precise symbolic encodings

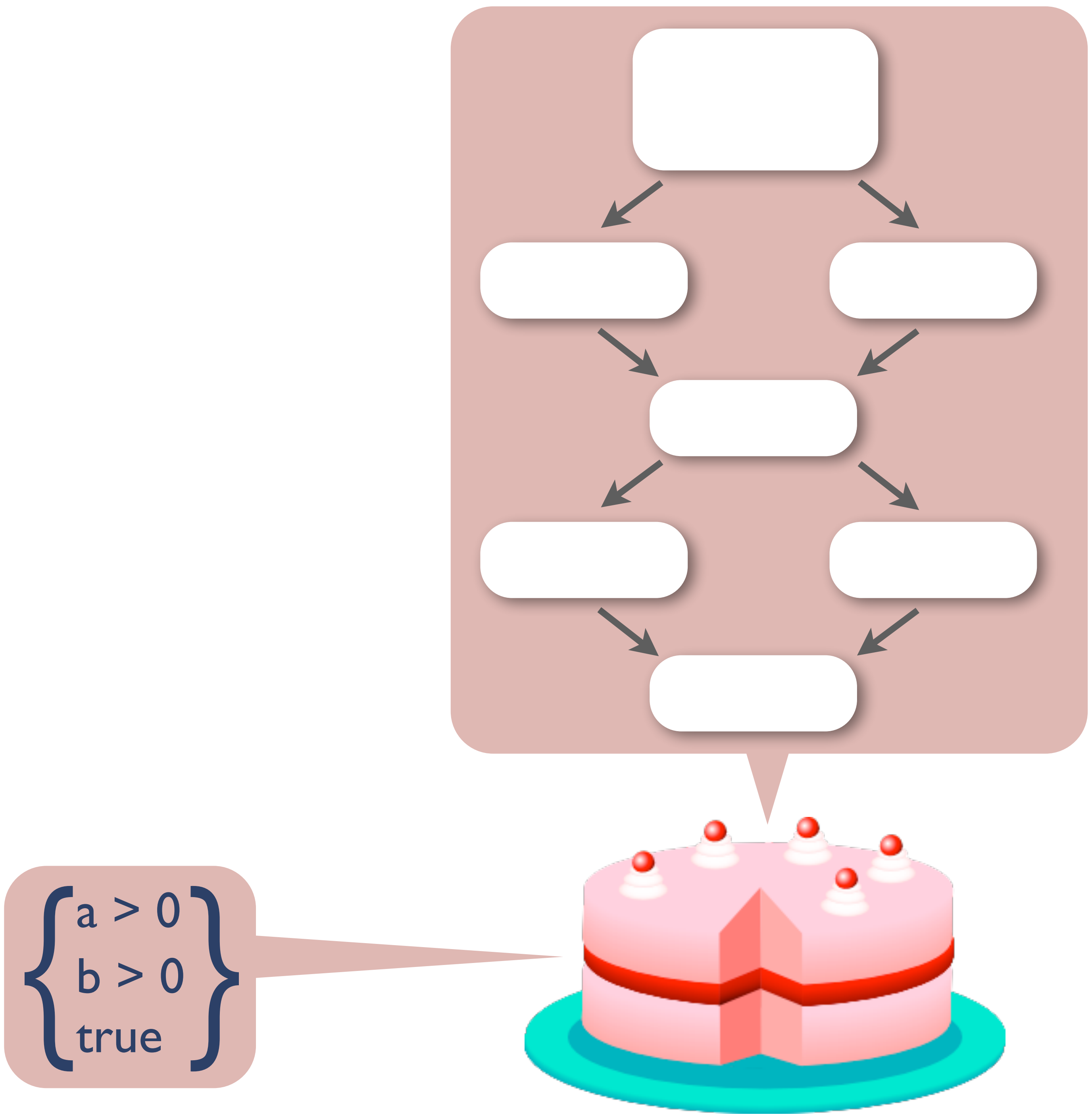
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```





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)
```

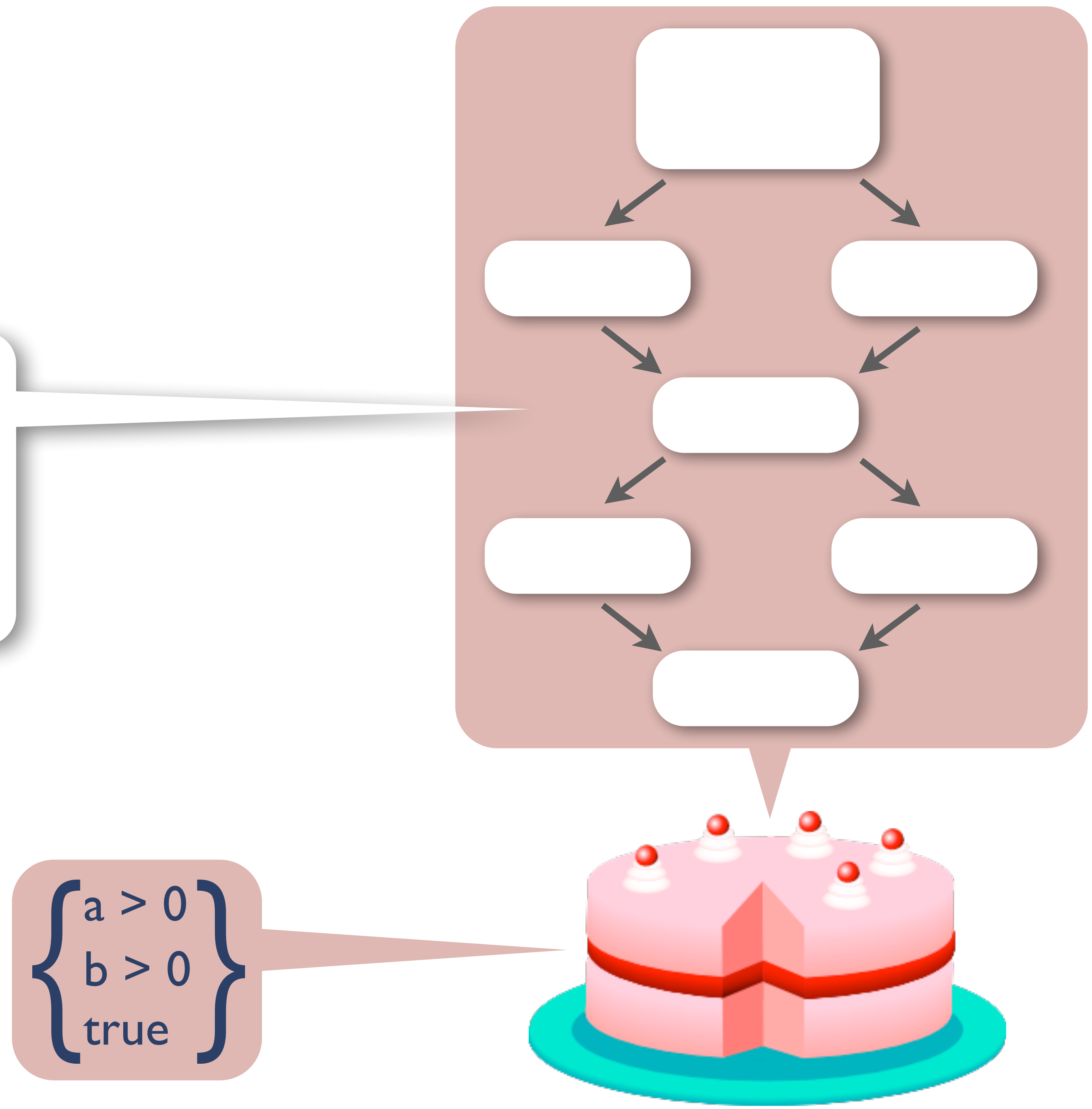


A new design: type-driven state merging

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

Merge values of

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

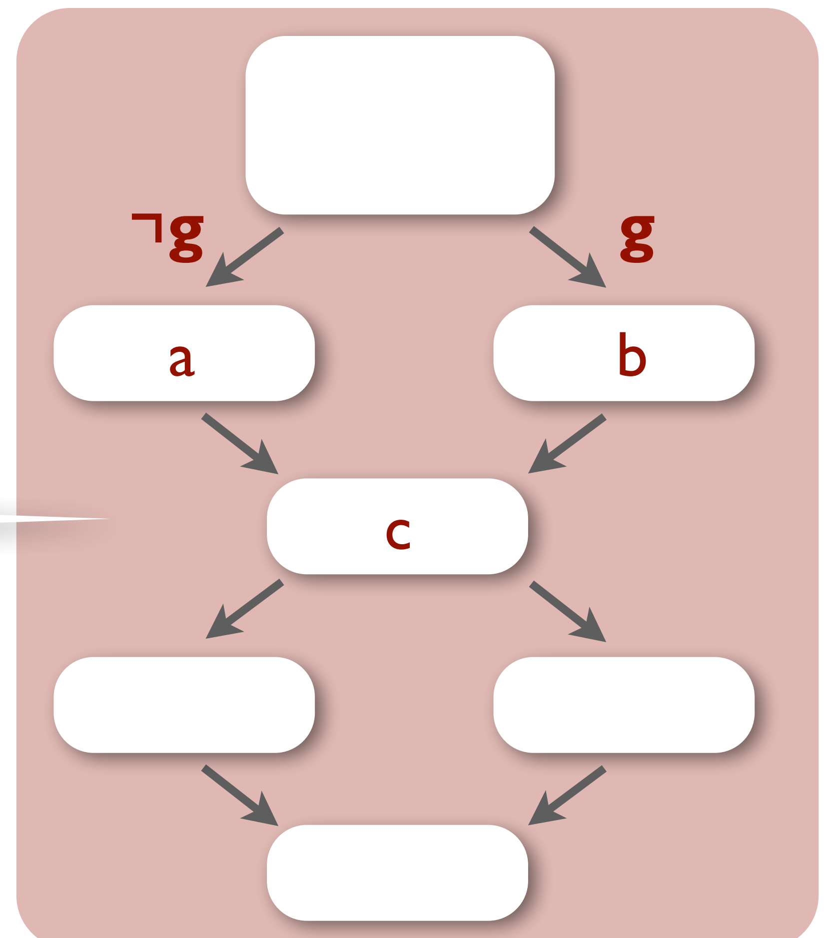


A new design: type-driven state merging

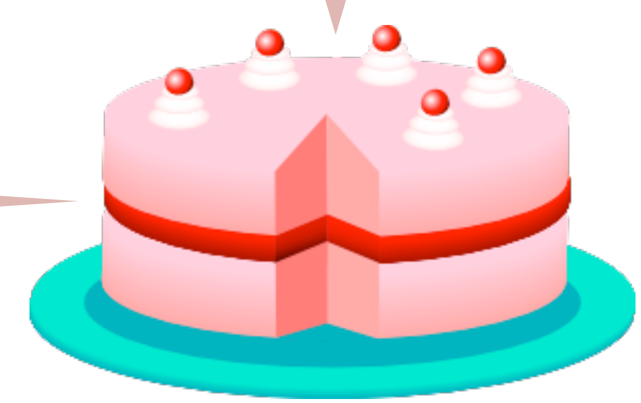
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

Merge values of

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



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

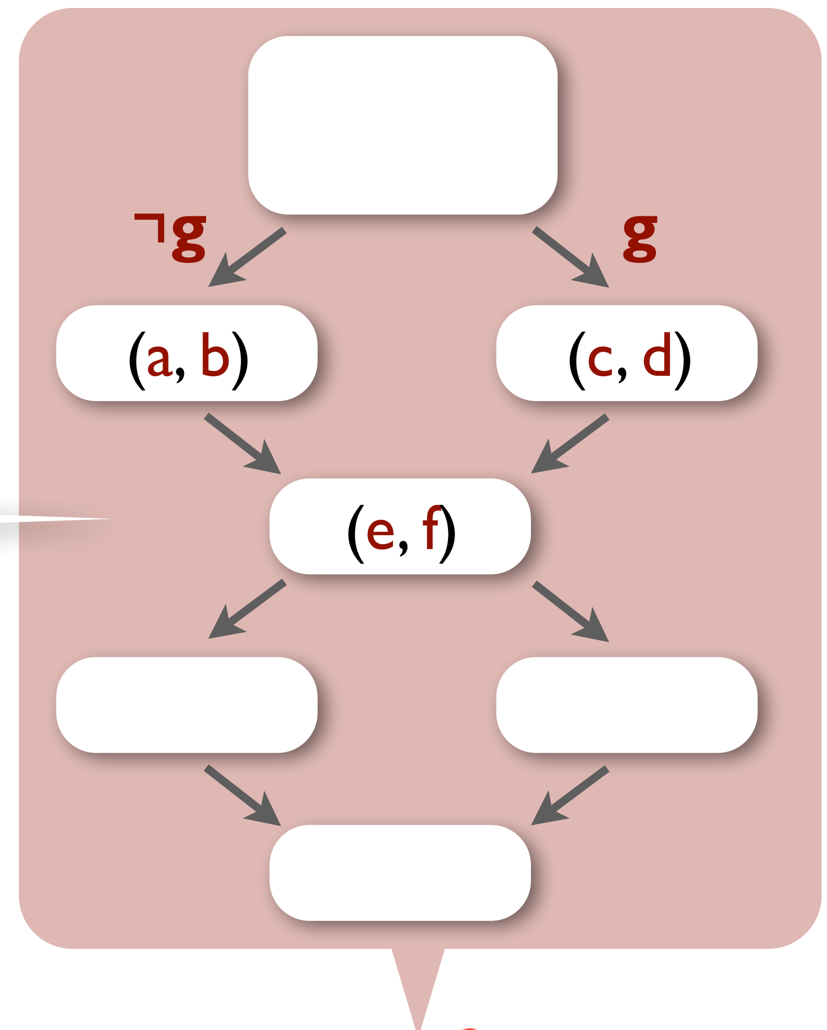


A new design: type-driven state merging

```
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

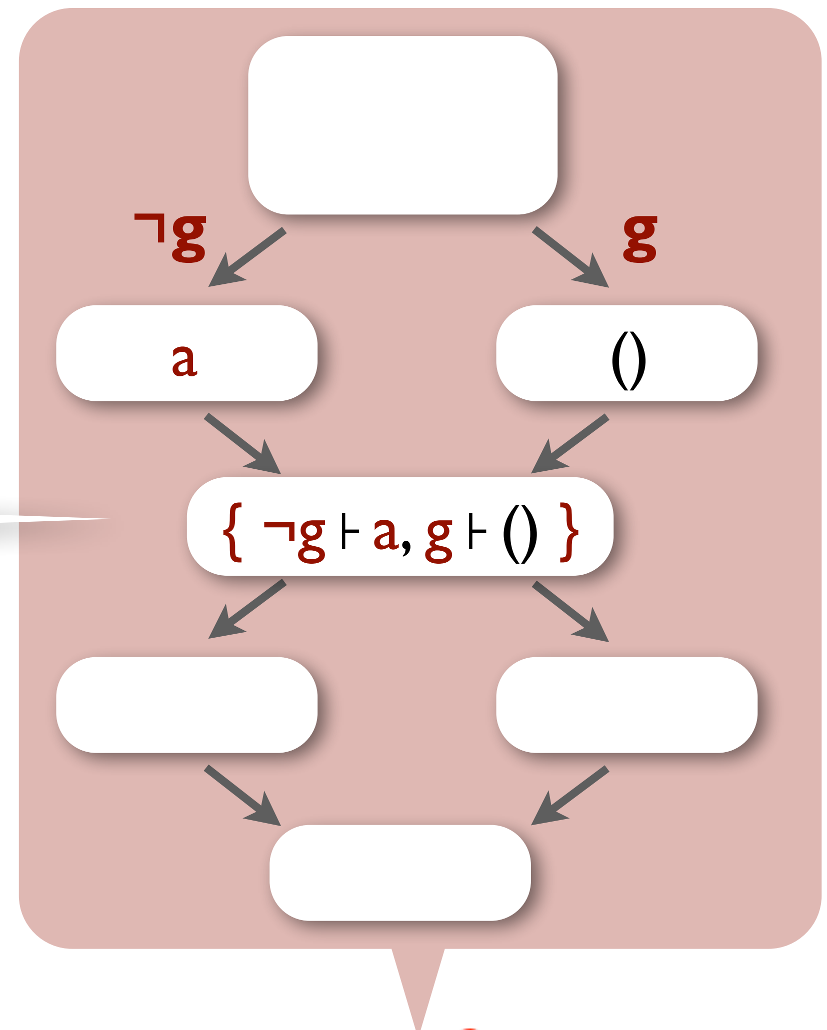


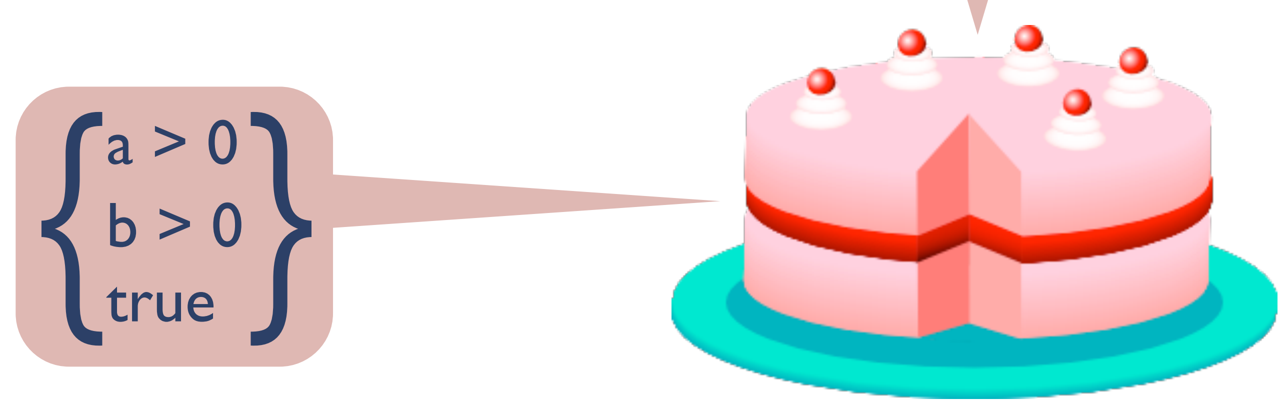
A new design: type-driven state merging

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

Merge values of

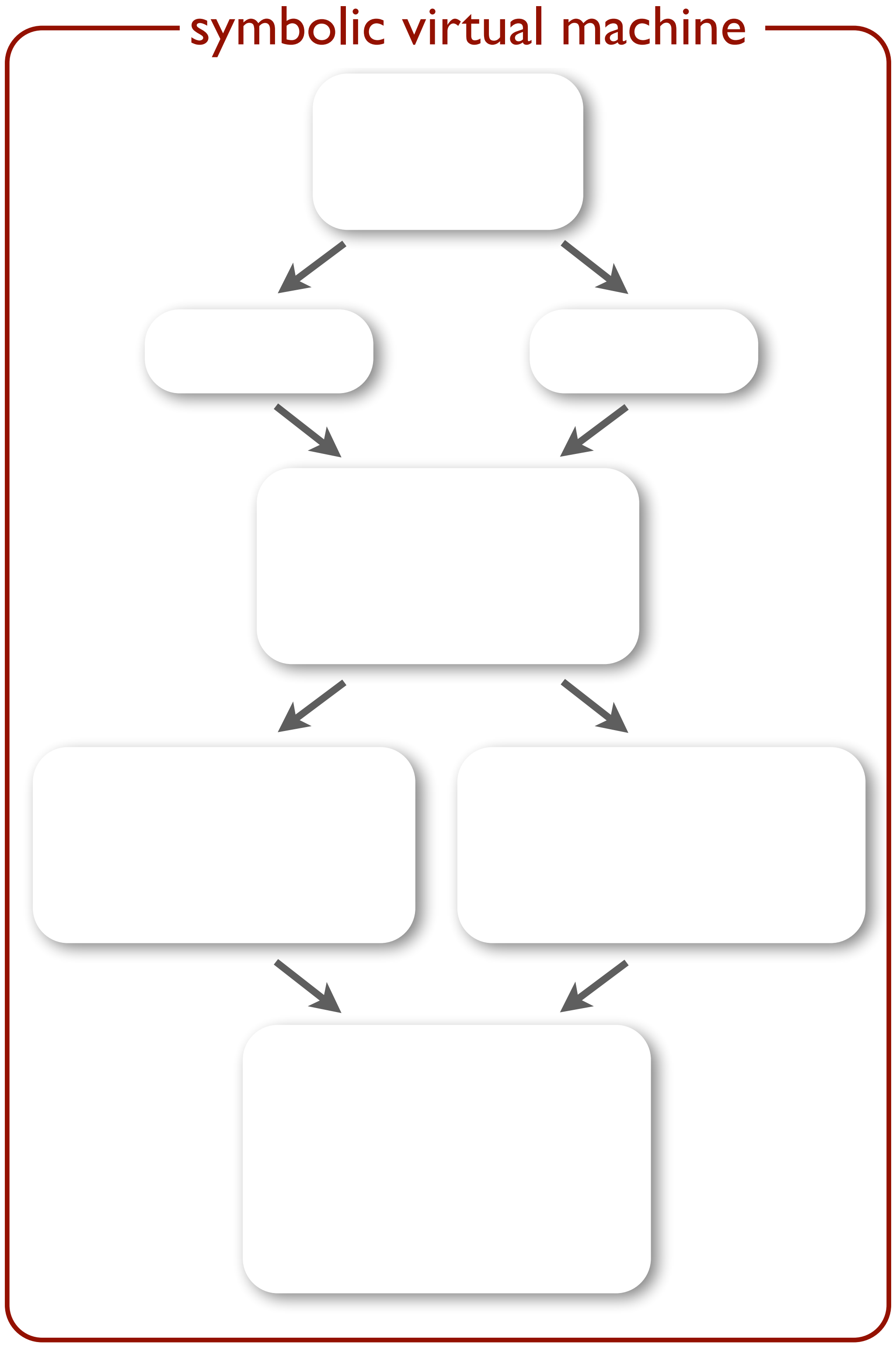
- primitive types: symbolically
- immutable types: structurally
- all other types: via unions





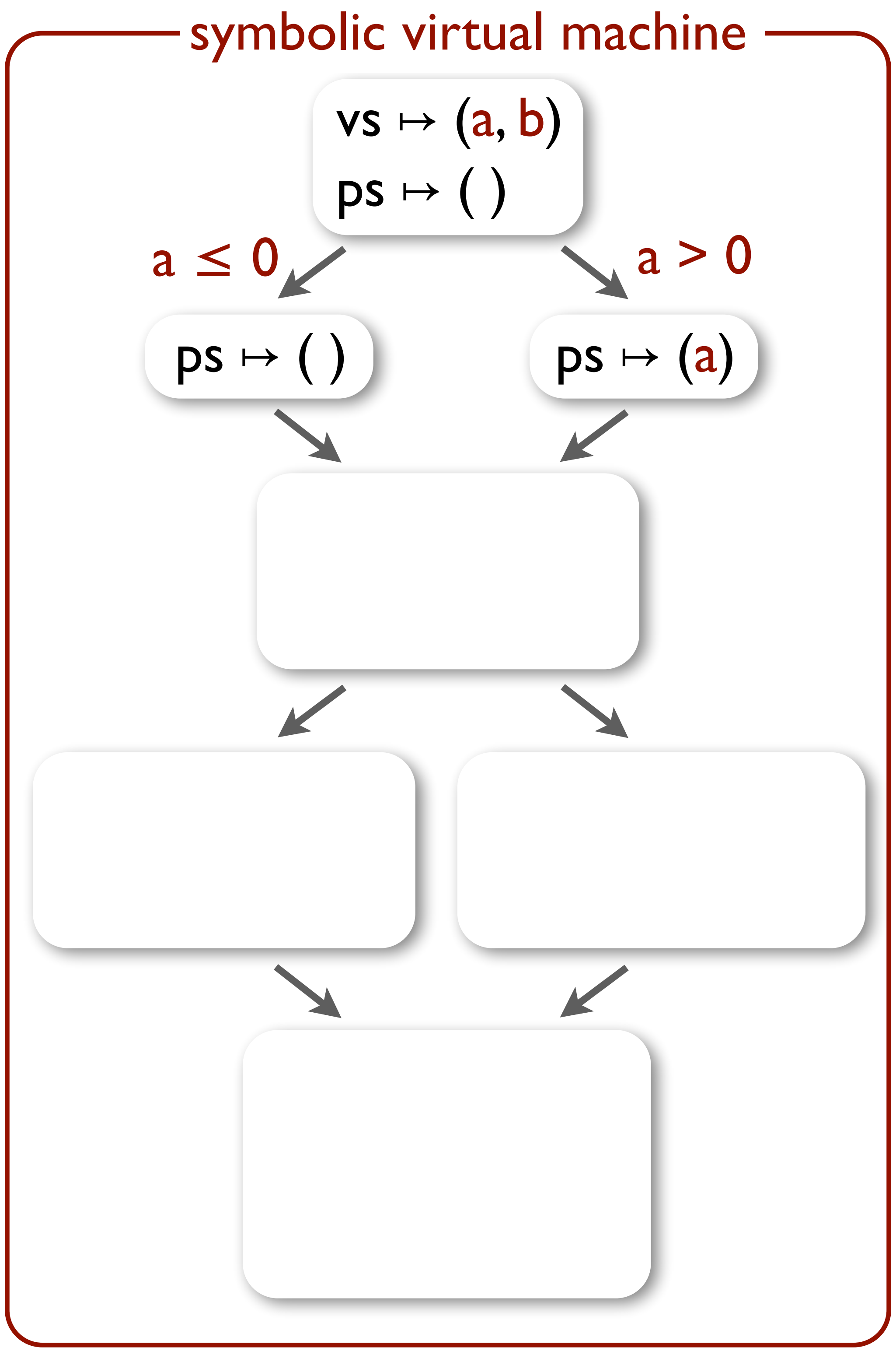
A new design: type-driven state merging

```
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)
```

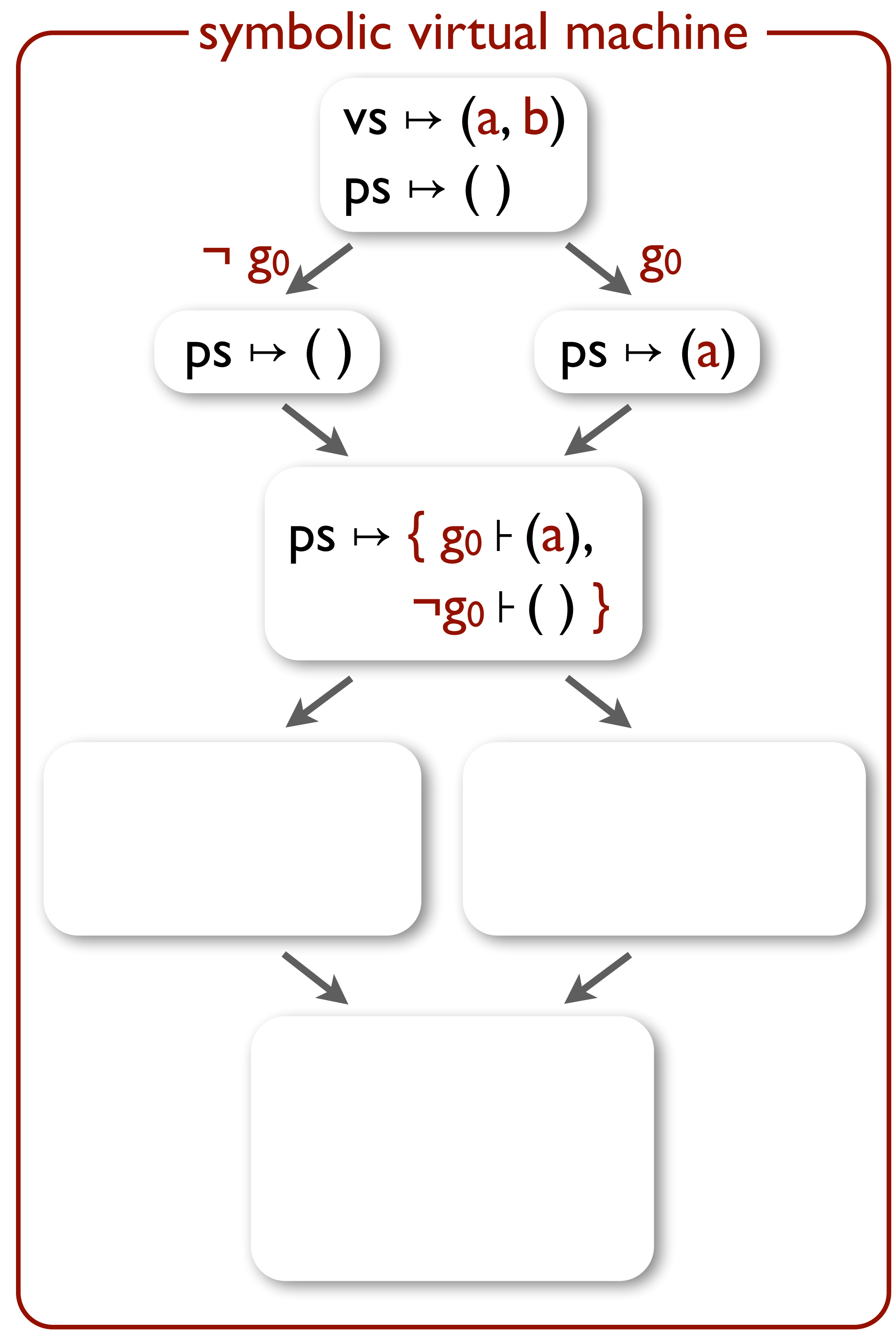


A new design: type-driven state merging

```
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.

$g_0 = a > 0$

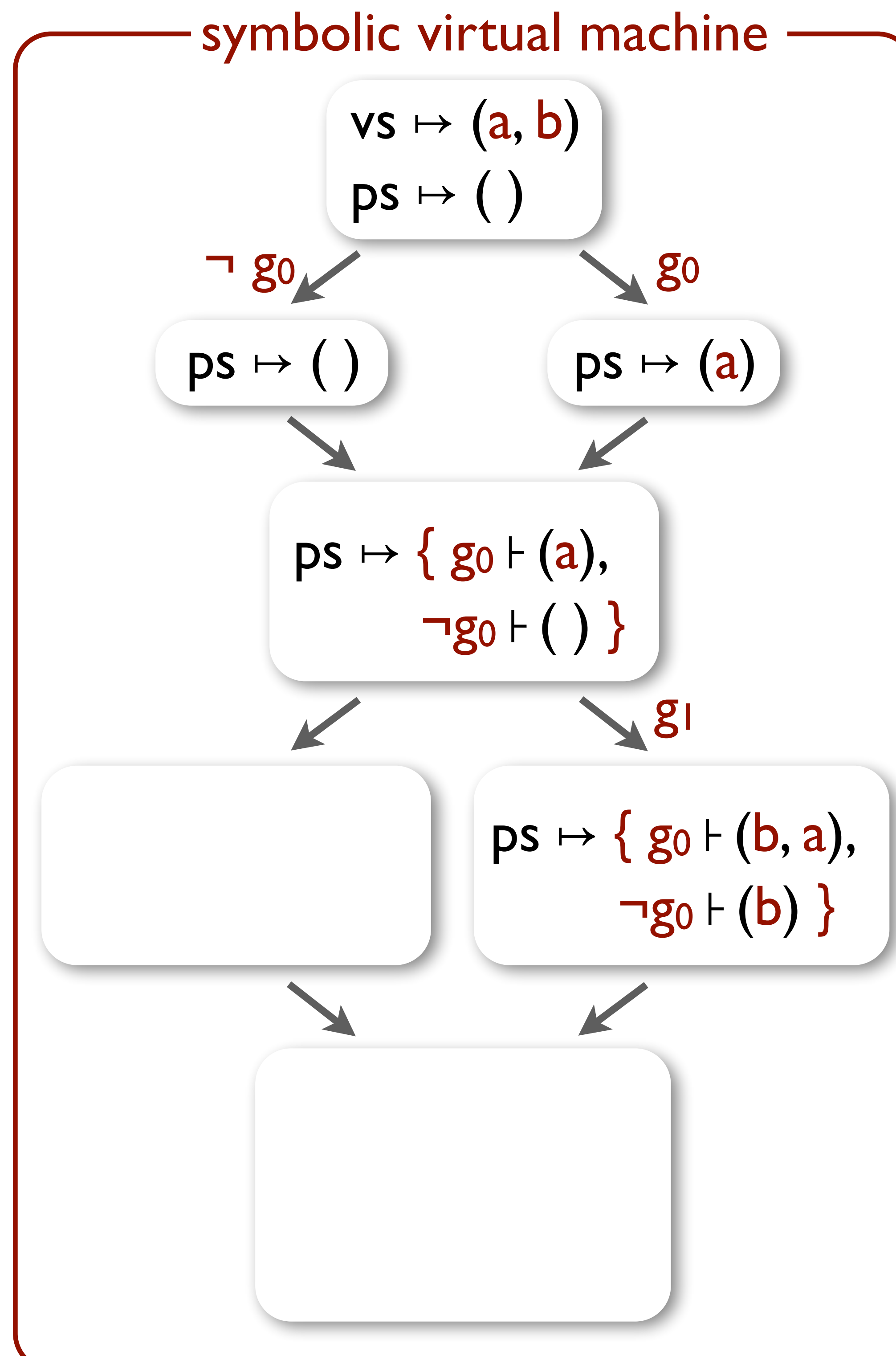


A new design: type-driven state merging

```
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.

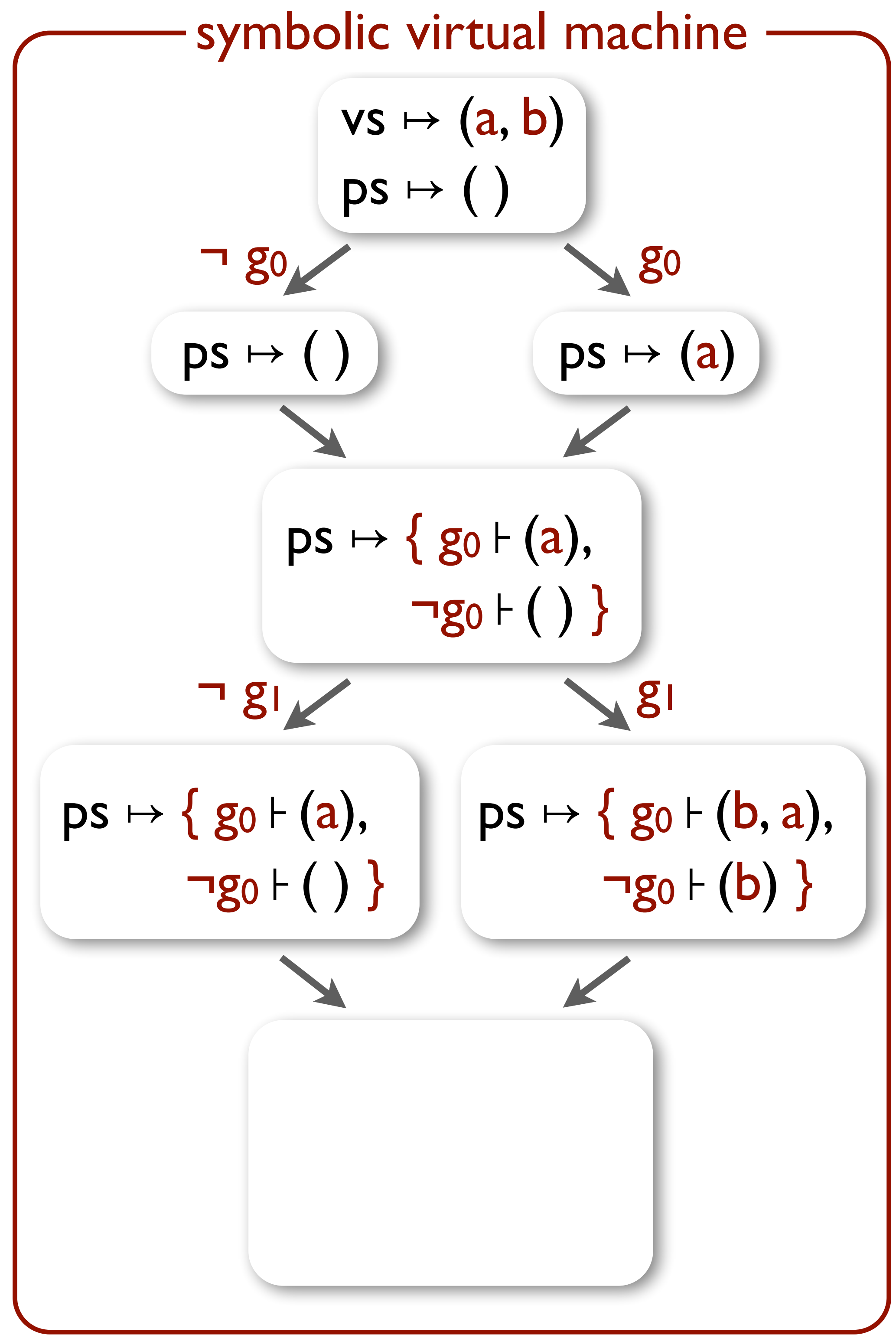
$g_0 = a > 0$
$g_1 = b > 0$



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)
```

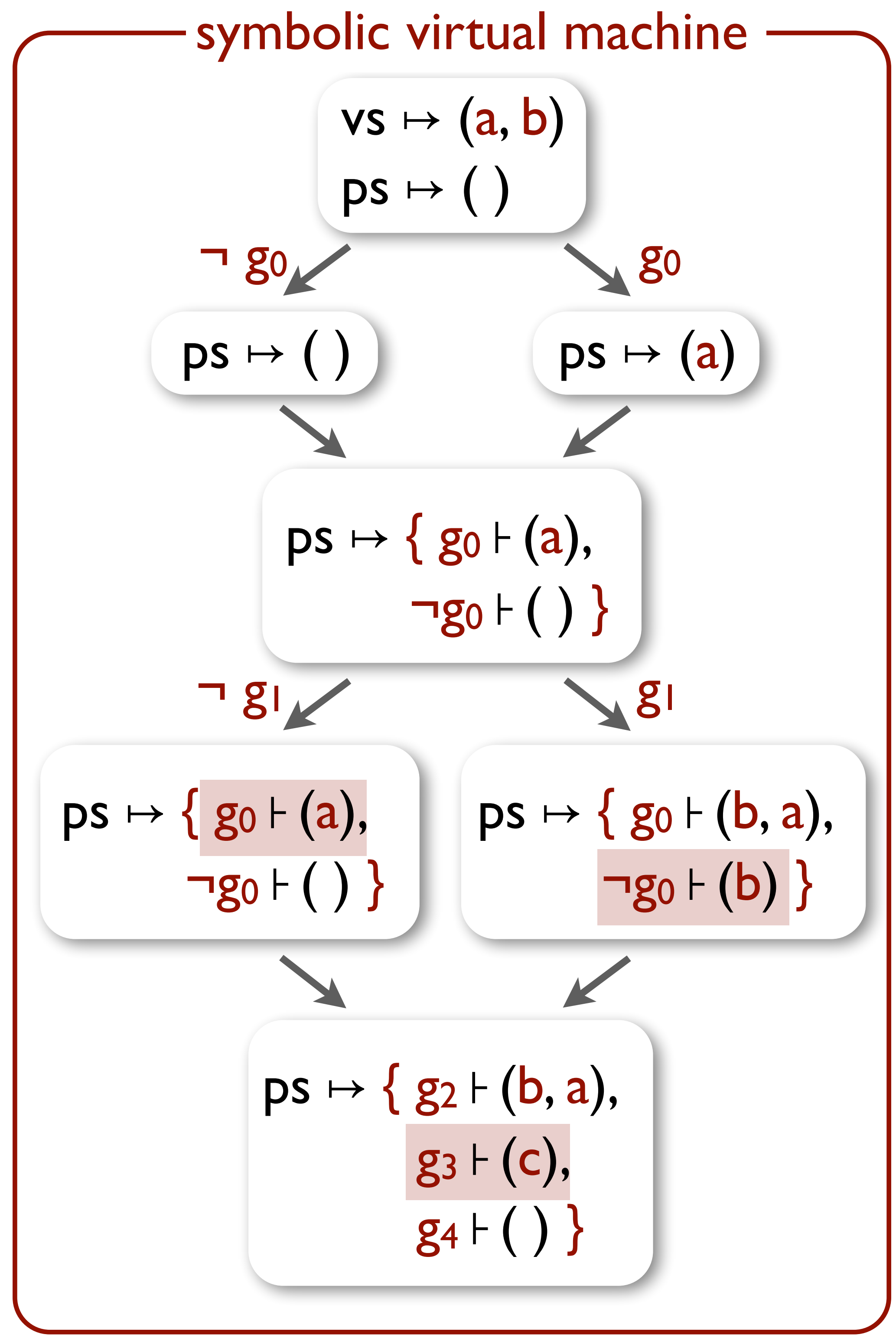
$g_0 = a > 0$
$g_1 = b > 0$



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)
```

$g_0 = a > 0$
$g_1 = b > 0$
$g_2 = g_0 \wedge g_1$
$g_3 = \neg(g_0 \Leftrightarrow g_1)$
$g_4 = \neg g_0 \wedge \neg g_1$
$c = \mathrm{ite}(g_1, b, a)$

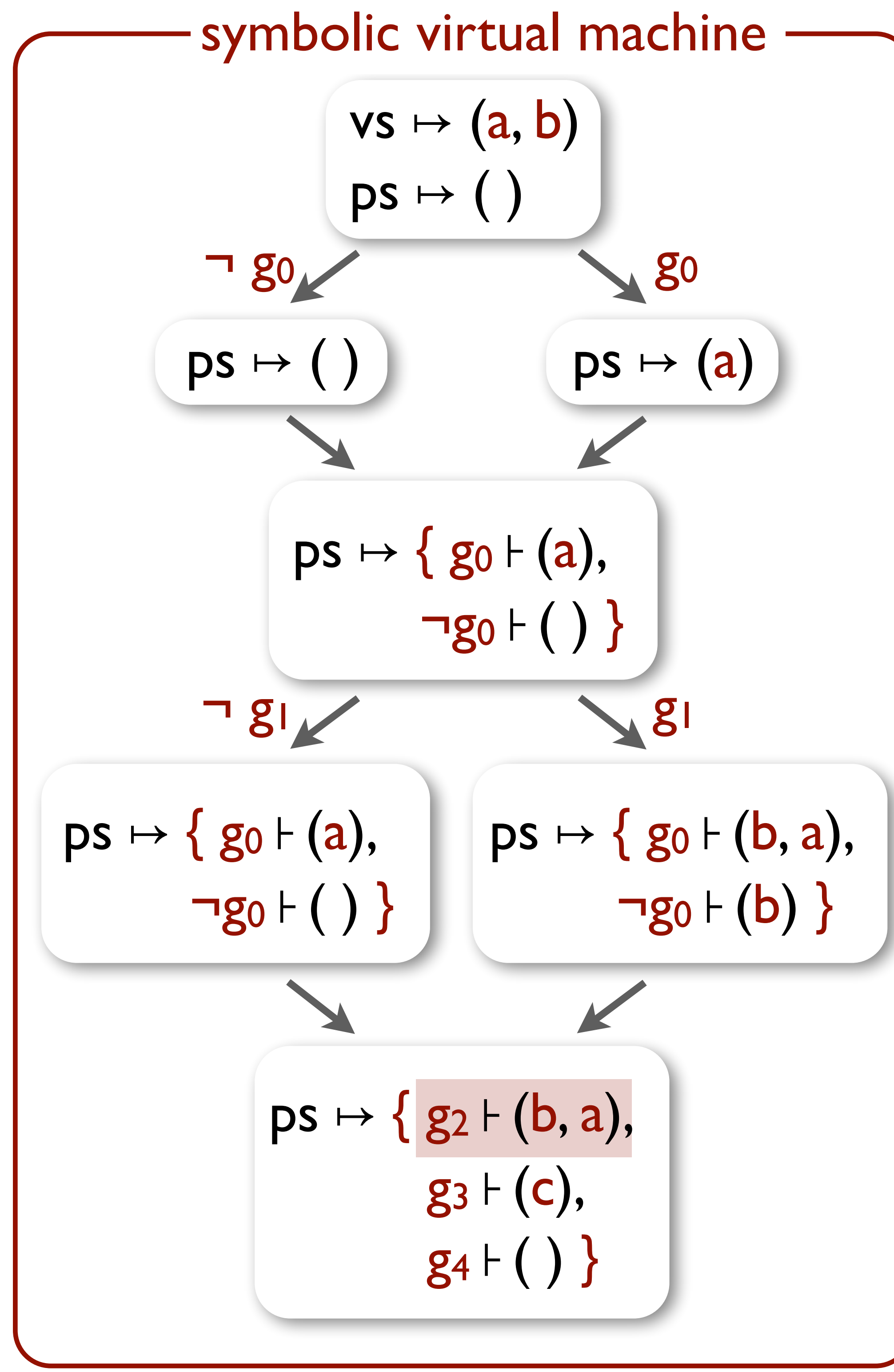


A new design: type-driven state merging

```
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



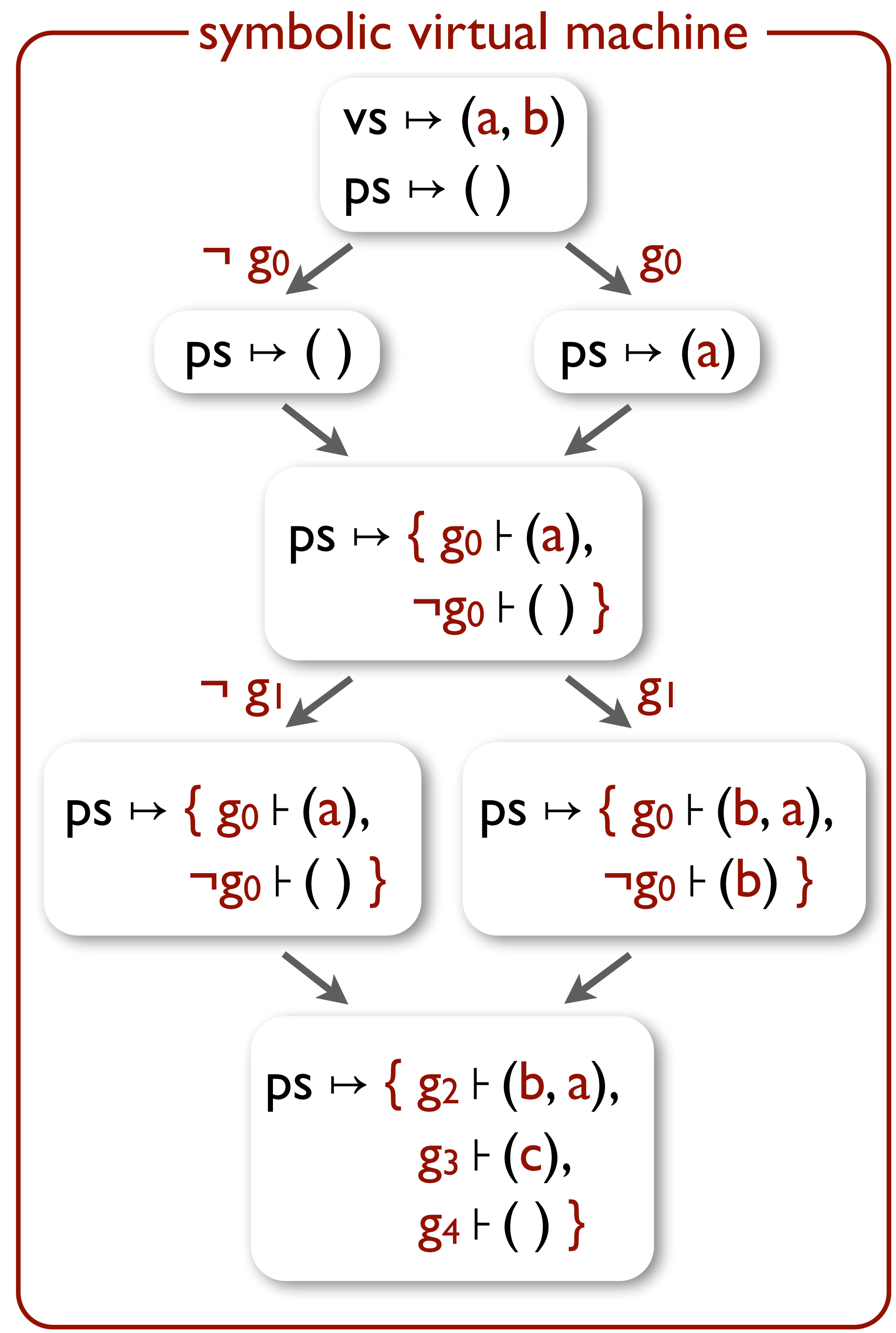
A new design: type-driven state merging

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

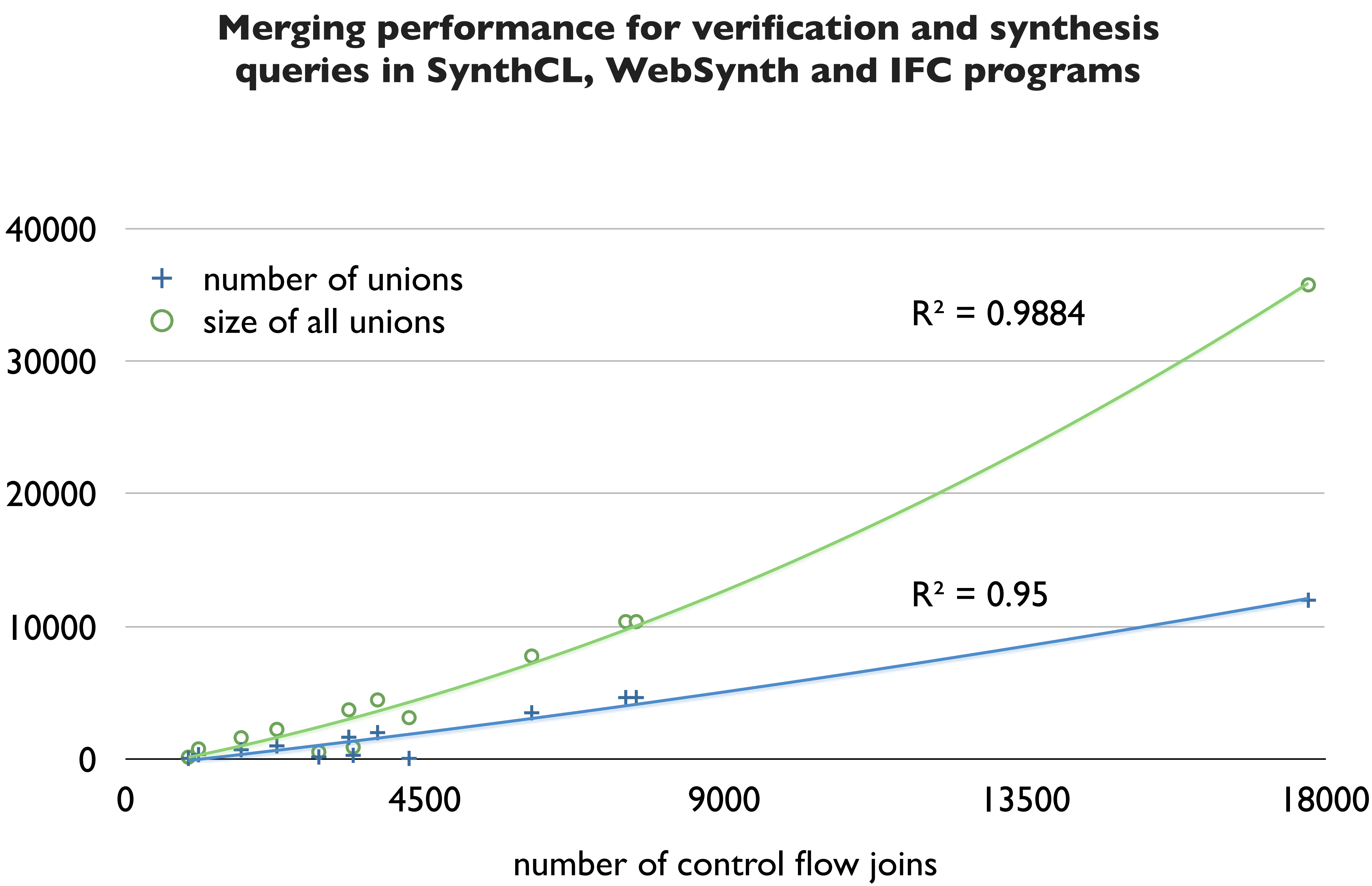
polynomial encoding

concrete evaluation

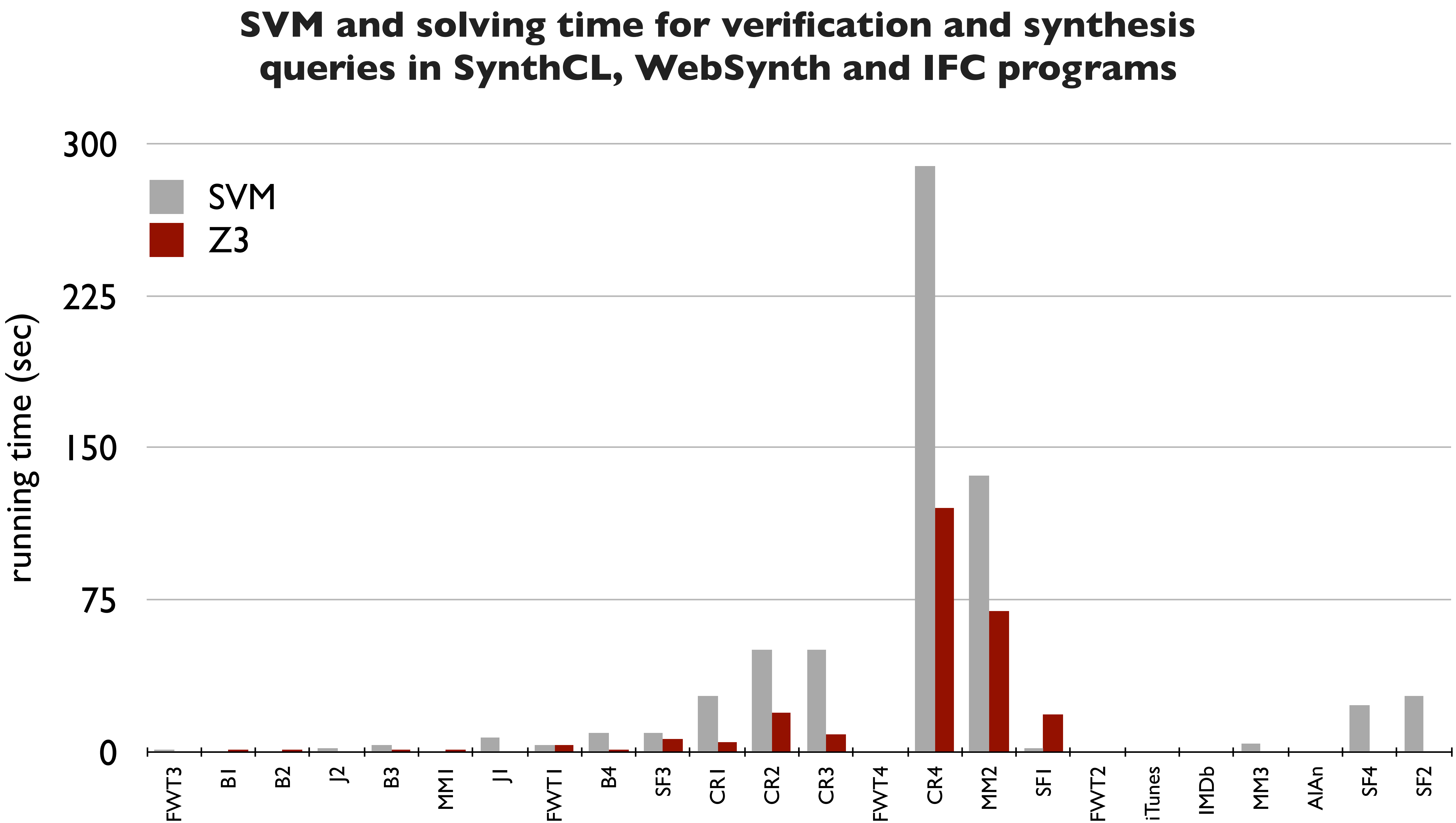
$g_0 = a > 0$
$g_1 = b > 0$
$g_2 = g_0 \wedge g_1$
$g_3 = \neg(g_0 \Leftrightarrow g_1)$
$g_4 = \neg g_0 \wedge \neg g_1$
$c = \mathrm{ite}(g_1, b, a)$
assert g_2



Effectiveness of type-driven state merging

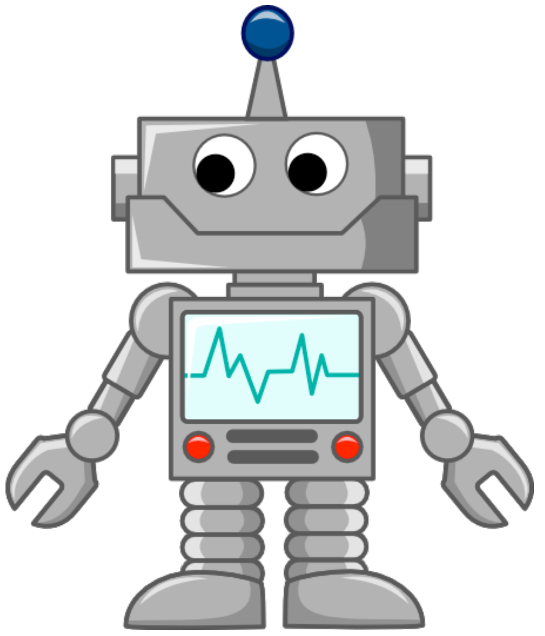


Effectiveness of type-driven state merging



apps

solver-aided programming for everyone



Chlorophyll: ultra low-power computing

Instructions/Second vs Power

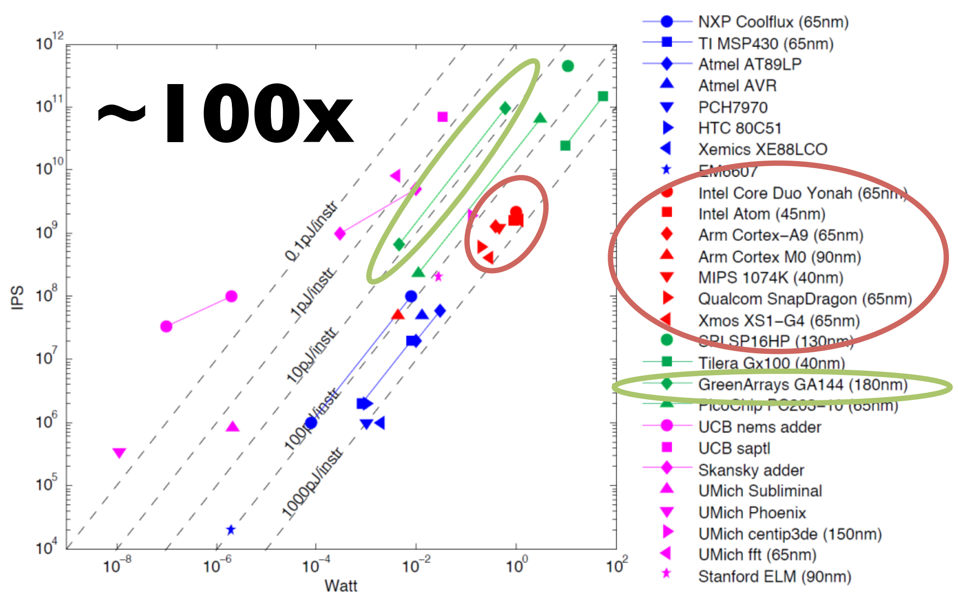
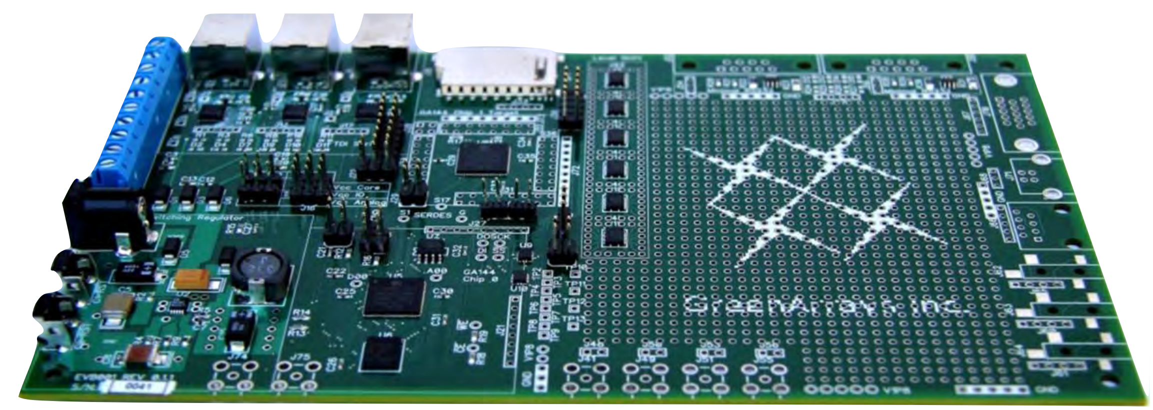


Figure by Per Ljung

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

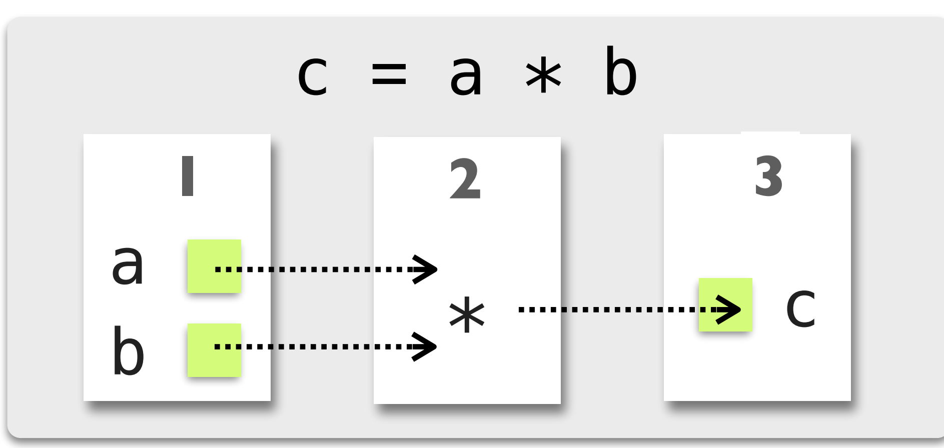
Manual program partitioning: break programs up into a pipeline with a few operations per core.

Drawing by Mangpo Phothilimthana

Chlorophyll: ultra low-power computing

GreenArrays GA144 Processor

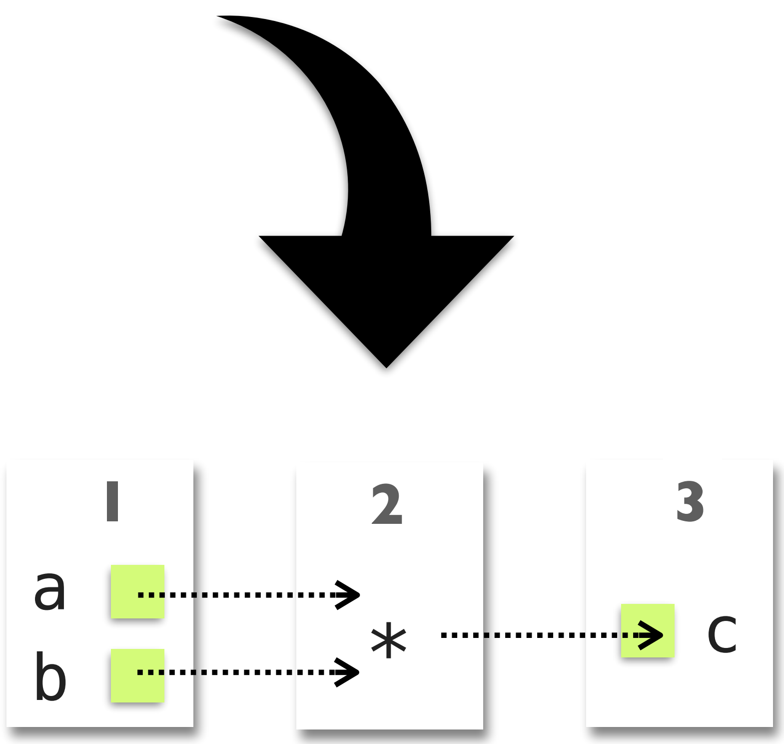
- Stack-based 18-bit architecture
- 32 instructions
- 8 x 18 array of asynchronous cores
- No shared resources (cache, memory)
- Limited communication, neighbors only
- < 300 byte memory per core



Drawing by Mangpo Phothilimthana

Chlorophyll: ultra low-power computing

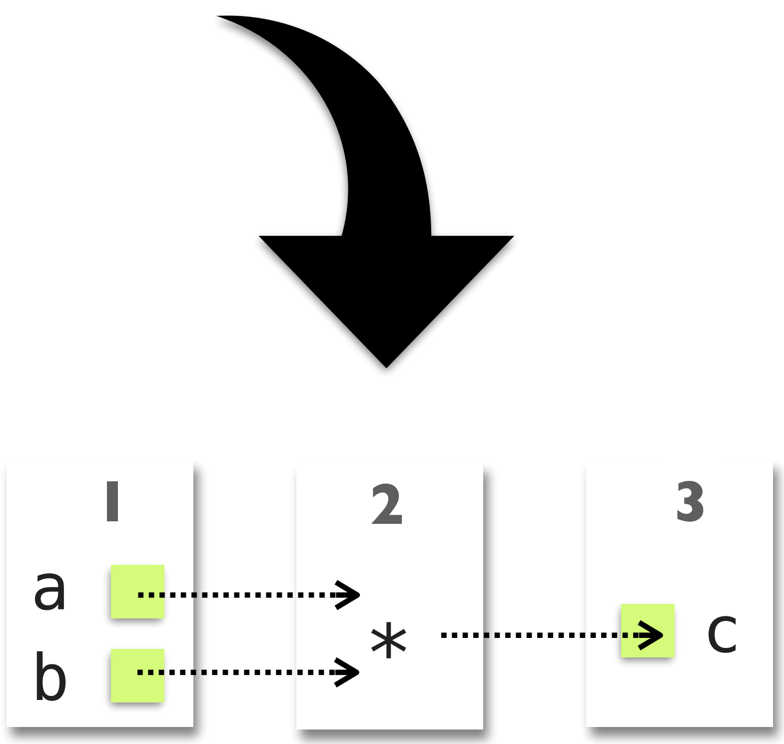
```
int a, b;
int c = a * b;
```



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;
```

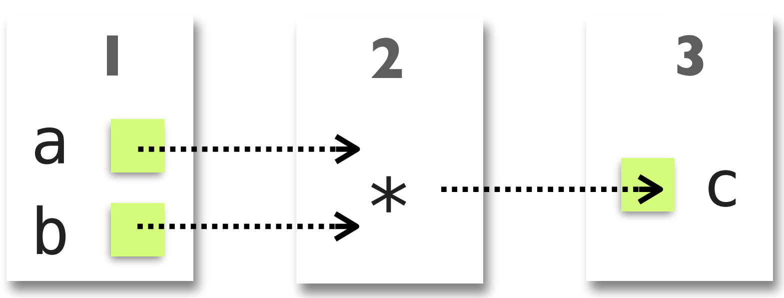


Synthesizes placement of code and data onto cores, by **type-checking a program** sketch in a C-like DSL.

Chlorophyll: ultra low-power computing

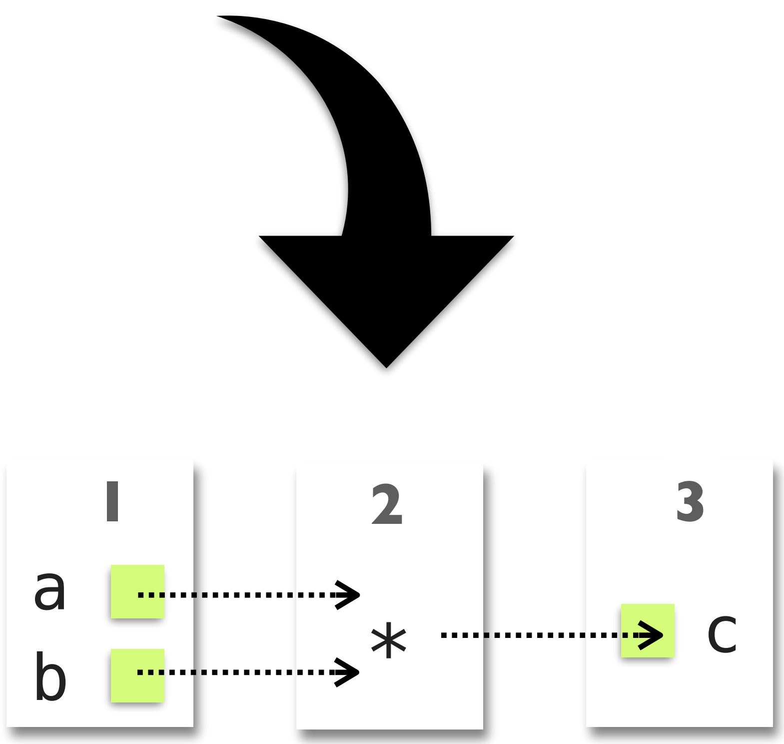
```
int@?? a, b;
int@?? c = a *@?? b;
```

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;
```

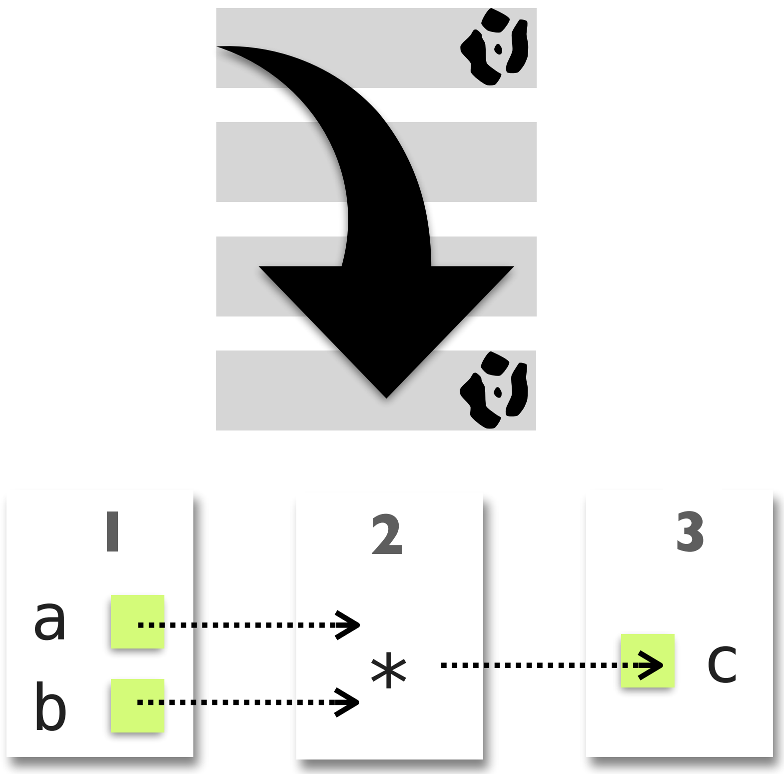


Built by a first-year grad in a few weeks

Phitchaya Mangpo Phothilimthana

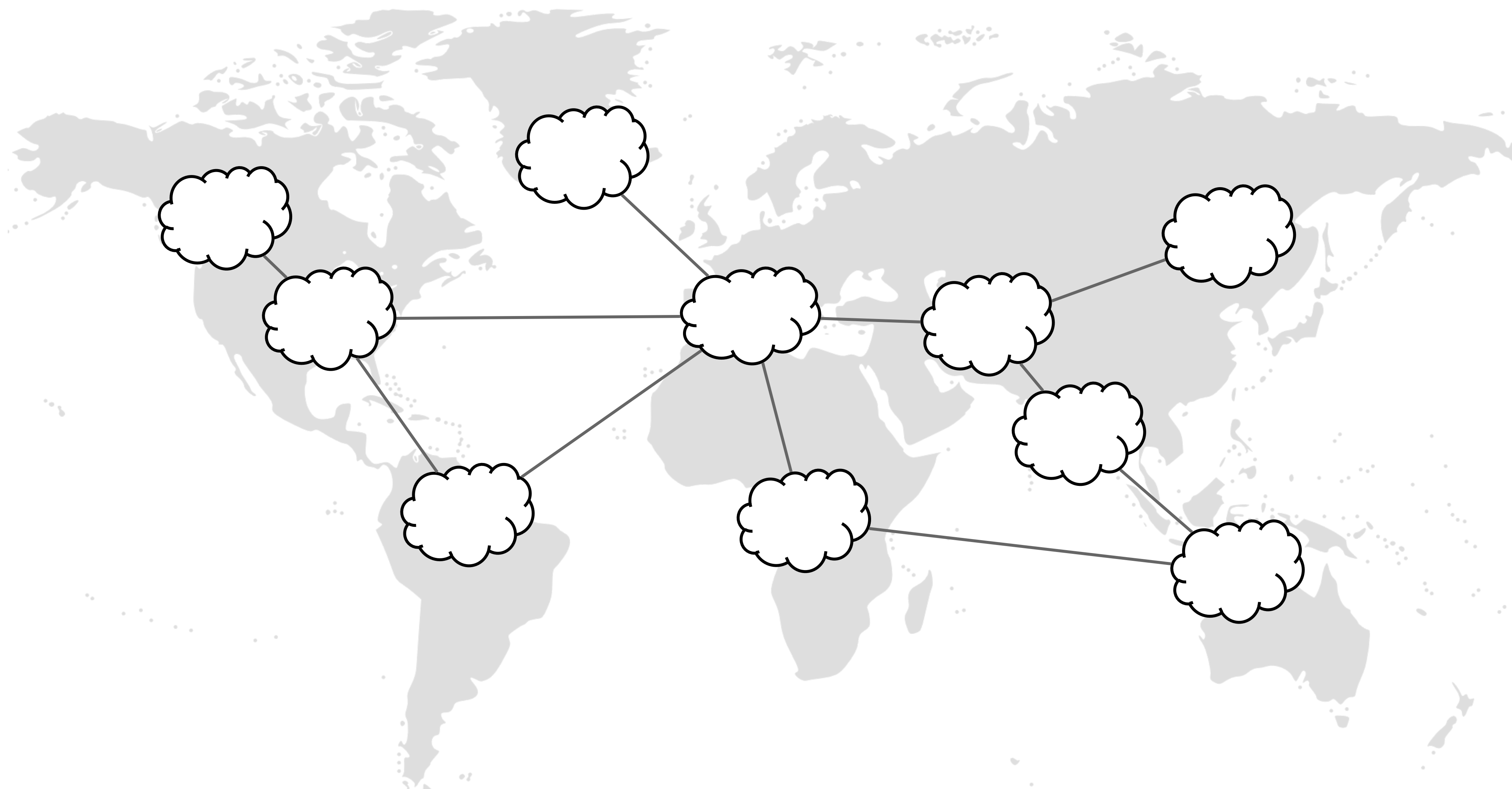
Chlorophyll: ultra low-power computing

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

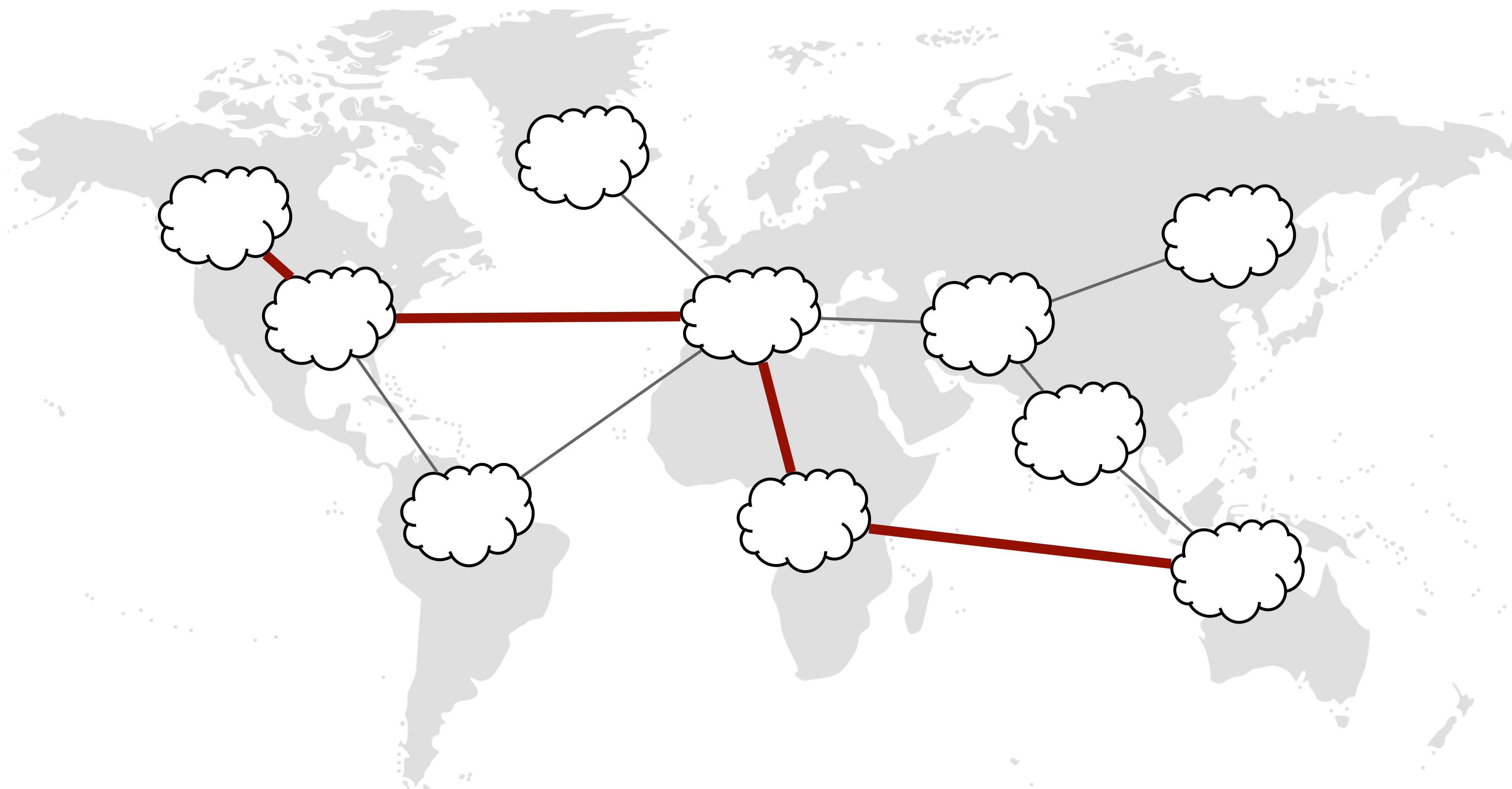


[Phothilimthana et al., **PLDI'14**]

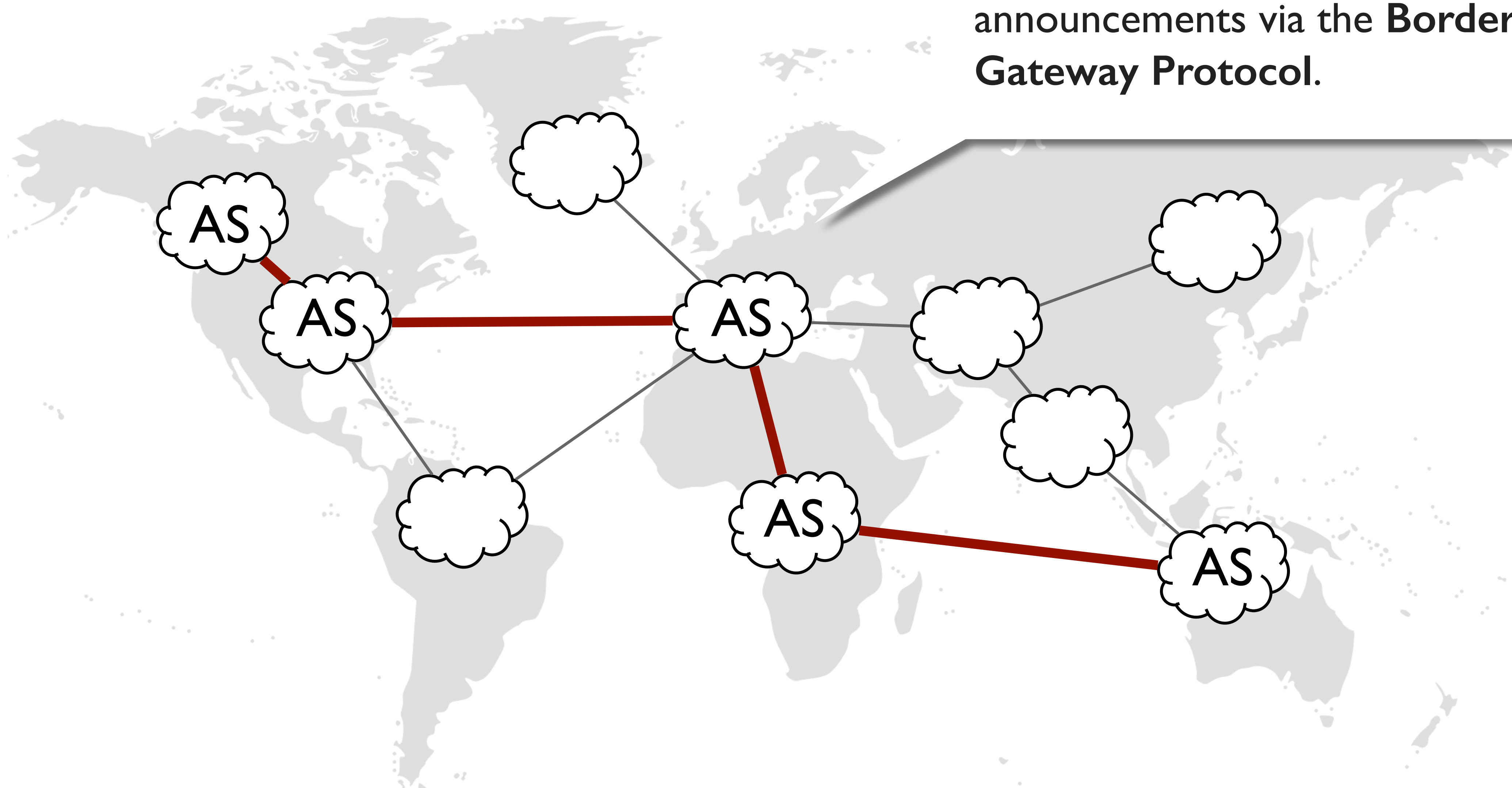
Bagpipe: verifying BGP router configurations



Bagpipe: verifying BGP router configurations

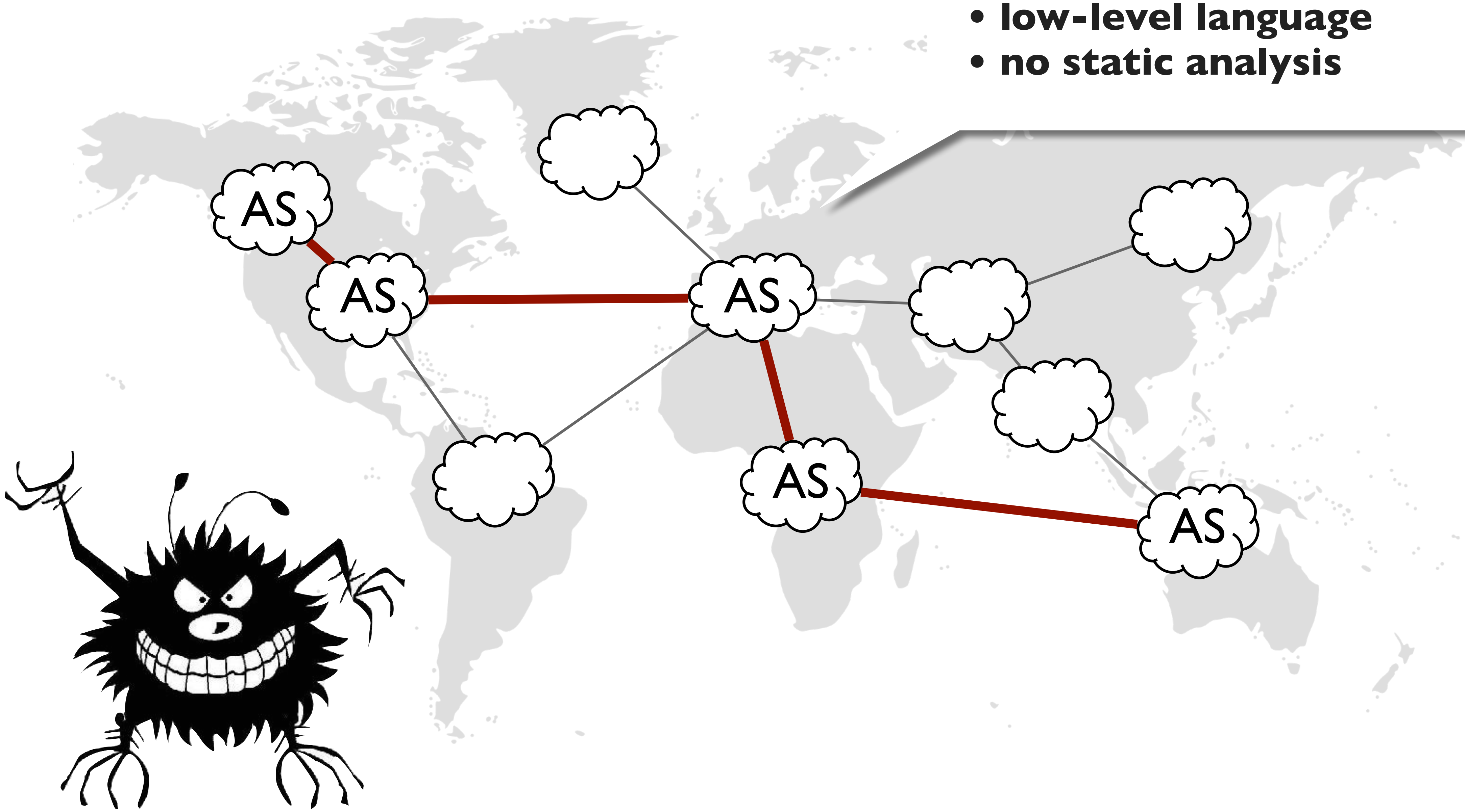


Bagpipe: verifying BGP router configurations



Autonomous systems communicate routing information by sending announcements via the **Border Gateway Protocol**.

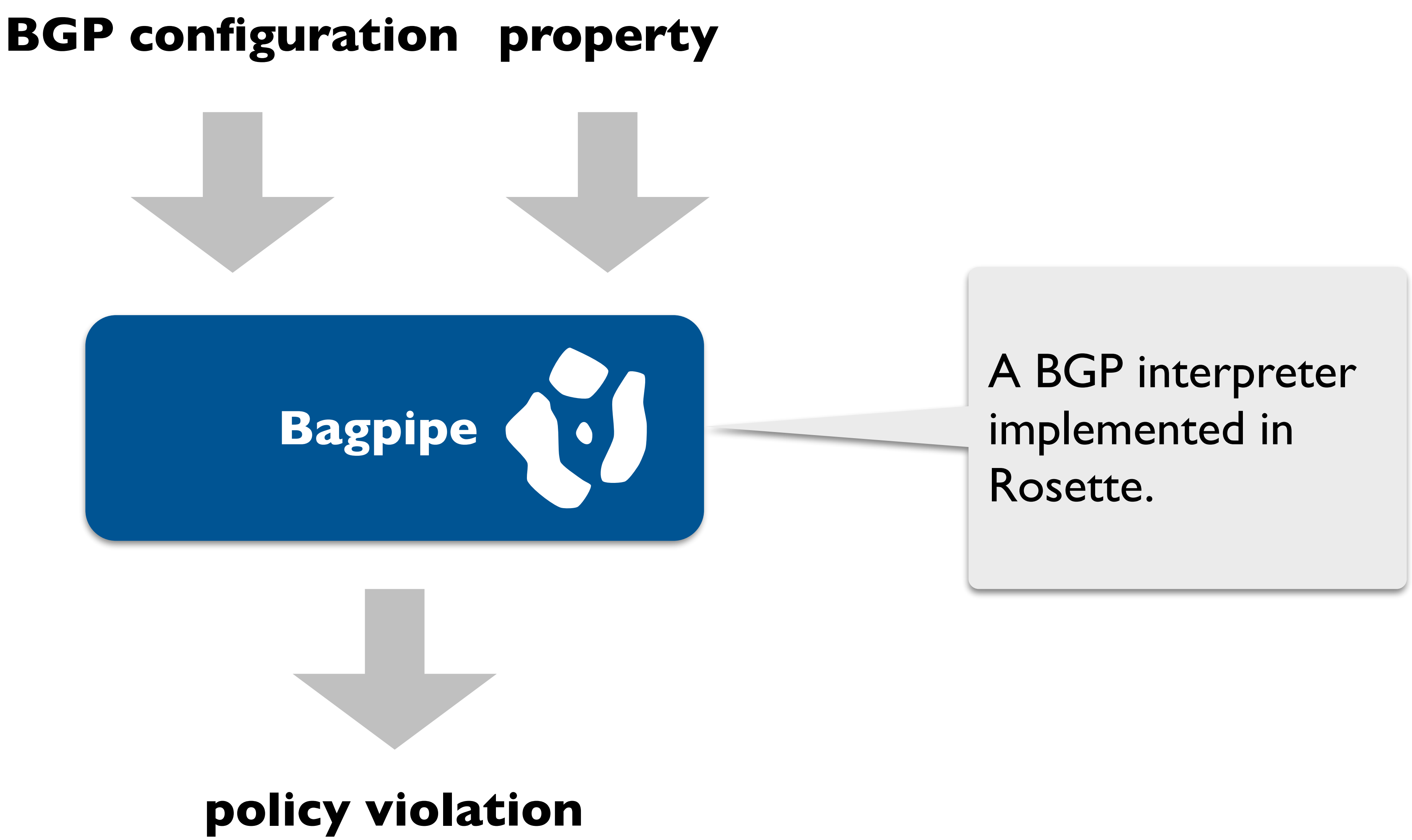
Bagpipe: verifying BGP router configurations



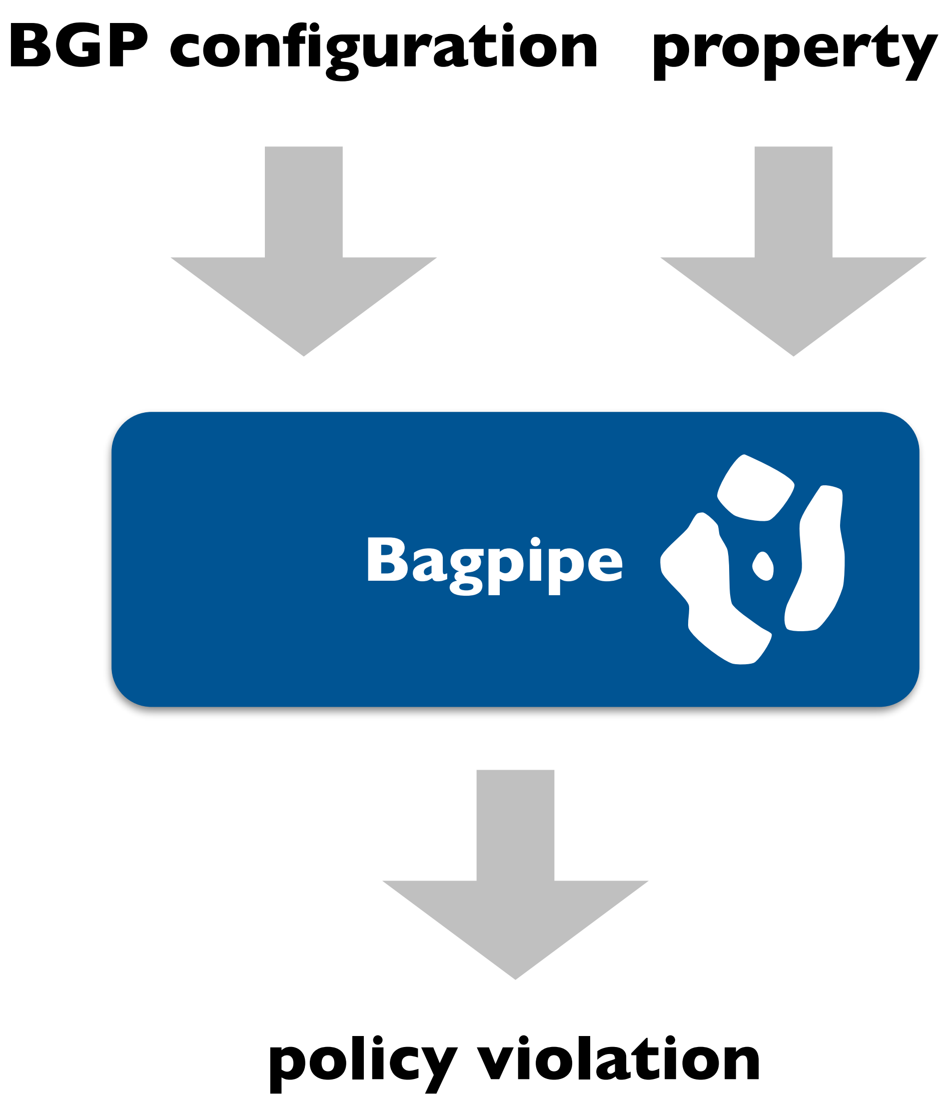
Configuring BGP is tricky

- **distributed system**
- **low-level language**
- **no static analysis**

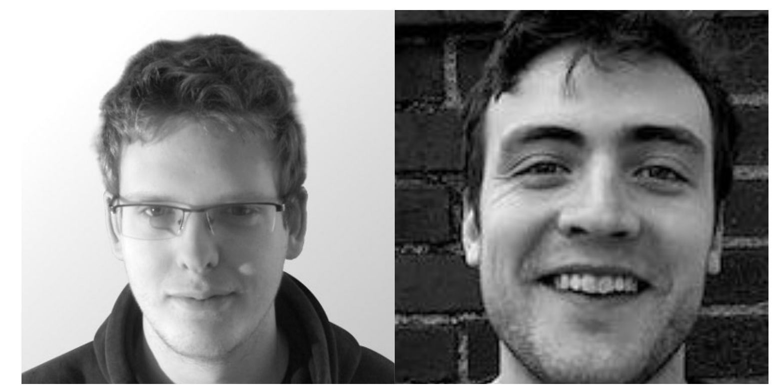
Bagpipe: verifying BGP router configurations



Bagpipe: verifying BGP router configurations

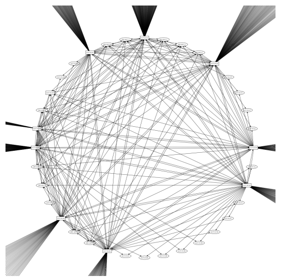


Built by two grads in a few weeks



Konstantin Weitz and Doug Woos

Bagpipe: verifying BGP router configurations

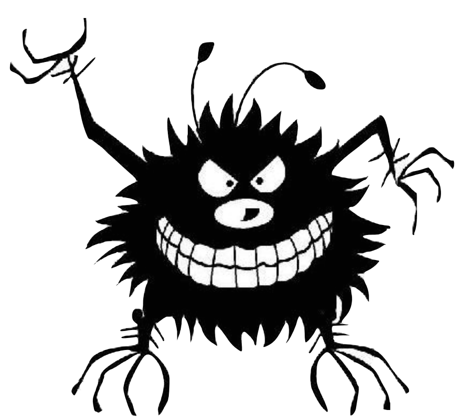


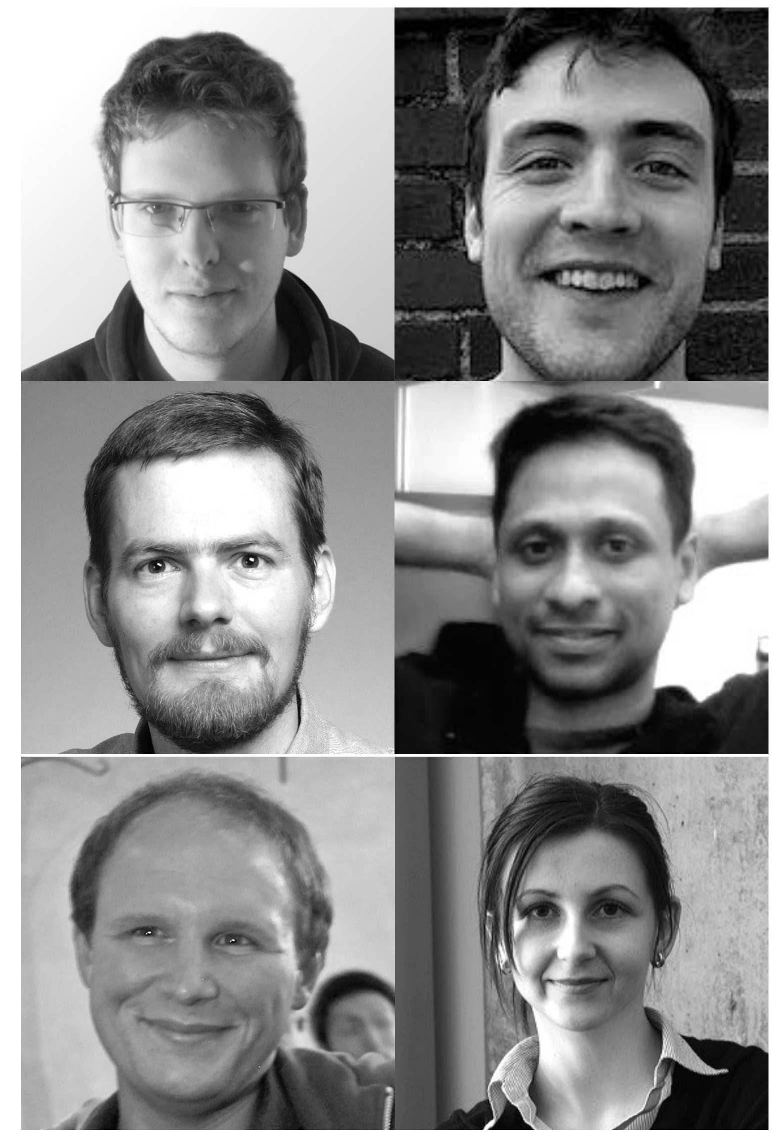
Internet2

private announcements are not leaked

Bagpipe

route leaks!

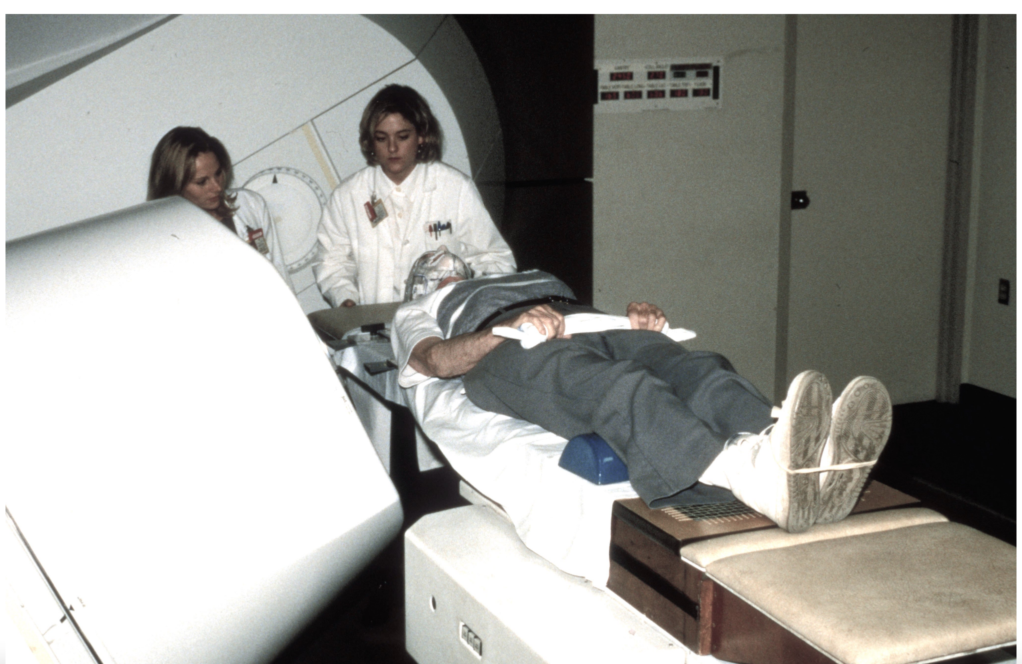




[Weitz et al., **OOPSLA'16**]

Neutrons: verifying a radiotherapy system

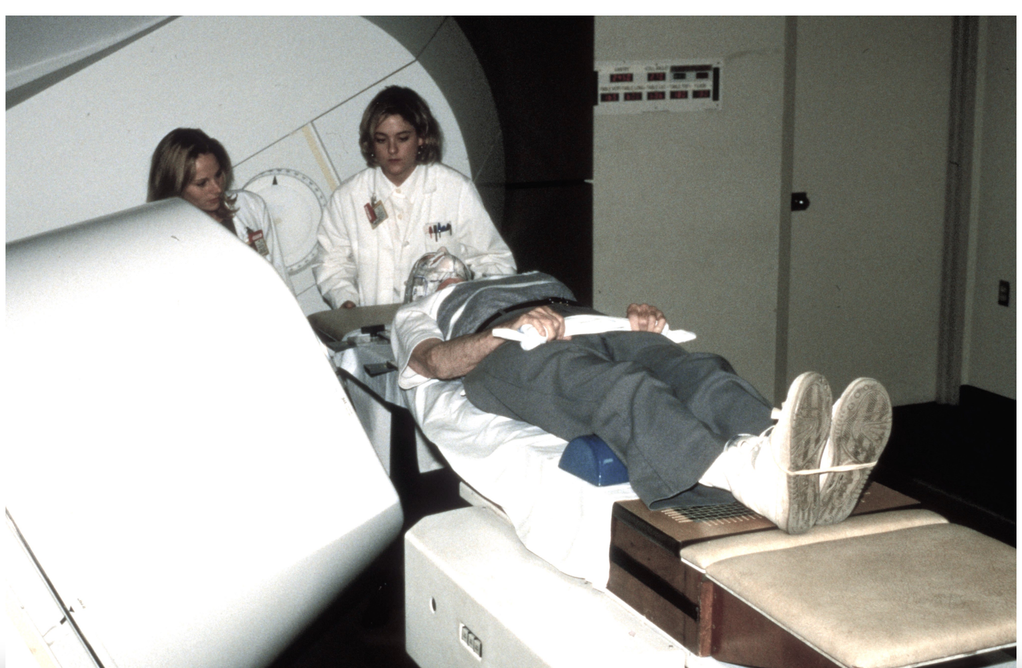
Clinical Neutron Therapy System (CNTS) at UW

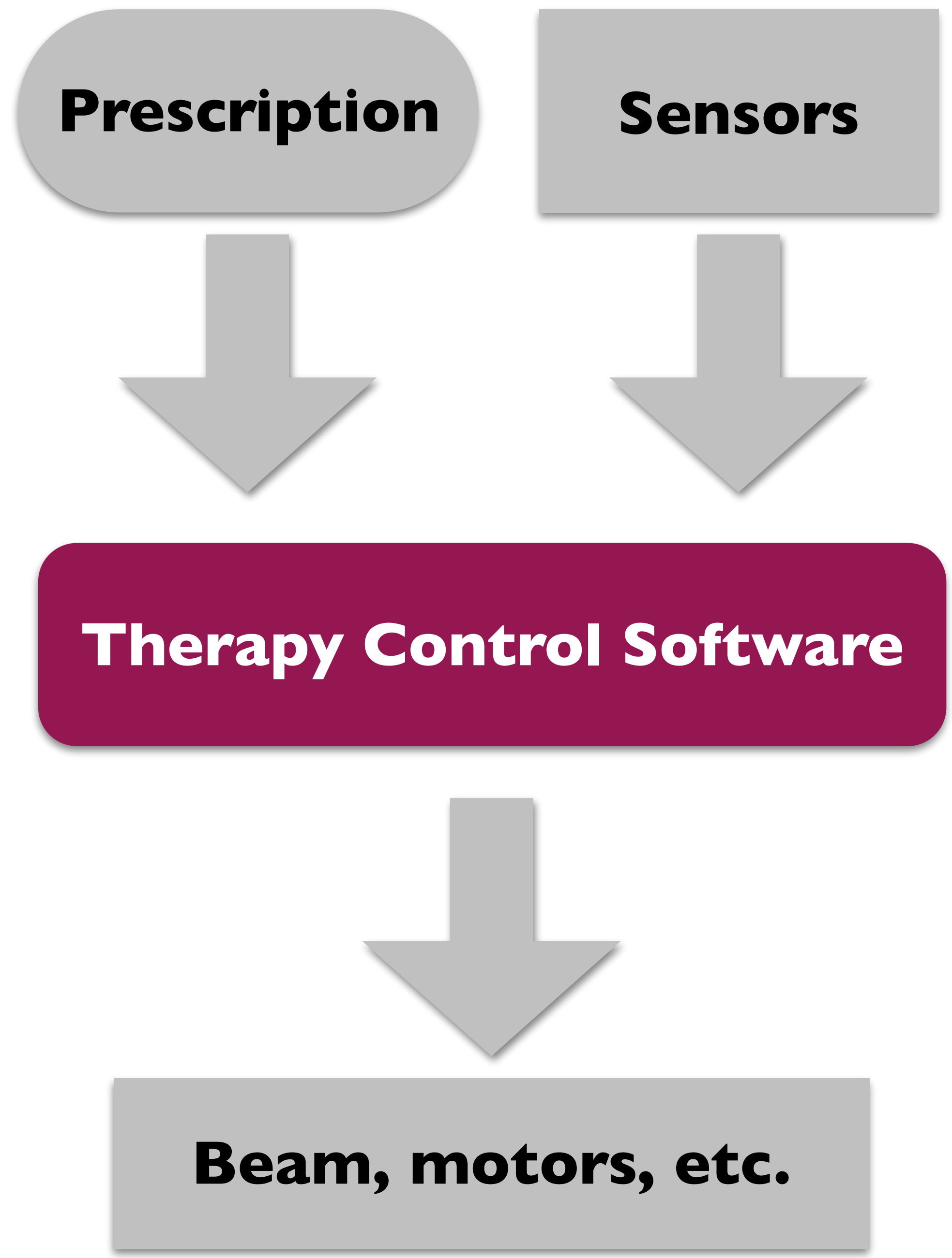


- 30 years of incident-free service.
- Controlled by custom software, built by CNTS engineering staff.
- Third generation of Therapy Control software built recently.

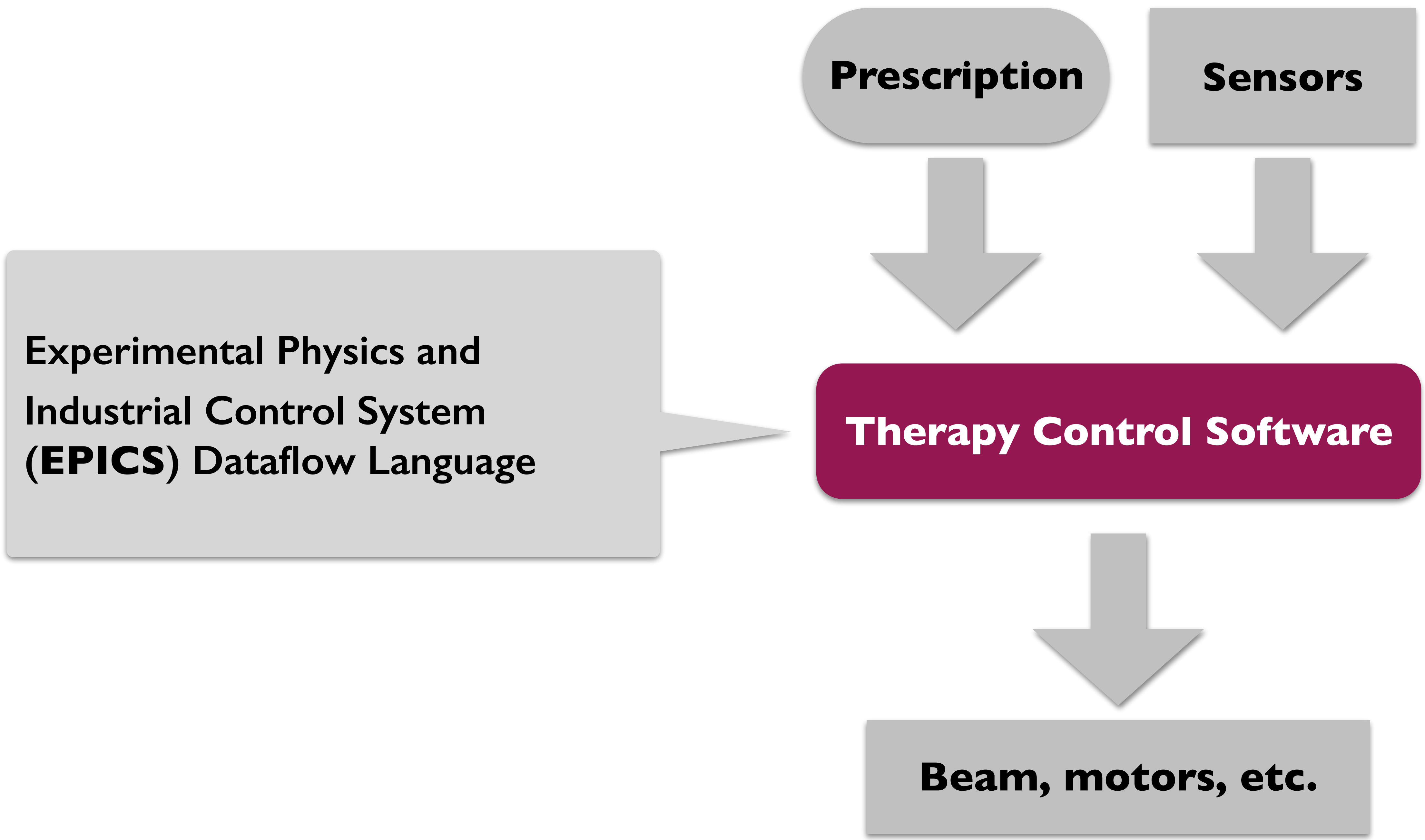
Neutrons: verifying a radiotherapy system

Clinical Neutron Therapy System (CNTS) at UW





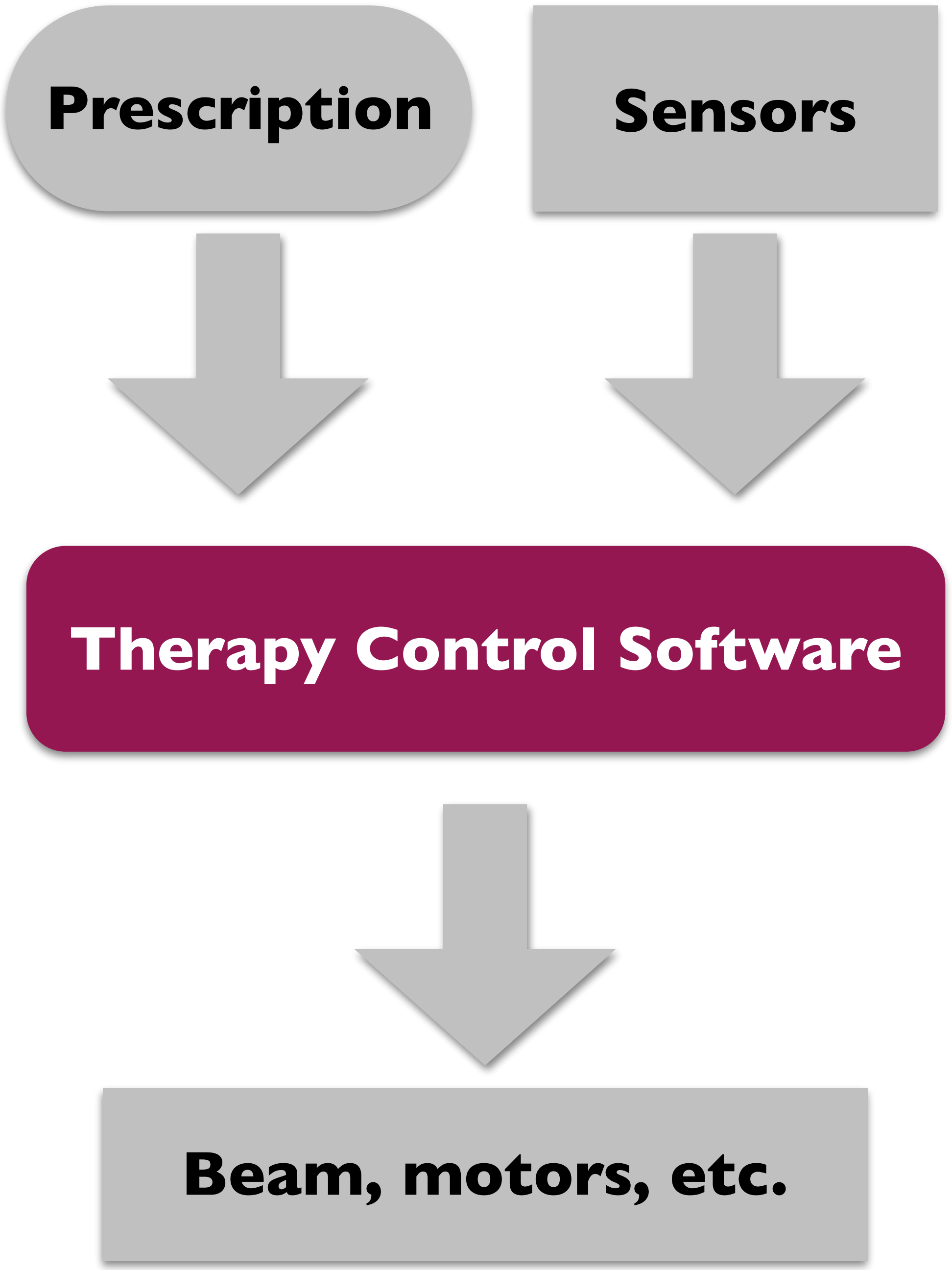
Neutrons: verifying a radiotherapy system



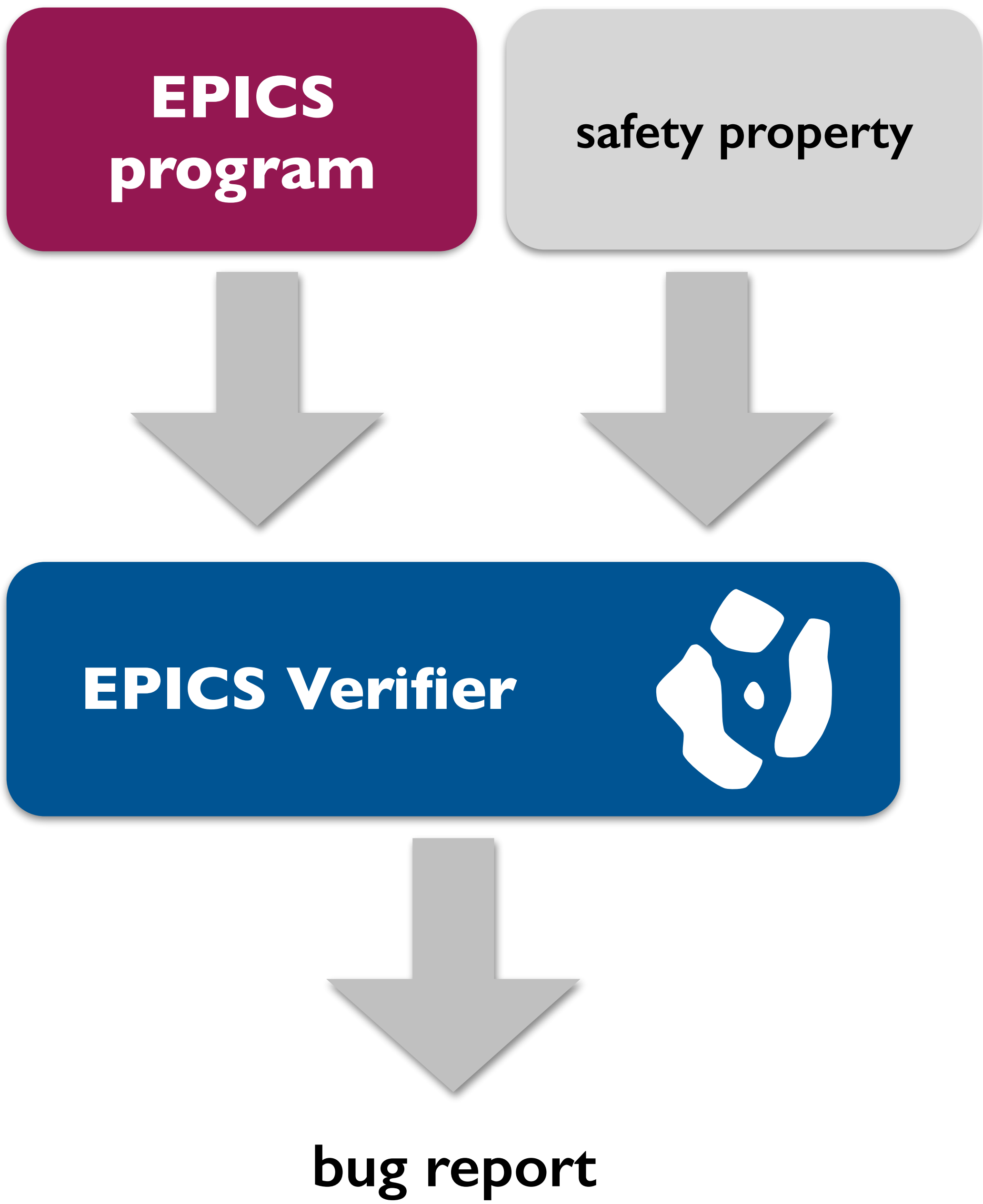
Neutrons: verifying a radiotherapy system

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.



Neutrons: verifying a radiotherapy system



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

Calvin Loncaric

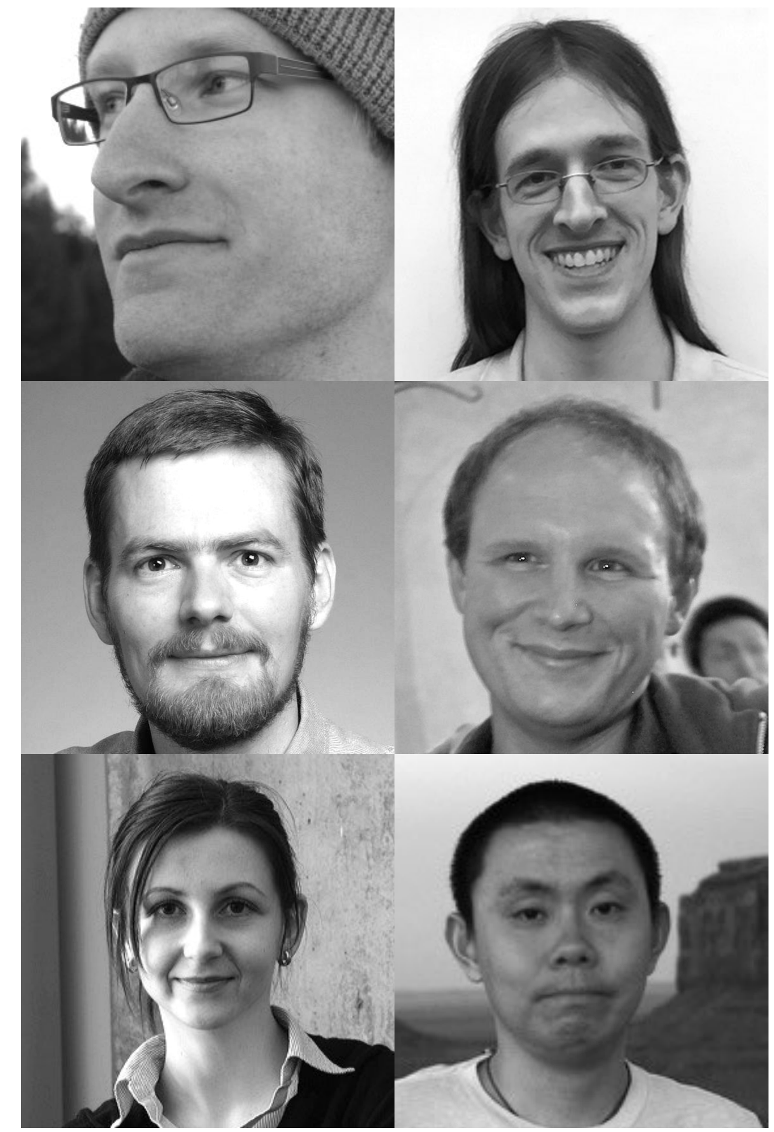
Neutrons: verifying a radiotherapy system

Therapy Control Software

safety property

EPICS Verifier

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



[Pernsteiner et al., CAV'16]

Thanks for a great quarter!