Computer-Aided Reasoning for Software

CSE507

Solver-Aided Programming II

Emina Torlak
emina@cs.washington.edu

Topics

Last lecture

- Getting started with solver-aided programming.

Today

- Going pro with solver-aided programming.

Announcements

- HW1 is out.

topics

A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

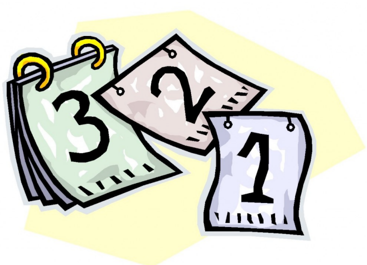
ROSETTE

Solver-aided programming in two parts: (1) getting started and (2) going pro

How to use a solver-aided language: the workflow, constructs, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.

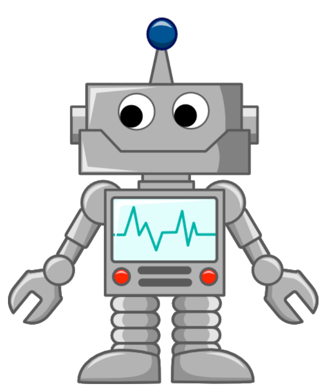
How to build your own solver-aided tool or language



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.

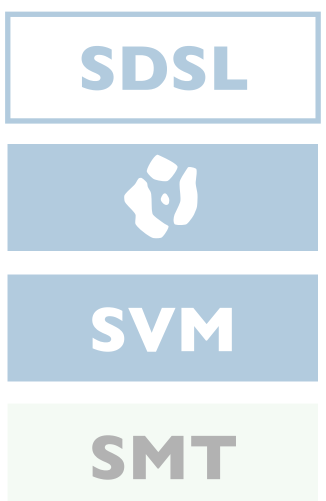
Behind the scenes: symbolic virtual machine
How Rosette works so you don't have to.



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

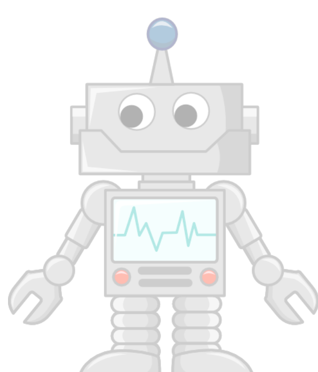
How to build your own solver-aided tool or language

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



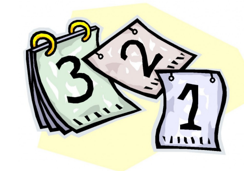
An easier way: tools as languages
How to build tools by stacking layers of languages.

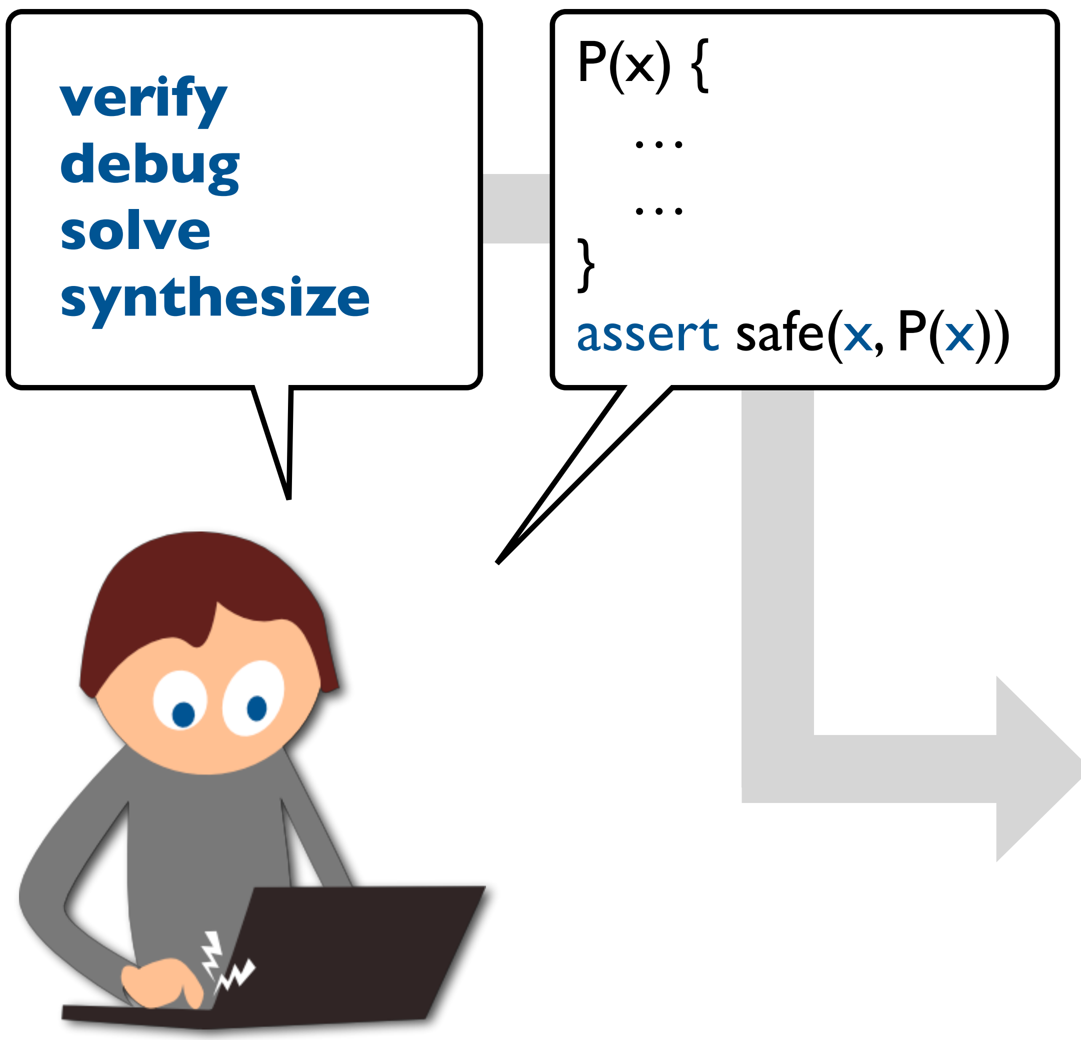
Behind the scenes: symbolic virtual machine
How Rosette works so you don’t have to.



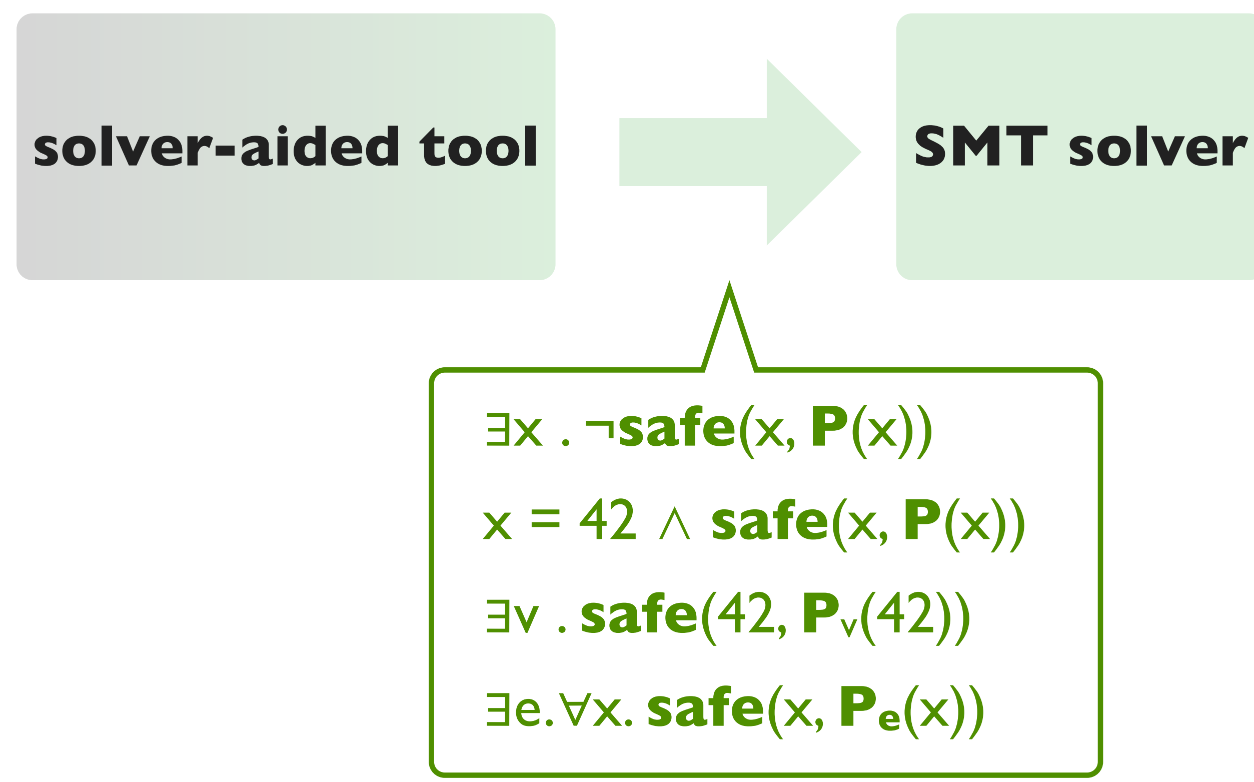
A last look: a few recent applications
Cool tools built with Rosette!

The classic (hard) way to build a tool

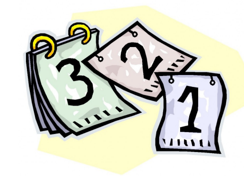


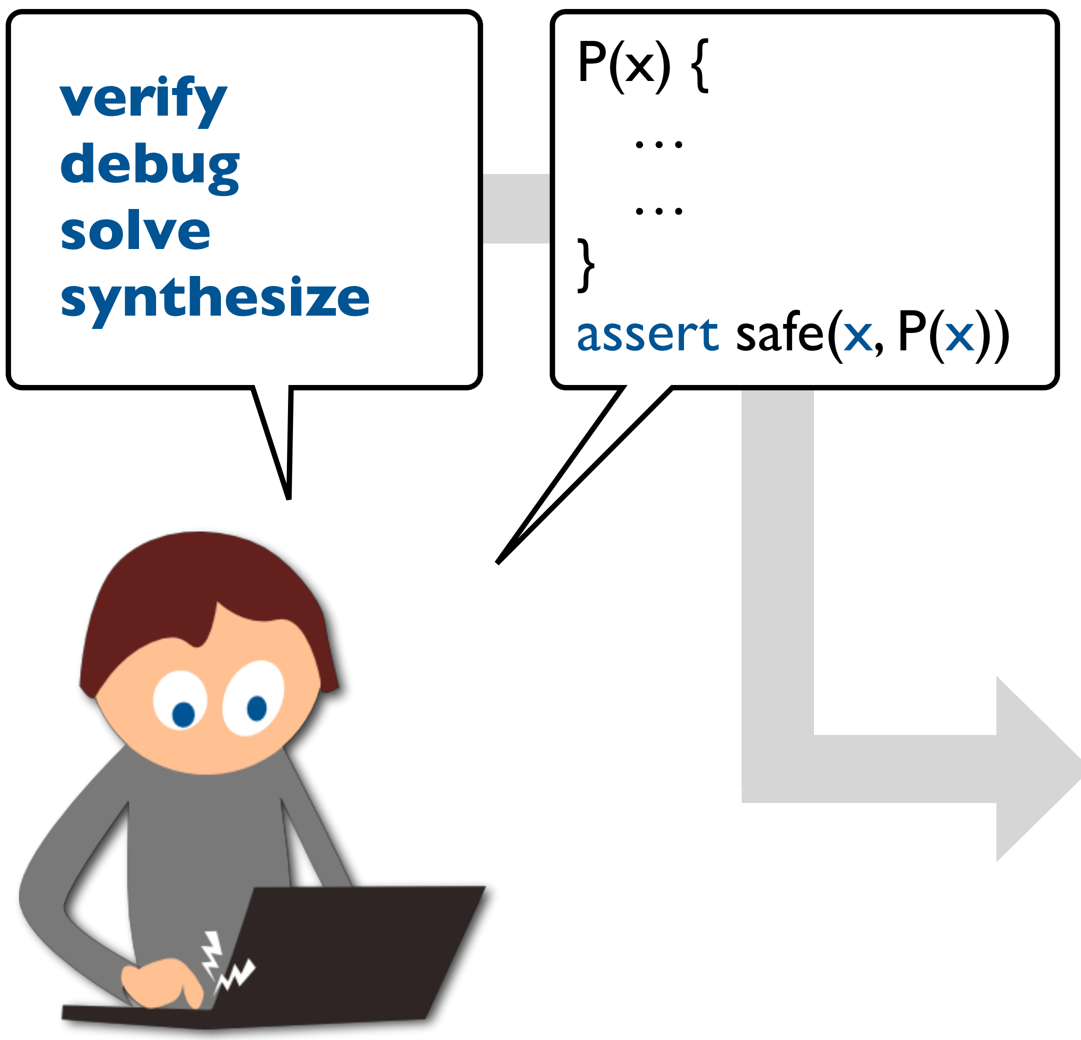


Recall the solver-aided programming tool chain: the tool reduces a query about program behavior to an SMT problem.



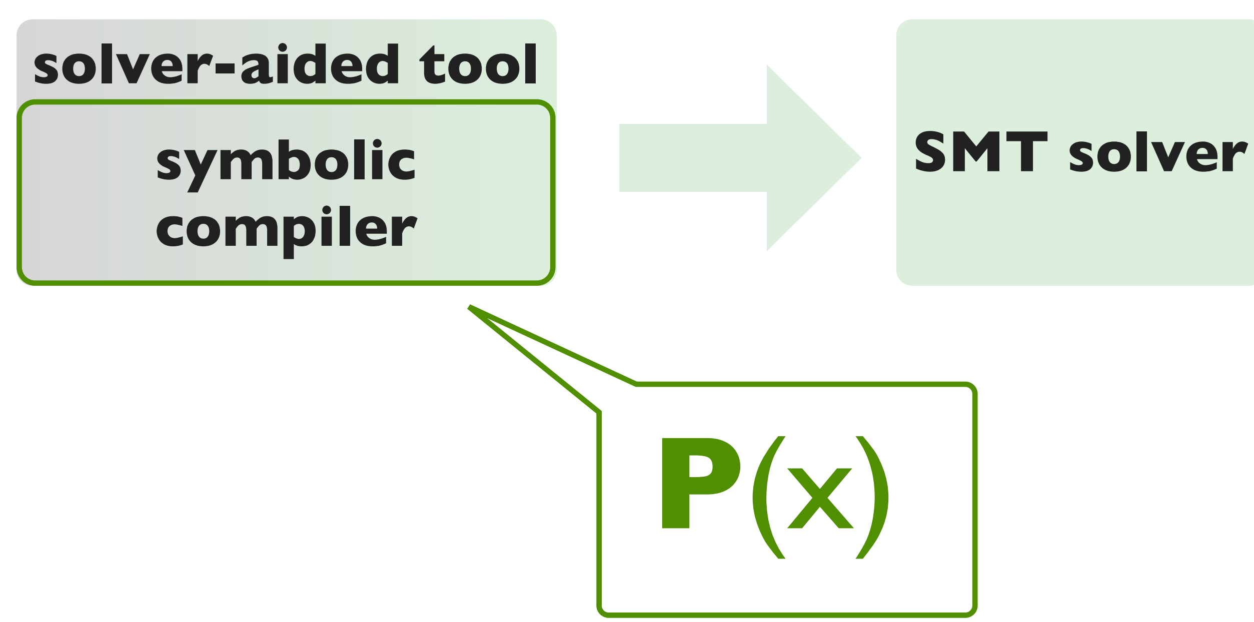
The classic (hard) way to build a tool



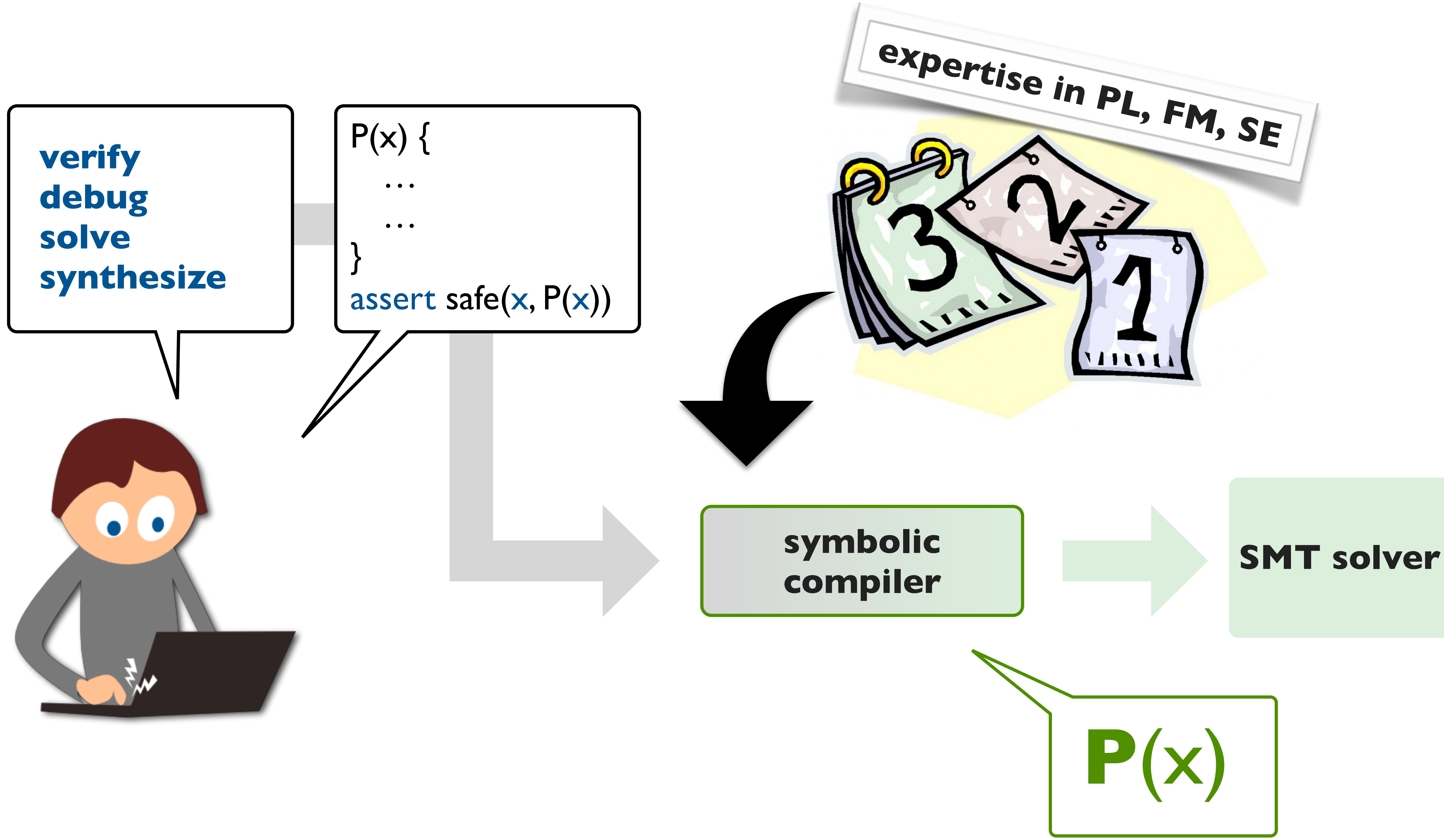


Recall the solver-aided programming tool chain: the tool reduces a query about program behavior to an SMT problem.

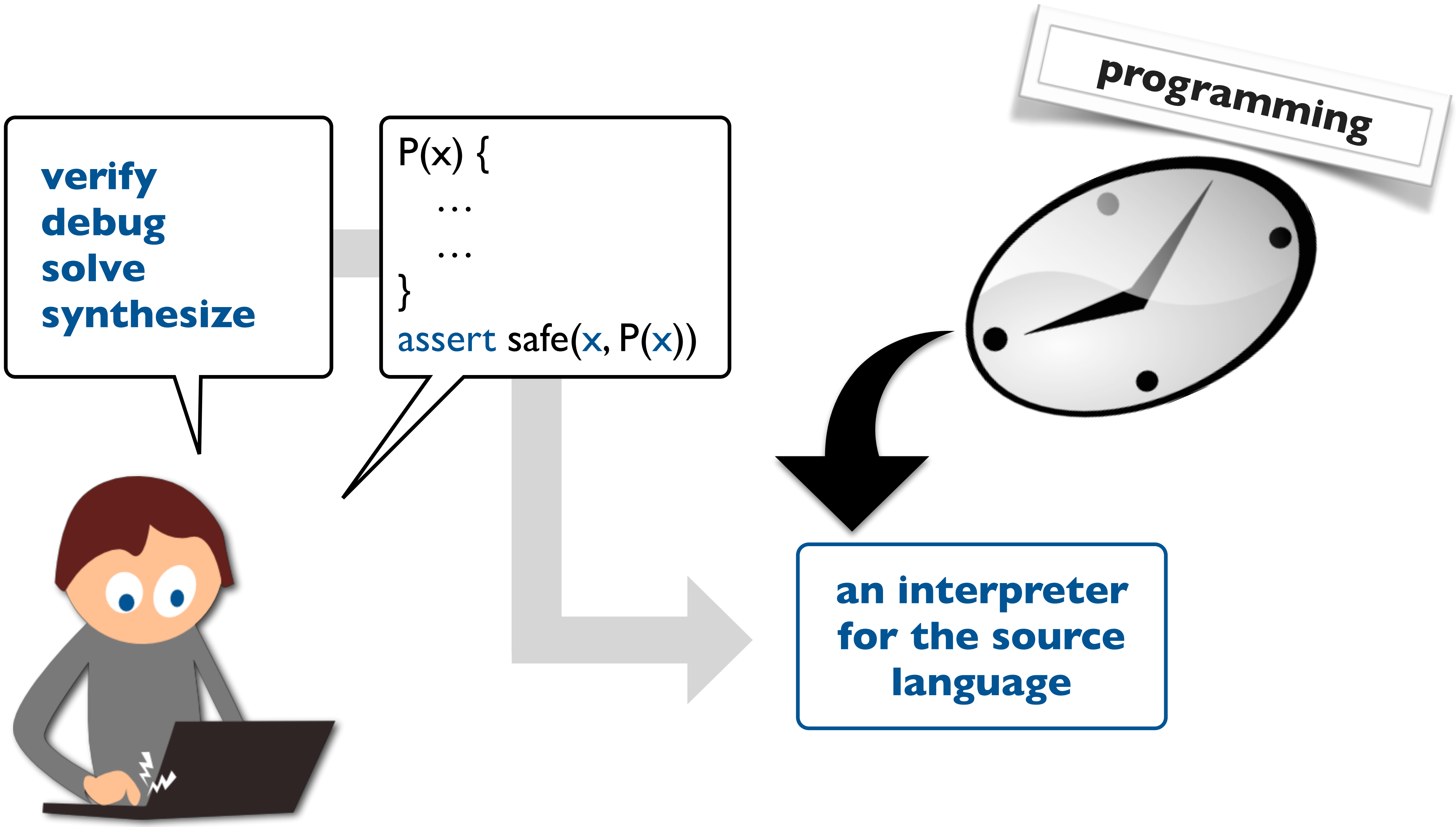
What all queries have in common: they need to translate programs to constraints!



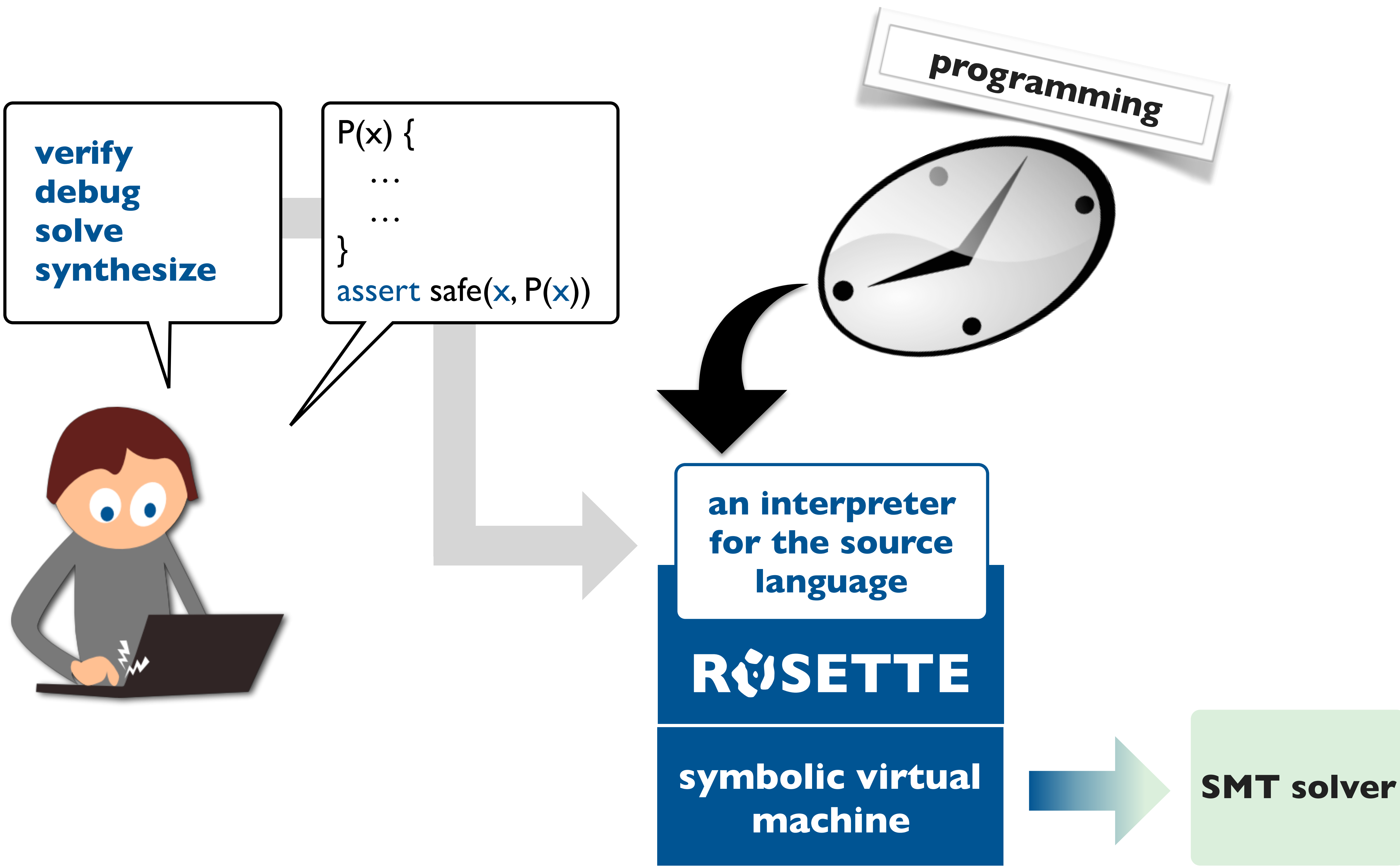
The classic (hard) way to build a tool



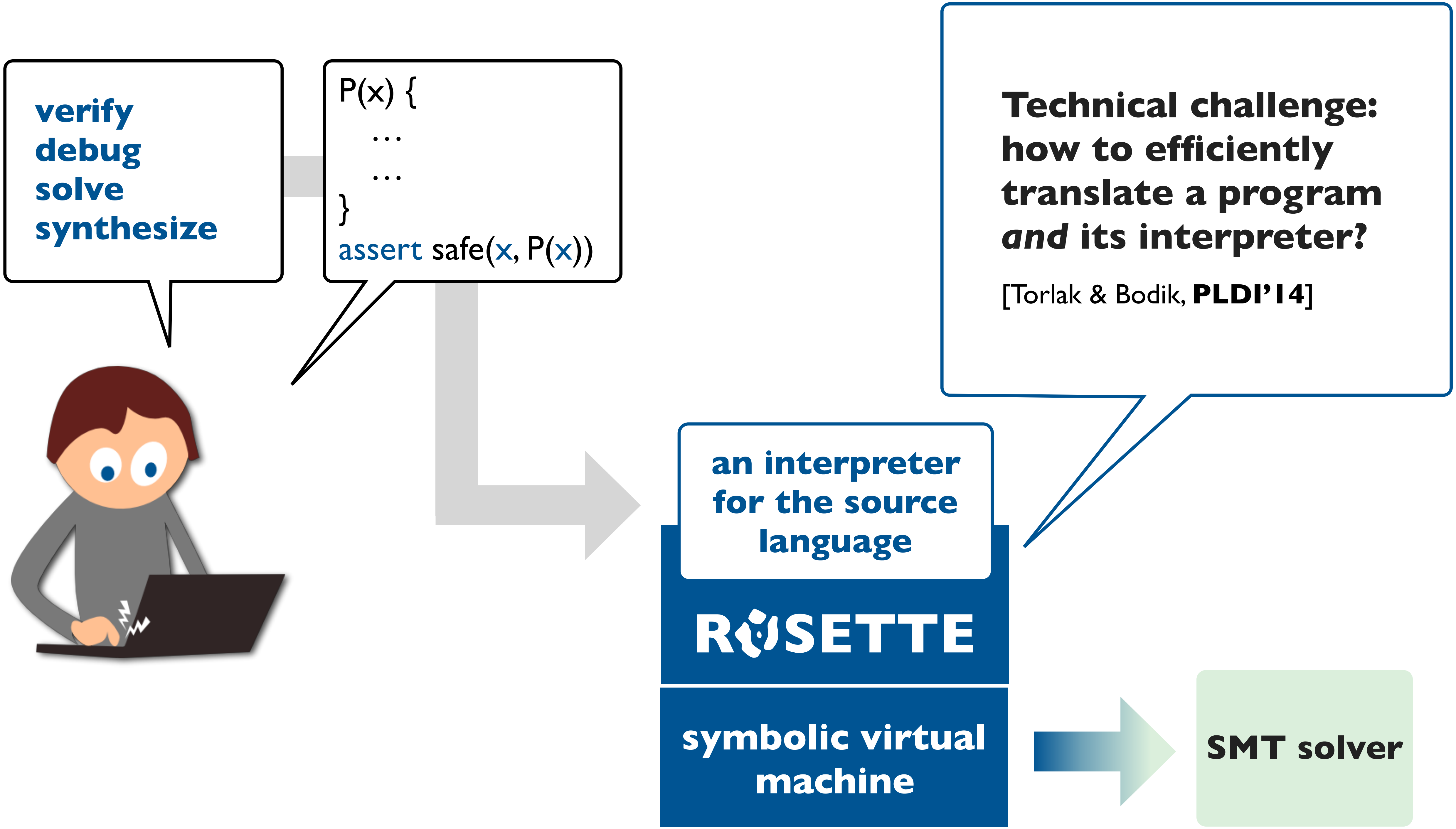
Wanted: an easier way to build tools



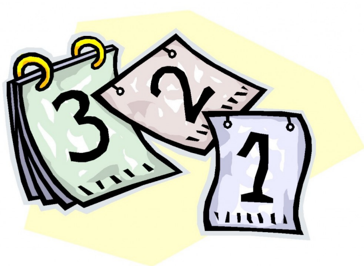
Wanted: an easier way to build tools



Wanted: an easier way to build tools



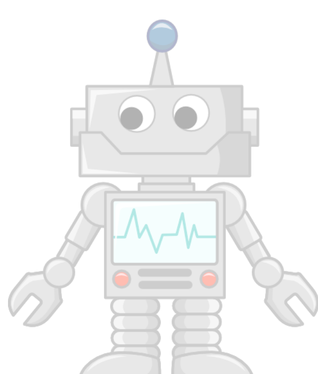
How to build your own solver-aided tool or language



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

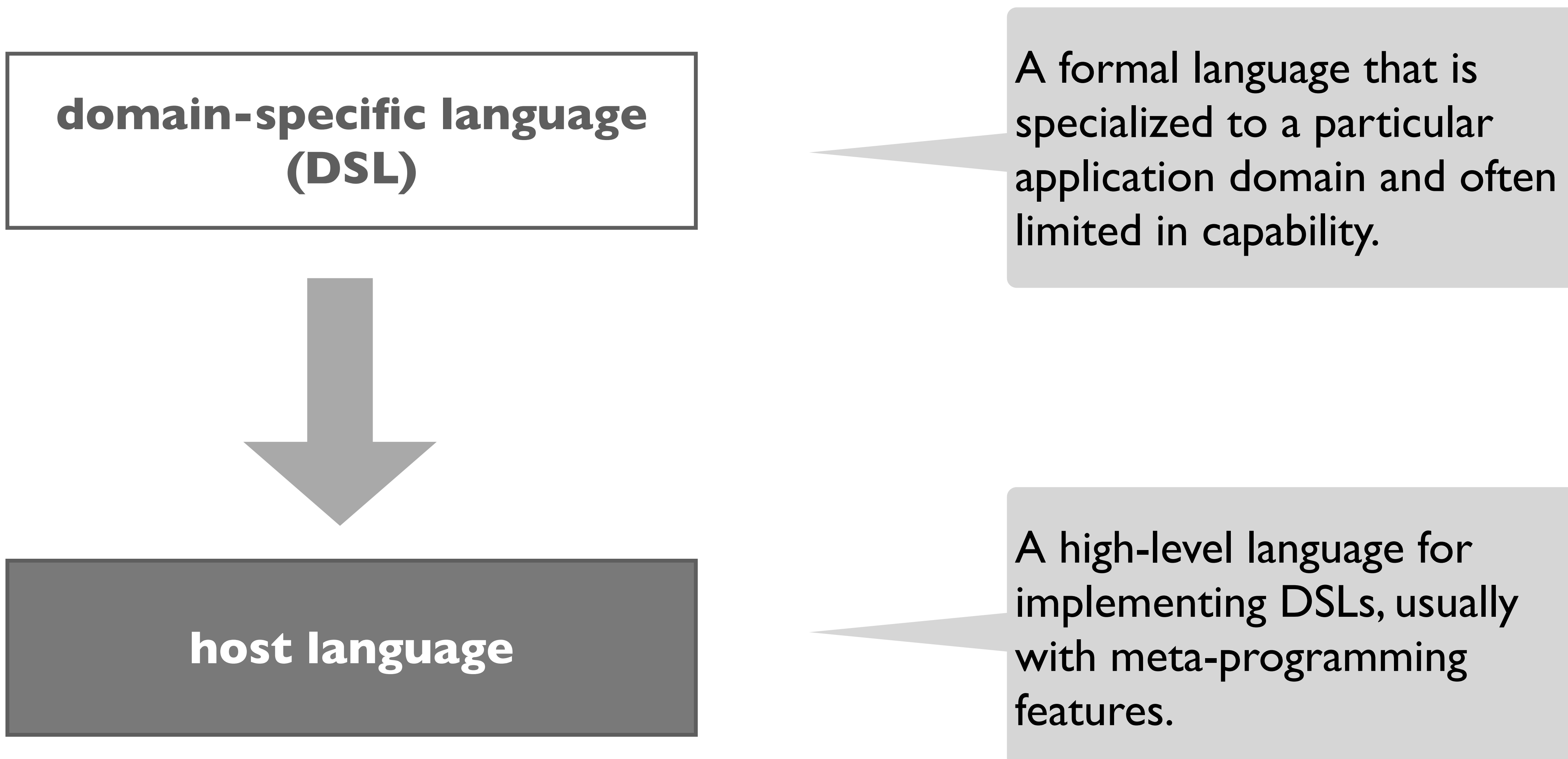
An easier way: tools as languages
How to build tools by stacking layers of languages.

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

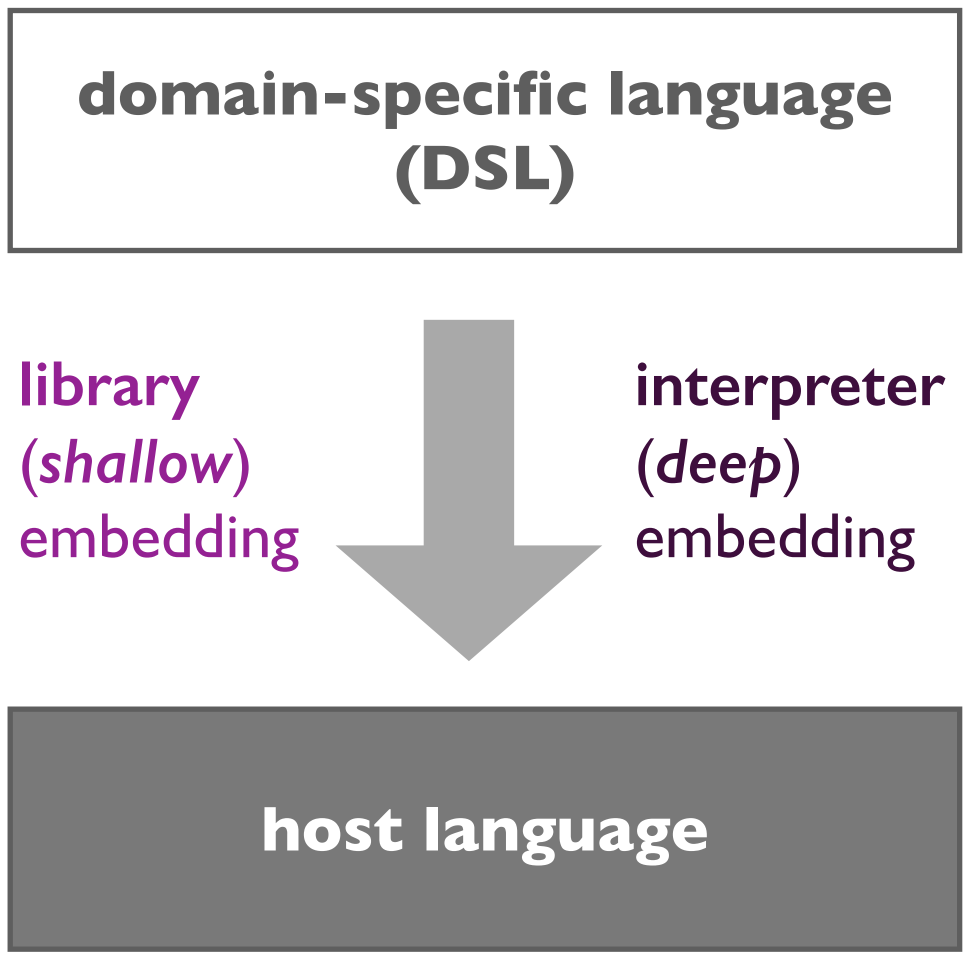


A last look: a few recent applications
Cool tools built with Rosette!

Layers of classic languages: DSLs and hosts



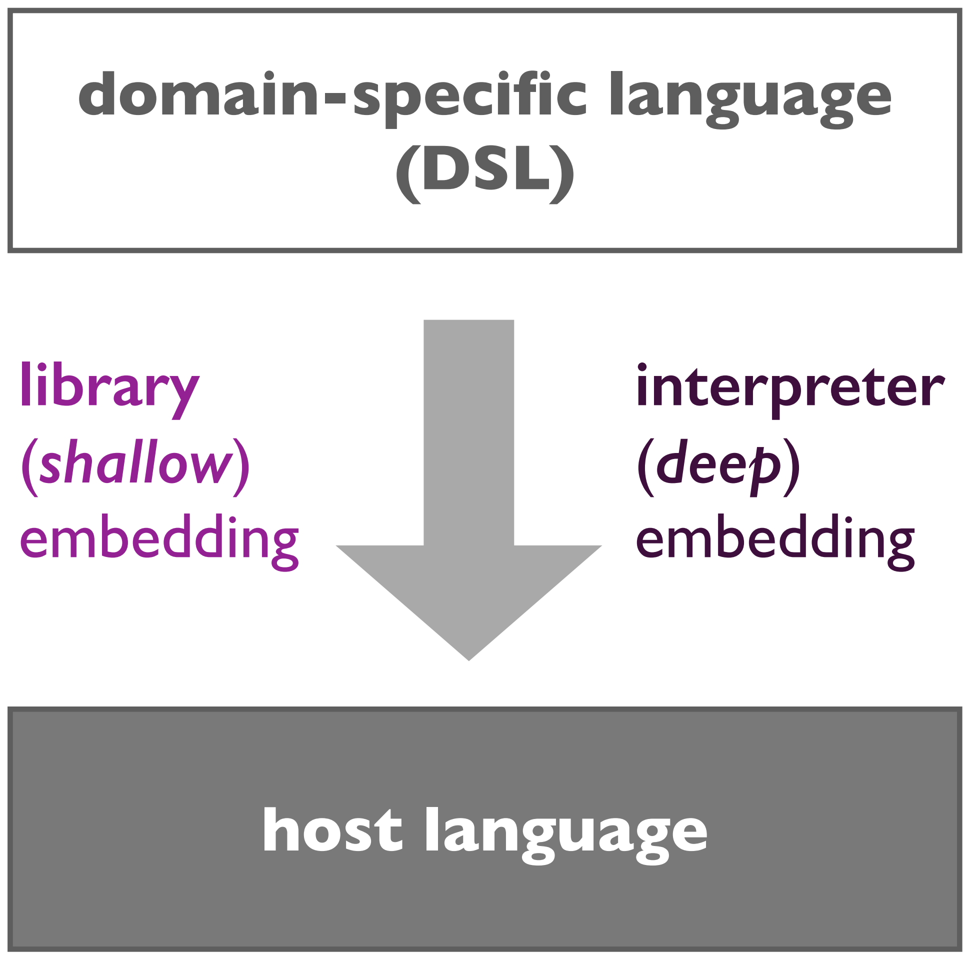
Layers of classic languages: DSLs and hosts



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



artificial intelligence
Church, BLOG

databases
SQL, Datalog

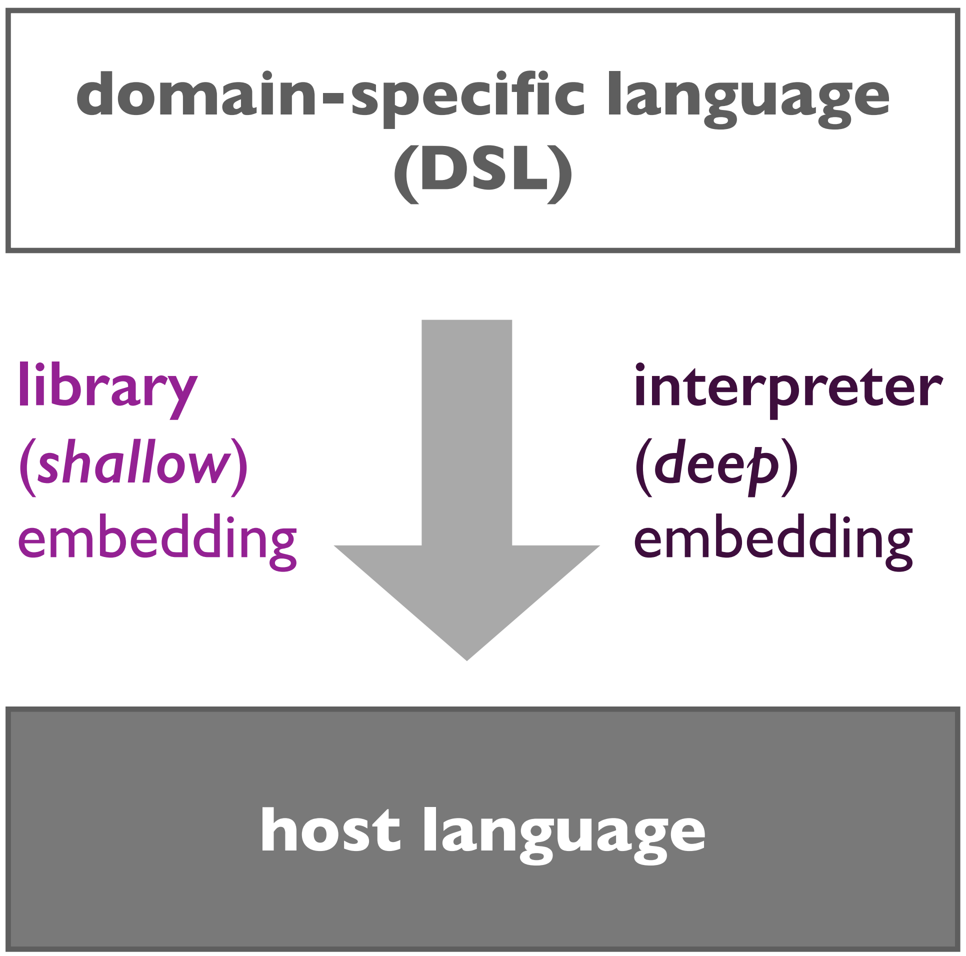
hardware design
Bluespec, Chisel, Verilog, VHDL

math and statistics
Eigen, Matlab, R

layout and visualization
LaTex, dot, dygraphs, D3

Racket, Scala, JavaScript, …

Layers of classic languages: why DSLs?



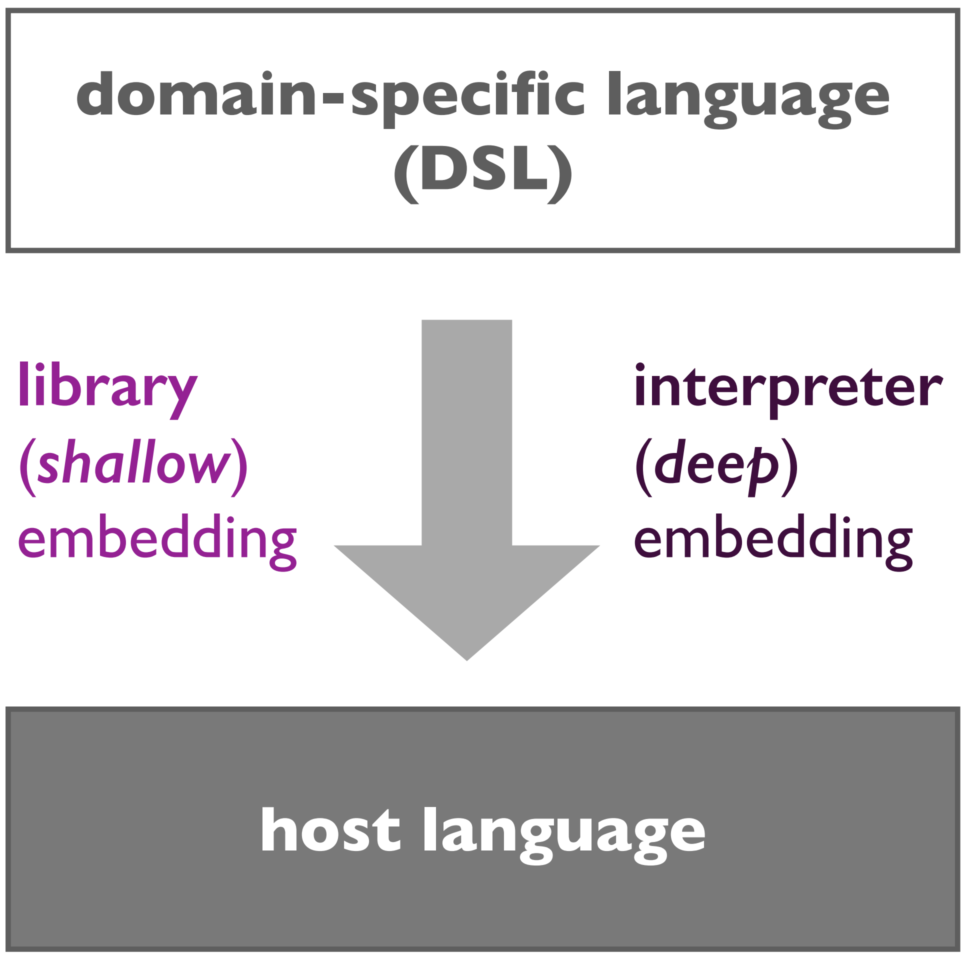
Eigen / Matlab

```
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?



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

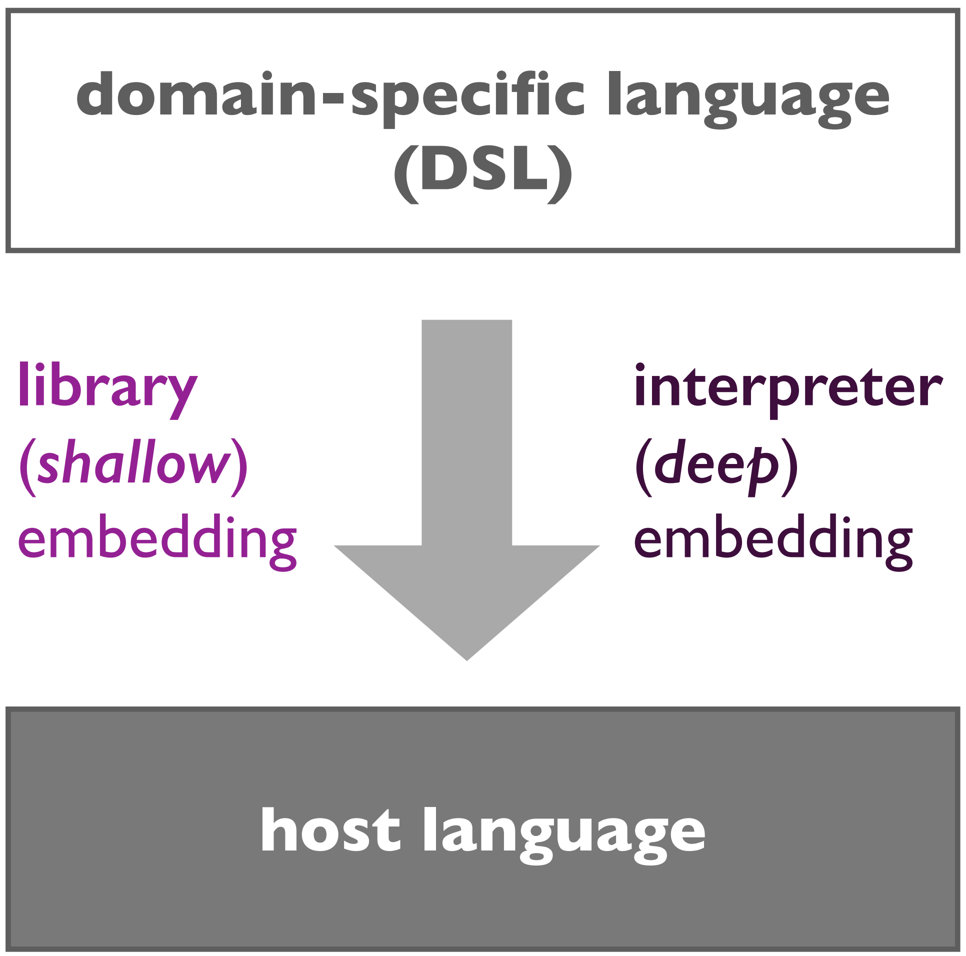
Eigen / Matlab

```
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?



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

Eigen / Matlab

```
C = A * B
```

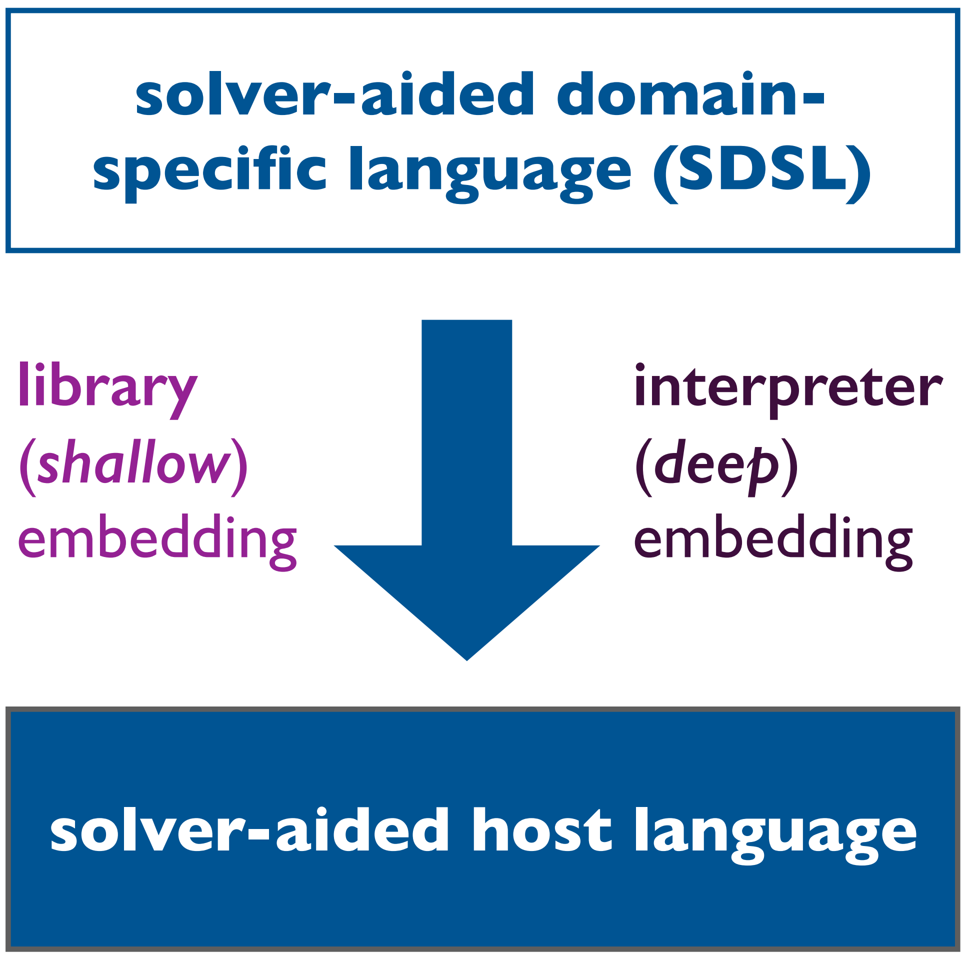
[associativity]

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



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)

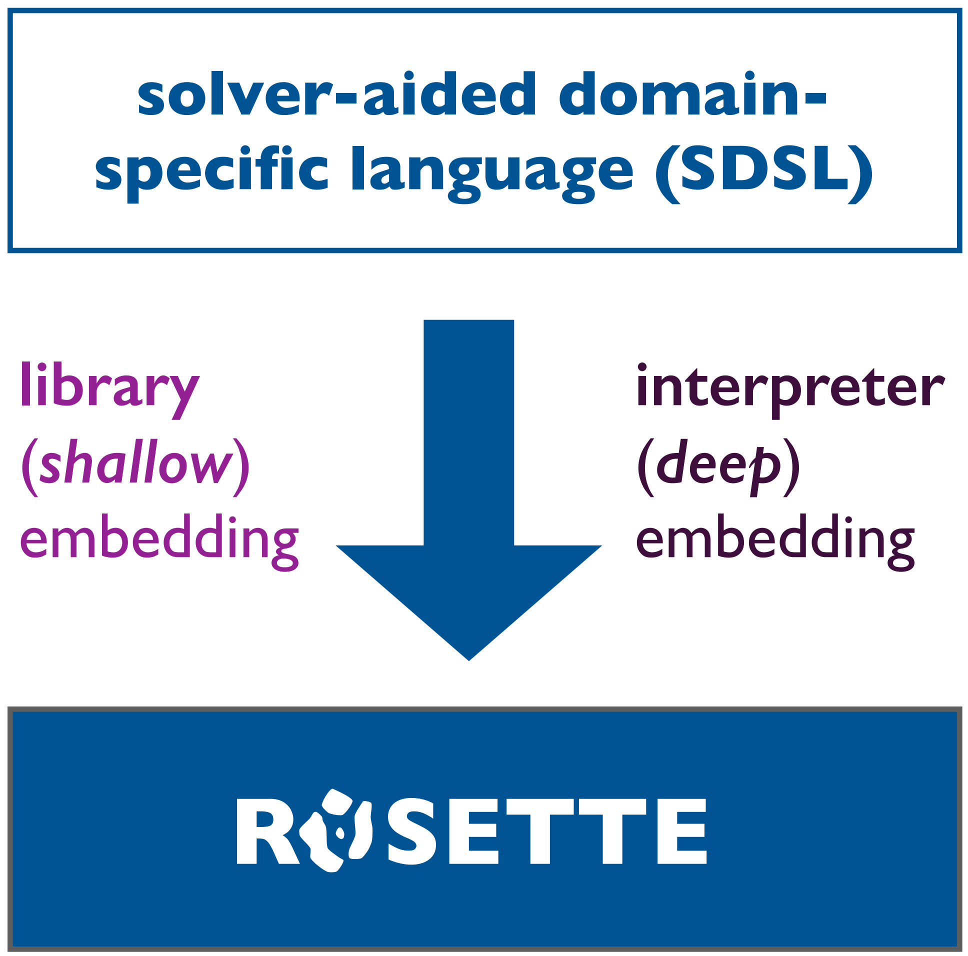
computer architecture
MemSynth (PLDI'17)

databases
Cosette (CIDR'17)

radiation therapy control
Neutrons (CAV'16)

… and more

Layers of solver-aided languages: tools as SDSLs



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

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
```

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.

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)
```

parse

```
(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

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)
```

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	

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 = 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	

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 = 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

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	

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 = 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	0
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 = 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	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 = bvsge(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 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

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

```
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 x y int32?)
(define in (list x y))
(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)
```

query

Creates two fresh symbolic values of type 32-bit integer and binds them to the variables x and y.

```
(define-symbolic x y int32?)
(define in (list x y))
(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)
```

query

Creates two fresh symbolic values of type 32-bit integer and binds them to the variables x and y.

```
(define-symbolic x y int32?)
(define in (list x y))
(verify
  (assert (equal? (interpret bvmax in)
                  (interpret max in))))
```

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

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

Creates two fresh symbolic values of type 32-bit integer and binds them to the variables x and y.

```
(define-symbolic x y int32?)
(define in (list x y))
(verify
  (assert (equal? (interpret bvmax in)
                  (interpret max in))))
```

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

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

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 x y int32?)
(define in (list x y))
(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 x y int32?)
(define in (list x y))
(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 (int32 0) (int32 -2)))
(debug [register?]
  (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 (int32 0) (int32 -2)))
(debug [register?]
  (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 x y int32?)
(define in (list x y))
(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)
```

query

```
(define-symbolic x y int32?)
(define in (list x y))
(synthesize
  #:forall in
  #:guarantee
  (assert (equal? (interpret bvmax in)
                  (interpret max in)))))
```

How to build your own solver-aided tool or language

The classic (hard) way to build a tool

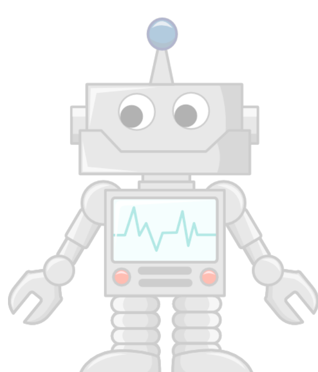
What is hard about building a solver-aided tool?

An easier way: tools as languages

How to build tools by stacking layers of languages.

Behind the scenes: symbolic virtual machine

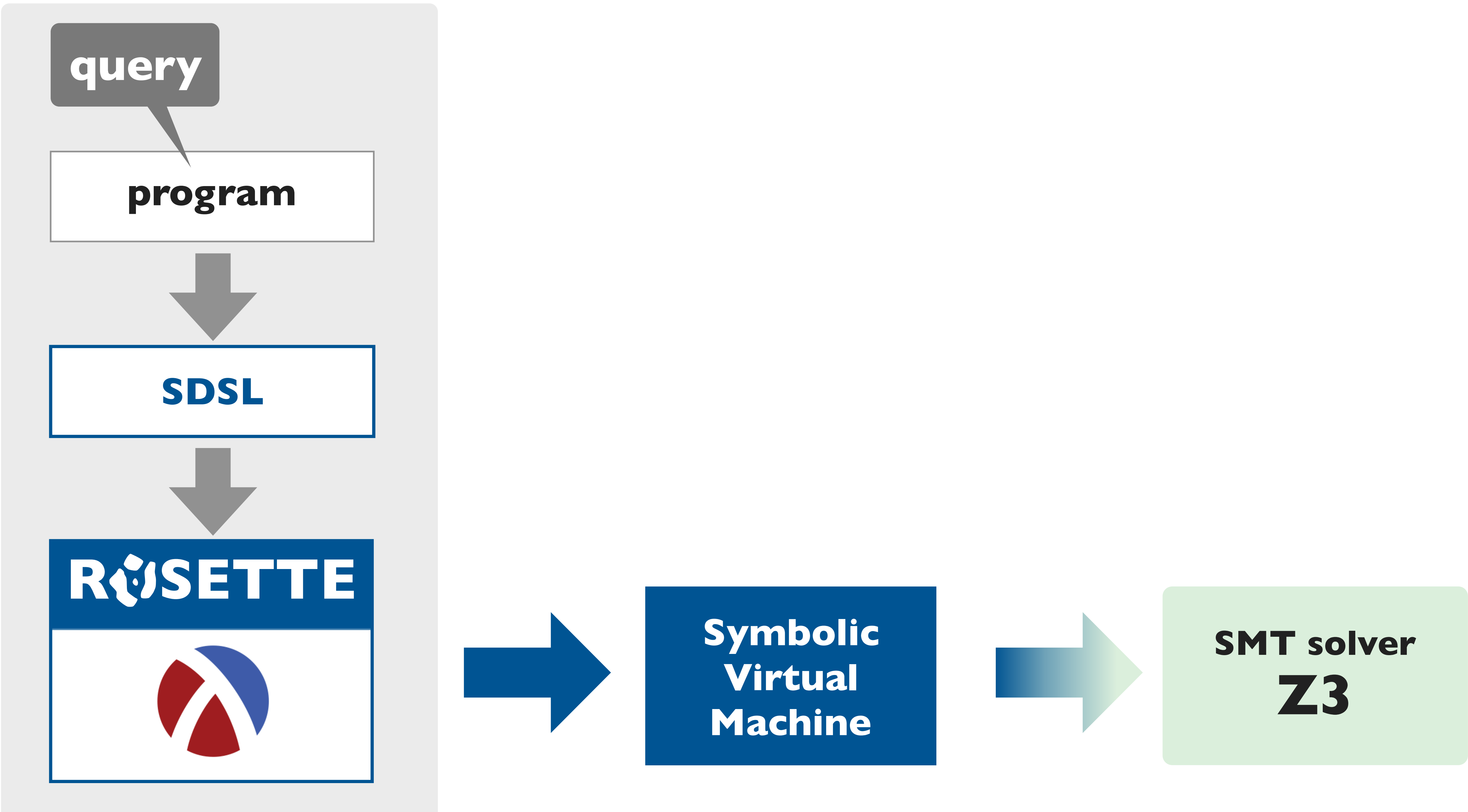
How Rosette works so you don't have to.



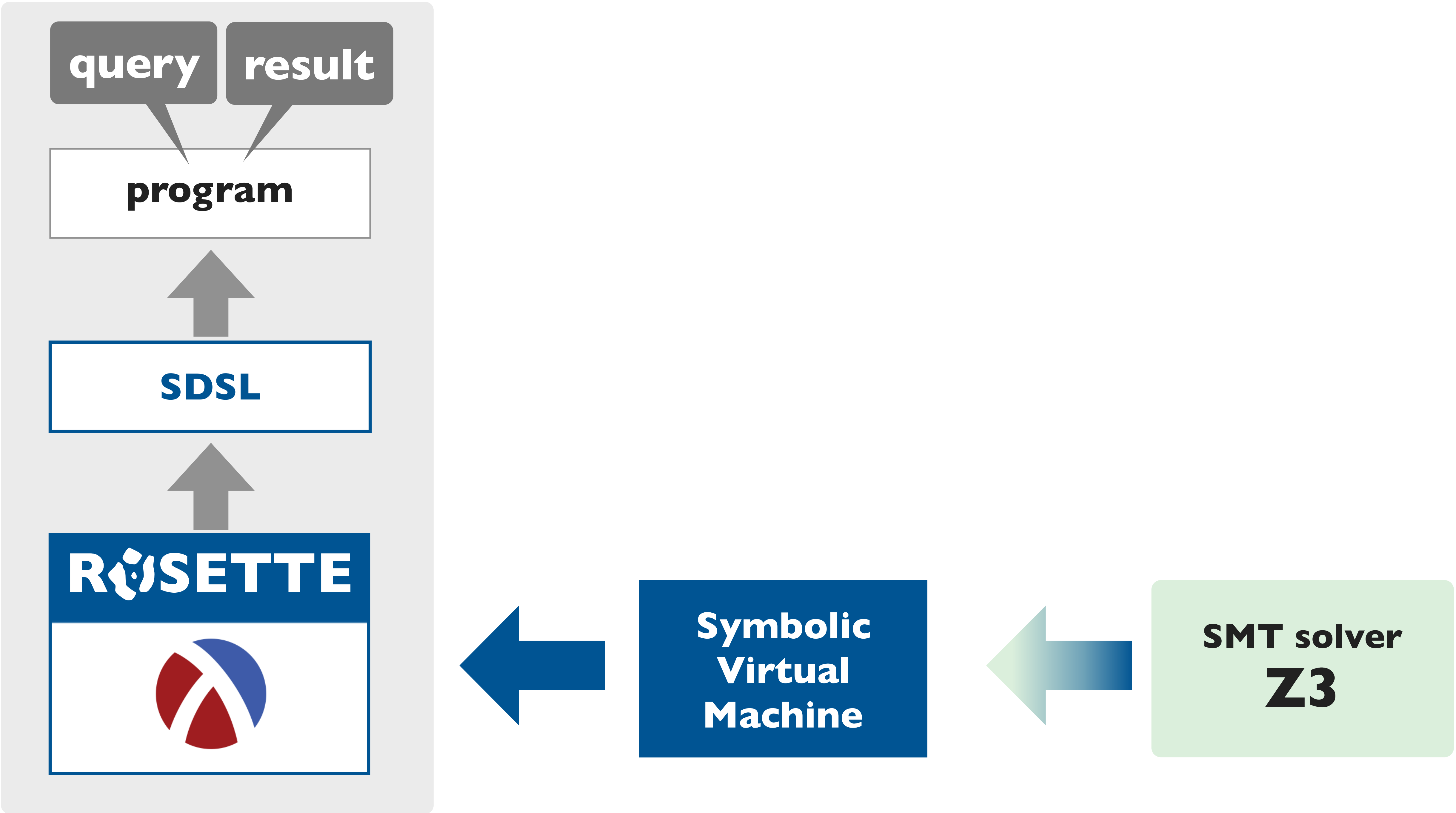
A last look: a few recent applications

Cool tools built with Rosette!

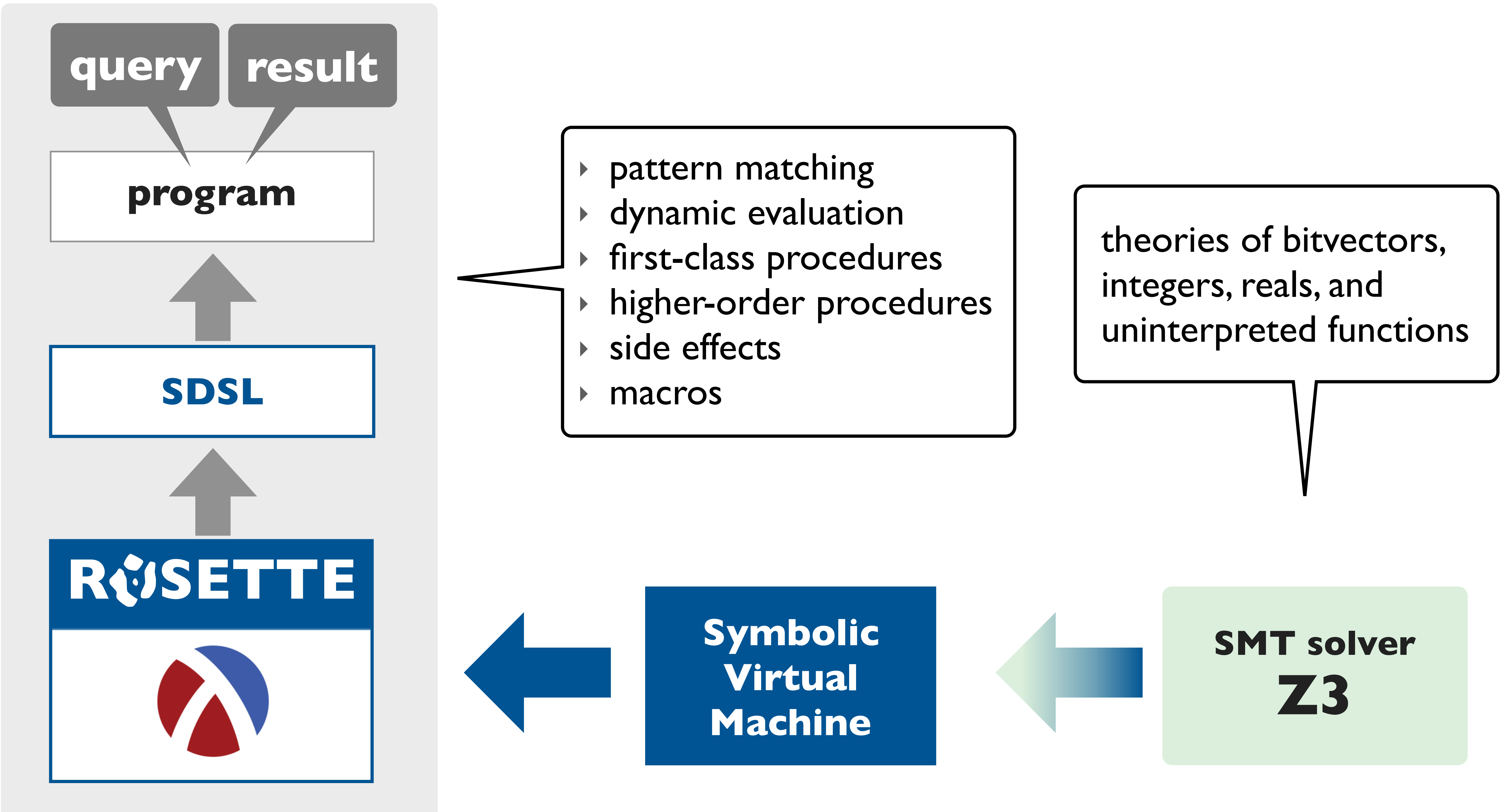
How it all works: a big picture view



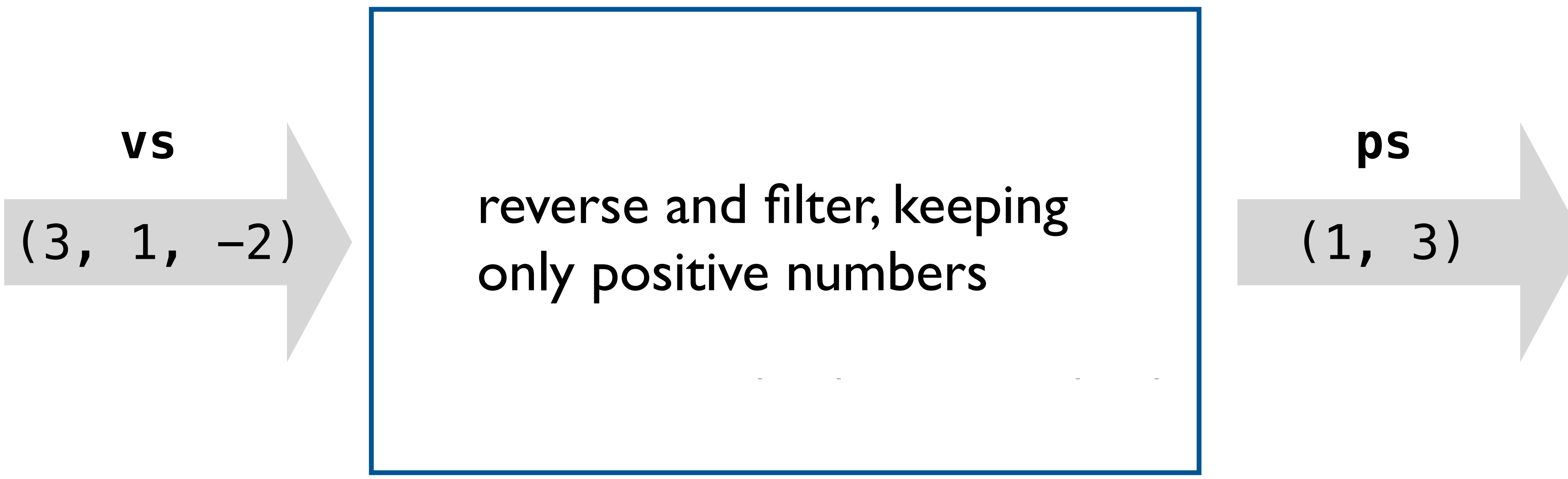
How it all works: a big picture view



How it all works: a big picture view



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)

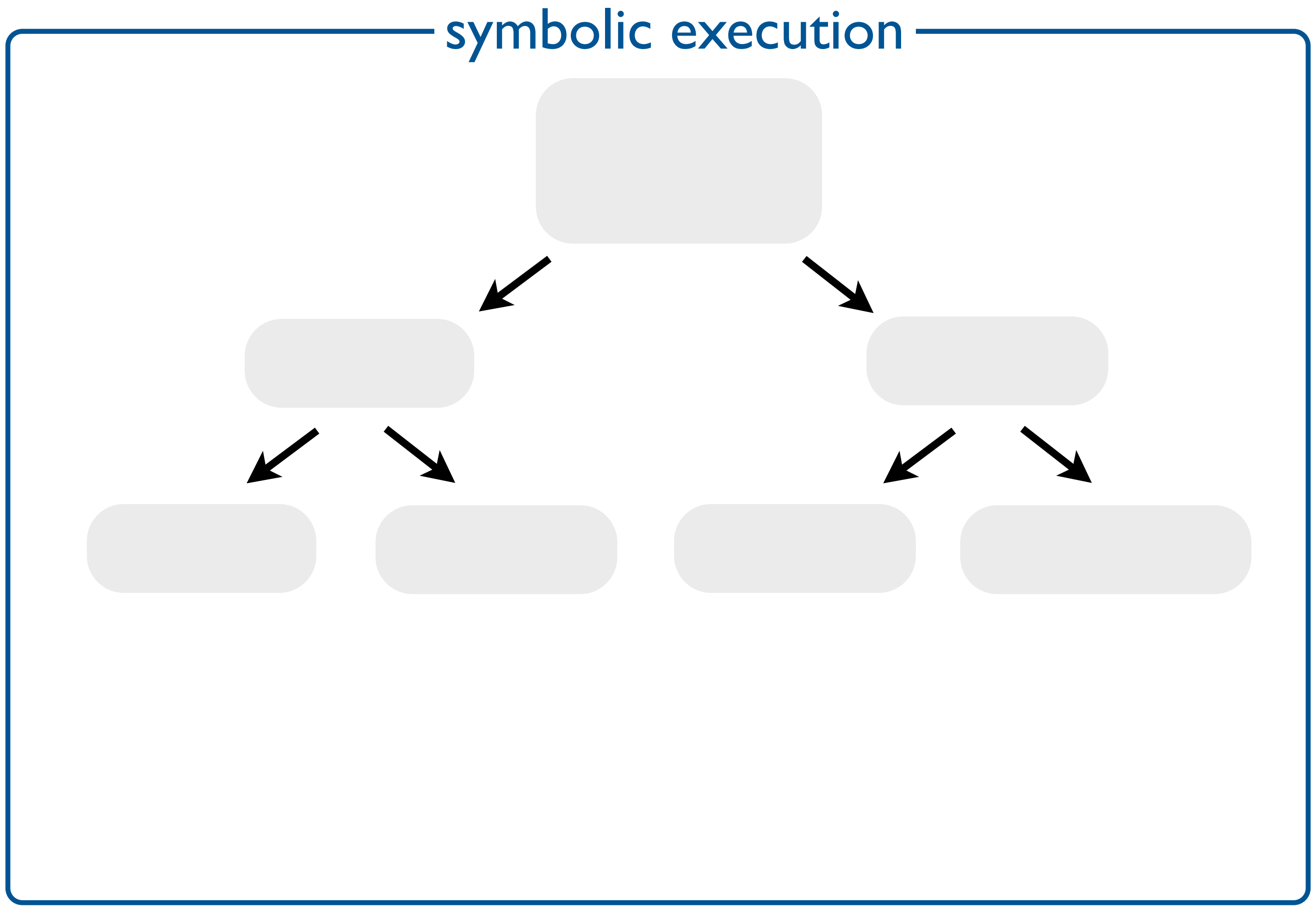
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

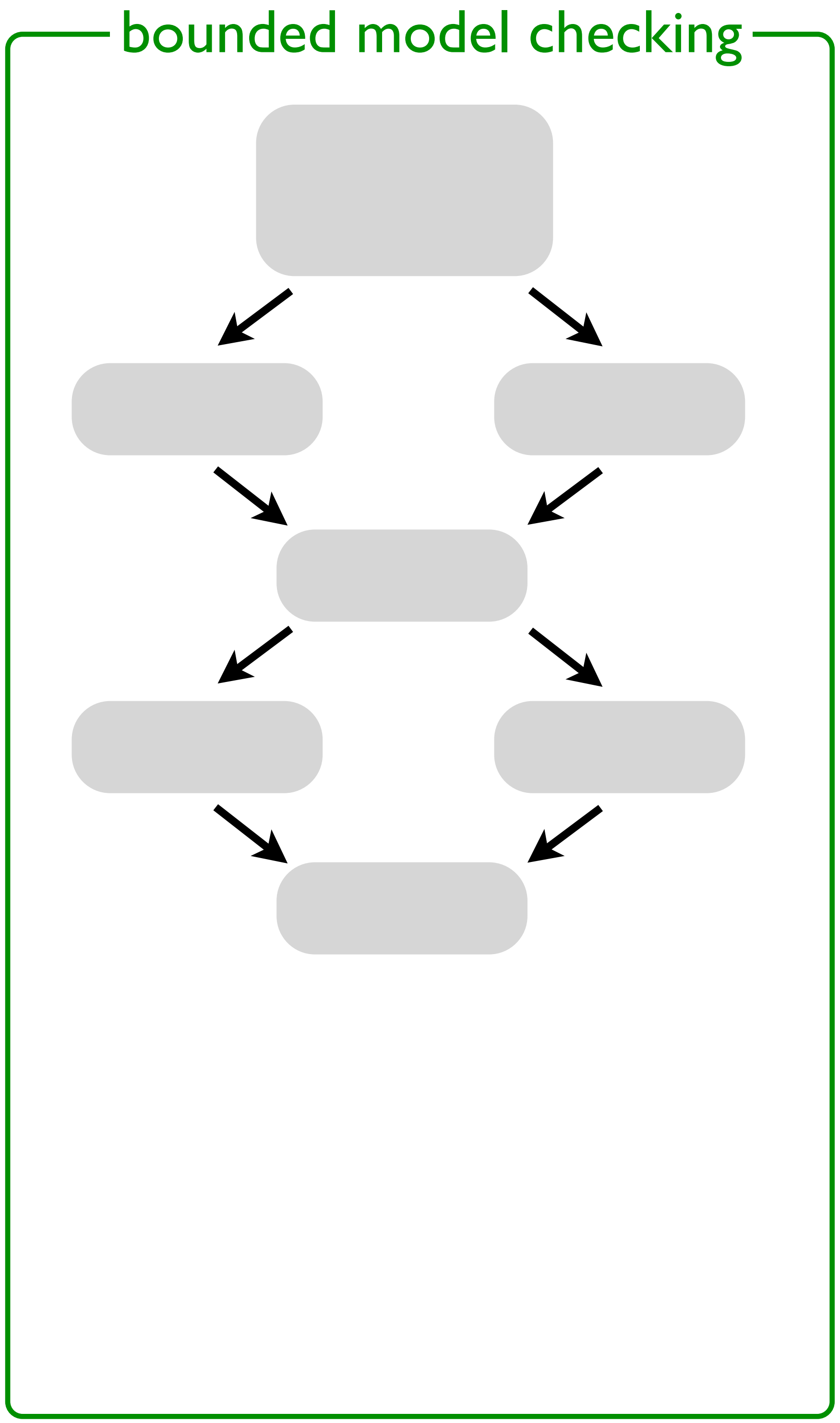
constraints

a>0 ∧ b>0

Design space of precise symbolic encodings

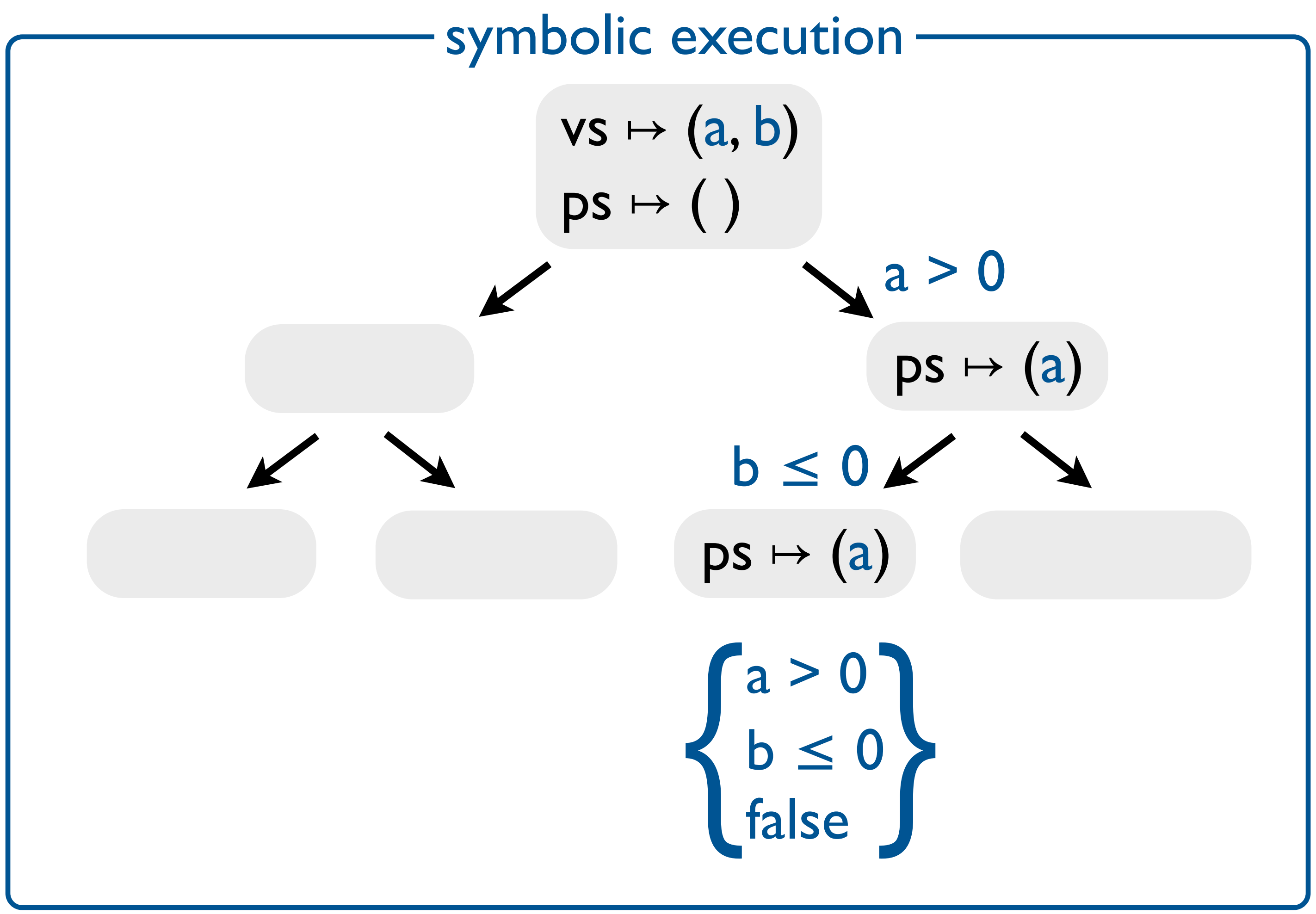
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

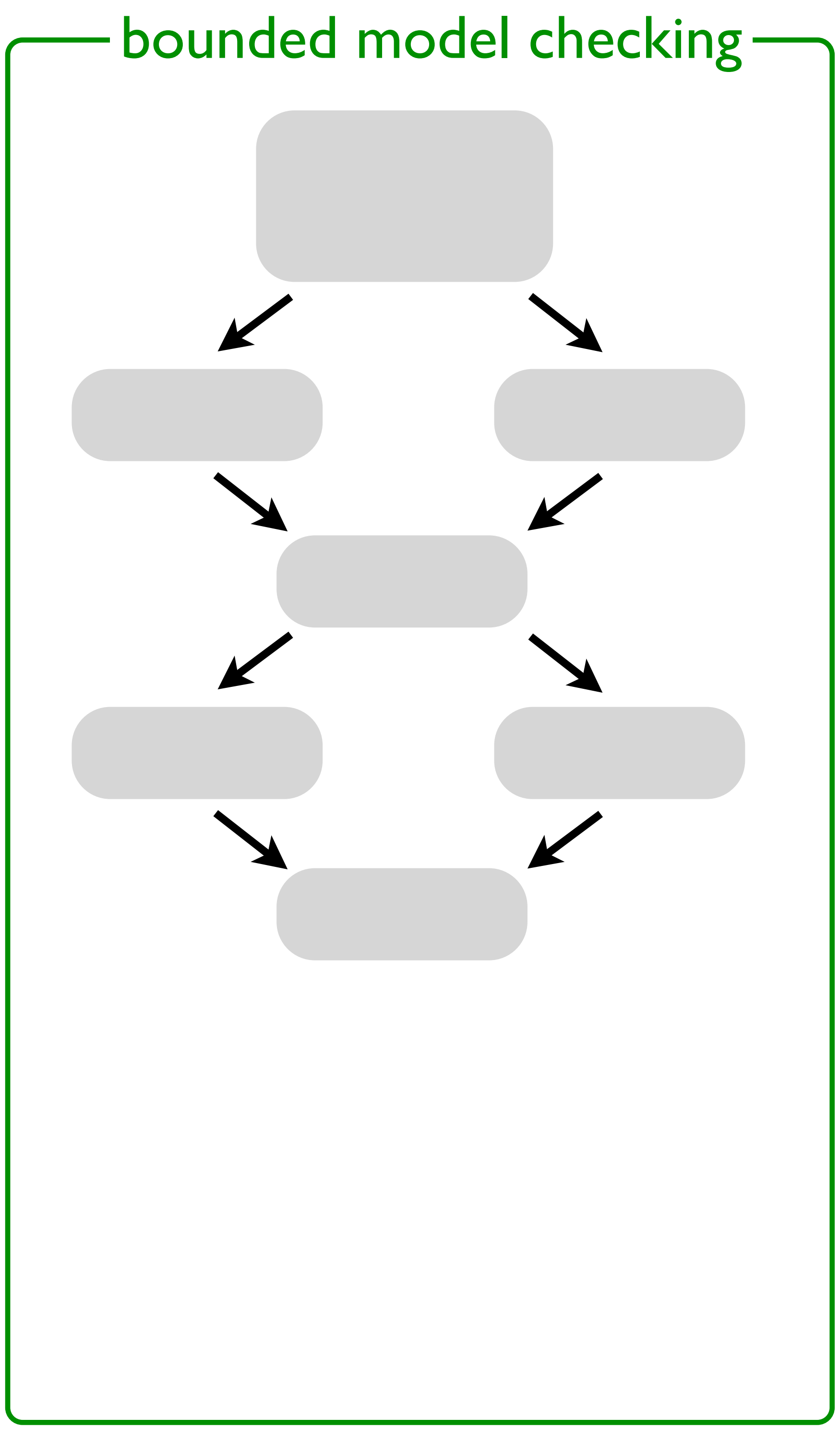




Design space of precise symbolic encodings

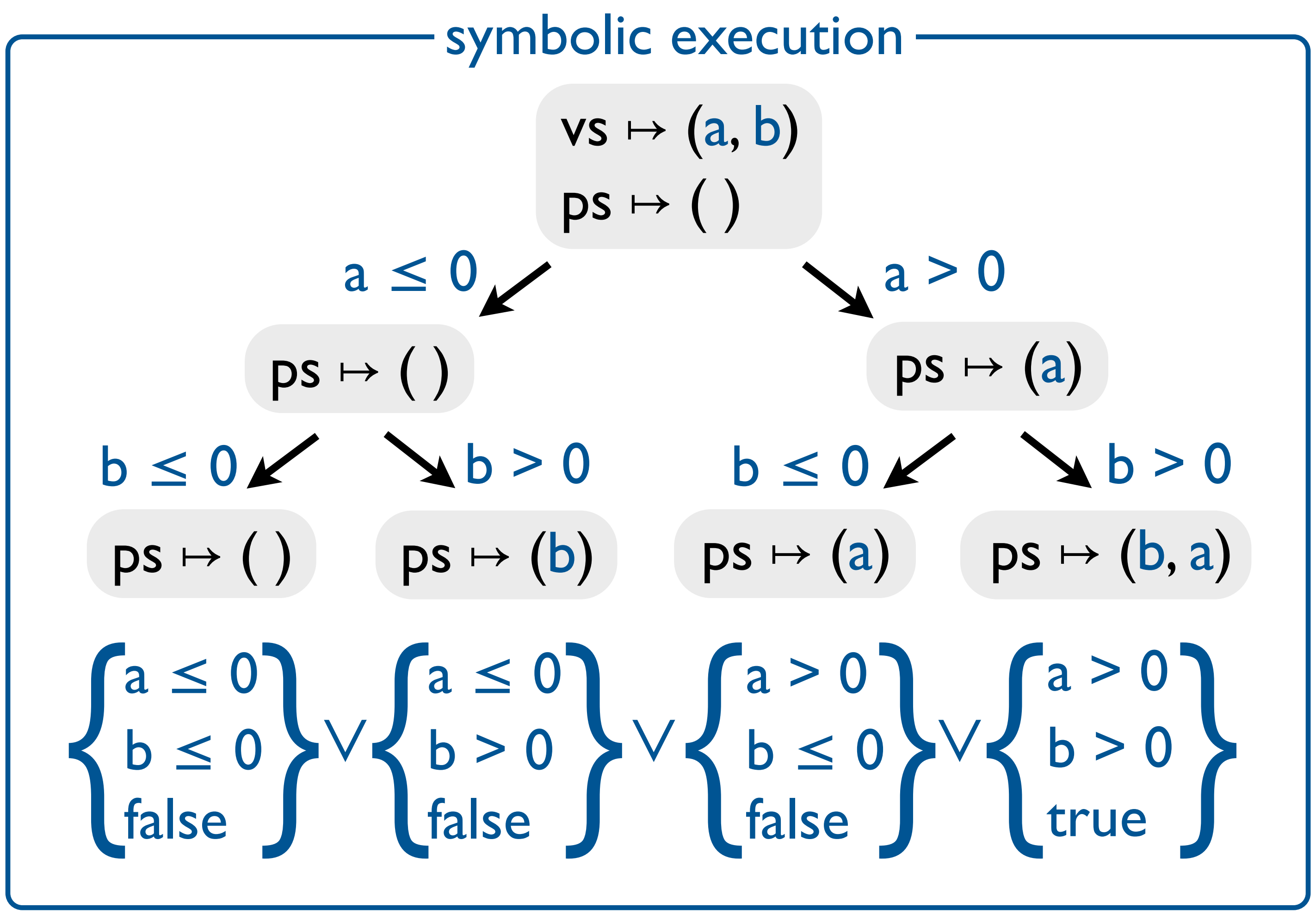
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

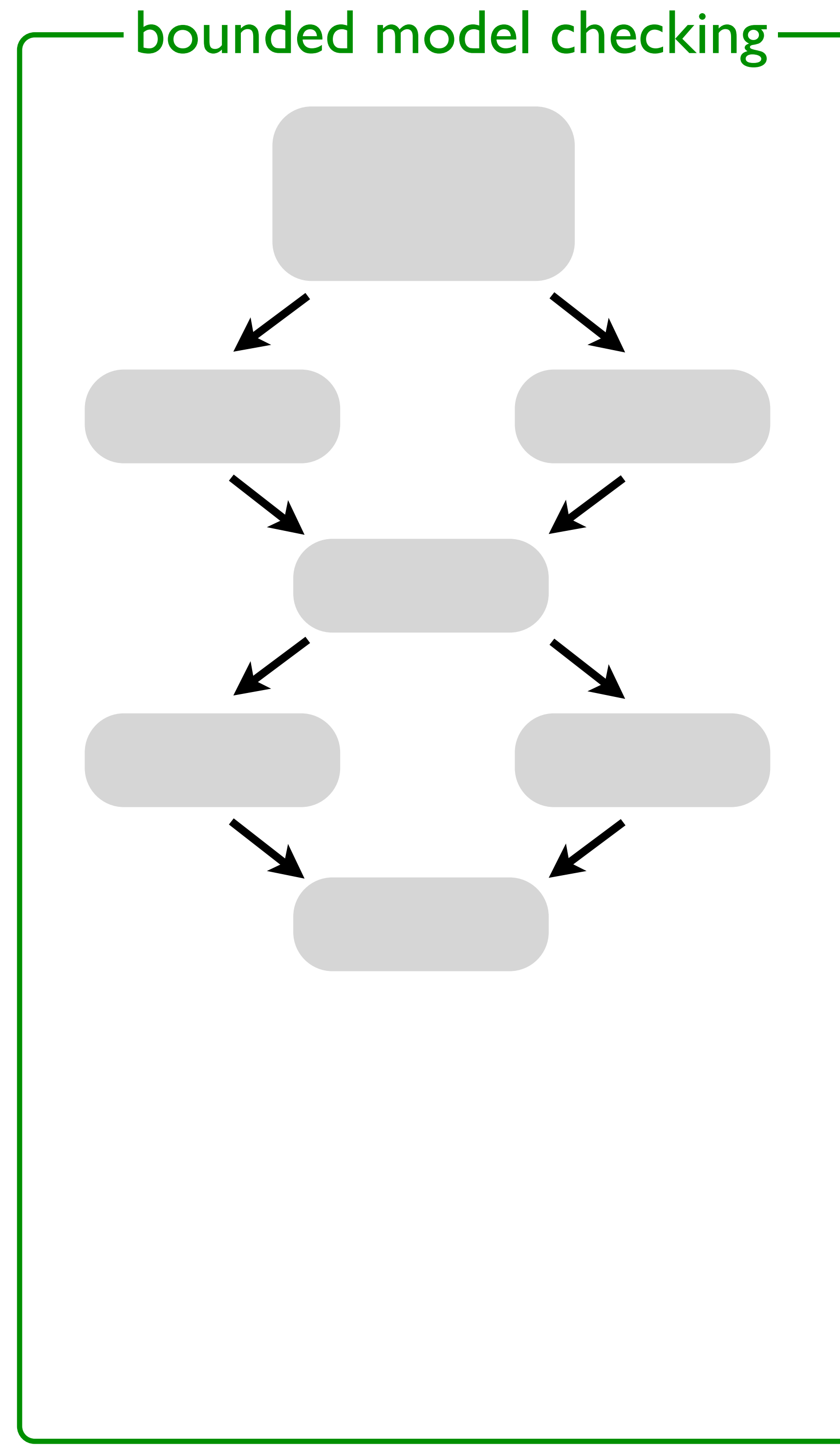




Design space of precise symbolic encodings

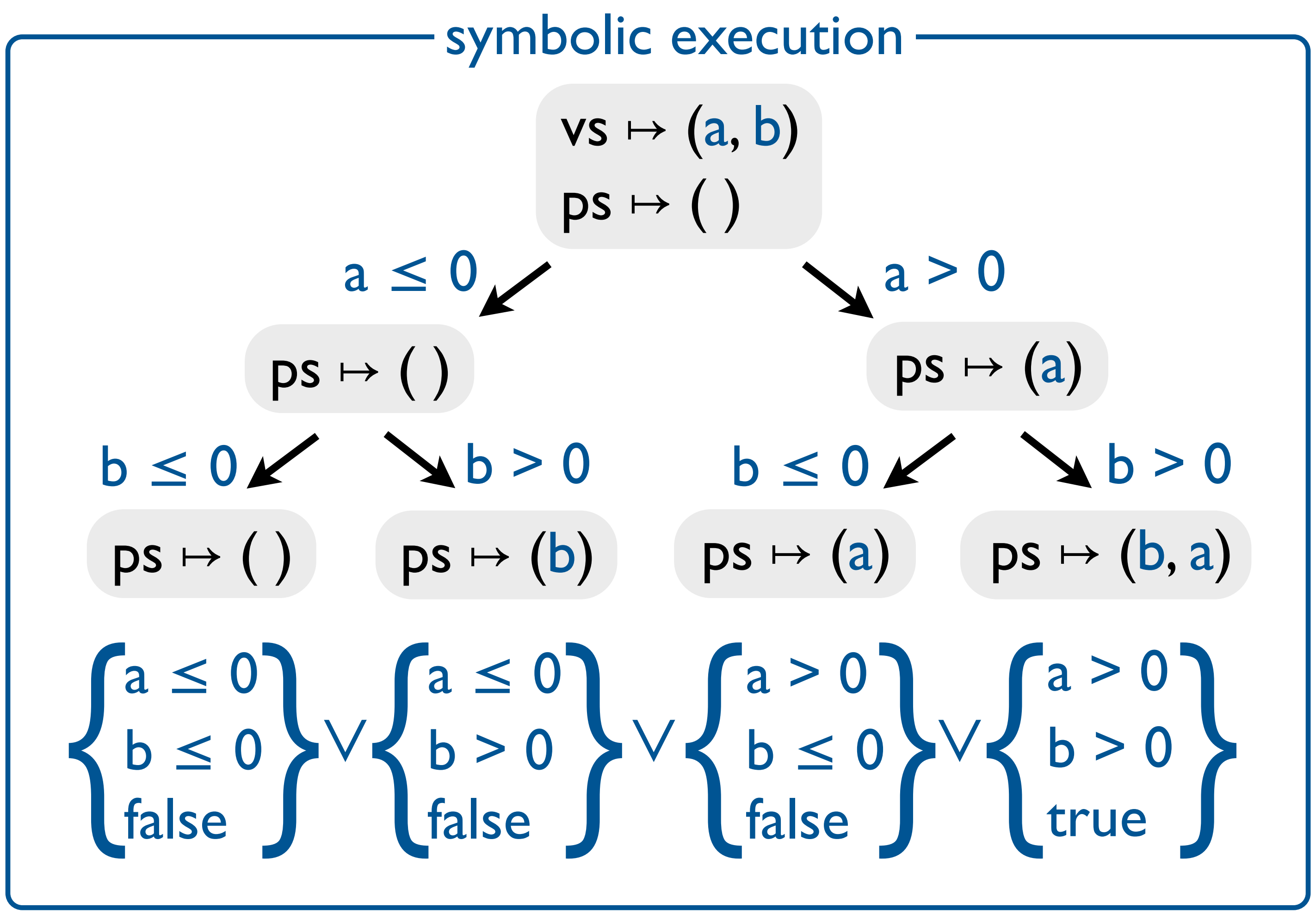
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

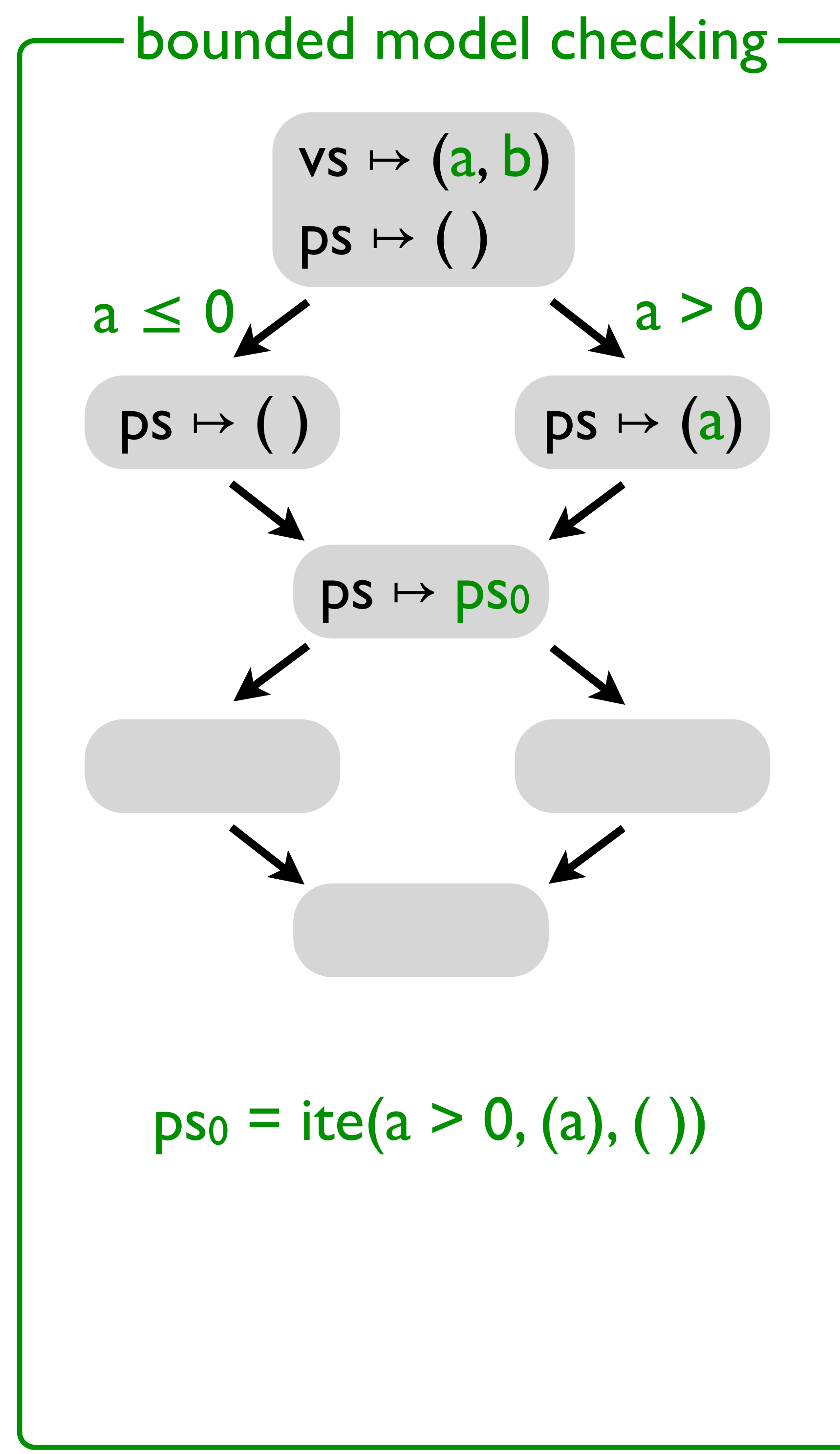




Design space of precise symbolic encodings

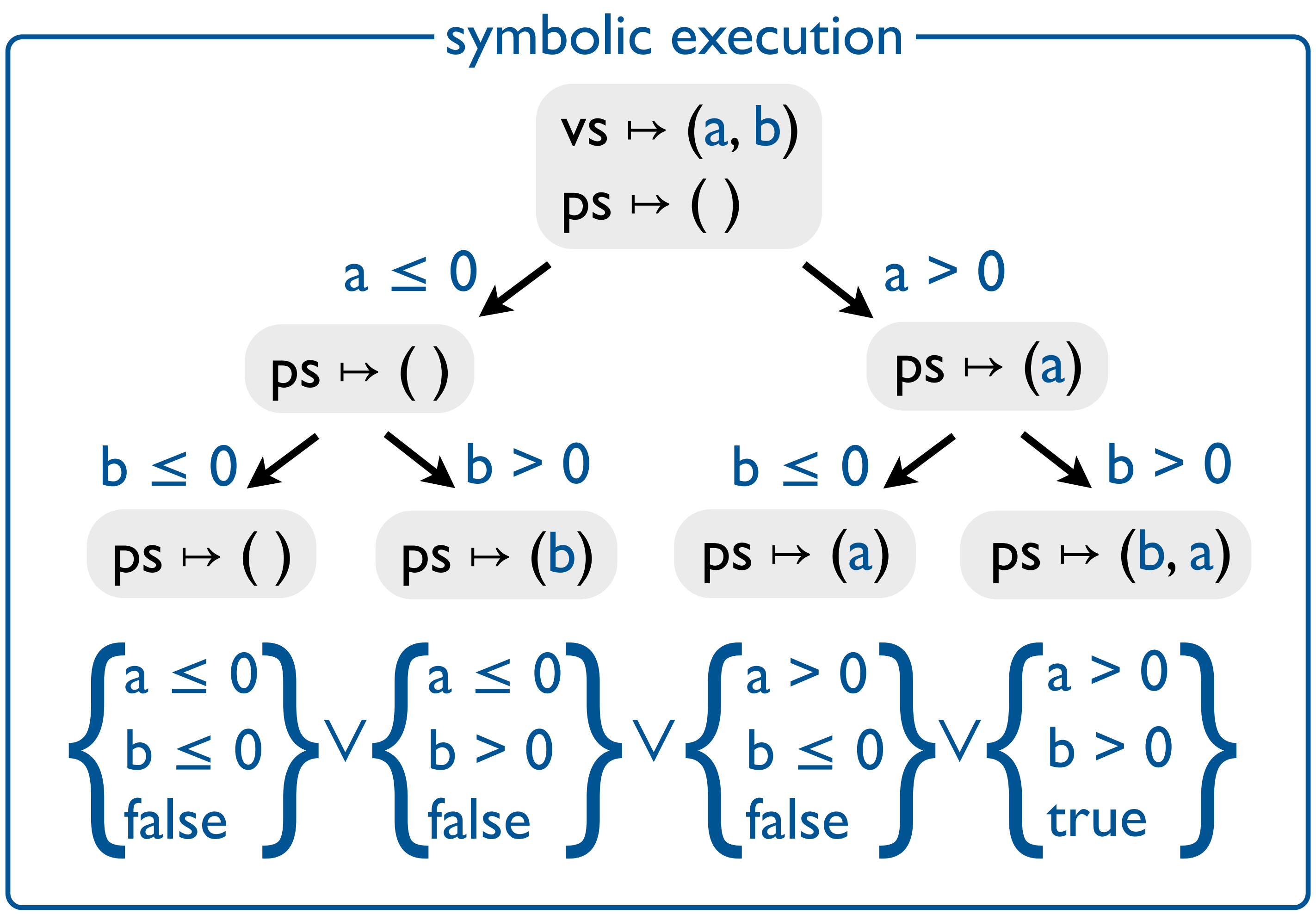
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

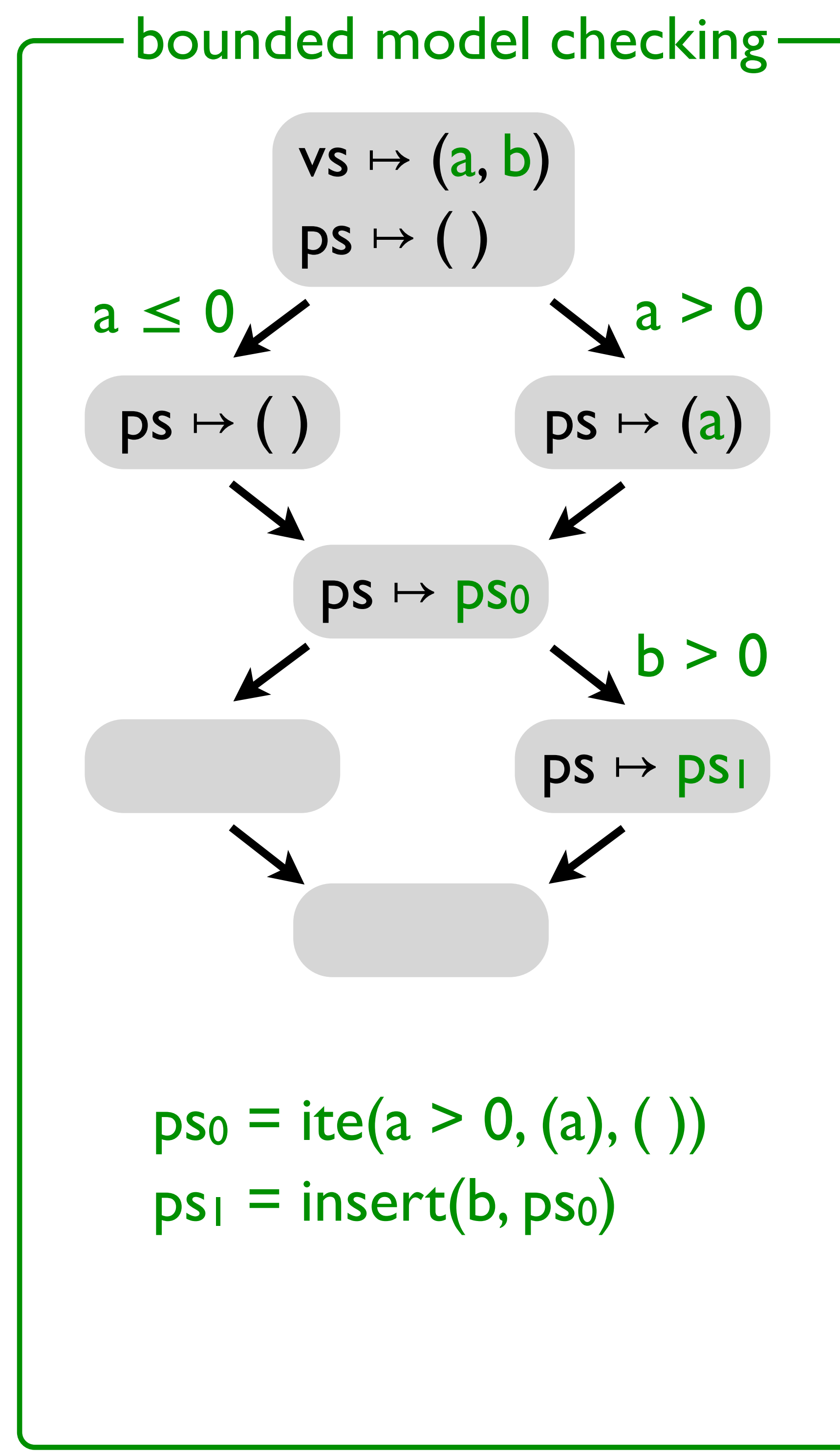




Design space of precise symbolic encodings

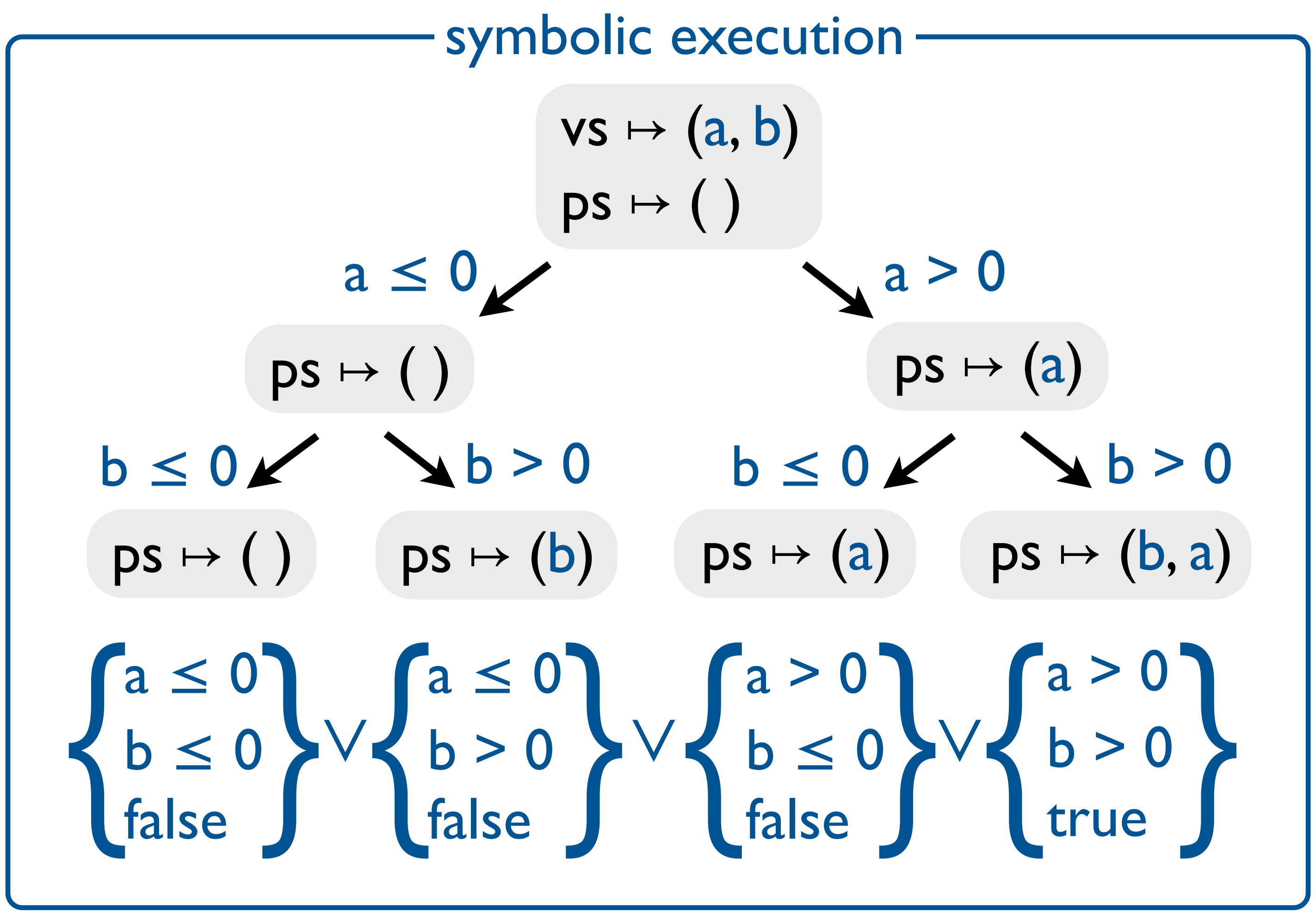
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```

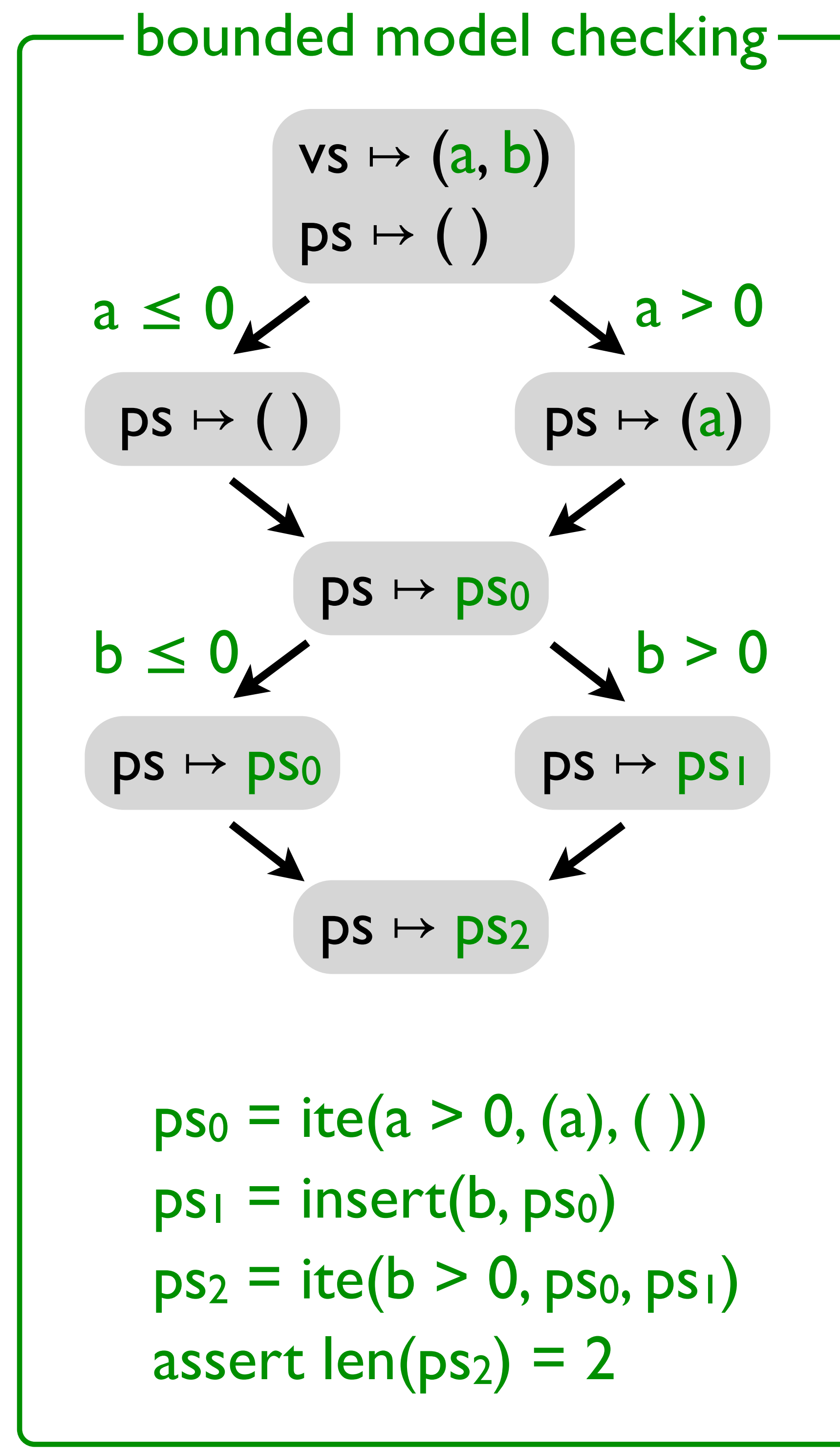




Design space of precise symbolic encodings

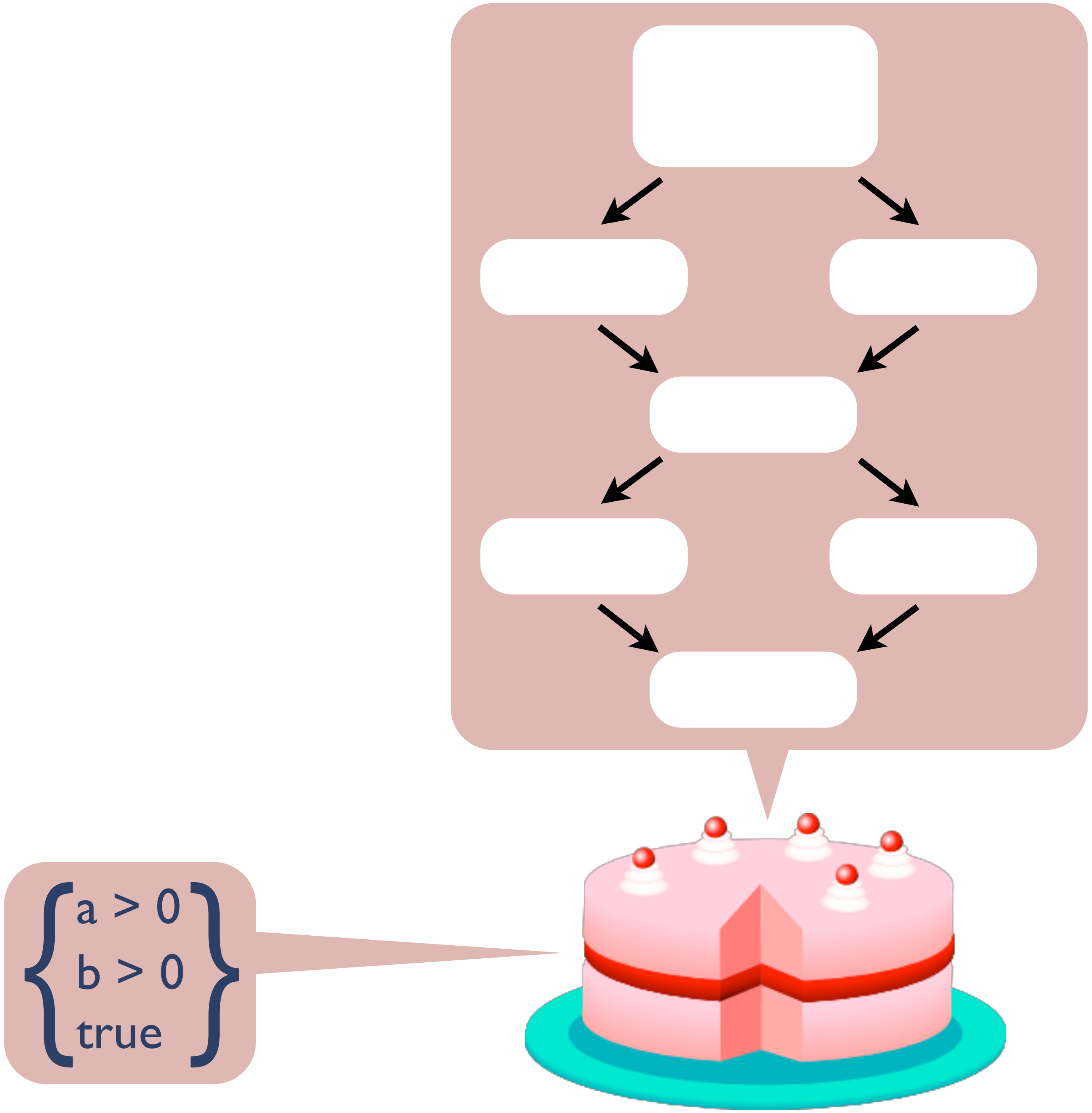
```
solve:
  ps = ()
  for v in vs:
    if v > 0:
      ps = insert(v, ps)
  assert len(ps) == len(vs)
```





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)
```

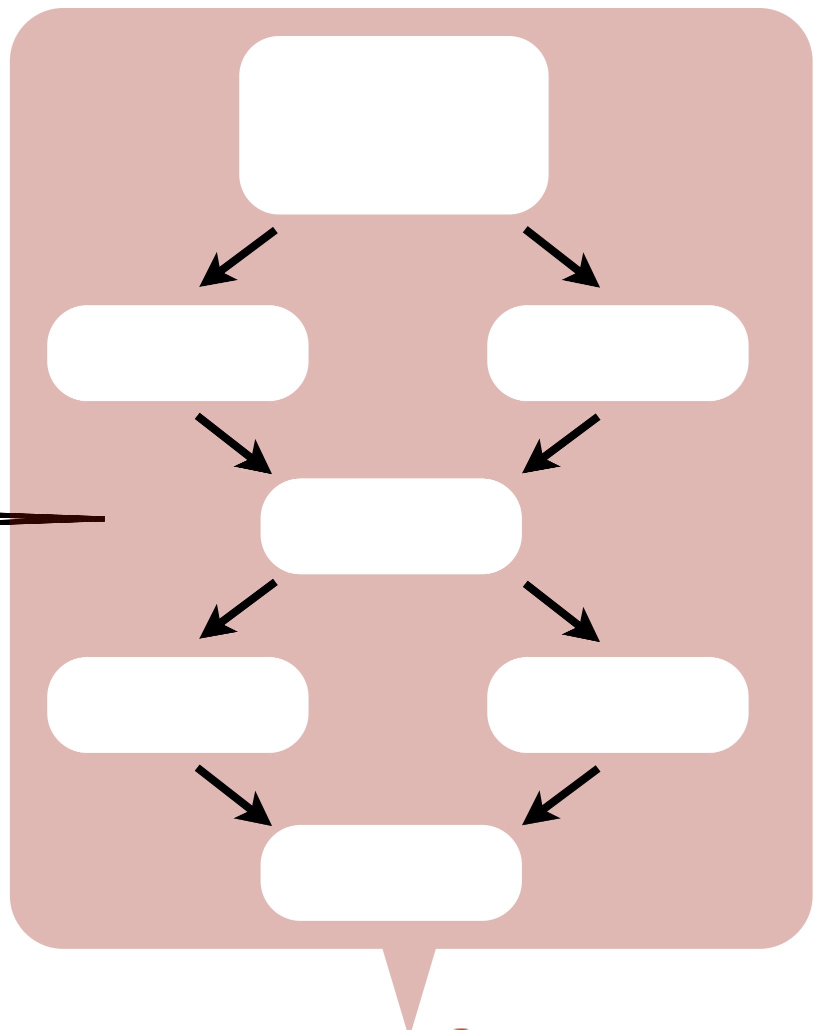


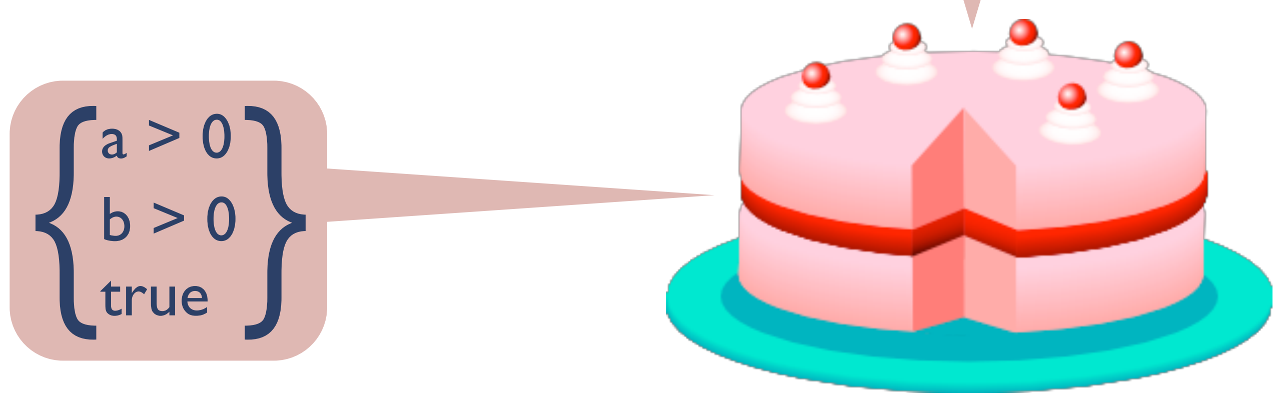
A new design: type-driven state merging

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

Merge instances of

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



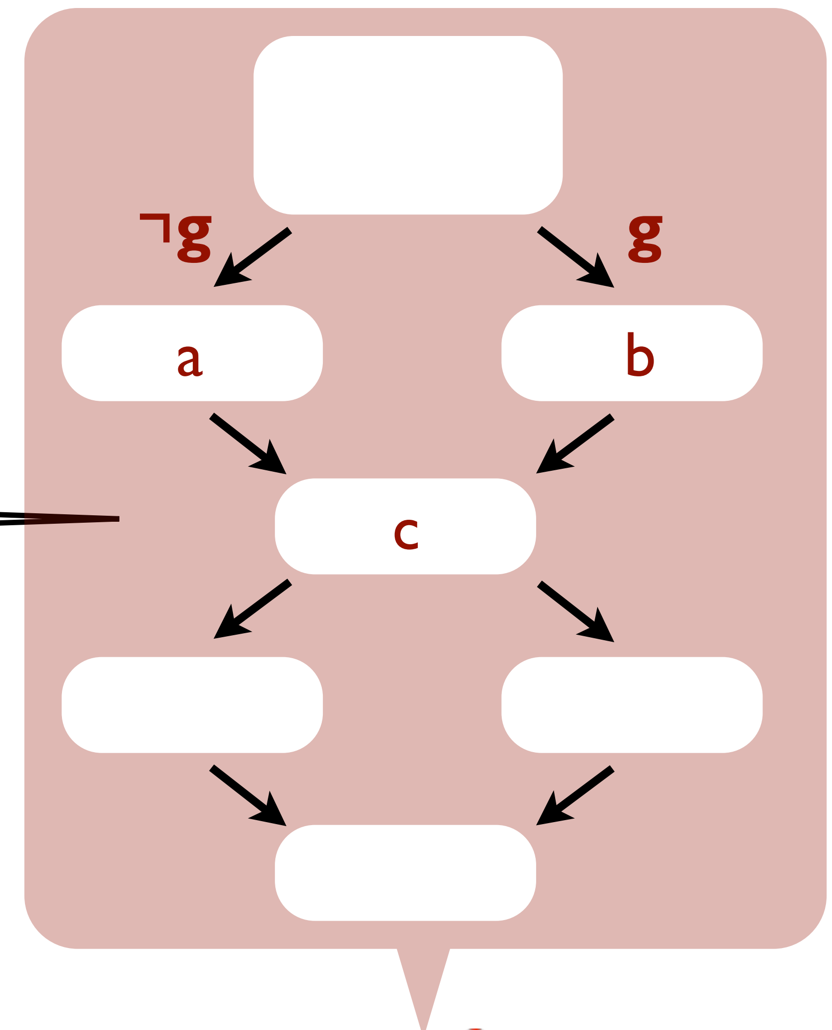


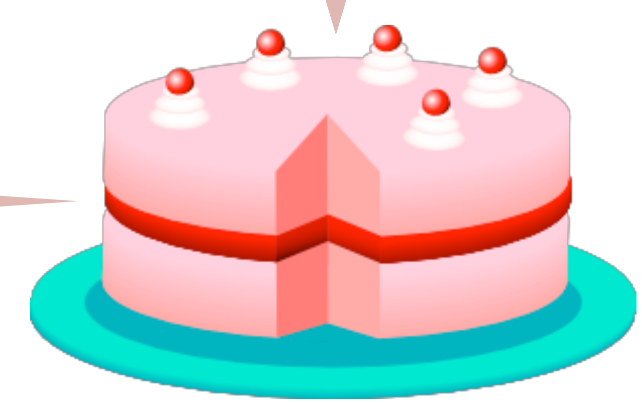
A new design: type-driven state merging

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

Merge instances of

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





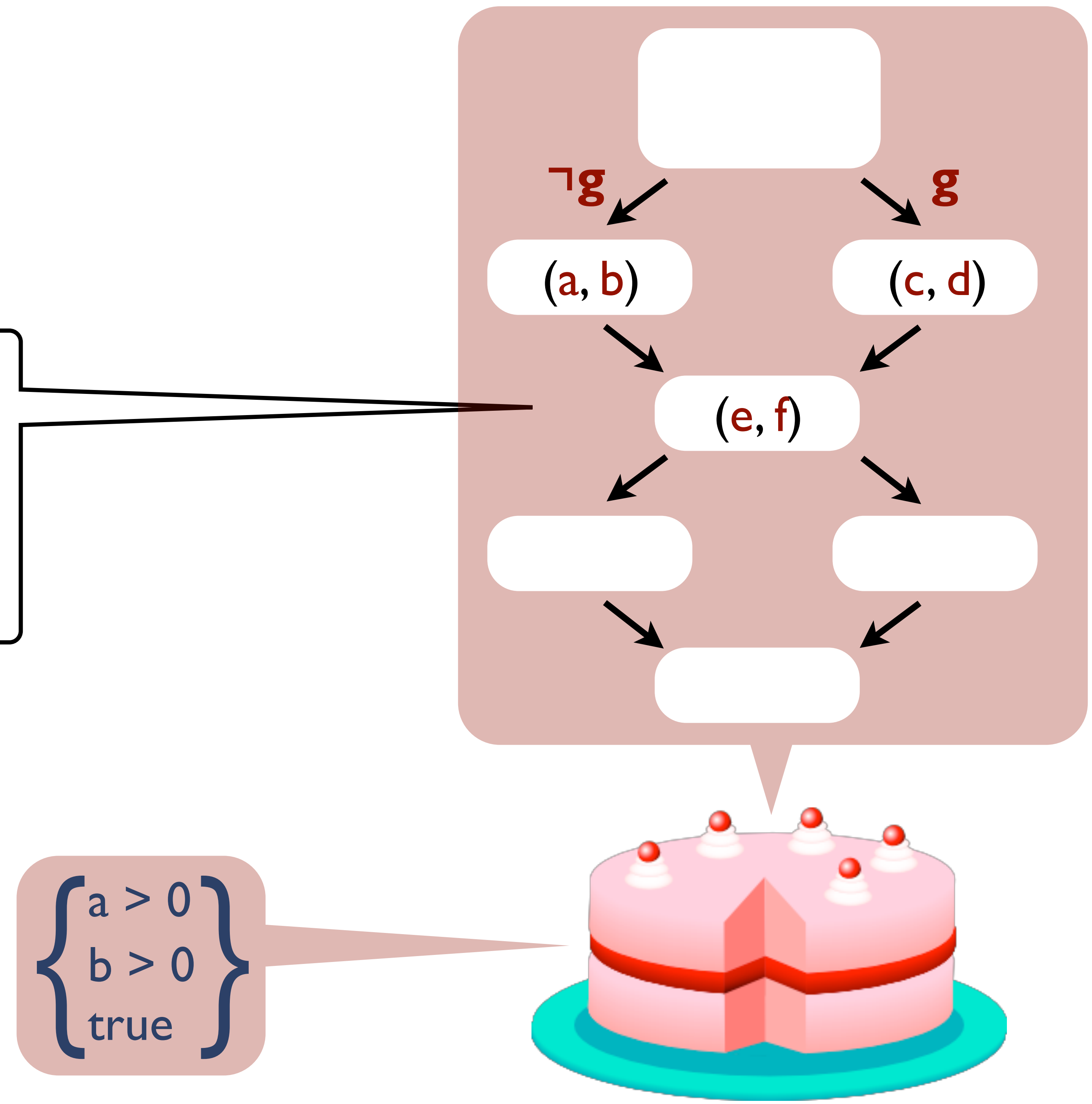
$$\left\{ \begin{array}{l} a > 0 \\ b > 0 \\ \text{true} \end{array} \right\}$$

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 instances of

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

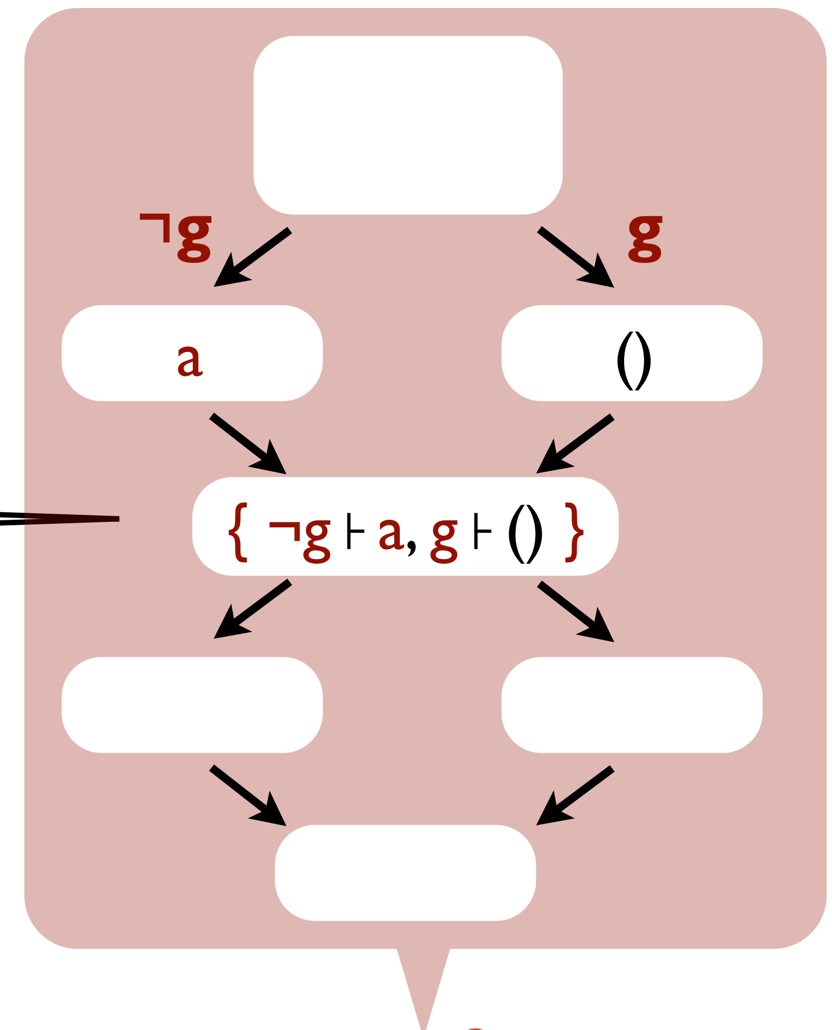


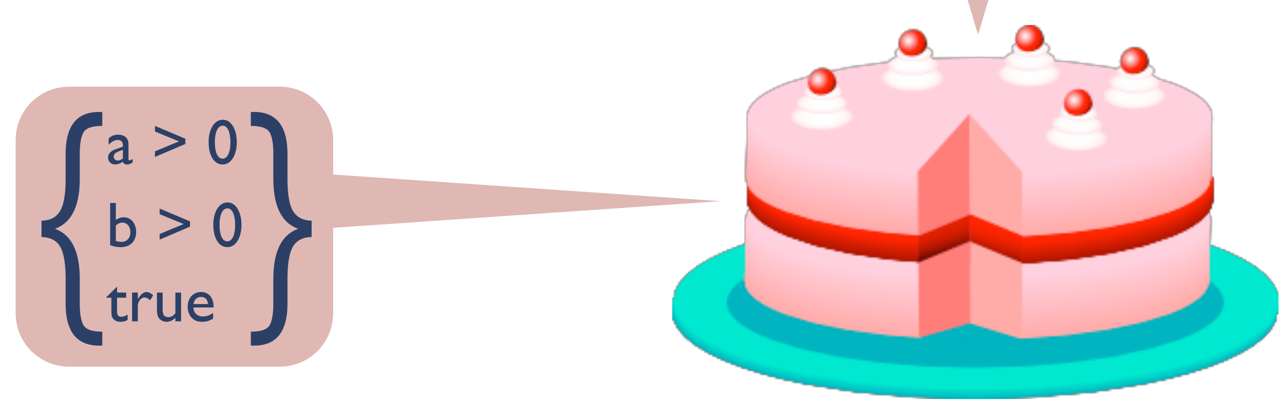
A new design: type-driven state merging

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

Merge instances of

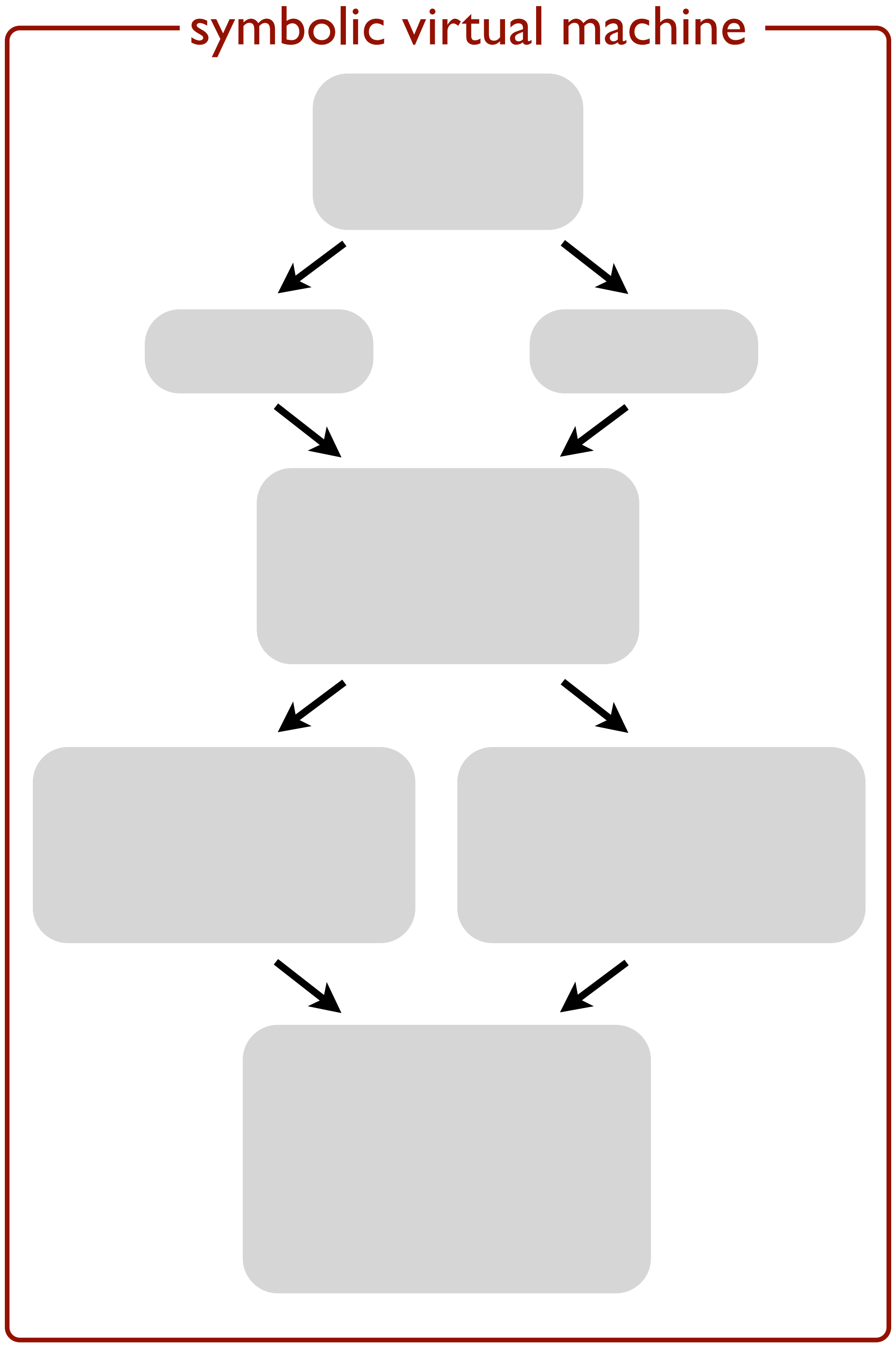
- primitive types: symbolically
- value types: structurally
- all other types: via unions





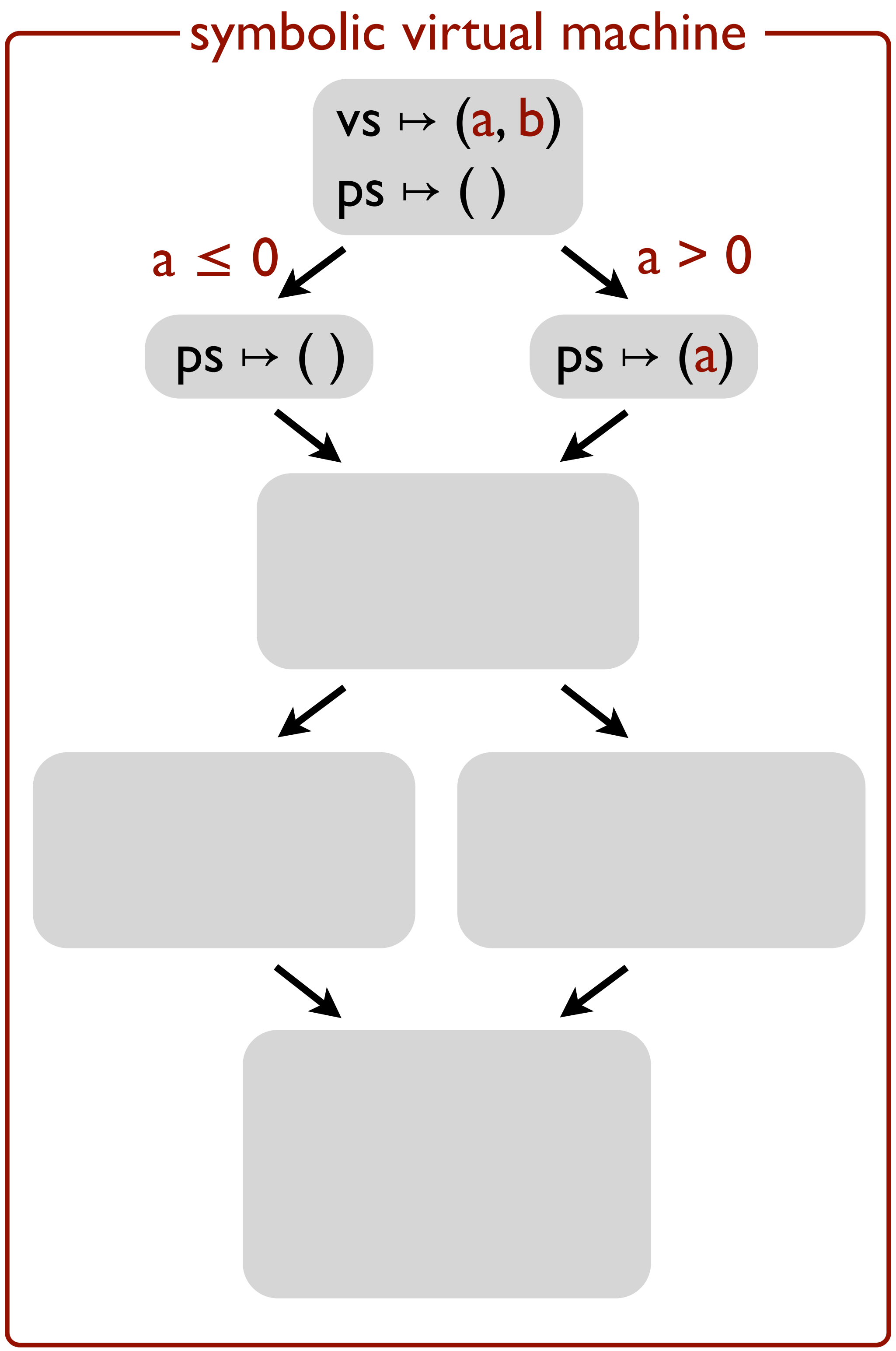
A new design: type-driven state merging

```
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)
```

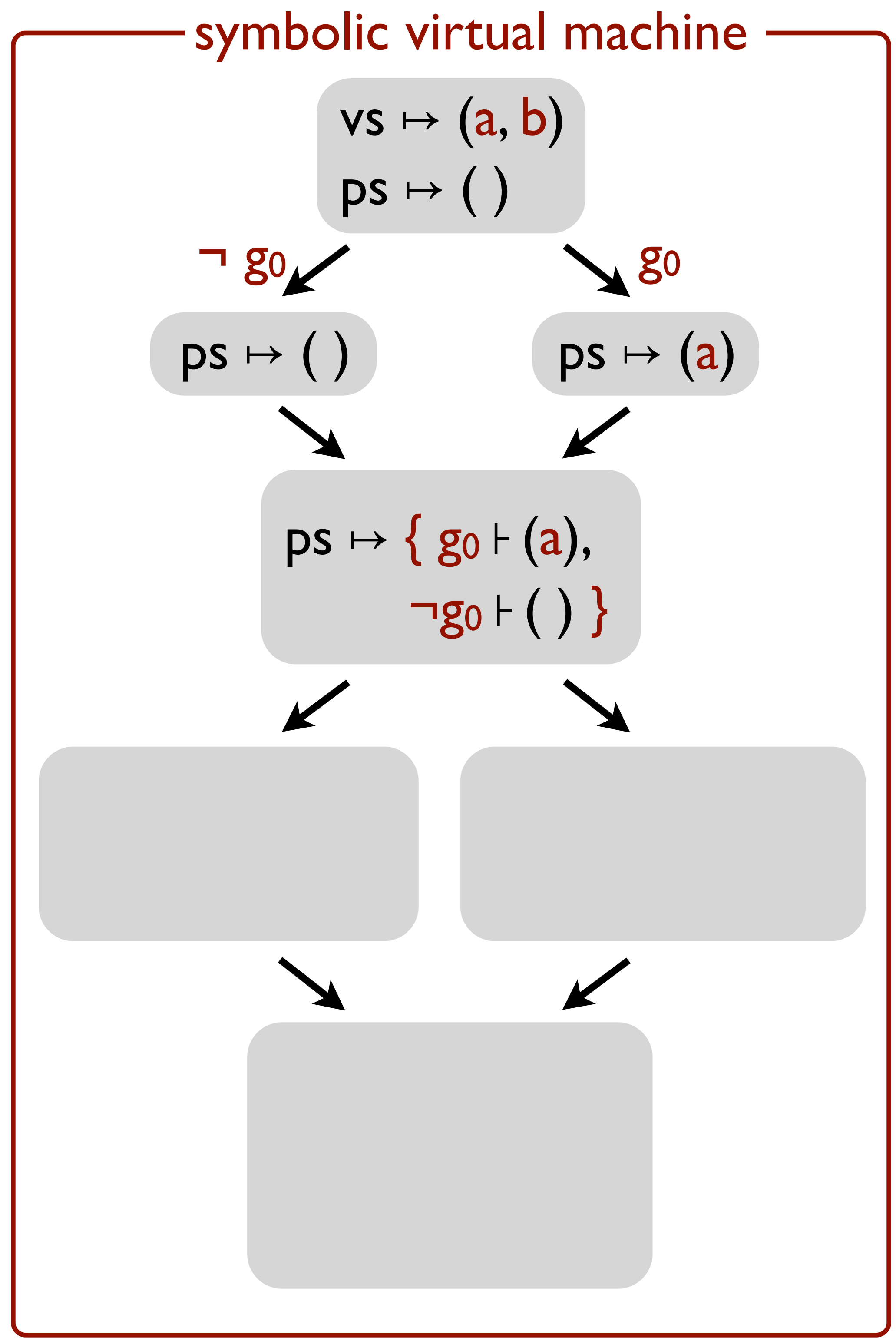


A new design: type-driven state merging

```
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.

$g_0 = a > 0$

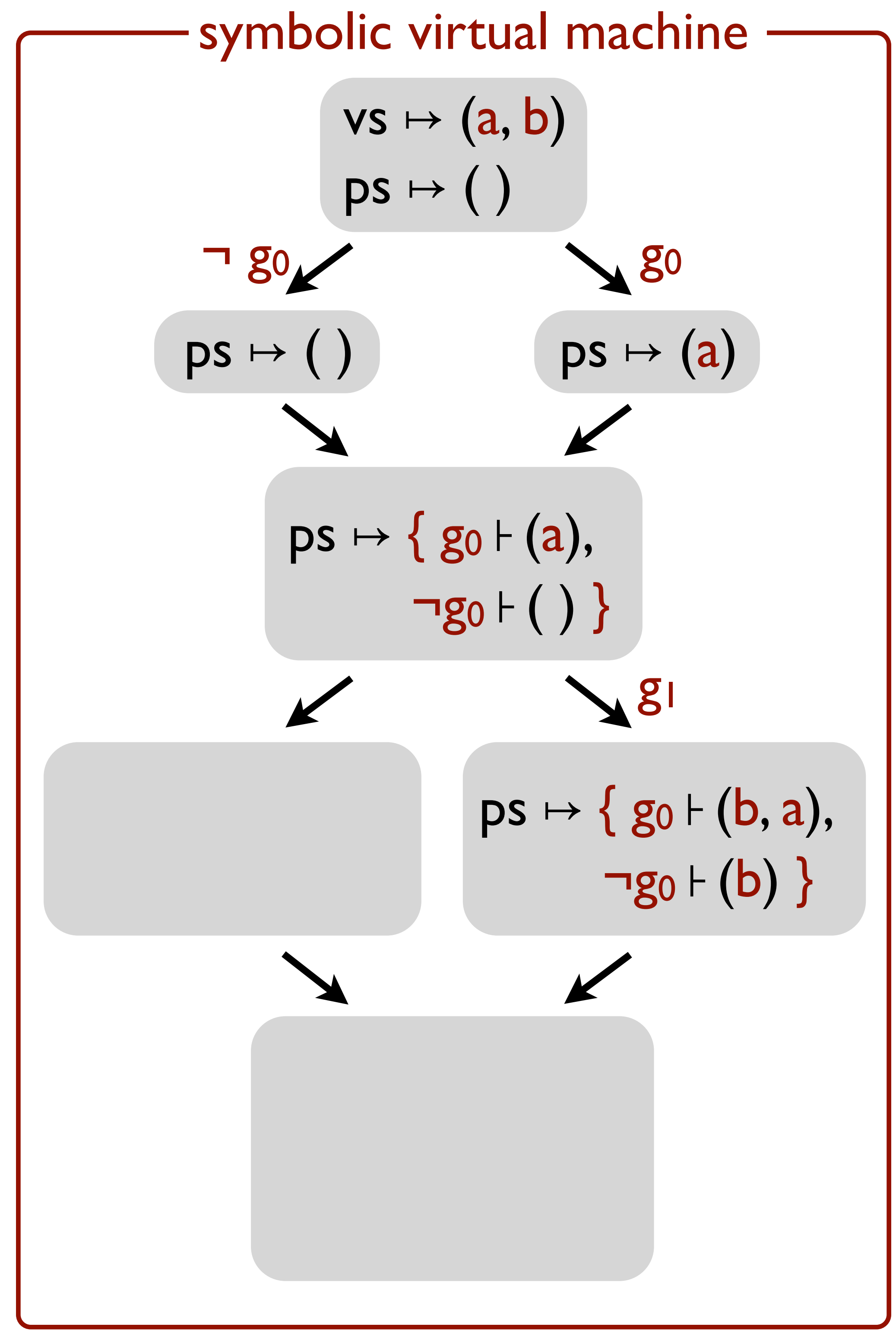


A new design: type-driven state merging

```
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.

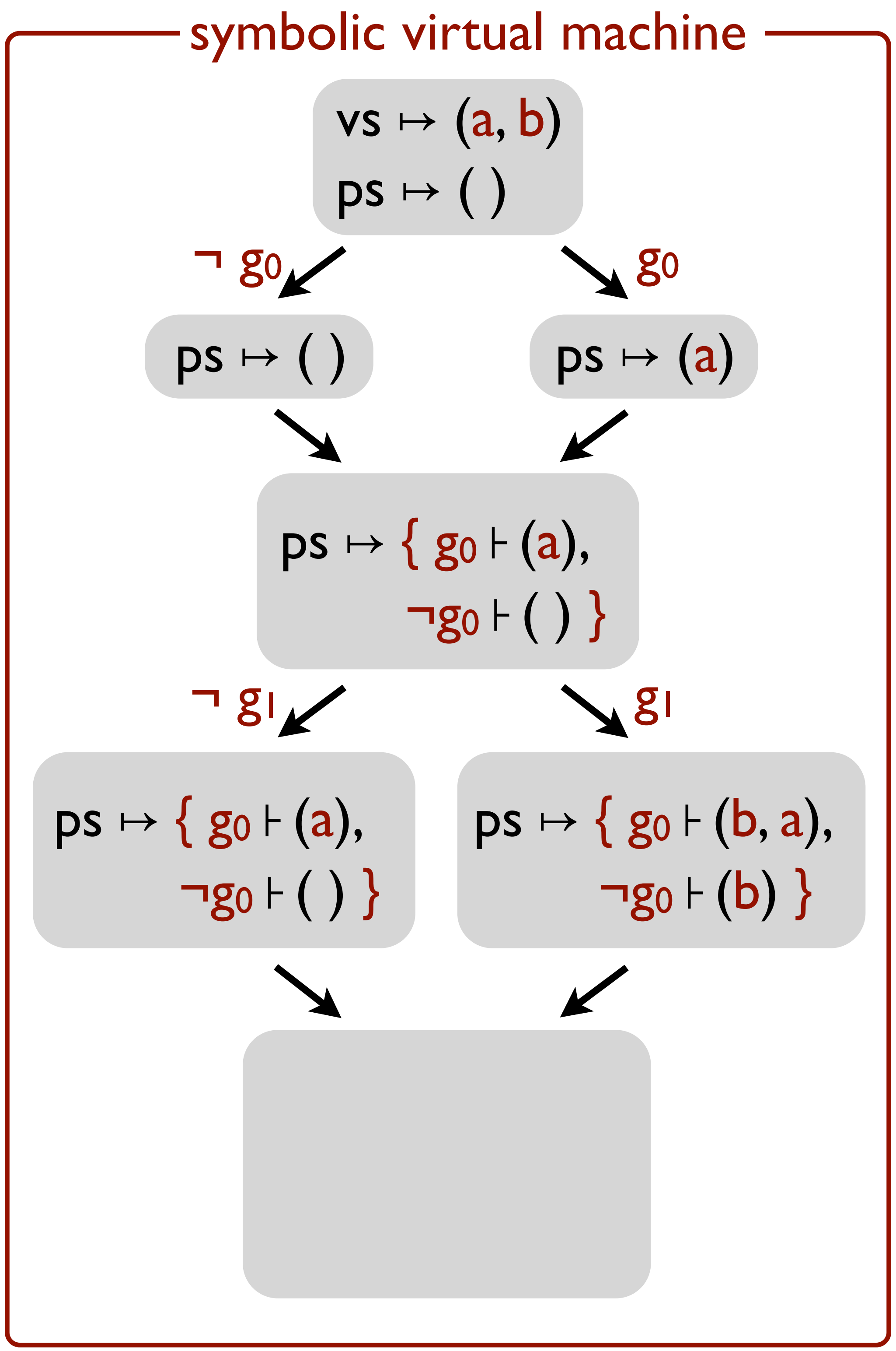
$g_0 = a > 0$
$g_1 = b > 0$



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)
```

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

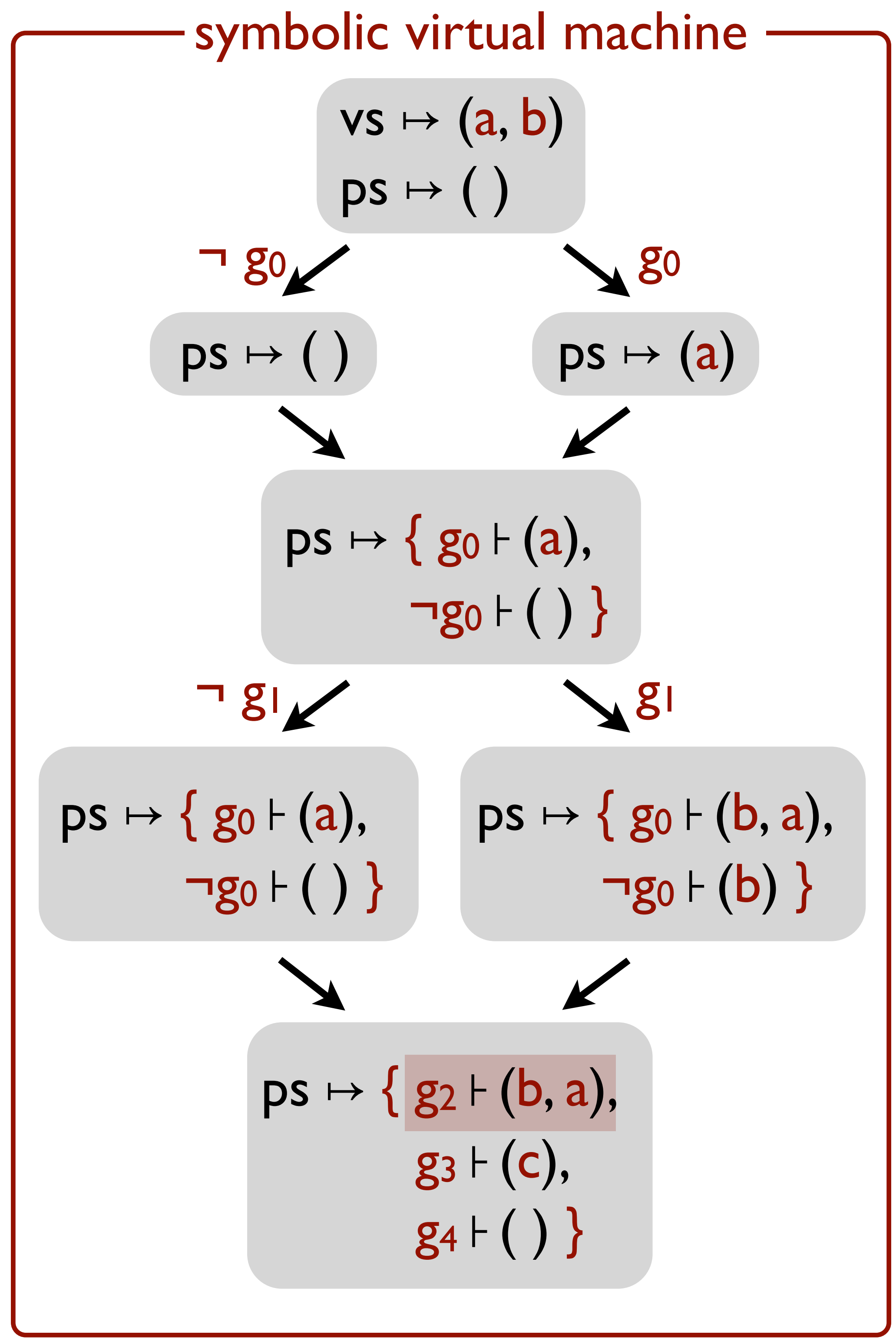


A new design: type-driven state merging

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

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

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



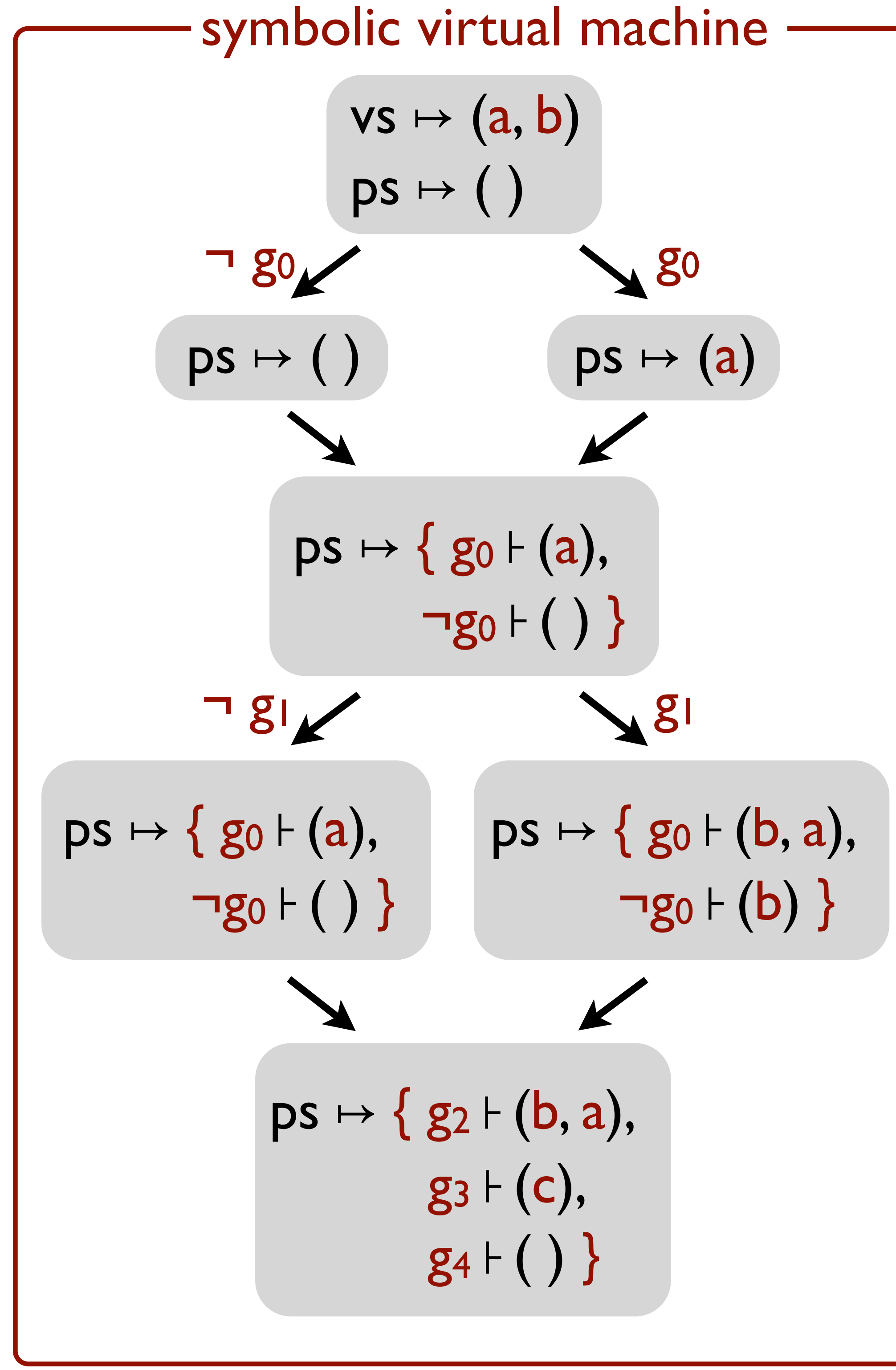
A new design: type-driven state merging

```
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



How to build your own solver-aided tool or language

The classic (hard) way to build a tool

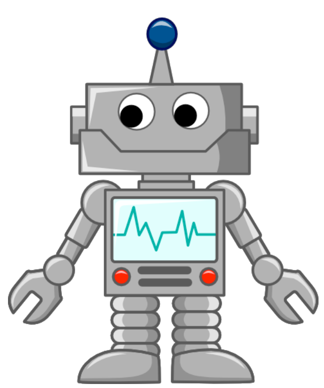
What is hard about building a solver-aided tool?

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.

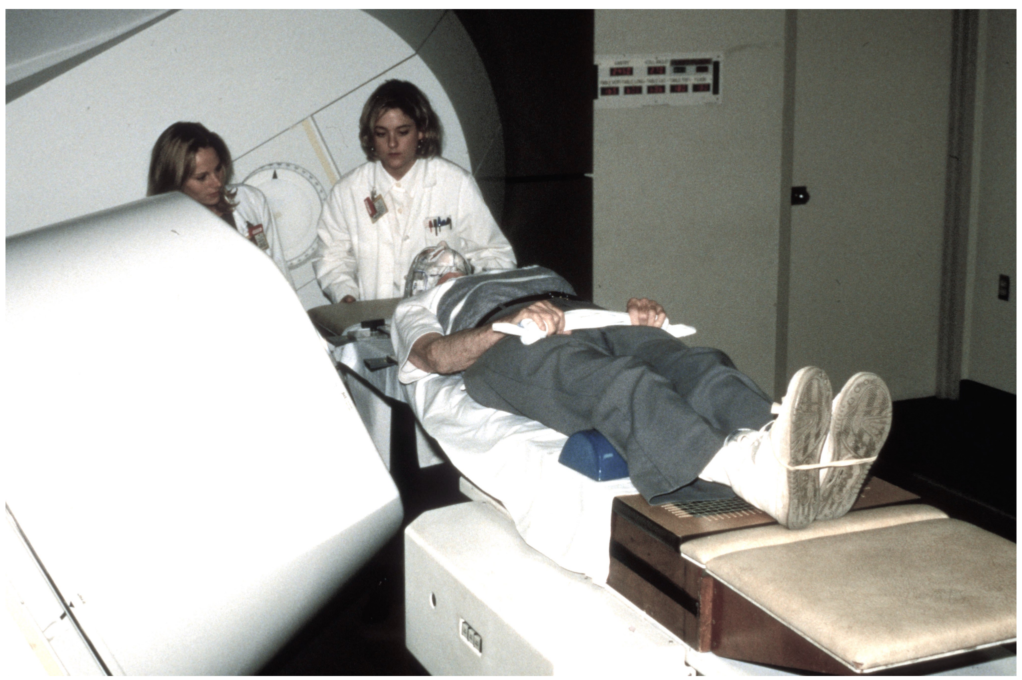


A last look: a few recent applications

Cool tools built with Rosette!

Verifying a radiation therapy system

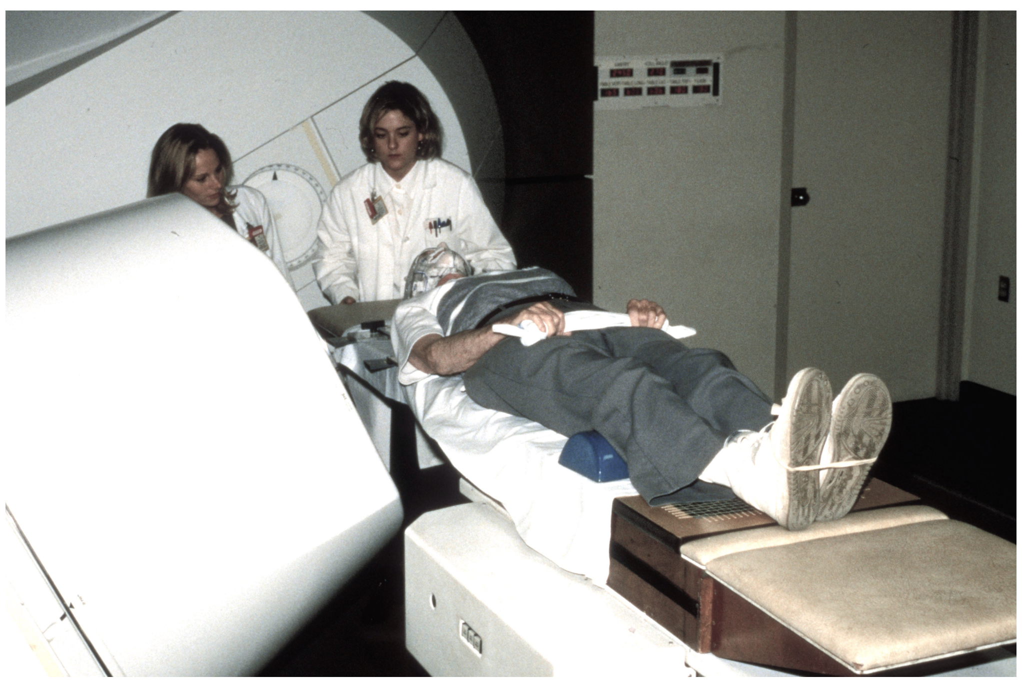
Clinical Neutron Therapy System (CNTS) at UW

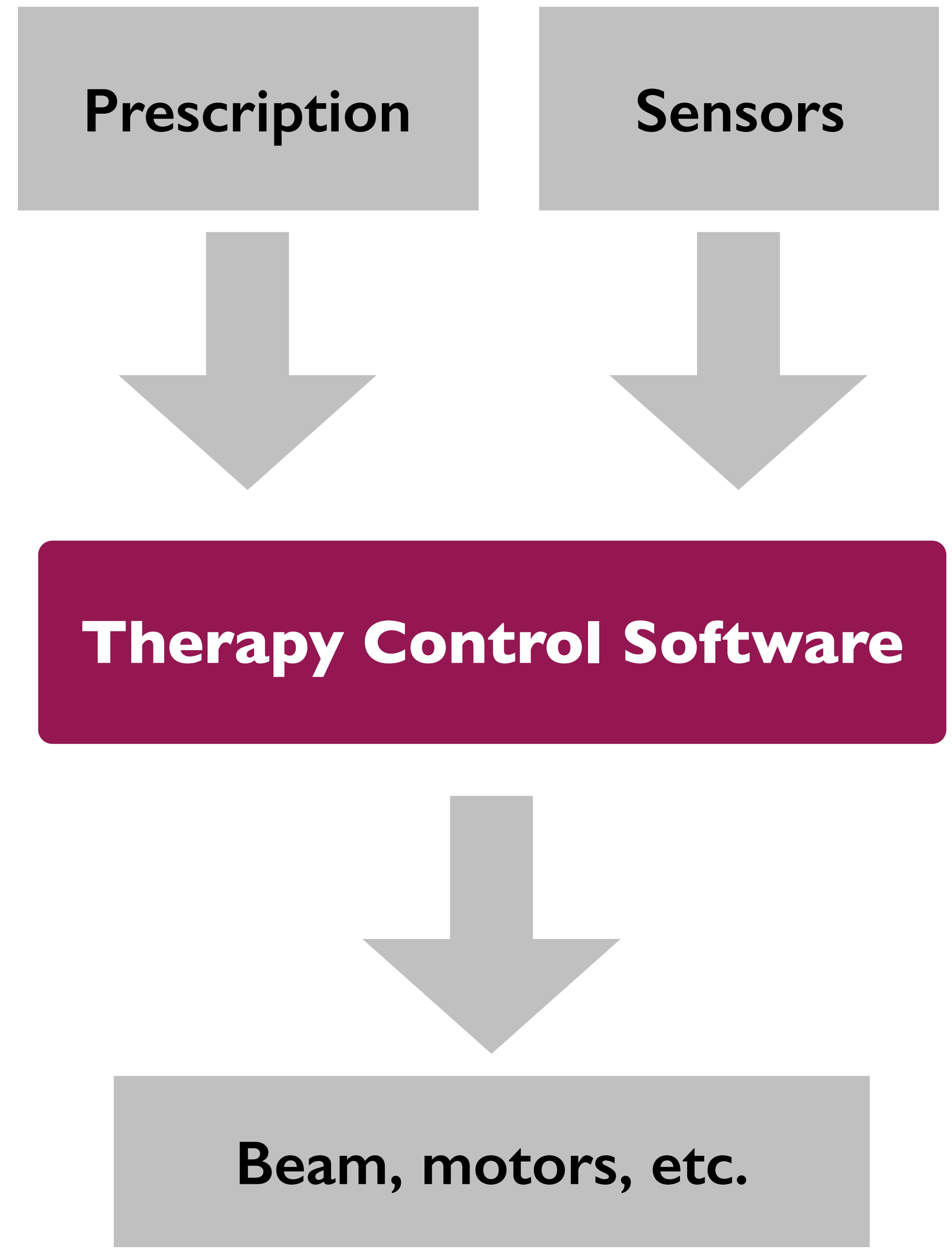


- 30 years of incident-free service.
- Controlled by custom software, built by CNTS engineering staff.
- Third generation of Therapy Control software built recently.

Verifying a radiation therapy system

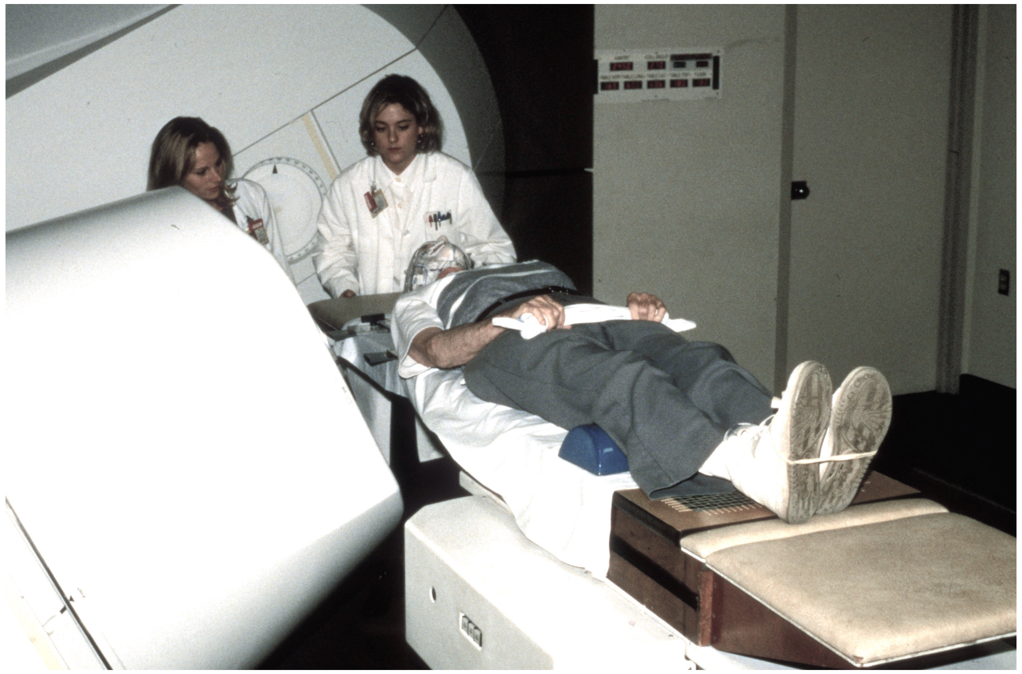
Clinical Neutron Therapy System (CNTS) at UW





Verifying a radiation therapy system

Clinical Neutron Therapy System (CNTS) at UW

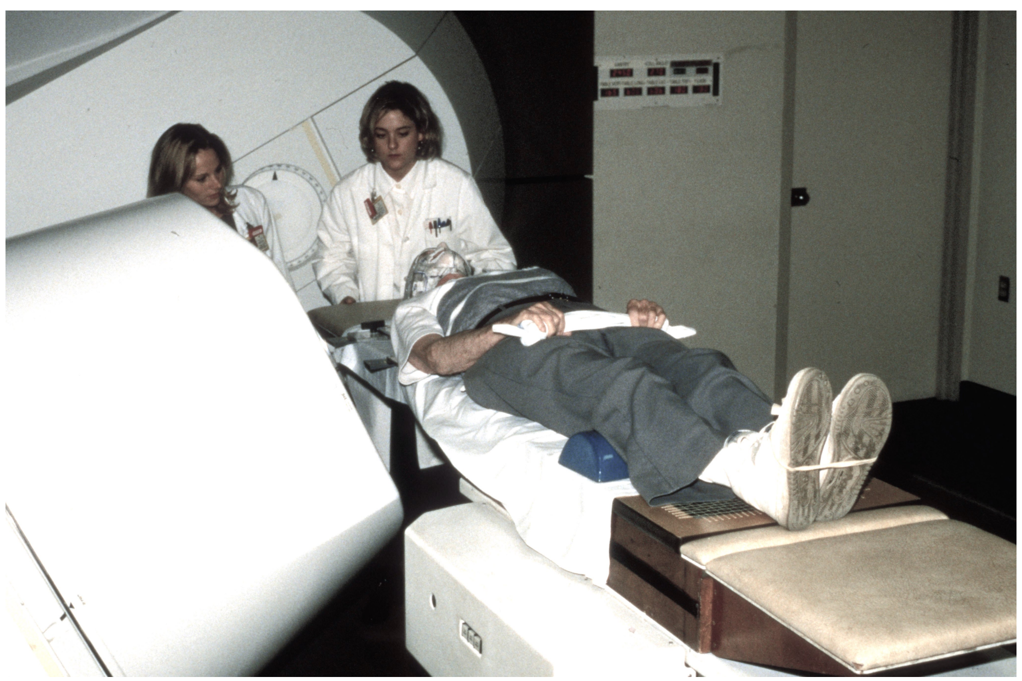


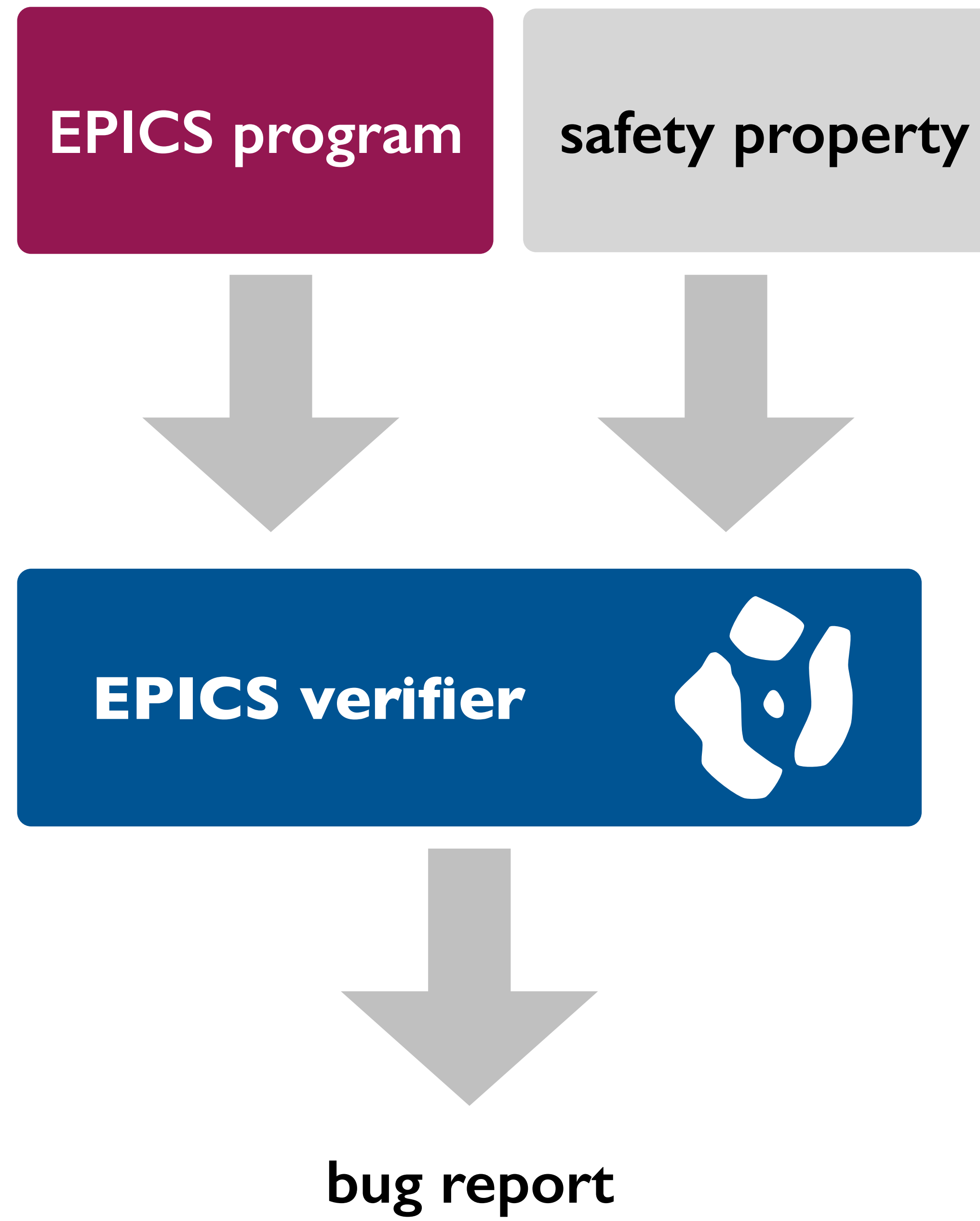
Experimental Physics and Industrial Control System (EPICS) Dataflow Language

Therapy Control Software

Verifying a radiation therapy system

Clinical Neutron Therapy System (CNTS) at UW





Verifying a radiation therapy system

EPICS program

safety property

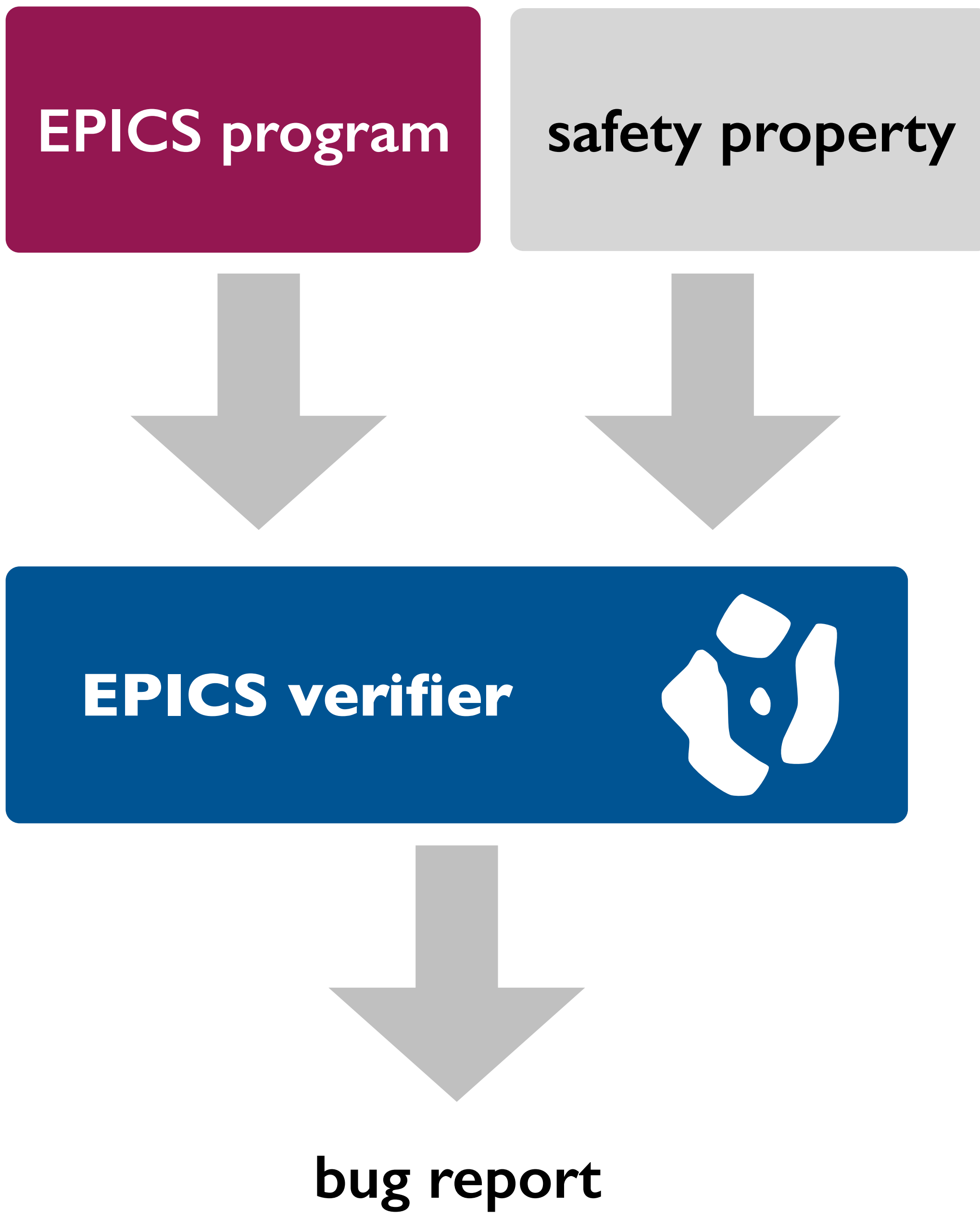
EPICS verifier

bug report

Prototyped in a few days and found bugs.

Calvin Loncaric

Verifying a radiation therapy system



[Pernsteiner et al., CAV'16]

Found safety-critical defects in a pre-release version of the therapy control software.

Used by CNTS staff to verify changes to the controller.

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`

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.

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.

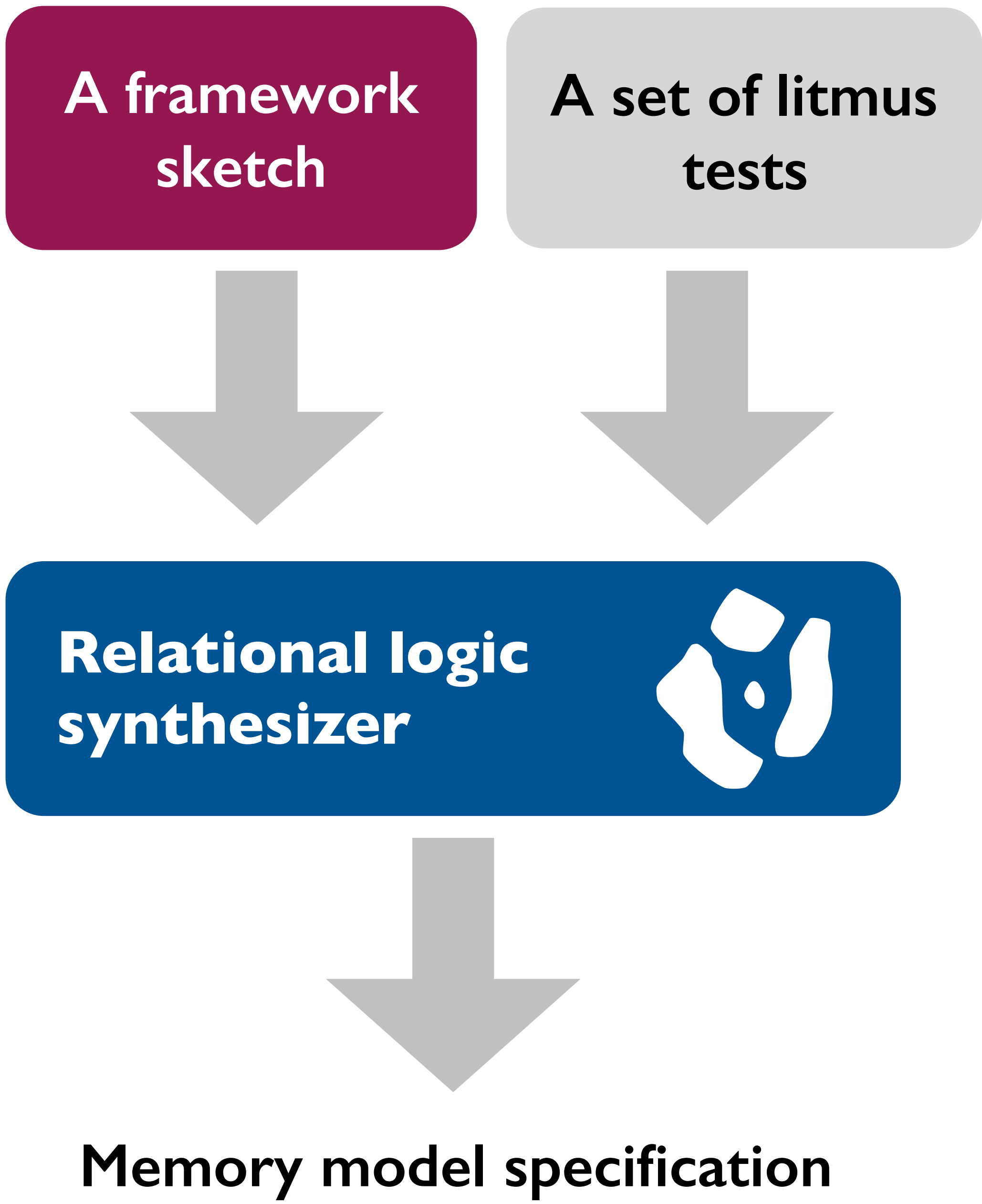
Synthesizing memory models

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.



Synthesizing memory models

A framework sketch

A set of litmus tests

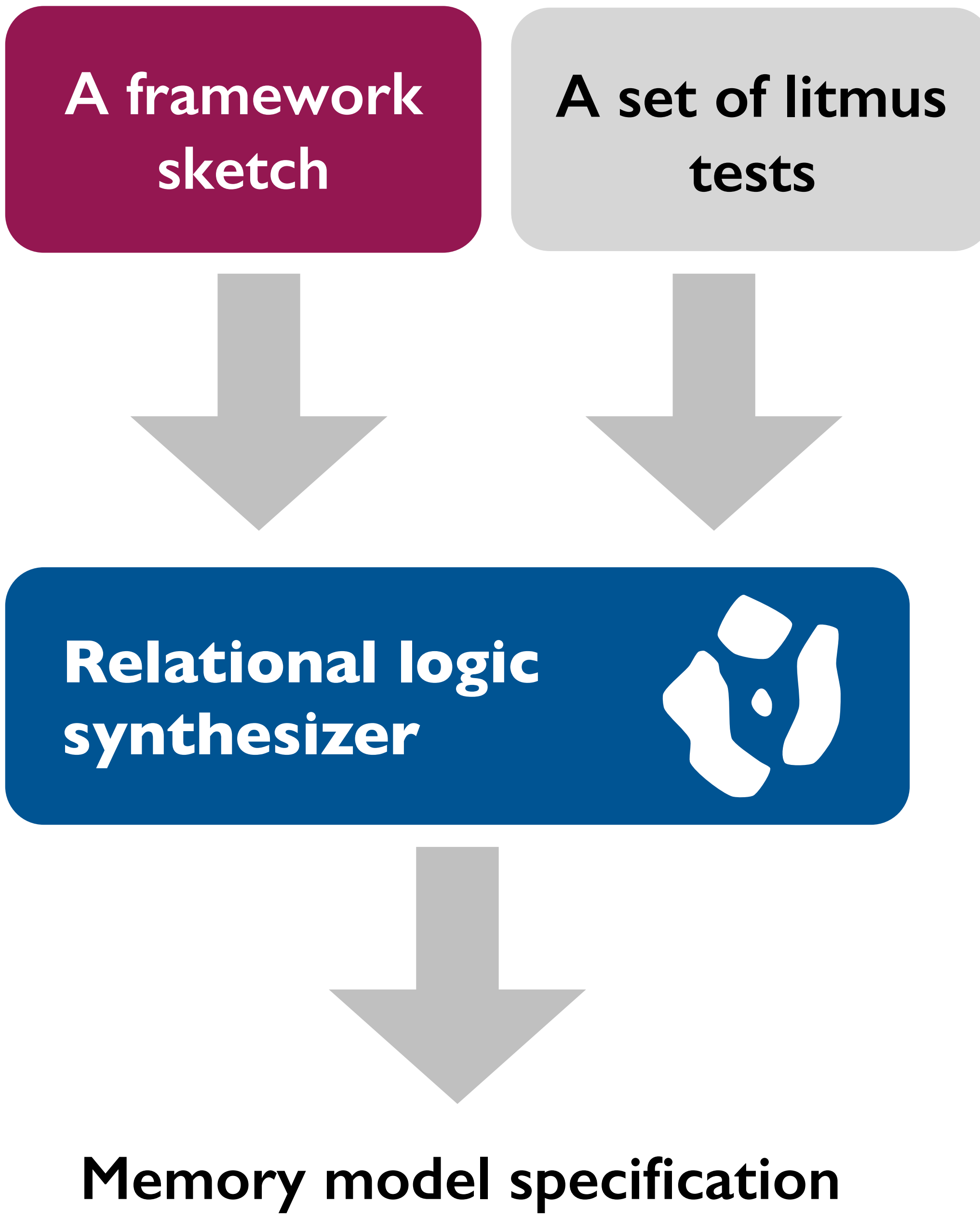
Relational logic synthesizer

Memory model specification

Prototyped in a few weeks and synthesized real memory models.

James Bornholt

Synthesizing memory models

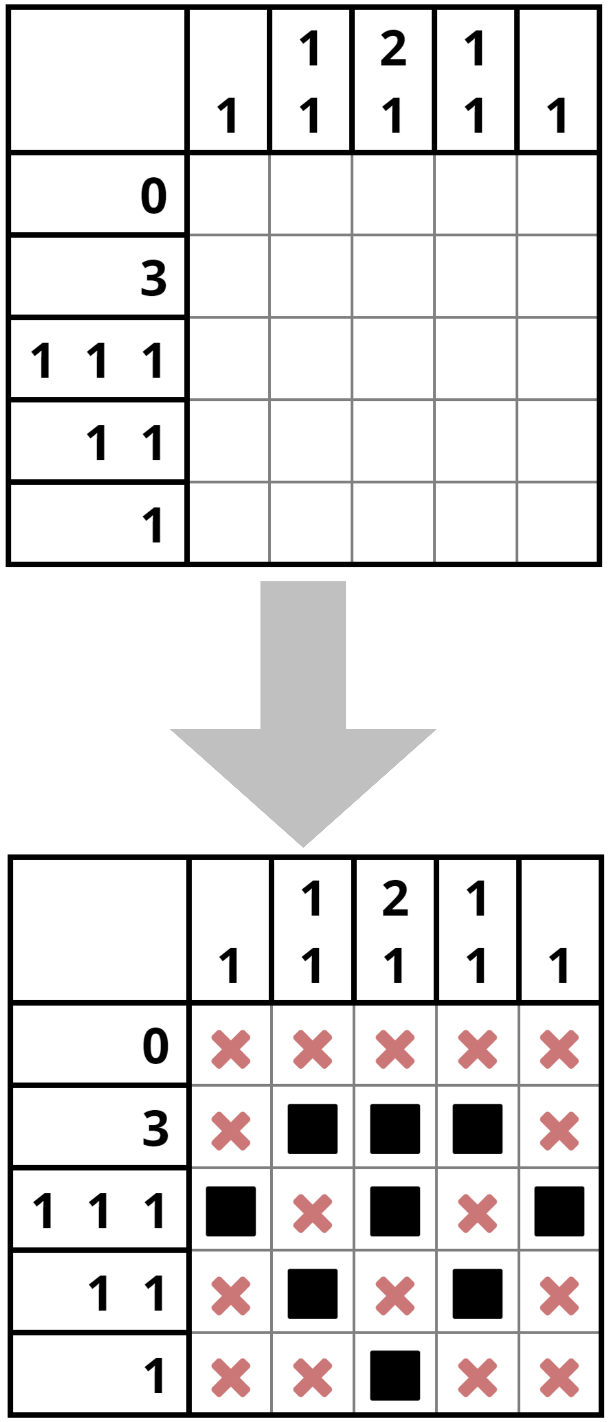


[Bornholt and Torlak, PLDI'17]

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.

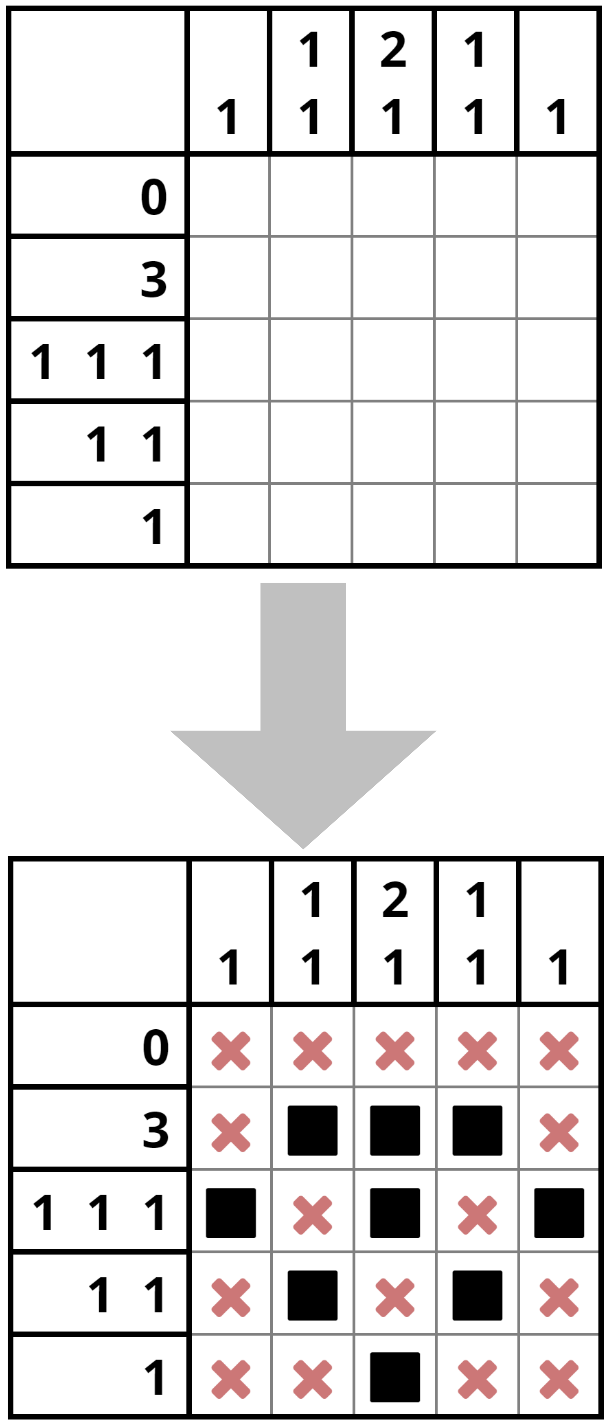
Synthesizing strategies for games and education



Nonograms game mechanics:

The numbered hints describe how many contiguous blocks of cells are filled with *true*. Cells filled with *true* are marked as a black square and cells filled with *false* as a red X.

Synthesizing strategies for games and education



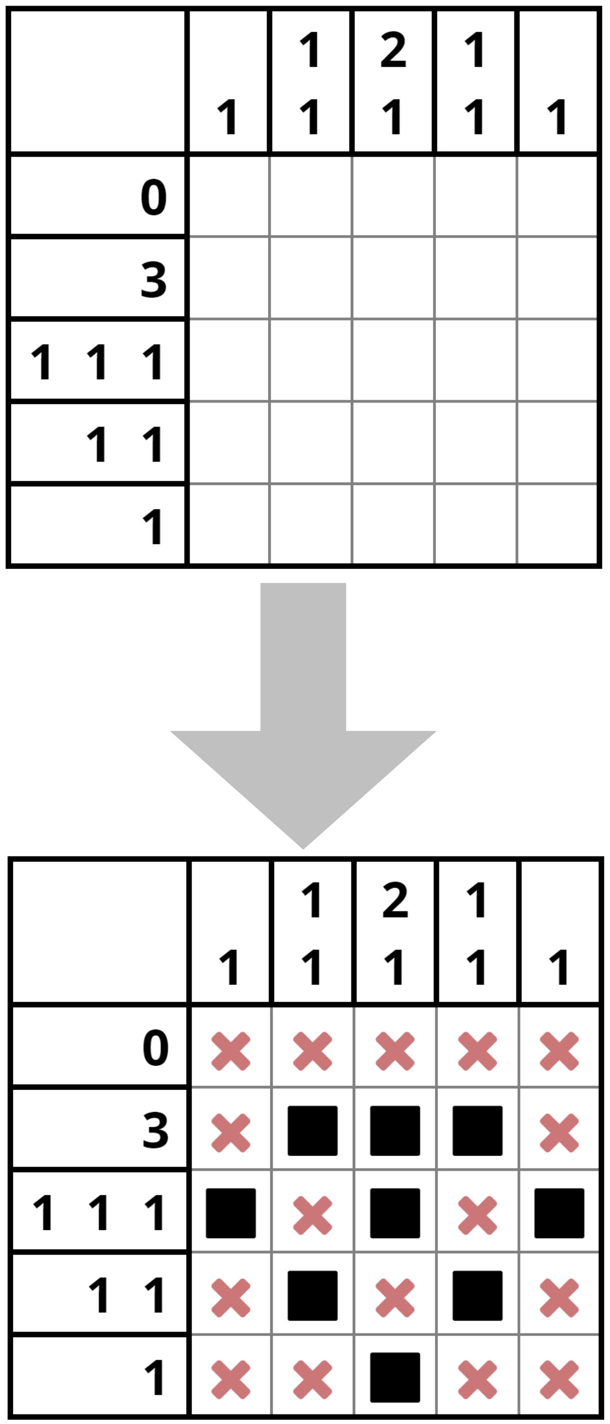
Nonograms game mechanics:

The numbered hints describe how many contiguous blocks of cells are filled with *true*. Cells filled with *true* are marked as a black square and cells filled with *false* as a red X.

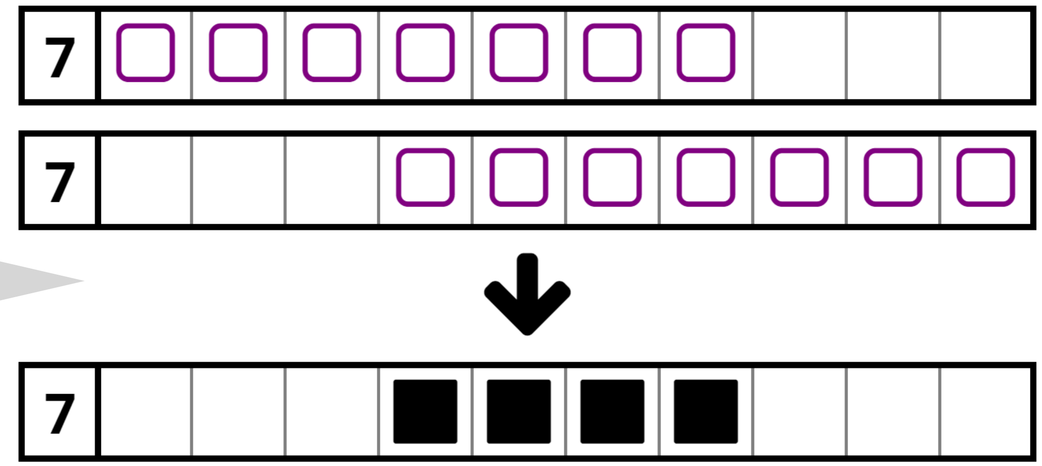
A computer solves puzzles by reducing the game mechanics to backtracking search, but human players solve puzzles by using multiple **strategies** to make progress without guessing.

Finding these strategies is a key challenge in game design, and is usually done through human testing.

Synthesizing strategies for games and education



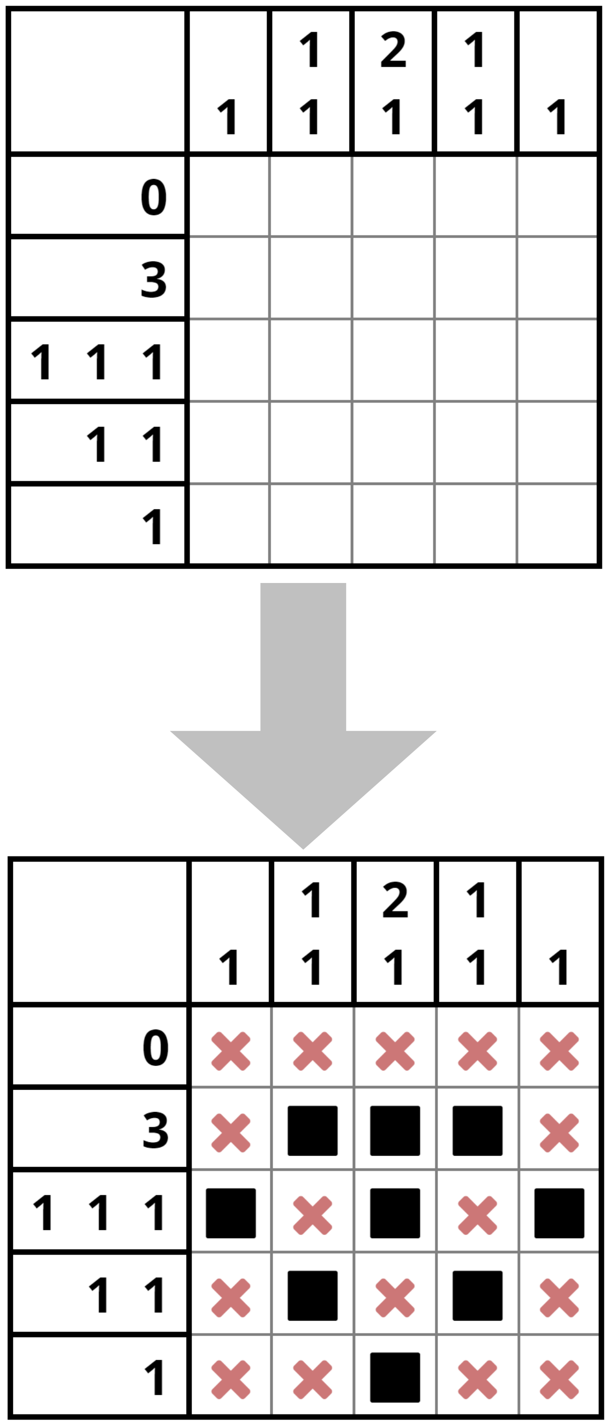
The 'big hint" strategy.



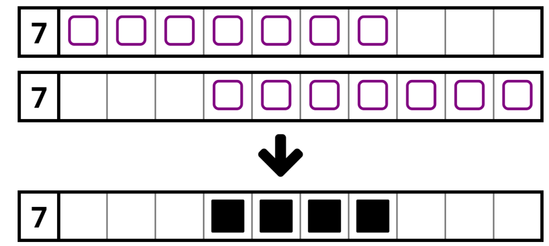
A computer solves puzzles by reducing the game mechanics to backtracking search, but human players solve puzzles by using multiple **strategies** to make progress without guessing.

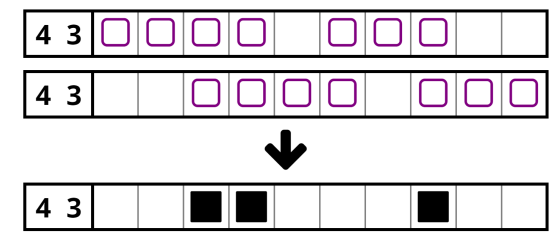
Finding these strategies is a key challenge in game design, and is usually done through human testing.

Synthesizing strategies for games and education



The 'big hint" strategy.

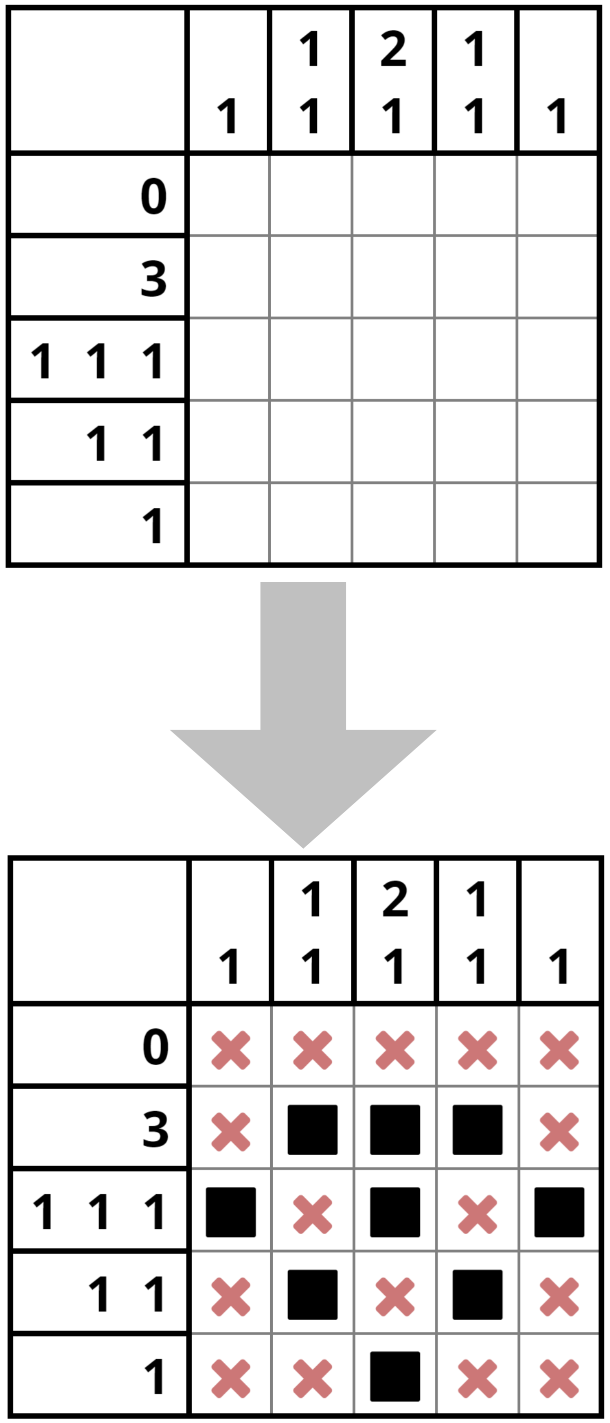


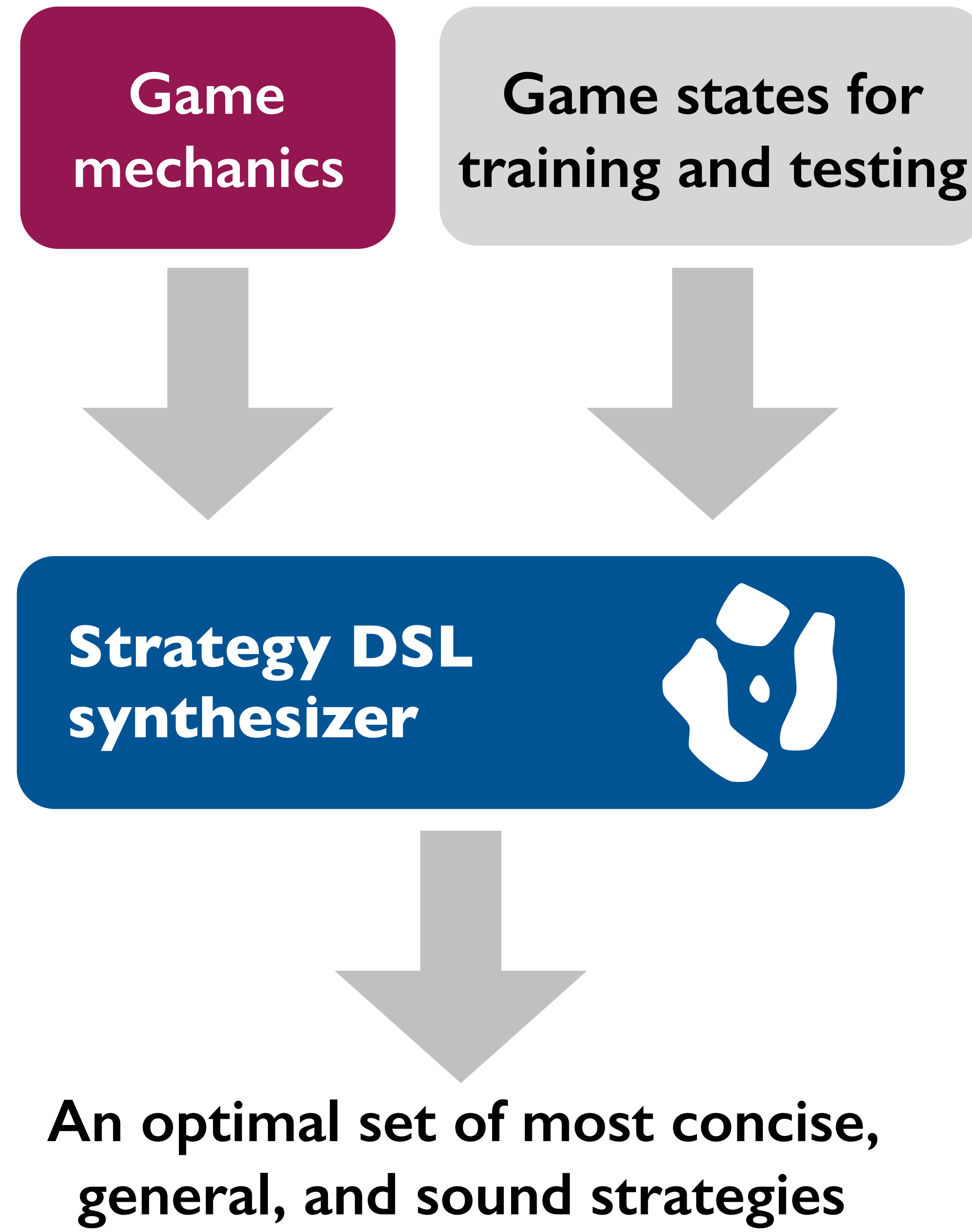


A computer solves puzzles by reducing the game mechanics to backtracking search, but human players solve puzzles by using multiple **strategies** to make progress without guessing.

Finding these strategies is a key challenge in game design, and is usually done through human testing.

Synthesizing strategies for games and education





Synthesizing strategies for games and education

Game mechanics

Game states for training and testing

Strategy DSL synthesizer

An optimal set of most concise, general, and sound strategies

Prototyped in a few weeks and synthesized real strategies.

Eric Butler

Synthesizing strategies for games and education

Game mechanics

Game states for training and testing

Strategy DSL synthesizer

An optimal set of most concise, general, and sound strategies

[Butler et al., FDG'17, VMCAI'18]

Synthesized strategies that outperform documented strategies for Nonograms, both in terms of coverage and quality.

Also used to synthesize strategies for solving K-12 algebra and proofs for propositional logic, recovering and outperforming textbook strategies for these domains.

Summary

Today

- Going pro with solver-aided programming.

Next lecture

- Getting started with SAT solving!