Computer-Aided Reasoning for Software

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

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.

```
verify solve synthesize

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

Solver-aided tool → SMT solver

- $\exists x. \neg \text{safe}(x, P(x))$
- $x = 42 \land \text{safe}(x, P(x))$
- $\exists e. \forall x. \text{safe}(x, P_e(x))$
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

```
P(x) {
  ...
  ...
} assert safe(x, P(x))
```
Wanted: an easier way to build tools

(assert safe(x, P(x)))
Wanted: an easier way to build tools

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

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

ROSETTE  
symbolic virtual machine  
an interpreter for the source language

SMT solver
Wanted: an easier way to build tools

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

[Torlak & Bodik, PLDI’14]
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: guests and hosts

guest language

library (shallow) embedding

interpreter (deep) embedding

host language
Layers of classic languages: guests and hosts

- D3
- JavaScript
- Python
- C

Guest language

- Library (shallow) embedding
- Interpreter (deep) embedding

Host language
 Layers of solver-aided languages

solver-aided guest language

library (shallow) embedding  interpreter (deep) embedding

solver-aided host language
Layers of solver-aided languages

- C (subset)
- **solver-aided guest language**
  - library ((shallow) embedding)
  - interpreter ((deep) embedding)
- ROSSETTE

**Jitterbug** (OSDI 2020): Verifying and synthesizing BPF JITs in Linux.

- BPF, x86 32, x86 64, ARM 32, ARM 64, RISC-V 32, RISC-V 64
A tiny example solver-aided guest language

```python
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 solver-aided guest language

```python
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**, and **synthesize** programs in the BV SDSL.
A tiny example solver-aided guest language

```python
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.

1. interpreter [50 LOC]
2. verifier [free]
3. synthesizer [free]

We want to **test**, **verify**, and **synthesize** programs in the BV SDSL.
A tiny example language

def bvmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bv xor(r0, r2)
    r5 = bv and(r3, r4)
    r6 = bv xor(r1, r5)
    return r6

> bvmax(-2, -1)
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 language

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

> bvmx(-2, -1)
A tiny example language

```python
def b vmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bv xor(r0, r2)
    r5 = bv and(r3, r4)
    r6 = bv xor(r1, r5)
    return r6

> b vmax(-2, -1)
```

```
(define b vmax
 `((2 bvsge 0 1)
   (3 bvneg 2)
   (4 bv xor 0 2)
   (5 bv and 3 4)
   (6 bv xor 1 5)))

`(-2 -1)
```

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

```python
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)
```

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

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

Rosette
A tiny example language

```python
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)
```

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

```plaintext`
(load (last))
```
A tiny example language

```python
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)
```

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

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

```python
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)
```

```python
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
```

```
(define (interpret prog inputs)
  (define op (lookup opcode))
  (define args (map load in))
  (store out (apply op args)))))
(load (last)))
```

\[
\begin{align*}
\text{define} & \quad \text{bvmax} \quad 0 \quad -2 \\
\text{``}( & \quad 2 \quad \text{bvsge} \quad 0 \quad 1) \\
\text{``}( & \quad 3 \quad \text{bvneg} \quad 2) \\
\text{``}( & \quad 4 \quad \text{bvxor} \quad 0 \quad 2) \\
\text{``}( & \quad 5 \quad \text{bvand} \quad 3 \quad 4) \\
\text{``}( & \quad 6 \quad \text{bvxor} \quad 1 \quad 5))) \\
\end{align*}
\]
A tiny example language

```python
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)
```

Interpretation:
```
(define (interpret prog inputs)
  (make-registers prog inputs)
  (for ([stmt prog])
    (match stmt
      [(list out opcode in ...)
        (define op (lookup opcode))
        (define args (map load in))
        (store out (apply op args))])))
  (load (last)))
```
A tiny example language

```python
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)))

load (last))
```

### Rosette

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

```
```

A tiny example language

```python
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 language

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

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

(b(bvmax 0 1) b(neg 2) b(xor 0 2) b(land 3 4) b(xor 1 5)))

> bvmx(-2, -1)
-1
A tiny example language

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

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

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

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

- pattern matching
- first-class & higher-order procedures
- side effects
A tiny example language

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

> verify(bvmx, max)
```

```lisp
(define-symbolic x y int32?)
(define in (list x y))
(verify
  (assert (equal? (interpret bvmx in) (apply max in))))
```
A tiny example language

def \text{bvmx}(\text{r0}, \text{r1}) :
    \text{r2} = \text{bvsge}(\text{r0}, \text{r1})
    \text{r3} = \text{bvneg}(\text{r2})
    \text{r4} = \text{bvxor}(\text{r0}, \text{r2})
    \text{r5} = \text{bvand}(\text{r3}, \text{r4})
    \text{r6} = \text{bvxor}(\text{r1}, \text{r5})
    \text{return} \quad \text{r6}

> \text{verify}(\text{bvmx}, \text{max})

(\text{define-symbolic} \ x \ y \ \text{int32}\?)
(\text{define} \ \text{in} \ (\text{list} \ x \ y))
(\text{verify}
    (\text{assert} \ (\text{equal?} \ (\text{interpret} \ \text{bvmx} \ \text{in})
    \ (\text{apply} \ \text{max} \ \text{in}))))

(\text{define} \ (\text{max} \ x \ y)
    (\text{if} \ (\text{bvsge} \ x \ y) \ x \ y))
**A tiny example language**

```python
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)
```

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

```scheme
(define-symbolic x y int32?)
(define in (list x y))
(verify
  (assert (equal? (interpret bvmax in) (apply max in))))
```
A tiny example language

```
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)
```

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) (apply max in)))
```

Symbolic values can be used just like concrete values of the same type.
A tiny example language

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

> verify(bvmx, max)
```

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 bvmx in) (apply 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 language

```
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]
```

---

```
(define-symbolic x y int32?)
(define in (list x y))
(verify
  (assert (equal? (interpret bvmax in) (apply max in))))
```
A tiny example language

```python
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
A tiny example language

def bvmax(r0, r1):
    r2...r6 = inst??(bvsge, bvneg, bvxor, bvand)
    return r6

> synthesize(bvmax, max)

(define-symbolic x y int32?)
(define in (list x y))
(synthesize
  #:forall in
  #:guarantee
  (assert (equal? (interpret bvmax in) (apply max in))))
A tiny example language

```python
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)
```

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

query

program

guest language

ROSETTE

Symbolic Virtual Machine

SMT solver

Z3
How it all works: a big picture view

query → result

program

guest language

ROSETTE

Symbolic Virtual Machine

SMT solver Z3
How it all works: a big picture view

- **query**
- **result**
- **program**
- **guest language**

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

theories of bitvectors, integers, reals, and uninterpreted functions

**ROSETTE**

Symbolic Virtual Machine

SMT solver **Z3**
Translation to constraints by example

\[ \text{reverse and filter, keeping only positive numbers} \]

\[ \text{vs} \quad (3, 1, -2) \]

\[ \text{ps} \quad (1, 3) \]
Translation to constraints by example

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

assert len(ps) == len(vs)
```

vs:
(3, 1, -2)

ps:
(1, 3)
Translation to constraints by example

```python
solve:
    ps = ()
    for v in vs:
        if v > 0:
            ps = insert(v, ps)
    assert len(ps) == len(vs)
```
Translation to constraints by example

solve:
ps = ()
for v in vs:
    if v > 0:
        ps = insert(v, ps)
assert len(ps) == len(vs)
Translation to constraints by example

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:

\[
\begin{align*}
\text{ps} & = () \\
\text{for } v \text{ in } \text{vs}: \\
\text{if } v > 0: \\
\text{ps} & = \text{insert}(v, \text{ps}) \\
\text{assert } \text{len(ps)} == \text{len(vs)}
\end{align*}
\]
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

bounded model checking
Design space of precise symbolic encodings

solve:
\[
\begin{align*}
\text{ps} &= () \\
\text{for } v \text{ in } \text{vs}: \\
\quad \text{if } v > 0: \\
\quad \quad &\text{ps} = \text{insert}(v, \text{ps}) \\
\text{assert } \text{len(ps)} == \text{len(vs)}
\end{align*}
\]

symbolic execution

bounded model checking

solve:
\[
\begin{align*}
\text{ps} &= () \\
\text{for } v \text{ in } \text{vs}: \\
\quad \text{if } v > 0: \\
\quad \quad &\text{ps} = \text{insert}(v, \text{ps}) \\
\text{assert } \text{len(ps)} == \text{len(vs)}
\end{align*}
\]
Design space of precise symbolic encodings

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

bounded model checking

vs ⊧ (a, b)
ps ⊧ ()
a ≤ 0
ps ⊧ ()
b > 0
ps ⊧ (a)
ps ⊧ ps₀

ps₀ = ite(a > 0, (a), ( ))
ps₁ = insert(b, ps₀)
Design space of precise symbolic encodings

solve:
\[
ps = ()
\]
\[
for v in vs:
    if v > 0:
        ps = insert(v, ps)
assert len(ps) == len(vs)
\]
Challenge: simple vs compact encoding (SE and BMC)

Can we have both a polynomially sized encoding (like BMC) and concrete evaluation of complex operations (like SE)?
Solution: type-driven state merging

solve:

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

Solution: type-driven state merging
solve:
   ps = ()
   for v in vs:
      if v > 0:
         ps = insert(v, ps)
   assert len(ps) == len(vs)

Solution: type-driven state merging

Merge instances of
   ‣ primitive types: symbolically
   ‣ value types: structurally
   ‣ all other types: via unions
**Solution: type-driven state merging**

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

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

Solution: type-driven state merging
Solution: 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.

\[ g_0 = a > 0 \]
\[ g_1 = b > 0 \]
**Solution: type-driven state merging**

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

g₀ = a > 0  
g₁ = b > 0  

```latex
g₀ \rightarrow (a, b)  
\neg g₀ \rightarrow ()  
\neg g₀ \rightarrow ()  
g₀ \rightarrow ()  
```

```latex
\neg g₁ \rightarrow (b)  
g₁ \rightarrow (a)  
\neg g₁ \rightarrow (a)  
g₁ \rightarrow (b)  
```

symbolic virtual machine

```latex
ps \mapsto \{ g₀ \vdash (a),  
\neg g₀ \vdash () \}  
ps \mapsto \{ g₀ \vdash (a),  
\neg g₀ \vdash () \}  
ps \mapsto \{ g₀ \vdash (b, a),  
\neg g₀ \vdash (b) \}  
```
Solution: type-driven state merging

solve:
    \[
    ps = ()
    \]
    \[
    \text{for } v \text{ in } vs:
    \]
    \[
    \text{if } v > 0:
    \]
    \[
    ps = \text{insert}(v, ps)
    \]
    \[
    \text{assert } \text{len}(ps) == \text{len}(vs)
    \]

\[
g_0 = a > 0
\]
\[
g_1 = b > 0
\]
\[
g_2 = g_0 \land g_1
\]
\[
g_3 = \neg(g_0 \leftrightarrow g_1)
\]
\[
g_4 = \neg g_0 \land \neg g_1
\]
\[
c = \text{ite}(g_1, b, a)
\]

\[
\begin{align*}
vs & \mapsto (a, b) \\
ps & \mapsto ()
\end{align*}
\]

\[
\begin{align*}
ps & \mapsto \{ g_0 \vdash (a), \\
\neg g_0 & \vdash () \}
\end{align*}
\]

\[
\begin{align*}
ps & \mapsto \{ g_1 \vdash (a), \\
\neg g_1 & \vdash () \}
\end{align*}
\]

\[
\begin{align*}
ps & \mapsto \{ g_2 \vdash (b, a), \\
g_3 & \vdash (c), \\
g_4 & \vdash () \}
\end{align*}
\]

\[
\begin{align*}
ps & \mapsto \{ g_0 \vdash (a), \\
\neg g_0 & \vdash () \}
\end{align*}
\]

\[
\begin{align*}
ps & \mapsto \{ g_0 \vdash (b, a), \\
\neg g_0 & \vdash (b) \}
\end{align*}
\]

\[
\begin{align*}
ps & \mapsto \{ g_0 \vdash (a), \\
\neg g_0 & \vdash (a) \}
\end{align*}
\]

symbolic virtual machine
Solution: type-driven state merging

```python
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 \land g_1
g_3 = \neg(g_0 \leftrightarrow g_1)
g_4 = \neg g_0 \land \neg g_1
```

```
c = ite(g_1, b, a)
```

```
assert g_2
```
Solution: type-driven state merging

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

symbolic virtual machine

polyominal encoding
cconcrete evaluation

\[
g_0 = a > 0 \\
g_1 = b > 0 \\
g_2 = g_0 \land g_1 \\
g_3 = \neg(g_0 \leftrightarrow g_1) \\
g_4 = \neg g_0 \land \neg g_1 \\
c = \text{ite}(g_1, b, a)
\]
assert \( g_2 \)
**Solution: type-driven state merging**

**SymPro (OOPSLA’18):** use **symbolic profiling** to find performance bottlenecks in solver-aided code.

**concrete evaluation**

**polynomial encoding**

solve:

```python
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 \land g_1  
g_3 = \neg(g_0 \Leftrightarrow g_1)  
g_4 = \neg g_0 \land \neg g_1  
c = \text{ite}(g_1, b, a)
assert g_2

**symbolic virtual machine**

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

```
\neg g_0  
\neg 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 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!
30+ tools
programming languages, software engineering, systems, architecture, networks, security, formal methods, databases, education, games, ...

programming languages, formal methods, and software engineering
- type systems and programming models
- compilation and parallelization
- safety-critical systems
- test input generation
- software diversification

education and games
- hints and feedback
- problem generation
- problem-solving strategies

systems, architecture, networks, security, and databases
- memory models
- OS components
- data movement for GPUs
- router configuration
- cryptographic protocols
programming languages, formal methods, and software engineering
- type systems and programming models
- compilation and parallelization
- safety-critical systems [CAV’16]
- test input generation
- software diversification

education and games
- hints and feedback
- problem generation
- problem-solving strategies

systems, architecture, networks, security, and databases
- memory models
- OS components
- data movement for GPUs
- router configuration
- cryptographic protocols
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

Prescription  
Sensors

Therapy Control Software

Beam, motors, etc.
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

Found safety-critical defects in a pre-release version of the therapy control software. Used by CNTS staff to verify changes to the controller.
Summary

Today
  • Going pro with solver-aided programming.

Next lecture
  • Getting started with SAT solving!