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.

Announcements
• HW1 is out.
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.
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

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

- verify
- debug
- solve
- synthesize

expertise in PL, FM, SE

symbolic compiler

SMT solver

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

verify
dev
solve
synthesize

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

an interpreter for the source language

programming
Wanted: an easier way to build tools

verify
dump
solve
synthesize

assert safe(x, P(x))

P(x) {
  ...
  ...
}

ROSSETTE
symbolic virtual machine

an interpreter for the source language

programming

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: DSLs and hosts

A domain-specific language (DSL) is a formal language that is specialized to a particular application domain and often limited in capability.

A host language is a high-level language for implementing DSLs, usually with meta-programming features.
A domain-specific language (DSL) is 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: DSLs and hosts

- **host language**
  - **library** *(shallow) embedding*
  - **interpreter** *(deep) embedding*

- **domain-specific language (DSL)**
Layers of classic languages: many DSLs and hosts

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

- **host language**

- **artificial intelligence**
  - Church, BLOG

- **databases**
  - SQL, Datalog

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

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

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

- Racket, Scala, JavaScript, …
Layers of classic languages: why DSLs?

domain-specific language (DSL)

library
(shallow) embedding

interpreter
(deep) embedding

host language

Eigen / Matlab

\[ C = A \times B \]

C / Java

\[
\begin{align*}
&\text{for } (i = 0; i < n; i++) \\
&\quad \text{for } (j = 0; j < m; j++) \\
&\quad \quad \text{for } (k = 0; k < p; k++) \\
&\quad \quad \quad C[i][k] += A[i][j] \times B[j][k]
\end{align*}
\]
Layers of classic languages: why DSLs?

C = A * B

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

C = A * B

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]

Eigen / Matlab

C / Java
Layers of classic languages: why DSLs?

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

- **host language**

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

- **Eigen / Matlab**
  \[ C = A \times B \]  
  [associativity]

Easier for tools to analyze.

- **C / Java**

```java
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    for (k = 0; k < p; k++)
      C[i][k] += A[i][j] * B[j][k]
```
Layers of solver-aided languages

solver-aided domain-specific language (SDSL)

library (shallow) embedding → interpreter (deep) embedding

solver-aided host language
Layers of solver-aided languages: tools as SDSLs

solver-aided domain-specific language (SDSL)

library (shallow) embedding

interpreter (deep) embedding

Rosette

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

synthesis-aided compilation
LinkiT tools, Chlorophyll (PLDI’14),
GreenThumb (ASPLOS’16)

type system soundness
Bonsai (POPL’18)

computer architecture
MemSynth (PLDI’17)

databases
Cosette (CIDR’17)

radiation therapy control
Neutrons (CAV’16)

... and more
Layers of solver-aided languages: tools as SDSLs

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

- **library** (shallow) embedding
- **interpreter** (deep) embedding

- ROSETTE

---

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

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

**type system soundness**
- Bonsai (POPL’18)

**computer architecture**
- MemSynth (PLDI’17)

**databases**
- Cosette (CIDR’17)

**radiation therapy control**
- Neutrons (CAV’16)

... and more
A tiny example SDSL

```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
```

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

```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**, **debug**, and **synthesize** programs in the BV SDSL.
A tiny example SDSL

```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 [10 LOC]
2. verifier [free]
3. debugger [free]
4. synthesizer [free]

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

```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 SDSL

```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)))

A tiny example SDSL

```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
```

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

```ruby
(out opcode in ...)
```
A tiny example SDSL

```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)

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

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

(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)))

0 -2
1 -1
2
3
4
5
6
A tiny example SDSL

```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)
```

(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)
```

```
(define bvmax 0 -2
 `(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 (eval opcode))
        (define args (map load in))
        (store out (apply op args)))]))
  (load (last)))
```
A tiny example SDSL

```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)))
```

```python
interpret
```
A tiny example SDSL

```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
```

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

```
(define bvmax (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**

```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 (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 (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

(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 (eval opcode))
        (define args (map load in))
        (store out (apply op args))])
    (load (last))))
A tiny example SDSL

```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)
-1
```

```scheme
(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

```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)
```

(\texttt{define-symbolic} \texttt{x} \texttt{y} \texttt{int32}?)
(\texttt{define in (list x y)})
(\texttt{verify})
(\texttt{assert (equal? (interpret bvmax in) (interpret max in))))
A tiny example SDSL

```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.

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

```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)
```

Creating 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) (interpret max in))))
```

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

```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)
```

**(define-symbolic x y int32?)**
**(define in (list x y))**
**(verify)**

(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

```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]
```

(\texttt{define-symbolic x y int32?})
(\texttt{define in (list x y)})
(\texttt{verify})
(\texttt{assert (equal? (interpret bvmax in) (interpret max in)))})
A tiny example SDSL

```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
```

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

> debug(bvmax, max, [0, -2])

(define in (list (int32 0) (int32 -2)))
(debug [register?]
    (assert (equal? (interpret bvmax in)
                    (interpret max in))))
A tiny example SDSL

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

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

> synthesize(bvmax, max)

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

> \textbf{synthesize}(bvmax, \textit{max})

\begin{verbatim}
(define-symbolic x y int32?)
(define in (list x y))
(synthesize
  #:forall in
  #:guarantee
  (assert (equal? (interpret bvmax in) (interpret max in))))
\end{verbatim}
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

1. Query
2. Program
3. SDSL
4. Rosetta
5. Symbolic Virtual Machine
6. SMT solver Z3
How it all works: a big picture view

- **query**
- **result**
- **program**
- **SDSL**
- **ROSETTE**
- **Symbolic Virtual Machine**
- **SMT solver**
  - **Z3**
How it all works: a big picture view

- **query**
- **result**

**program**

- SDSL

**ROSETTE**

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

Theories of bitvectors, integers, reals, and uninterpreted functions

**Symbolic Virtual Machine**

**SMT solver Z3**
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)

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

ps = (1, 3)
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)
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:
\[ ps = () \]
\[ \text{for } v \text{ in } vs: \]
\[ \quad \text{if } v > 0: \]
\[ \quad \quad ps = \text{insert}(v, ps) \]
\[ \text{assert } \text{len}(ps) == \text{len}(vs) \]

symbolic execution

bounded model checking

\[ \{ a > 0 \} \]
\[ \{ b \leq 0 \} \]
\[ \{ \text{false} \} \]
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

\[
\begin{align*}
    & a \leq 0 \\
    & b \leq 0 \\
    & \text{false} \\
\end{align*}
\]

\[
\begin{align*}
    & a \leq 0 \\
    & b \leq 0 \\
    & \text{false} \\
\end{align*}
\]

\[
\begin{align*}
    & a \leq 0 \\
    & b > 0 \\
\end{align*}
\]

\[
\begin{align*}
    & a > 0 \\
    & b \leq 0 \\
\end{align*}
\]

\[
\begin{align*}
    & a > 0 \\
    & b > 0 \\
\end{align*}
\]

\[
\begin{align*}
    & a > 0 \\
    & b > 0 \\
    & \text{true} \\
\end{align*}
\]
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 ↦ ()

b > 0
ps ↦ (b)

a > 0
ps ↦ (a)

ps ↦ ps₀

ps₀ = ite(a > 0, (a), ( ))

symbolic execution
vs ↦ (a, b)
ps ↦ ()

a ≤ 0
ps ↦ ()

b ≤ 0
ps ↦ ()

b > 0
ps ↦ (b)

a > 0
ps ↦ (a)

{ a ≤ 0 \lor b > 0 \lor \text{false} } \lor { a > 0 \lor b ≤ 0 \lor \text{false} } \lor { a > 0 \lor b > 0 \lor \text{true} }
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 \mapsto (a, b)
ps \mapsto ()

\begin{align*}
    a & \leq 0 \quad & b & > 0 \\
    ps & \mapsto () & ps & \mapsto (b) \\
    \{a \leq 0 \} \lor \{b > 0 \} & \lor \{false\}
\end{align*}

\begin{align*}
    a & > 0 \quad & b & \leq 0 \\
    ps & \mapsto (a) & ps & \mapsto (b, a) \\
    \{a > 0 \} \lor \{b \leq 0 \} & \lor \{false\}
\end{align*}

\begin{align*}
    a & > 0 \quad & b & > 0 \\
    ps & \mapsto () & ps & \mapsto (a) \\
    \{a > 0 \} \lor \{b > 0 \} & \lor \{true\}
\end{align*}

\begin{align*}
    a & \leq 0 \quad & b & \leq 0 \\
    ps & \mapsto () & ps & \mapsto () \\
    \{a \leq 0 \} \lor \{b \leq 0 \} & \lor \{false\}
\end{align*}

ps_0 = \text{ite}(a > 0, (a), ( ))
ps_1 = \text{insert}(b, ps_0)
Design space of precise symbolic encodings

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

symbolic execution

bounded model checking

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

a \leq 0
ps \mapsto ()

b \leq 0
ps \mapsto ()

\{ a \leq 0, b \leq 0 \}
\{ false \}
\lor
\{ a \leq 0, b > 0 \}
\{ false \}
\lor
\{ a > 0, b \leq 0 \}
\{ true \}
\lor
\{ a > 0, b > 0 \}
\{ true \}

ps_0 = \text{ite}(a > 0, (a), ( ))
ps_1 = \text{insert}(b, ps_0)
ps_2 = \text{ite}(b > 0, ps_0, ps_1)
assert \text{len}(ps_2) = 2
A new design: type-driven state merging

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

\{ ¬g ⊨ a, g ⊨ () \}
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.

`go = a > 0`

```
symbolic virtual machine
```

```
vs ⇔ (a, b)
ps ⇔ ()
g0
ps ⇔ ( )
g0
ps ⇔ (a)
```

```
ps ⇔ { go \vdash (a),
         \neg go \vdash ( ) }
```

```
...
```
A new design: type-driven state merging

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

Execute insert concretely on all lists in the union.

G_0 = a > 0
G_1 = b > 0

\[\begin{align*}
    \text{symbolic virtual machine}\end{align*}\]

\[\begin{align*}
    \text{vs} &\mapsto (a, b) \\
    \text{ps} &\mapsto () \\
    \neg G_0 &\mapsto () \\
    G_0 &\mapsto (a) \\
    ps &\mapsto () \\
    ps &\mapsto \{ G_0 \vdash (a), \\
    \neg G_0 \vdash () \}\}
\end{align*}\]
A new design: type-driven state merging

solve:

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

\[ g_0 = a > 0 \]
\[ g_1 = b > 0 \]
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`.

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

symbolic virtual machine

polynomial encoding
concrete evaluation

g₀ = a > 0
g₁ = b > 0
g₂ = g₀ ⋀ g₁
g₃ = ¬(g₀ ⇔ g₁)
g₄ = ¬g₀ ⋀ ¬g₁

assert g₂
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

- 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

EPICS program

EPICS verifier

safety property

bug report

Prototyped in a few days and found bugs.

Calvin Loncaric
Verifying a radiation therapy system

EPICS program \rightarrow \text{EPICS verifier} \rightarrow \text{bug report}

EPICS verifier

safety property

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

[Persteiner et al., CAV'16]
Synthesizing memory models

Memory consistency models define memory reordering behaviors on multiprocessors.

\[
\begin{array}{c|c|c|c|c}
\text{x} &=& \text{y} &=& 0 \\
\hline
\text{a} &=& \text{x} &|& \text{b} &=& \text{y} \\
\text{y} &=& 1 &|& \text{x} &=& 1 \\
\hline
\text{a} &\equiv& \text{b} &\equiv& 1
\end{array}
\]
Synthesizing memory models

Memory consistency models define memory reordering behaviors on multiprocessors.

\[
\begin{array}{c|c|c}
\hline
x &= y &= 0 \\
\hline
a &= x & b = y \\
y &= 1 & x = 1 \\
\hline
a &\equiv& b &\equiv 1
\end{array}
\]

Forbidden by sequential consistency.
Allowed by x86 and other hardware memory models.
Synthesizing memory models

Memory consistency models define memory reordering behaviors on multiprocessors.

\[
\begin{align*}
\text{x} &= \text{y} = 0 \\
\text{a} &= \text{x} \\
\text{b} &= \text{y} \\
\text{y} &= 1 \\
\text{x} &= 1 \\
\text{a} &\equiv \text{b} \equiv 1
\end{align*}
\]

Forbidden by sequential consistency.

Allowed by x86 and other hardware memory models.

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

Memory consistency models define memory reordering behaviors on multiprocessors.

\[
\begin{align*}
    x &= y = 0 \\
    a &= x \\
    b &= y \\
    y &= 1 \\
    x &= 1 \\
    a \equiv b \equiv 1
\end{align*}
\]

Forbidden by sequential consistency.

Allowed by x86 and other hardware memory models.

A framework sketch

A set of litmus tests

Relational logic synthesizer

Memory model specification
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

A framework sketch

A set of litmus tests

Relational logic synthesizer

Memory model specification

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.

[Bornholt and Torlak, PLDI’17]
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.
**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

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.

The 'big hint” strategy.
Synthesizing strategies for games and education

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.

The ‘big hint’ strategy.
Synthesizing strategies for games and education

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

Game mechanics

Game states for training and testing

Strategy DSL synthesizer

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

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

- Game mechanics
- Strategy DSL synthesizer
- Game states for training and testing

[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!