Computer-Aided Reasoning for Software

Solver-Aided Programming II

Emina Torlak
drolak@cs.washington.edu
Topics

Last lecture

• Getting started with solver-aided programming.

Today

• Going pro with solver-aided programming.
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.

\[
\exists x . \neg \text{safe}(x, P(x)) \\
\text{x} = 42 \land \text{safe}(x, P(x)) \\
\exists v . \text{safe}(42, P_v(42)) \\
\exists e . \forall x . \text{safe}(x, P_e(x))
\]
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!

```
P(x) {
  ...
  ...
}
assert safe(x, P(x))
```
The classic (hard) way to build a tool

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

\[ P(x) \begin{cases} \ldots \\ \ldots \end{cases} \]
assert safe(x, P(x))
Wanted: an easier way to build tools

- verify
- debug
- solve
- synthesize

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

- programming
- Rosette
- SMT solver
- symbolic virtual machine
- an interpreter for the source language
Wanted: an easier way to build tools

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

[Torlak & Bodik, PLDI’14]

\[
P(x) \{
  ...
  ...
}\]

assert safe(x, P(x))

verify debug solve synthesize
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

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

- **host language**
  - A high-level language for implementing DSLs, usually with meta-programming features.
Layers of classic languages: DSLs and hosts

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

- **host language**

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

**A high-level language for implementing DSLs, usually with meta-programming features.**
Layers of classic languages: many DSLs and hosts

- **domain-specific language (DSL)**

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

- **host language**

- **artificial intelligence**
  - Church, BLOG

- **databases**
  - SQL, Datalog

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

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

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

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

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

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

- **C / Java**
  
  ```c
  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.

\[
C = A \times B
\]

```c
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]
```

```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]
```

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**

\[ C = A \times B \]  

**Easier for tools to analyze.**

**C / Java**

```cpp
for (i = 0; i < n; i++)
for (j = 0; j < m; j++)
for (k = 0; k < p; k++)
C[i][k] += A[i][j] \times 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**
Chlorophyll (PLDI’14), GreenThumb (ASPLOS’16)

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

**systems software**
Serval (SOSP’19)

**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

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

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

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

- **systems software**
  - Serval (SOSP’19)

- **databases**
  - Cosette (CIDR’17)

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

... and more
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.
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
```

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

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
`\((2 \ bv\text{\texttt{ge}} 0 1)\)
(3 \ bv\text{\texttt{neg}} 2)
(4 \ bv\text{\texttt{xor}} 0 2)
(5 \ bv\text{\texttt{and}} 3 4)
(6 \ bv\text{\texttt{xor}} 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

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

```
(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

```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)
```

```rossette
(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)))
```

> `(-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)))
```

```plaintext
(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

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

> \text{bvmax}(-2, -1)

\[
(\text{define } \text{bvmax} \\
\quad `((2 \text{ bvsge } 0 1) \\
\quad (3 \text{ bvneg } 2) \\
\quad (4 \text{ bvxor } 0 2) \\
\quad (5 \text{ bvand } 3 4) \\
\quad (6 \text{ bvxor } 1 5)))
\]

\[
(\text{define} (\text{interpret prog inputs}) \\
\quad (\text{make-registers prog inputs}) \\
\quad (\text{for} ([\text{stmt prog}]) \\
\quad \quad (\text{match stmt}) \\
\quad \quad \quad \quad ([\text{list out opcode in ...}]) \\
\quad \quad \quad \quad \quad (\text{define op (eval opcode)}) \\
\quad \quad \quad \quad \quad (\text{define args (map load in)}) \\
\quad \quad \quad \quad \quad (\text{store out (apply op args)}))) \\
\quad (\text{load (last)}))
\]
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 `((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 (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)
```

```
(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))
```

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

```
(define (interpret 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
```

```rossette
(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)
-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

> 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

```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)
```

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

```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)
```

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

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 bvmx 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

```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]
```

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

---

> `define` symbolically `x` and `y` as `int32?`
> `define` `in` as a list of `x` and `y`
> `verify` the assertion that `equal?` (`interpret bvmax in`) and `interpret max in` should be equal

---

**query**
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
```

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

> bvmax(0, -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

> 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 = 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

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

> synthesize(bvmax, max)
```

```scheme
(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

> synthesize(bvmax, max)
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. **Rosette**
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

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

theories of bitvectors, integers, reals, and uninterpreted functions

SDSL

ROSETTE

Symbolic Virtual Machine

SMT solver Z3
Translation to constraints by example

\[
\text{solve:}
\] \[\text{ps = ()}
\text{for v in vs: if v > 0: ps = insert(v, ps)}
\text{assert len(ps) == len(vs) reverse and filter, keeping only positive numbers}
\] \[\text{vs} (3, 1, -2) \quad \text{ps} (1, 3)\]
Translation to constraints by example

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

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

**Constraints:**

```
a > 0 \land b > 0
```
Design space of symbolic encodings: SE and BMC

solve:
    ps = ()
    for v in vs:
        if v > 0:
            ps = insert(v, ps)
    assert len(ps) == len(vs)
Design space of symbolic encodings: SE and BMC

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 symbolic encodings: SE and BMC

solve:
ps = ()
for v in vs:
    if v > 0:
        ps = insert(v, ps)
assert len(ps) == len(vs)
Design space of symbolic encodings: SE and BMC

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

bounded model checking

vs ↦ (a, b)
p0 ↦ ()

assert len(ps) = 2

a > 0
b ≤ 0
false

ps0 = ite(a > 0, (a), ( ))

Design space of symbolic encodings: SE and BMC

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

symbolic execution

bounded model checking

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

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

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

\[
\begin{align*}
\text{ps} & \mapsto \text{ps}_0 \\
\text{ps} & \mapsto \text{ps}_1
\end{align*}
\]

\[
\begin{align*}
\text{ps}_0 & = \text{ite}(a > 0, (a), ( )) \\
\text{ps}_1 & = \text{insert}(b, \text{ps}_0)
\end{align*}
\]
Design space of symbolic encodings: SE and BMC

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

bounded model checking

\[ vs \mapsto (a, b) \]
\[ ps \mapsto ( ) \]
\[ a \leq 0 \]
\[ ps \mapsto ( ) \]
\[ b \leq 0 \]
\[ ps \mapsto ( ) \]
\[ a > 0 \]
\[ ps \mapsto (a) \]
\[ b > 0 \]
\[ ps \mapsto (b, a) \]
\[ \{ a \leq 0 \} \lor \{ a > 0 \} \lor \{ b \leq 0 \} \lor \{ b > 0 \} \]

\[ ps_0 = \text{ite}(a \geq 0, (a), ( )) \]
\[ ps_1 = \text{insert}(b, ps_0) \]
\[ ps_2 = \text{ite}(b > 0, ps_0, ps_1) \]
assert len(ps_2) = 2
Design space of symbolic encodings: best of all worlds?

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

Can we have both a polynomially sized encoding (like BMC) and concrete evaluation of complex operations (like SE)?
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)

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

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:

```python
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.

\[
\begin{align*}
g_0 &= a > 0 \\
g_1 &= b > 0
\end{align*}
\]

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

symbolic virtual machine
A new design: type-driven state merging

solve:

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

\[
\begin{align*}
g_0 &= a > 0 \\
g_1 &= b > 0
\end{align*}
\]
A new design: type-driven state merging

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

g_0 = a > 0
g_1 = b > 0
g_2 = g_0 ∧ g_1
\neg g_0 = \neg (g_0 ↔ g_1)
g_3 = \neg g_0 ∧ \neg g_1
c = \text{ite}(g_1, b, a)

symbolic virtual machine

ps \mapsto ( )
ps \mapsto (a)
ps \mapsto (b)

\neg g_0 \mapsto ( )
g_0 \mapsto (a)
\neg g_0 \mapsto (b)
g_0 \mapsto (a, b)
\neg g_0 \mapsto (c)
g_0 \mapsto ( )
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 \( \text{len} \) 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
solve:
ps = ()
for v in vs:
    if v > 0:
        ps = insert(v, ps)
assert len(ps) == len(vs)

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

\textbf{solve:}

\begin{verbatim}
ps = ()
for v in vs:
    if v > 0:
        ps = insert(v, ps)
assert len(ps) == len(vs)
\end{verbatim}

\textbf{SymPro (OOPSLA’18): use \textit{symbolic profiling} to find performance bottlenecks in solver-aided code.}

\textbf{polynomial encoding}

\begin{verbatim}
g0 = a > 0
g1 = b > 0
g2 = g0 \land g1
g3 = \neg(g0 \leftrightarrow g1)
g4 = \neg g0 \land \neg g1
c = \text{ite}(g1, b, a)
assert g2
\end{verbatim}

\textbf{concrete evaluation}

\verb|vs| $\mapsto$ \((a, b)\)
\verb|ps| $\mapsto$ \((\ )\)
\verb|\neg g0| $\mapsto$ \((\ )\)
\verb|\neg g1| $\mapsto$ \((\ )\)
\verb|ps| $\mapsto$ \{ \verb|g0| $\vdash$ \(a\), \newline \verb|\neg g0| $\vdash$ \((\ )\) \}
\verb|ps| $\mapsto$ \{ \verb|g0| $\vdash$ \(b, a\), \newline \verb|\neg g0| $\vdash$ \(b\) \}
\verb|ps| $\mapsto$ \{ \verb|g2| $\vdash$ \(b, a\), \newline \verb|g3| $\vdash$ \(c\), \newline \verb|g4| $\vdash$ \((\ )\) \}

\textbf{concrete evaluation}
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
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 [CAV’16]
- test input generation
- software diversification

education and games
- hints and feedback
- problem generation
- problem-solving strategies [VMCAI’18, FDG’17]

systems, architecture, networks, security, and databases
- memory models
- OS components [SOSP’19]
- 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

EPICS program

safety property

EPICS verifier

bug report
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

EPICS program

safety property

EPICS verifier

bug report

[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 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

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.
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
Synthesizing strategies for games and education

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

Eric Butler

Prototyped in a few weeks and synthesized real strategies.
Synthesizing strategies for games and education

A strategy DSL synthesizer takes game mechanics and game states for training and testing to produce 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.
Verifying systems software

An OS is a set of software components that mediate access to hardware and provide services to user applications.

<table>
<thead>
<tr>
<th>applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>JIT compilers</td>
</tr>
<tr>
<td>security monitors</td>
</tr>
<tr>
<td>file systems</td>
</tr>
<tr>
<td>device drivers</td>
</tr>
<tr>
<td>OS kernel</td>
</tr>
<tr>
<td>hardware</td>
</tr>
</tbody>
</table>
Verifying systems software

An OS is a set of software components that mediate access to hardware and provide services to user applications.

<table>
<thead>
<tr>
<th>applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>JIT compilers</td>
</tr>
<tr>
<td>security monitors</td>
</tr>
<tr>
<td>file systems</td>
</tr>
<tr>
<td>device drivers</td>
</tr>
</tbody>
</table>

OS kernel

hardware

Bugs in OS components are bad news for reliability, security, and performance of computer systems.
Verifying systems software

An OS is a set of software components that mediate access to hardware and provide services to user applications.

<table>
<thead>
<tr>
<th>applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>JIT compilers</td>
</tr>
<tr>
<td>security monitors</td>
</tr>
<tr>
<td>file systems</td>
</tr>
<tr>
<td>device drivers</td>
</tr>
<tr>
<td>OS kernel</td>
</tr>
<tr>
<td>hardware</td>
</tr>
</tbody>
</table>

Bugs in OS components are bad news for reliability, security, and performance of computer systems.

Verifying OS components is hard: e.g., the Komodo security monitor took 2 person-years to prove, with a proof-to-implementation ratio of 6:1.
Verifying systems software

An OS is a set of software components that mediate access to hardware and provide services to user applications.

<table>
<thead>
<tr>
<th>applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>JIT compilers</td>
</tr>
<tr>
<td>security monitors</td>
</tr>
<tr>
<td>file systems</td>
</tr>
<tr>
<td>device drivers</td>
</tr>
<tr>
<td>OS kernel</td>
</tr>
<tr>
<td>hardware</td>
</tr>
</tbody>
</table>

Bugs in OS components are bad news for reliability, security, and performance of computer systems.

- **System specification**
- **System (binary) implementation**
- **Serval verifiers**
  - RISCV, x86, LLVM, BPF

Bug report or correctness guarantee
Verifying systems software

System specification → System (binary) implementation → Serval verifiers (RISCV, x86, LLVM, BPF) → Bug report or correctness guarantee

Each verifier took a couple of weeks to build!

Luke Nelson
Verified three existing security monitors (CertiKOS, Komodo, Keystone) fully automatically.

Found 15 new bugs in the Linux BPF JITs for RISCV64 and x86-32, all confirmed and fixed by developers.

[Verification process diagram]

[Verification process diagram description]
Summary

Today

• Going pro with solver-aided programming.

Next lecture

• Getting started with SAT solving!