Solver-Aided Languages

Emina Torlak
eminaw@cs.washington.edu
Today

Last lecture
• Program synthesis

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

Reminders
• Next lecture: metasketches!
• Project presentations next Friday in class
  • 11 min per team: 8 min presentation + 3 min questions
• Project reports and prototypes due next Friday at 11:00pm
a little programming for everyone
A little programming for everyone

Every knowledge worker wants to program …
A little programming for everyone

Every knowledge worker wants to program …

› spreadsheet data manipulation
A little programming for everyone

Every knowledge worker wants to program …

› spreadsheet data manipulation
› models of cell fates
A little programming for everyone

Every knowledge worker wants to program …

- spreadsheet data manipulation
- models of cell fates
- cache coherence protocols
- memory models
A little programming for everyone

Every knowledge worker wants to program …

- spreadsheet data manipulation [Flashfill, POPL’11]
- models of cell fates [SBL, POPL’13]
- cache coherence protocols [Transit, PLDI’13]
- memory models [MemSAT, PLDI’10]
A little programming for everyone

We all want to build programs …

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

solver-aided languages

less time

less expertise

hardware designer
biologist
social scientist
outline
solver-aided tools
solver-aided tools, languages
solver-aided tools, languages, and applications
solver-aided tools
Programming ...

specification

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

8
Programming ...

```
P(x) {
    ...
    ...
}
assert safe(P(2))
```
Programming with a solver-aided tool

P(x) {
  ...
  ...
}
assert safe(P(2))
Programming with a solver-aided tool

∃ \ x . ¬ safe(P(x))

Find an input on which the program fails.

verify

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

\[ 42 \]

Find an input on which the program fails.

SAT/SMT solver

CBMC [Kroening et al., DAC’03]
Dafny [Leino, LPAR’10]
Miniatur [Vaziri et al., FSE’07]
Klee [Cadar et al., OSDI’08]
Programming with a solver-aided tool

Find an input on which the program fails.
Localize bad parts of the program.

P(x) {
    v = x + 2
    ...
}  
assert safe(P(x))

x = 42 ∧ safe(P(x))

BugAssist [Jose & Majumdar, PLDI'11]
Programming with a solver-aided tool

P(x) {
  v = choose()
  ...
} assert safe(P(x))

Find an input on which the program fails.
Localize bad parts of the program.
Find values that repair the failing run.

∃v. safe(P(42, v))

Kaplan [Koksal et al, POPL’12]
PBnJ [Samimi et al., ECOOP’10]
Squander [Milicevic et al., ICSE’11]
Programming with a solver-aided tool

P(x) {
  v = ??
  ...
}  
assert safe(P(x))

Find an input on which the program fails.
Localize bad parts of the program.
Find values that repair the failing run.
Find code that repairs the program.

Sketch [Solar-Lezama et al., ASPLOS’06]
Comfusy [Kuncak et al., CAV’10]
The standard (hard) way to build a tool

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

expertise in PL, FM, SE

verify debug solve synth

translator

SAT/SMT solver
A new, easy way to build tools

P(x) {
  ...
  ...
} assert safe(P(x))
A new, easy way to build tools

 Implement a language for an application domain, get the tools for free!

 `P(x) { ...
    ...
} assert safe(P(x))`
A new, easy way to build tools

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

Implement a language for an application domain, get the tools for free!

an interpreter or a library
A new, easy way to build tools

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

Implement a language for an application domain, get the tools for free!

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

[Rosette: Symbolic Virtual Machine](Torlak & Bodik, PLDI’14, Onward’13)
design

solver-aided languages
Layers of languages

**domain-specific language (DSL)**

- library
- interpreter

**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 languages

- **domain-specific language (DSL)**
  - library
  - interpreter
  - host language

- **artificial intelligence**
  - Church, BLOG

- **databases**
  - SQL, Datalog

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

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

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

- Scala, Racket, JavaScript
Layers of languages

domain-specific language (DSL)

library interpreter

host language

C = A * B

[associativity]

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
  - interpreter

- **solver-aided host language**

- **symbolic virtual machine**
Layers of **solver-aided languages**

- **solver-aided domain-specific language (SDSL)**
  - library
  - interpreter
- **solver-aided host language**
- **symbolic virtual machine**

**Rosette**

[Torlak & Bodik, *Onward’13, PLDI’14*]
Layers of solver-aided languages

- solver-aided domain-specific language (SDSL)
  - library
  - interpreter

- solver-aided host language

- symbolic virtual machine

Applications:
- spatial programming
  - Chlorophyll
- intelligent tutoring
  - RuleSynth
- memory models
  - MemSynth
- optimal synthesis
  - Synapse
- radiotherapy controllers
  - Neutrons
- BGP router configurations
  - BagPipe

[Torlak & Bodik, Onward’13, PLDI’14]
Layers of solver-aided languages

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

- **solver-aided host language**

- **symbolic virtual machine**

- **spatial programming**
  - Chlorophyll

- **intelligent tutoring**
  - RuleSynth

- **memory models**
  - MemSynth

- **optimal synthesis**
  - Synapse

- **radiotherapy controllers**
  - Neutrons

- **BGP router configurations**
  - BagPipe

[Torlak & Bodik, *Onward’13, PLDI’14*]
Anatomy of a solver-aided host language

Modern descendent of Scheme with macro-based metaprogramming.

Racket
Anatomy of a solver-aided host language

(define-symbolic id type)
(assert expr)
(verify expr)
(debug [expr] expr)
(solve expr)
(synthesize [expr] expr)
A tiny example SDSL

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

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

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

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

test  debug
verify synth
def bvmax(r0, r1) :
    r2 = bvge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

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

1. interpreter       [10 LOC]
2. verifier           [free]
3. debugger           [free]
4. synthesizer        [free]
A tiny example SDSL:  \textsc{Rosette}

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

> bvmax(-2, -1)
def bvmax(r0, r1) :
    r2 = bvge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> bvmax(-2, -1)
def bvmax(r0, r1):
    r2 = bvge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> bvmax(-2, -1)

A tiny example SDSL:

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

(out opcode in ...)

ROSETTE
A tiny example SDSL:

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

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

```scheme
(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)))
```

ROSETTE

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

`(-2 -1)`
def bvmax(r0, r1):
    r2 = bvge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> bvmax(-2, -1)
-1
A tiny example SDSL:

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

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

```
(define bvmax
  `((2 bvge 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)))
```

>` bvmax(-2, -1)`

-2

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-2</td>
<td>1</td>
<td>-1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>6</td>
<td>-1</td>
<td>2</td>
<td>___</td>
<td>___</td>
</tr>
</tbody>
</table>
```python
def bvmax(r0, r1):
    r2 = bvge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

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

```
(define interpret prog inputs) 0 -2
(make-registers prog inputs) 1 -1
(for ([stmt prog]) 2
  (match stmt 3
    [(list out opcode in ...)] 4
      (define op (eval opcode)) 5
        (define args (map load in)) 6
          (store out (apply op args)])
    (load (last)))
```

**A tiny example SDSL:**

R"\*SETTE"
def bvmax(r0, r1) :
    r2 = bvge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> bvmax(-2, -1)

A tiny example SDSL:

(define bvmax
  `(2 bvge 0 1)
    (3 bvneg 2)
    (4 bvxor 0 2)
    (5 bvand 3 4)
    (6 bvxor 1 5)))
A tiny example SDSL:

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

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

```plaintext
0 -2
1 -1
2 0
3
4
5
6
```

(Rosette)

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

> bvmax(-2, -1)

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

```
(interpret (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  0
 3  0
 4 -2
 5  0
 6 -1
```
A tiny example SDSL:

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

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

Rosette:

```scheme
(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)))
```

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

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

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

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

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

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

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

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

> verify(bvmax, max)
```

(query)

```
(define-symbolic n0 n1 integer?)
(define inputs (list n0 n1))
(verify
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```
def bvmax(r0, r1):
    r2 = bvge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> verify(bvmax, max)
A tiny example SDSL:

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

> verify(bvmax, max)

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

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

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

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

(verify expr) searches for a concrete interpretation of symbolic constants that causes expr to fail.
A tiny example SDSL:

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

```lisp
(define-symbolic n0 n1 integer?)
(define inputs (list n0 n1))
(verify
 (assert (= (interpret bvmax inputs) (interpret max inputs))))
```
A tiny example SDSL:

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

```lisp
(define inputs (list 0 -2))
(debug [input-register?]
    (assert (= (interpret bvmax inputs)
                (interpret max inputs))))
```

> `debug(bvmax, max, (0, -2))`
A tiny example SDSL:

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

> `debug(bvmax, max, (0, -2))`

```query
(define inputs (list 0 -2))
(debug [input-register?]
  (assert (= (interpret bvmax inputs) (interpret max inputs))))
```
A tiny example SDSL:

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

> `synthesize(bvmax, max)`

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

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

> synthesize(bvmax, max)
```

query

```
(define-symbolic n0 n1 integer?)
(define inputs (list n0 n1))
(synthesize [inputs]
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```
symbolic virtual machine (SVM)
How it all works: a big picture view

[RoSETTE]

query

program

SDSL

[Torlak & Bodik, Onward’13]

symbolic

virtual machine

[solver]

Z3

[Torlak & Bodik, PLDI’14]
How it all works: a big picture view

[Torlak & Bodik, Onward’13]

Symbolic
virtual machine

[Torlak & Bodik, PLDI’14]

Solver

Z3

Program

SDSL

Result
How it all works: a big picture view

- Program
- SDSL
- Rosette

- Pattern matching
- Dynamic evaluation
- First-class procedures
- Higher-order procedures
- Side effects
- Macros

Theories of bitvectors, integers, reals, and uninterpreted functions

[Torlak & Bodik, Onward’13]

[Torlak & Bodik, PLDI’14]
Translation to constraints by example

vs
(3, 1, -2) → reverse and filter, keeping only positive numbers → ps
(1, 3)
Translation to constraints by example

vs
(3, 1, -2)

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

ps
(1, 3)
Translation to constraints by example

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

Translate to constraints:

\[(a, b)\]

\[a > 0 \land b > 0\]

```python
solve:
    vs = ()
    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)

symbolic execution

bounded model checking
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 ( )

a \leq 0
ps \mapsto ( )
a > 0
ps \mapsto (a)
b \leq 0
ps \mapsto ( )
b > 0
ps \mapsto (b)
b \leq 0
ps \mapsto (a)
b > 0
ps \mapsto (b, a)

ps_0 = \text{ite}(a > 0, (a), ( ))
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

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:**

\[ ps = () \]

\[ \text{for } v \text{ in } vs: \]

\[ \text{if } v > 0: \]

\[ ps = \text{insert}(v, ps) \]

\[ \text{assert len(ps) == len(vs)} \]

```
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:

\[
\begin{align*}
    ps &= () \\
    \text{for } v \text{ in } vs: \\
    &\quad \text{if } v > 0: \\
    &\quad\quad ps = \text{insert}(v, ps) \\
    \text{assert } \text{len}(ps) == \text{len}(vs)
\end{align*}
\]
A new design: type-driven state merging

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

Merge values of
  ‣ primitive types: symbolically
  ‣ immutable types: structurally
  ‣ all other types: via unions
A new design: type-driven state merging

solve:

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

Merge values of
- primitive types: symbolically
- immutable types: structurally
- all other types: via unions
A new design: type-driven state merging

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

Merge values of
   ‣ primitive types: symbolically
   ‣ immutable types: structurally
   ‣ all other types: via unions
A new design: type-driven state merging

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

**Merge values of**
- primitive types: symbolically
- immutable types: structurally
- all other types: via unions

\[ \{ \neg g \vdash a, g \vdash () \} \]
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.

symbolic virtual machine

\n\begin{align*}
\text{vs} & \mapsto (a, b) \\
\text{ps} & \mapsto ( ) \\
\neg \text{g}_0 & \mapsto ( ) \\
\text{ps} & \mapsto ( ) \\
\text{ps} & \mapsto (a) \\
\text{ps} & \mapsto \{ \text{g}_0 \vdash (a), \\
& \neg \text{g}_0 \vdash ( ) \} \\
\text{ps} & \mapsto \{ \text{g}_0 \vdash (b, a), \\
& \neg \text{g}_0 \vdash (b) \} \\
\text{g}_0 & = a > 0 \\
\text{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)

gs = a > 0
gs = b > 0

g0 = a > 0
gs = 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)

g₀ = a > 0
g₁ = b > 0
g₂ = g₀ ∧ g₁
g₃ = ¬(g₀ ⇔ g₁)
g₄ = ¬g₀ ∧ ¬g₁
c = ite(g₁, b, a)
**A new design: type-driven state merging**

**solve:**
```
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 \( g_2 \).

```symbolic virtual machine```
```
vs ⊢ (a, b)
ps ⊢ ()

\( \neg g_0 \)
ps ⊢ ()
\( g_0 \)
ps ⊢ (a)

\( \neg g_1 \)
ps ⊢ \{ g_0 ⊨ (a), 
\neg g_0 ⊨ () \}
\( g_1 \)
ps ⊢ \{ \neg g_0 ⊨ ( ) \}

\( g_2 = g_0 \land g_1 \)
g_3 = \neg(g_0 \leftrightarrow g_1)
\( \neg g_0 \land \neg g_1 \)
c = 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)

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

\[ \neg g_0 \]
\[ g_0 \]
\[ \neg g_0 \]
\[ g_3 \]
\[ g_4 \]
\[ \neg g_0 \]
\[ g_0 \]
\[ \neg g_0 \]
\[ g_1 \]
\[ \neg g_0 \]
\[ \neg g_0 \]
\[ g_1 \]
\[ \neg g_0 \]
\[ g_0 \]
\[ \neg g_0 \]
\[ g_1 \]
\[ \neg g_0 \]
\[ g_0 \]
\[ \neg g_0 \]
\[ g_1 \]
Effectiveness of type-driven state merging

Merging performance for verification and synthesis queries in SynthCL, WebSynth and IFC programs

\[ R^2 = 0.9884 \]

\[ R^2 = 0.95 \]
Effectiveness of type-driven state merging

SVM and solving time for verification and synthesis queries in SynthCL, WebSynth and IFC programs
solver-aided programming for everyone
Chlorophyll: ultra low-power computing

Instructions/Second vs Power

GreenArrays GA144 Processor

~100x

Figure by Per Ljung
Chlorophyll: ultra low-power computing

GreenArrays GA144 Processor

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

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

Drawing by Mangpo Phothilimthana
Chlorophyll: ultra low-power computing

GreenArrays GA144 Processor

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

\[ c = a \times b \]

Drawing by Mangpo Phothilimthana
Chlorophyll: ultra low-power computing

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

Synthesizes placement of code and data onto cores, by type-checking a program sketch in a C-like DSL.
Chlorophyll: ultra low-power computing

int@1 a, b;
int@3 c = a *@2 b;

type-checking a program sketch in a C-like DSL.

Synthesizes placement of code and data onto cores, by
Chlorophyll: ultra low-power computing

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

Synthesizes placement of code and data onto cores, by type-checking a program sketch in a C-like DSL.
Chlorophyll: ultra low-power computing

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

Built by a first-year grad in a few weeks

Phitchaya Mangpo Phothilimthana
Chlorophyll: ultra low-power computing

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

[Phothilimthana et al., PLDI’14]
Bagpipe: verifying BGP router configurations
Bagpipe: verifying BGP router configurations
Bagpipe: verifying BGP router configurations

Autonomous systems communicate routing information by sending announcements via the Border Gateway Protocol.
Bagpipe: verifying BGP router configurations

Configuring BGP is tricky

- distributed system
- low-level language
- no static analysis
Bagpipe: verifying BGP router configurations

BGP configuration property

Bagpipe

A BGP interpreter implemented in Rosette.

policy violation
Bagpipe: verifying BGP router configurations

BGP configuration → property → Bagpipe → policy violation

Built by two grads in a few weeks

Konstantin Weitz and Doug Woos
Bagpipe: verifying BGP router configurations

Internet2 announcements are not leaked

Bagpipe

route leaks!

[Weitz et al., OOPSLA’16]
Neutrons: verifying a radiotherapy system

Clinical Neutron Therapy System (CNTS) at UW

- 30 years of incident-free service.
- Controlled by custom software, built by CNTS engineering staff.
- Third generation of Therapy Control software built recently.
Neutrons: verifying a radiotherapy system

Clinical Neutron Therapy System (CNTS) at UW

Prescription

Sensors

Therapy Control Software

Beam, motors, etc.
Neutrons: verifying a radiotherapy system

Experimental Physics and Industrial Control System (EPICS) Dataflow Language

Prescription

Sensors

Therapy Control Software

Beam, motors, etc.
Neutrons: verifying a radiotherapy system

**EPICS documentation / semantics**

The Maximize Severity attribute is one of NMS (Non-Maximize Severity), MS (Maximize Severity), MSS (Maximize Status and Severity) or MSI (Maximize Severity if Invalid). It determines whether alarm severity is propagated across links. If the attribute is MSI only a severity of INVALID_ALARM is propagated; settings of MS or MSS propagate all alarms that are more severe than the record's current severity. For input links the alarm severity of the record referred to by the link is propagated to the record containing the link. For output links the alarm severity of the record containing the link is propagated to the record referred to by the link. If the severity is changed the associated alarm status is set to LINK_ALARM, except if the attribute is MSS when the alarm status will be copied along with the severity.
Neutrons: verifying a radiotherapy system

EPICS program

safety property

EPICS Verifier

bug report

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

Calvin Loncaric
Neutrons: verifying a radiotherapy system

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

Therapy Control Software

EPICS Verifier

safety property

[Pernsteiner et al., CAV’16]
Thanks for a great quarter!