Computer-Aided Reasoning for Software

Solver-Aided Languages

Emina Torlak
emina@cs.washington.edu
Today
Today

Last lecture

• Program synthesis
Today

Last lecture
  • Program synthesis

Today
  • The next N years: Solver-Aided Languages (?)
Today

Last lecture

• Program synthesis

Today

• The next N years: Solver-Aided Languages (?)

Reminders

• Please fill out the course evaluation form (Dec 02-08)
• 8 min final presentations on Monday, Dec 08, 10:30am, MGH 254
• Final projects due on Monday, Dec 08, at 11pm
a little programming for everyone
A little programming for everyone

We all want to build programs …
A little programming for everyone

We all want to build programs …

- spreadsheet data manipulation
A little programming for **everyone**

We all want to build programs …

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

We all want to build programs …

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

We all want to build programs …

› 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 code

less time

less effort

hardware designer

biologist

social scientist
A little history

- **program logics** (Floyd, Hoare, Dijkstra)
- **mechanization of logic** (Milner, Pnueli)
- **mechanized tools** (Clarke, Emerson, Sifakis)
A little history

1960 software crisis
1970 program logics (Floyd, Hoare, Dijkstra)
1980 mechanization of logic (Milner, Pnueli)
1990 mechanized tools (Clarke, Emerson, Sifakis)
2000 better programs
A little history

1960
software crisis

1970
program logics (Floyd, Hoare, Dijkstra)

1980
mechanization of logic (Milner, Pnueli)

1990
mechanized tools (Clarke, Emerson, Sifakis)

2000

better programs

6TH SENSE [IBM]

ASTRÉE [AbsInt]

SLAM [MSR]
A little history

- 1960: software crisis
- 1970: program logics (Floyd, Hoare, Dijkstra)
- 1980: mechanization of logic (Milner, Pnueli)
- 1990: mechanized tools (Clarke, Emerson, Sifakis)
- 2000: software gap
A little history

1960  software crisis
1970  program logics (Floyd, Hoare, Dijkstra)
1980  mechanization of logic (Milner, Pnueli)
1990  mechanized tools (Clarke, Emerson, Sifakis)
2000  software gap

2000  SAT/SMT solvers and tools
A little history

1960  software crisis
1970  program logics (Floyd, Hoare, Dijkstra)
1980  mechanization of logic (Milner, Pnueli)
1990  mechanized tools (Clarke, Emerson, Sifakis)
2000  software gap
2010  SAT/SMT solvers and tools

solver-aided languages

better programs

more easily
solver-aided tools, languages and beyond
solver-aided tools
solver-aided tools, languages
solver-aided tools, languages and beyond
solver-aided tools
Programming ...

specification

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

9
Programming ...

test case

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

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

translate(...) → SAT/SMT solver
Solver-aided tools: verification

Find an input on which the program fails.

\[ \exists x . \neg \text{safe}(P(x)) \]

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

assert safe(P(x))

\[ \exists x . \neg \text{safe}(P(x)) \]

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]
Solver-aided tools: verification

Find an input on which the program fails.

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

\[ \exists x . \neg \text{safe}(P(x)) \]

SAT/SMT solver

values

model

CBMC [Kroening et al., DAC’03]
Dafny [Leino, LPAR’10]
Miniatur [Vaziri et al., FSE’07]
Klee [Cadar et al., OSDI’08]
Solver-aided tools: debugging

Localize bad parts of the program.

\begin{verbatim}
P(x) {
  v = x + 2
  ...
}
assert safe(P(x))
\end{verbatim}

\[ x = 42 \land safe(P(x)) \]

SAT/SMT solver

BugAssist [Jose & Majumdar, PLDI’11]
Angelina [Chandra et al., ICSE’11]
Solver-aided tools: debugging

Localize bad parts of the program.

$P(x) \{\quad v = x + 2 \quad \ldots \quad \}$
$\text{assert safe}(P(x))$

$x = 42 \land \text{safe}(P(x))$

SAT/SMT solver

expressions

min core

BugAssist [Jose & Majumdar, PLDI’11]
Angelina [Chandra et al., ICSE’11]
Solver-aided tools: angelic execution

Find values that repair the failing execution.

\[ \exists v . \text{safe}(P(42, v)) \]

42

\[
P(x) \{ 
    v = \text{choose}() 
    
    \text{...} 
}\]

assert safe(P(x))

\[
\exists v . \text{safe}(P(42, v))
\]

SAT/SMT solver

Kaplan [Koksal et al, POPL’12]
PBNJ [Samimi et al., ECOOP’10]
Squander [Milicevic et al., ICSE’11]
Solver-aided tools: angelic execution

Find values that repair the failing execution.

\[
P(x) \{
  v = \text{choose}()
  
  \ldots
\}
\]

assert safe(P(x))

\[\exists v \ . \ safe(P(42, v))\]

Kaplan [Koksal et al, POPL’12]
PBNj [Samimi et al., ECOOP’10]
Squander [Milicevic et al., ICSE’11]
Synthesize code that repairs the program.

$P(x) \{ 
  v = ?? 
  ... 
\}

assert safe(P(x))

$\exists e \ . \ \forall x \ . \ safe(P_e(x))$

SAT/SMT solver

Sketch [Solar-Lezama et al., ASPLOS'06]
Comfusy [Kuncak et al., CAV'10]
Solver-aided tools: synthesis

Synthesize code that repairs the program.

\[ P(x) \{
    v = x - 2
    ...
} \]

assert safe(P(x))

\[ \exists e . \forall x . \text{safe}(P_e(x)) \]

expressions

model

SAT/SMT solver

Sketch [Solar-Lezama et al., ASPLOS'06]
Comfusy [Kuncak et al., CAV'10]
more solver-aided tools ...
wanted

more solver-aided tools ...
Building solver-aided tools: state-of-the-art
Building solver-aided tools: state-of-the-art

I need a tool to create models of biological cells …

learn the problem domain

tools expert

domain expert
Building solver-aided tools: state-of-the-art

- Learn the problem domain
- Design a domain language

I need a tool to create models of biological cells …

Abstractions for cells, components, interactions, …

Tools expert

Domain expert
Building solver-aided tools: state-of-the-art

learn the problem domain

design a domain language

build a symbolic compiler from the domain language to constraints

months or years of work

I need a tool to create models of biological cells …

A solver-aided tool for creating biological models

tools expert
domain expert

verify execute debug synth
Can we do better?

- design a domain language
- implement an interpreter for the language, get a symbolic compiler for free

weeks

verify execute debug synth

domain expert
Can we do better?

Design a domain language

Implement an interpreter for the language, get a symbolic compiler for free

Verify, execute, debug, synthesize

A solver-aided domain-specific language (SDSL)

A solver-aided host language

Domain expert
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**
  - Scala, Racket, JavaScript

---

**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**

**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]
```

**Eigen / Matlab**

```plaintext
C = A * B
```

[associativity]
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)
- solver-aided host language
- symbolic virtual machine

[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
- data-parallel programming
  - SynthCL
- web scraping
  - WebSynth
- secure stack machines
  - IFC

[Torlak & Bodik, Onward’13, PLDI’14]
SDSLs developed with ROSETTE

<table>
<thead>
<tr>
<th>SDSL</th>
<th>Development Time (weeks)</th>
<th>Developer</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chlorophyll</td>
<td>16</td>
<td>first-year grad</td>
<td>(grad)</td>
</tr>
<tr>
<td>SynthCL</td>
<td>12</td>
<td>expert</td>
<td></td>
</tr>
<tr>
<td>WebSynth</td>
<td>4</td>
<td>undergrad</td>
<td></td>
</tr>
<tr>
<td>IFC</td>
<td>1</td>
<td>expert</td>
<td></td>
</tr>
</tbody>
</table>
SDSLs developed with ROSETTE

Spatial programming for a low-power chip, using synthesis to partition code and data across 144 tiny cores.

- Chlorophyll (first-year grad)
- SynthCL (expert)
- WebSynth (undergrad)
- IFC (expert)

X + Z

GreenArrays GA144
SDSLs developed with Rosette

Optimal partitioning synthesized in minutes, while manual partitioning takes days [Phothilimthana et al., PLDI'14].

<table>
<thead>
<tr>
<th>SDSL</th>
<th>Development Time (weeks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chlorophyll</td>
<td>16 (first-year grad)</td>
</tr>
<tr>
<td>SynthCL</td>
<td>12 (expert)</td>
</tr>
<tr>
<td>WebSynth</td>
<td>4 (undergrad)</td>
</tr>
<tr>
<td>IFC</td>
<td>2 (expert)</td>
</tr>
</tbody>
</table>

GreenArrays
GA144

\[
X + Z
\]
SDSLs developed with Rosette

Verification and synthesis for data-parallel programming with OpenCL.
SDSLs developed with Rosette

Used by a novice to develop new vectorized kernels that are as fast as expert code.
SDSLs developed with Rosette

Synthesis of web scraping scripts from examples (PBE).
SDSLs developed with Rosette

Works on real web pages (e.g., iTunes) in seconds.
SDSLs developed with ROSETTE

Verification for executable specifications of secure stack machines.
SDSLs developed with ROSETTE

Finds all bugs reported by a specialized tool [Hritcu et al., ICFP’13].

Chlorophyll (first-year grad)
SynthCL (expert)
WebSynth (undergrad)
IFC (expert)
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
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.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. interpreter</td>
<td>[10 LOC]</td>
</tr>
<tr>
<td>2. verifier</td>
<td>[free]</td>
</tr>
<tr>
<td>3. debugger</td>
<td>[free]</td>
</tr>
<tr>
<td>4. synthesizer</td>
<td>[free]</td>
</tr>
</tbody>
</table>
A tiny example SDSL: 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 = bxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bxor(r1, r5)
    return r6

> bvmax(-2, -1)

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

```scheme
(define bvmax
  `((2 bvge 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)))
```
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)
-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)))

(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)
```

(interpret prog inputs)

```
(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)))
```
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 (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 (interpret prog inputs) 0 -2
  `(2 bvge 0 1) 1 -1
    (3 bvneg 2) 2
    (4 bvxor 0 2) 3
    (5 bvand 3 4) 4
    (6 bvxor 1 5)) 5
  (load (last)) 6

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

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)))

(0 -2
 1 -1
 2 0
 3
 4
 5
 6)

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

```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 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))]
  )
)
```

```
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))]
    )
  )
)
```

```
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)
```
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)))

(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 = 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)))

(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)))
```

- pattern matching
- dynamic evaluation
- first-class & higher-order procedures
- side effects
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 number?)
(define inputs (list n0 n1))
(verify
  (assert (= (interpret bvmax inputs)
             (interpret max inputs))))
```
A tiny example SDSL: Rosette

```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)

Creates two fresh symbolic constants of type number and binds them to variables n0 and n1.

```
(define-symbolic n0 n1 number?)
(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

> verify(bvmax, max)
```

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

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

```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
```

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

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

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

A tiny example SDSL:
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)
(0, -2)

> bvmax(0, -2)
-1
```

```
(define-symbolic n0 n1 number?)
(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
```

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

query

```lisp
(define inputs (list 0 -2))
(debug [input-register?]
  (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
```

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

```plaintext
(define inputs (list 0 -2))
(debug [input-register?]
    (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(??, ??)
    r5 = bvand(r3, ??)
    r6 = bvxor(??, ??)
    return r6

> synthesize(bvmax, max)
```

(query)

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

```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)`

(define-symbolic n0 n1 number?)
(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

[Torlak & Bodik, Onward’13]

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

[Torlak & Bodik, Onward’13]

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

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

[Torlak & Bodik, Onward’13]

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

\[ \text{vs} \quad (3, 1, -2) \quad \text{reverse and filter, keeping only positive numbers} \quad \text{ps} \quad (1, 3) \]
Translation to constraints by example

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

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

Define a function `solve`:

```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)
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:

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

symbolic execution

```
<table>
<thead>
<tr>
<th>Condition</th>
<th>ps \mapsto (a, b)</th>
<th>ps \mapsto (a)</th>
<th>ps \mapsto (b)</th>
<th>ps \mapsto (b, a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a \leq 0</td>
<td>\top</td>
<td>\top</td>
<td>\top</td>
<td>\top</td>
</tr>
<tr>
<td>b \leq 0</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
</tr>
<tr>
<td>false</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
</tr>
<tr>
<td>a &gt; 0</td>
<td>\top</td>
<td>\top</td>
<td>\top</td>
<td>\top</td>
</tr>
<tr>
<td>b &gt; 0</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
</tr>
<tr>
<td>a \leq 0</td>
<td>\top</td>
<td>\top</td>
<td>\top</td>
<td>\top</td>
</tr>
<tr>
<td>b \leq 0</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
</tr>
<tr>
<td>false</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
</tr>
</tbody>
</table>
```

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)

symbolic execution

bounded model checking

ps0 = 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)
Design space of precise symbolic encodings

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

symbolic execution

<table>
<thead>
<tr>
<th>Condition</th>
<th>ps \mapsto ()</th>
<th>ps \mapsto (b)</th>
<th>ps \mapsto (a)</th>
<th>ps \mapsto (b, a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a \leq 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b \leq 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a \leq 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b &gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a &gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b \leq 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a &gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b &gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

bounded model checking

\[
\begin{align*}
&vs \mapsto (a, b) \\
&ps \mapsto ( ) \\
&a \leq 0 \\
&ps \mapsto ps_0 \\
&b \leq 0 \\
&ps \mapsto ps_1 \\
&b > 0 \\
&ps \mapsto ps_2 \\
&ps_0 = \text{ite}(a > 0, (a), ( )) \\
&ps_1 = \text{insert}(b, ps_0) \\
&ps_2 = \text{ite}(b > 0, ps_0, ps_1) \\
&\text{assert } \text{len}(ps_2) = 2
\end{align*}
\]
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)
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 > 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 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 = insert(v, ps)
    assert len(ps) == len(vs)

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

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

vs \mapsto (a, b)
ps \mapsto ()
\neg g_0 \mapsto ()
g_0 \mapsto (a)

ps \mapsto \{ g_0 \vdash (a),\neg g_0 \vdash () \}
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.

A new design: type-driven state merging

\[ g_0 = a > 0 \]
\[ g_1 = b > 0 \]
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)

g₀ = a > 0
ɡ₁ = b > 0
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)
```

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

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

```plaintext
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)
assert g2
```

polynomial encoding

concrete evaluation

symbolic virtual machine

```
vs ⊨ (a, b)
p0 ⊨ ()
¬g0 ⊨ ()

ps ⊨ ()
¬g1 ⊨ ()
g1 ⊨ ( )

ps ⊨ { g0 ⊨ (a),
        ¬g0 ⊨ ( ) }

ps ⊨ { g0 ⊨ (a),
        ¬g0 ⊨ ( ) }

ps ⊨ { g0 ⊨ (a),
        ¬g0 ⊨ ( ) }

ps ⊨ { g2 ⊨ (b, a),
        g3 ⊨ (c),
        g4 ⊨ ( ) }
```

assert g2
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

running time (sec)

SVM
Z3
advanced programming for everyone
Where next?

- secure stack machines
- web scraping scripts
- spatial programs
- data-parallel programs
- less code
- less time
- less effort
- solver-aided languages
Where next?

- harder programs
- new kinds of programs
- advanced solver-aided languages
- less time
- less code
- less effort
Keeping the programmer in the loop

- verify
- debug
- execute
- synth

SDSL program

SVM

SAT/SMT solver
Keeping the programmer in the loop

verify debug execute synth

SDSL program

SVM

SAT/SMT solver
Keeping the programmer in the loop

- verify
- debug
- execute
- synth

domain properties, invariants, insight

SDSL program

SVM

SAT/SMT solver
Keeping the programmer in the loop

verify
deploy
effect
execute
synth

domain properties, invariants, insight

SDSL program

SVM

SAT/SM solver
Keeping the programmer in the loop

verify
domain properties, invariants, insight
debug
execute
synth

SDSL program

symbolic profiling

SVM

SAT/SMT solver
Keeping the system in the loop

- verify
- debug
- execute
- synth

domain properties, invariants, insight

SDSL program

- dynamic programming fibonacci

SVM

recursive fibonacci

SAT/SMT solver
Keeping the system in the loop

symbolic design patterns

domain properties, invariants, insight

verify debug execute synth

SDSL program

SVM

SAT/SMT solver
Domain-specific solvers

Sometimes you need a special-purpose solver …
Domain-specific solvers for everyone

verify
dbg
eexecute

synth

synthesis of domain-specific solvers
So long, and thanks for all the fish!