Programming for everyone: from solvers to solver-aided languages and beyond

Emina Torlak
U.C. Berkeley
emina@eecs.berkeley.edu
http://people.csail.mit.edu/ema/
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

**Tools**
- 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

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**: 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**
2010 **solver-aided languages**

better programs
more easily
outline
solver-aided tools
solver-aided tools, languages
solver-aided tools, languages and demos
solver-aided tools
Programming ...

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

specification
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.

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

∃x. ¬ safe(P(x))

SAT/SMT solver

CBMC [CMU], Dafny [MSR], Miniatur [IBM], Klee [Stanford], etc.
Solver-aided tools: verification

Find an input on which the program fails.

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

∃ x . ¬ safe(P(x))

SAT/SMT solver

values

model

CBMC [CMU], Dafny [MSR], Miniatur [IBM], Klee [Stanford], etc.
Solver-aided tools: debugging

Localize bad parts of the program.

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

x = 42 ∧ safe(P(x))

SAT/SMT solver
Solver-aided tools: debugging

Localize bad parts of the program.

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

SAT/SMT solver

```
x = 42 ∧ safe(P(x))
```

expressions

min core
Solver-aided tools: angelic execution

Find values that repair the failing execution.

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

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

SAT/SMT solver
Solver-aided tools: angelic execution

Find values that repair the failing execution.

\[
P(x) \{
\begin{align*}
v &= \text{choose}() \\
&\quad \ldots \\
\end{align*}
\}
\text{assert safe}(P(x))
\]

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

values

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

model

SAT/SMT solver

Kaplan [EPFL], PBn [UCLA], Skalch [Berkeley], Squander [MIT], etc.
Solver-aided tools: synthesis

Synthesize code that repairs the program.

\[
\begin{align*}
P(x) \{ \\
v &= ?? \\
... \\
\} \\
\text{assert safe}(P(x))
\end{align*}
\]

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

SAT/SMT solver

Comfusy [EPFL], Sketch [Berkeley / MIT]
Synthesize code that repairs the program.

\[
P(x) \{
    \begin{align*}
    v &= x - 2 \\
    \ldots
    \end{align*}
    
    \text{assert safe}(P(x))
\]

Solver-aided tools: synthesis

Comfusy [EPFL], Sketch [Berkeley / MIT]
more solver-aided tools ...
more solver-aided tools ...
Building solver-aided tools: state-of-the-art
Building solver-aided tools: state-of-the-art

learn the problem domain

I need a tool to create models of biological cells ...

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

- learn the problem domain
- design a domain language
- Abstractions for cells, components, interactions, …

I need a tool to create models of biological cells …

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

verify execute debug synth

I need a tool to create models of biological cells …

A solver-aided tool for creating biological models

tools expert domain expert
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
Can we do better?

design a domain language

weeks

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

verify execute debug synth

a solver-aided domain-specific language (SDSL)

a solver-aided host language

domain expert
solver-aided languages
Layers of languages

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

- **library interpreter**: 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

Eigen / Matlab

C = A * B

[associativity]

C / Java

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

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

library interpreter

[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

- Chlorophyll (first-year grad)
- SynthCL (expert)
- WebSynth (undergrad)
- IFC (expert)
Spatial programming for a low-power chip, using synthesis to partition code and data across 144 tiny cores.

<table>
<thead>
<tr>
<th>SDSDLs developed with ROSETTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chlorophyll</td>
</tr>
<tr>
<td>(first-year grad)</td>
</tr>
<tr>
<td>SynthCL</td>
</tr>
<tr>
<td>(expert)</td>
</tr>
<tr>
<td>WebSynth</td>
</tr>
<tr>
<td>(undergrad)</td>
</tr>
<tr>
<td>IFC</td>
</tr>
<tr>
<td>(expert)</td>
</tr>
</tbody>
</table>

GreenArrays GA144
Spatial programming for a low-power chip, using synthesis to partition code and data across 144 tiny cores. Optimal partitioning synthesized in minutes, while manual partitioning takes days [Phothilimthana et al., PLDI'14].

SDSLs developed with Rosette

X + Z

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

GreenArrays GA144
SDSLs developed with ROSSETTE

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.

<table>
<thead>
<tr>
<th>SDSL</th>
<th>Development Time (weeks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chlorophyll (first-year grad)</td>
<td>16</td>
</tr>
<tr>
<td>SynthCL (expert)</td>
<td>12</td>
</tr>
<tr>
<td>WebSynth (undergrad)</td>
<td>4</td>
</tr>
<tr>
<td>IFC (expert)</td>
<td>0</td>
</tr>
</tbody>
</table>
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.

- Chlorophyll (first-year grad)
- SynthCL (expert)
- WebSynth (undergrad)
- IFC (expert)
SDSLs developed with ROSETTE

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

(define-symbolic id type)
(assert expr)
(verify expr)
(debug [expr] expr)
(solve expr)
(synthesize [expr] expr)
Anatomy of a solver-aided host language

symbolic constants
(define-symbolic id type)

assertions
(assert expr)
(verify expr)
(debug [expr] expr)
(solve expr)
(synthesize [expr] expr)
Anatomy of a solver-aided host language

symbolic constants
(define-symbolic id type)

assertions
(assert expr)

queries
(verify expr)
(debug [expr] expr)
(solve expr)
(synthesize [expr] expr)
Anatomy of a solver-aided host language

- **symbolic constants**
  - `(define-symbolic id type)`

- **assertions**
  - `(assert expr)`

- **queries**
  - `(verify expr)`
  - `(debug [expr] expr)`
  - `(solve expr)`
  - `(synthesize [expr] expr)`

\[ \exists v . \text{property}(P(v)) \]
Anatomy of a solver-aided host language

- Symbolic constants: `(define-symbolic v number?)`
- Assertions: `(assert expr)`
- Queries: `(verify expr)`
  *(debug [expr] expr) (solve expr) (synthesize [expr] expr)*

\[ \exists v . \text{property}(P(v)) \]
symbolic constants

(assert (define-symbolic v number?))

assertions

(assert (odd? (P v)))

queries

(verify expr)
(debug [expr] expr)
(solve expr)
(synthesize [expr] expr)

∃ v. property(P(v))

Anatomy of a solver-aided host language
Anatomy of a solver-aided host language

- **symbolic constants**: `(define-symbolic v number?)`
- **assertions**: `(assert (odd? (P v)))`
- **queries**: `(verify expr)`
  `(debug [expr] expr)`
  `(solve (check P))`
  `(synthesize [expr] expr)`

\[ \exists v . \text{property}(P(v)) \]
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.
A tiny example SDSL:  \texttt{ROSETTE}

def bvmax(r0, r1) :
    
    r2 = \texttt{bvge}(r0, r1)
    r3 = \texttt{bvneg}(r2)
    r4 = \texttt{bvxor}(r0, r2)
    r5 = \texttt{bvand}(r3, r4)
    r6 = \texttt{bvxor}(r1, r5)

    return  r6

> bvmax(\texttt{-2}, \texttt{-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)

```
(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)
```

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

(out opcode in ...)

parse
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
```

```racket```
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)))```

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

`(-2 -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)`

```
(define (interpret prog inputs)
  (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)

(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
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)))
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)))

interpret

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

INTERPRET

```scheme
(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
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)))

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)
```

Interpretation:

```
(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)
-1
```

Interpret:

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

```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
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
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))))

- 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)

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

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

> verify(bvmax, max)
```

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) (apply 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.

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

(RUSETTE)

(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

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

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

(R)OSSETTE

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

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

> **debug**(bvmax, max, (0, −2)) **query**
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))

(define inputs (list 0 -2))
(define [input-register?]
  (assert (= (interpret bvmax inputs)
             (apply max inputs))))
def bvmax(r0, r1):
    r2 = bvge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(??, ??)
    r5 = bvand(r3, ??)
    r6 = bvxor(??, ??)
    return r6

> \texttt{synthesize(bvmax, max)}
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)
```

```
(define-symbolic n0 n1 number?)
(define inputs (list n0 n1))
(synthesize [inputs]
  (assert (= (interpret bvmax inputs) (apply max inputs))))
```
symbolic virtual machine (SVM)
Anatomy of a symbolic virtual machine

SDSL + program → SVM

describe
solve
verify
synthesize

evaluate & compile to constraints

solver

[Torlak & Bodik, Onward’13]

[Torlak & Bodik, PLDI’14]
Anatomy of a symbolic virtual machine

- debug
- solve
- verify
- synthesize

lift solution to program level

SDSL + program

ROSETTE

evaluate & compile to constraints

solver

[Torlak & Bodik, Onward’13]

[RISSETTE SVM]

[Torlak & Bodik, PLDI’14]

Z3
Anatomy of a symbolic virtual machine

- Debug
- Solve
- Verify
- Synthesize
- Lift solution to program level
- Evaluate & compile to constraints

[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

\[ \text{solve:} \]
\[ \text{ps} = () \]
\[ \text{for } v \text{ in } \text{vs:} \]
\[ \quad \text{if } v > 0: \]
\[ \quad \quad \text{ps} = \text{insert}(v, \text{ps}) \]
\[ \quad \text{assert } \text{len(ps)} == \text{len(vs)} \]
Translation to constraints by example

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

constraints
a>0 ∧ b>0
Design space of precise symbolic encodings

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

ps₀ = ite(a > 0, (a), ( ))
ps₁ = insert(b, ps₀)

Design space of precise symbolic encodings

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

symbolic execution

bounded model checking

vs ⇔ (a, b)
ps ⇔ ()

a ≤ 0

ps ⇔ ()

{a ≤ 0} V {b ≤ 0}
false

{a > 0} V {b > 0}
true

ps0 = ite(a > 0, (a), ( ))
ps1 = insert(b, ps0)
ps2 = ite(b > 0, ps0, ps1)
assert len(ps2) = 2
A new design: type-driven state merging

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

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

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

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)
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

symbolic virtual machine
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 and exhaustive guards.

g_0 = a > 0

symbolic virtual machine

vs ↦ (a, b)
ps ↦ ()
¬g_0 ↦ ( )
g_0 ↦ ( )
ps ↦ ( )
ps ↦ (a)

ps ↦ { g_0 ⊨ (a), ¬g_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.

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

symbolic virtual machine

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

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

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

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

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

\[
\begin{align*}
  ps & \mapsto \{ g_0 \vdash (b, a), \neg g_0 \vdash (b) \}
\end{align*}
\]
solve:
ps = ()
for v in vs:
    if v > 0:
        ps = insert(v, ps)
assert len(ps) == len(vs)

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

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

Evaluate `len` concretely on all lists in the union; assertion true only on the list guarded by \( g_2 \).

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

solve:

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

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) \\
\text{assert } g_2
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

queries in SynthCL, WebSynth and IFC programs

running time (sec)
a little SDSL for finite state automata
thanks
thanks

emina@eecs.berkeley.edu