Solver-Aided Languages

Emina Torlak
emina@cs.washington.edu
Today

Last lecture
• Program synthesis

Today
• Solver-aided languages

Announcements
• Next Wednesday: guest lecture by James Bornholt
• Project presentations next Friday in class
  • 13 min per team: 10 min presentation + 3 min questions
• Project reports and prototypes due next Friday at 11:00pm
How to build your own solver-aided tool

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

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.

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

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

Find an input on which the program fails.

- verify
- debug
- solve
- synthesize

42

∃x. ¬safe(x, P(x))

solver-aided tool

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

The classic (hard) way to build a tool

verify
dbg
solve
synthesize

42

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

solver-aided tool

SMT solver

∃x \cdot \neg safe(x, P(x))

x = 42 \land safe(x, P(x))
The classic (hard) way to build a tool

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

```
verify
direct
solve
synthesize
```

```
P(x) {  
v = choice()  
...  
assert safe(x, P(x))
```

```
∃x . ¬safe(x, P(x))
x = 42 ∧ safe(x, P(x))
∃v . safe(42, P(v)(42))
```

solver-aided tool  SMT solver
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.

∃x . ¬safe(x, P(x))
x = 42 ∧ safe(x, P(x))
∃v . safe(42, P_v(42))
∃e. ∀x. safe(x, P_e(x))

The classic (hard) way to build a tool

verify
debug
solve
synthesize
The classic (hard) way to build a tool

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

verify debug solve synthesize

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

expertise in PL, FM, SE

generate
symbolic compiler

P(x)

SMT solver
Wanted: an easier way to build tools

```
verify
dbg
solve
synthesize
```

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

```
programming
an interpreter for the source language
```
Wanted: an easier way to build tools

verify
dbg
solve
synthesize

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

an interpreter for the source language
symbolic virtual machine

SMT solver
Wanted: an easier way to build tools

- verify
- debug
- solve
- synthesize

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

Technical challenge: how to efficiently translate a program and its interpreter?  
[Torlak & Bodik, PLDI’14]
How to build your own solver-aided tool

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!
Layer 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

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

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

```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 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.

Easier for tools to analyze.

```
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]
```

```
C = A * B
```

- `C / Java`
- `Eigen / Matlab`

[associativity]
Layers of solver-aided languages

solver-aided domain-specific language (SDSL)

library (shallow) embedding → interpreter (deep) embedding

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

- **Solver-aided domain-specific language (SDSL)**
  - Library (shallow) embedding
  - Interpreter (deep) embedding

- **Rosette**

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

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

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

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

- **Databases**
  - Cosette (CIDR’17)

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

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

- **Solver-aided domain-specific language (SDSL)**
- **Library (shallow) embedding**
- **Interpreter (deep) embedding**

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

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

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

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

- **Databases**
  - Cosette (CIDR’17)

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

... and more
The anatomy of a solver-aided host language

=  +

(symbolic values)
(assertions)
(queries)

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
  #:forall expr
  #:guarantee expr)
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
```

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

- interpreter [10 LOC]
- verifier [free]
- debugger [free]
- synthesizer [free]

We want to **test, verify, debug**, and **synthesize** programs in the BV SDSL.
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)
```

```scheme
(define bvmax
  `((2 bvsge 0 1)
     (3 bvneg 2)
     (4 bvxor 0 2)
     (5 bvand 3 4)
     (6 bvxor 1 5)))
```
**A tiny example SDSL**

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

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

```scheme
(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

```
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)))
`(-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)))
```
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)))
```

(de-definition-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))
```

```
```

interpret
A tiny example SDSL

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

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

```text
(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)))
```

```
(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
3
4
5
6
```
A tiny example SDSL

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

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

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

```python
(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 b vmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(r0, r2)
    r5 = bvand(r3, r4)
    r6 = bvxor(r1, r5)
    return r6

> b vmax(-2, -1)

(define b vmax
  `((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)
```

```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)))
```
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 `((2 bvsge 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)))))

(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

> 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)))

(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 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)
```

(query)

```
(define-symbolic* in (bitvector 32) [2])
(verify
 (assert (equal? (interpret bvmx in) (interpret max in))))
```
A tiny example SDSL

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

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

(query)

```lisp
(define-symbolic* in (bitvector 32) [2])
(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)(0, -2)

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

```scheme
(define-symbolic* in (bitvector 32) [2])
(verify
  (assert (equal? (interpret bvmax in) (interpret max in))))
```
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 (bv 0 32) (bv -2 32)))
(debug [integer?]
  (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

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

```Scheme
(define in (list (bv 0 32) (bv -2 32)))
(debug [integer?]
  (assert (equal? (interpret bvmax in) (interpret max in))))
```
def bvmax(r0, r1):
    r2 = bvsge(r0, r1)
    r3 = bvneg(r2)
    r4 = bvxor(??, ??)
    r5 = bvand(r3, ??)
    r6 = bvxor(??, ??)
    return r6

> synthetize(bvmax, max)

(define-symbolic* in (bitvector 32) [2])
(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)

(define-symbolic* in (bitvector 32) [2])
(synthesize
  #:forall in
  #:guarantee
  (assert (equal? (interpret bvmax in) (interpret max in))))
How to build your own solver-aided tool

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

query

program

SDSL

ROSETTE

Symbolic Virtual Machine

SMT solver Z3
How it all works: a big picture view

ROSETTE

query

result

program

SDSL

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

query -> result

program

SDSL

ROSETTE

Symbolic Virtual Machine

SMT solver Z3
Translation to constraints by example

\[
\begin{align*}
\text{solve:} & \\
\text{ps} = () & \\
\text{for} & \text{v in vs:} \\
\text{if} & \text{v > 0:} \\
\text{ps} = & \text{insert(v, ps)} \\
\text{assert} & \text{len(ps) == len(vs)} \\
& \text{reverse and filter, keeping only positive numbers} \\
\text{vs} & \text{(3, 1, -2)} \\
\text{ps} & \text{(1, 3)}
\end{align*}
\]
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

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

```
vs = (a, b)

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

symbolic execution

- \(a \leq 0\) → \(\text{ps} \mapsto ()\)
- \(a > 0\) → \(\text{ps} \mapsto (a)\)
- \(b \leq 0\) → \(\text{ps} \mapsto ()\)
- \(b > 0\) → \(\text{ps} \mapsto (b)\)
- \(a \leq 0\) → \(\text{ps} \mapsto (a)\)
- \(b \leq 0\) → \(\text{ps} \mapsto (b)\)
- \(a > 0\) → \(\text{ps} \mapsto (b, a)\)
- \(b > 0\) → \(\text{ps} \mapsto (b)\)

\[
\{a \leq 0\} \lor \{b \leq 0\} \\
\{a > 0\} \lor \{b > 0\} \\
\{a > 0\} \lor \{b > 0\} \\
\{a > 0\} \lor \{b > 0\} \\
\{a > 0\} \lor \{b > 0\}
\]
Design space of precise symbolic encodings

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

bounded model checking

a ≤ 0
ps ↦ ()

a > 0
ps ↦ (a)

b ≤ 0
ps ↦ (b)

b > 0
ps ↦ (a, b)

{a ≤ 0} ∨ {b > 0} \text{false}
{a ≤ 0} \text{false}
{a > 0} \text{false}
{a > 0} \text{true}

ps0 = ite(a > 0, (a), ( ))
Design space of precise symbolic encodings

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

**symbolic execution**

**bounded model checking**

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

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

```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, b), (c, d), (e, f)\}
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)
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.

g0 = a > 0

symbolic virtual machine

vs ⟷ (a, b)
ps ⟷ ()
¬g0 ⟷ ()
g0 ⟷ (a)

ps ⟷ { g0 ⊨ (a), ¬g0 ⊨ () }
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

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

\[
\begin{align*}
g_0 &= a > 0 \\
g_1 &= b > 0
\end{align*}
\]
solve:
    ps = ()
    for v in vs:
        if v > 0:
            ps = insert(v, ps)
    assert len(ps) == len(vs)

\[ g_0 = a > 0 \]
\[ g_1 = b > 0 \]
A new design: type-driven state merging

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

Evaluate len concretely on all lists in the union; assertion true only on the list guarded by $g_2$. 

**symbolic virtual machine**

<table>
<thead>
<tr>
<th>$vs$</th>
<th>$(a, b)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ps$</td>
<td>$(\ )$</td>
</tr>
<tr>
<td>$g_0$</td>
<td>$\neg$</td>
</tr>
<tr>
<td>$g_1$</td>
<td>$g_0$</td>
</tr>
<tr>
<td>$g_2$</td>
<td>$g_0 \land g_1$</td>
</tr>
<tr>
<td>$g_3$</td>
<td>$\neg(g_0 \leftrightarrow g_1)$</td>
</tr>
<tr>
<td>$g_4$</td>
<td>$\neg g_0 \land \neg g_1$</td>
</tr>
<tr>
<td>$c$</td>
<td>$\text{ite}(g_1, b, a)$</td>
</tr>
</tbody>
</table>

assert $g_2$
A new design: type-driven state merging

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

symbolic virtual machine

vs ↦ (a, b)
solve:
  vs = (a, b)
solve:
  vs = (a, b)
solve:
  vs = (a, b)
solve:
  vs = (a, b)

ps ↦ ( )
solve:
  ps = ()
solve:
  ps = ()
solve:
  ps = ()
solve:
  ps = ()

ps ↦ ( )
solve:
  ps = ()
solve:
  ps = ()
solve:
  ps = ()
solve:
  ps = ()

polynomial encoding
concrete evaluation

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

assert \( g_2 \)
How to build your own solver-aided tool

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!
Chlorophyll: ultra low-power computing

Instructions/Second vs Power

GreenArrays GA144 Processor

Figure by Per Ljung
Chlorophyll: ultra low-power computing

2. Basic Architecture

The purpose of this board is to facilitate evaluation and application prototyping using GreenArrays chips. Because no single I/O complement would be suitable for all likely uses, this board has two GA144 chips: One (called “Host”) configured with sufficient I/O for intensive software development, and the other (called “Target”) with as little I/O committed as possible so that pure, dedicated applications may be prototyped.

2.1 Highlights

- Three FTDI USB to serial chips provide high speed (960 kBaud) communications for interactive software development and general-purpose host communications.
- An onboard switching regulator takes power from the USB connectors and/or a conventional “wall wart” power supply. Whichever of these is offering the highest voltage is used by the regulator.
- A barrier strip provides for connection of bench power supplies.
- Each of the power buses of the two GA144 chips may selectively be run from external power in lieu of the onboard regulator, allowing you to run either chip from any desired $V_{DD}$ voltage and also facilitating current measurements.
- The Host chip is supplied with an SPI boot flash holding 1 MByte of nonvolatile data, an external SRAM with 1 MWord (2 MBytes) of memory; and may optionally use a dual voltage MMC card such as the 2 Gigabyte unit we have selected for in-house use. These memory resources may be used in conjunction with Virtual Machines such as eForth and polyFORTH, or for direct use by your own F18 code.
- The Target chip is committed to as few I/O connections as possible. The sources for its reset signal are fully configurable, and with the exception of a SERDES line connecting it with the Host chip, all other communications (two 2-wire serial interfaces) may be disconnected so that the chip is fully isolated and thus all practical I/O is available for any desired use.

Roughly half the board is prototyping area, mainly populated with a grid of plated through holes on 0.1 inch centers. By soldering suitable headers to this grid, you can provide for expansion using various prototyping fixtures such as those made by SchmartBoard. The grid is intentionally large enough to support an 8- or 16-bit PC-104 socket.

The periphery of the prototyping area is provided with hole patterns for many popular connectors, and there are six 8-bit bidirectional level shifters for interfacing with external circuits that may not run on 1.8v. In addition, one 1.8v 2-input OR and three NANDs are available for use in external circuitry.

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

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

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

With Chlorophyll, it took one afternoon to build a set of apps that took 3 months to build manually.

[Phothilimthana et al., PLDI’14]
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.
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

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 depended on this bug for correct operation.

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

[Fernstein et al., CAV'16]
MemSynth: synthesizing memory models

Memory consistency models define memory reordering behaviors on multiprocessors.

\[
\begin{align*}
x &= y &= 0 \\
a &= x & b &= y \\
y &= 1 & x &= 1 \\
a \equiv b \equiv 1
\end{align*}
\]
MemSynth: synthesizing memory models

Memory consistency models define memory reordering behaviors on multiprocessors.

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

Forbidden by sequential consistency.

Allowed by x86 and other hardware memory models.
MemSynth: synthesizing memory models

Memory consistency models define memory reordering behaviors on multiprocessors.

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

Forbidden by sequential consistency.

Allowed by x86 and other hardware memory models.

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

Memory consistency models define memory reordering behaviors on multiprocessors.

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

Forbidden by sequential consistency.

Allowed by x86 and other hardware memory models.
MemSynth: synthesizing memory models

A framework sketch → A set of litmus tests → Relational logic → Memory model specification

Built by a 2nd year grad in a few weeks

James Bornholt
MemSynth: synthesizing memory models

A framework sketch

A set of litmus tests

Relational logic

Memory model specification

Synthesized PowerPC in 12 seconds from 768 previously published tests. Synthesized x86 in 2 seconds from Intel's litmus tests. Discovered 4 tests are missing from the Intel manual.

[Birtholt and Torlak, PLDI’17]
Thanks for a great quarter!