Computer-Aided Reasoning for Software

Solver-Aided Languages

Emina Torlak
emina@cs.washington.edu
Today

Last lecture
  • Angelic execution

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

Reminders
  • No lecture next Wednesday, HW4 due at 11:00pm
  • Project presentations (8 min per team) next Friday in class
  • Project reports and prototypes due next Friday at 11:00pm
a little programming for everyone
A little programming for everyone

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

Every knowledge worker wants to program …

› spreadsheet data manipulation
A little programming for everyone

Every knowledge worker wants to program …

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

Every knowledge worker wants to program …

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

Every knowledge worker wants to program …

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

Every knowledge worker wants to program …

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

We all want to build programs …

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

solver-aided languages

less time

less expertise

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

```
P(x) {
  ...
  ...
}
```
Programming ...

test case

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

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

Find an input on which the program fails.

∃ \( x \) . ¬ safe(\( P(x) \))

Programming with a solver-aided tool

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

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

x = 42 ∧ safe(P(x))

SAT/SMT solver

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

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

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

∃v . safe(P(42, v))

SAT/SMT solver

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

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.

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

∃e . ∀x . safe(P_e(x))

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

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

expertise in PL, SE, FM

verify debug solve synth
A new, easy way to build tools

verify debug solve synth

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

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

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

an interpreter or a library

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

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

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

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

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

[Rosette, symbolic virtual machine]

[torlak & bodik, PLDI’14, Onward’13]
design

solver-aided languages
Layers of languages

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

- **library**

- **interpreter**

- **host language**
  - 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

**Expression Equations**

- **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)

library

interpreter

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
- intelligent tutoring
  - RuleSynth
- memory models
  - MemSynth
- optimal synthesis
  - Synapse
- radiotherapy controllers
  - Neutrons
- BGP router configurations
  - BagPipe

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

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

Library interpreter

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

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

Modern descendent of Scheme with macro-based metaprogramming.

Racket
Anatomy of a solver-aided host language

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

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

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

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

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

1. interpreter [10 LOC]
2. verifier [free]
3. debugger [free]
4. synthesizer [free]
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)

(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)
```

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

(out opcode in ...)
```
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)
    (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 bvge 0 1)
   (3 bvneg 2)
   (4 bvxor 0 2)
   (5 bvand 3 4)
   (6 bvxor 1 5)))

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

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

```
(bdefine 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)))
```
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:

$$\text{interpret}$$

(define bvmax (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)
```

### Interpretation

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

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

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

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

interpret

(definition (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
```

ROSETTE

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

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

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

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

> verify(bvmax, max)
```

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

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

> verify(bvmax, max)
```

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

ROSETTE

Creates two fresh symbolic constants of type number and binds them to variables n0 and n1.
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 integer?)
(define inputs (list n0 n1))
(verify
 (assert (= (interpret bvmax inputs) (interpret max inputs))))
```
A tiny example SDSL:

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

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

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

(query)

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

> `bvmax(0, -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

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

(query)

```schema
(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
```

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

(query)

```prolog
(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)`

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

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

> `synthesize(bvmax, max)`

query

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

[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

Theories of bitvectors, integers, reals, and uninterpreted functions

[Rosette, Onward’13]

[Symbolic virtual machine]

[Solver Z3]

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

\[
\text{solve:}\quad ps = ()
\]

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

\[
\text{if } v > 0:\quad ps = \text{insert}(v, ps)
\]

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

reverse and filter, keeping only positive numbers

\[
vs = (3, 1, -2)
\]

\[
ps = (1, 3)
\]
Translation to constraints by example

\[
\begin{align*}
\text{vs} & = (3, 1, -2) \\
\text{ps} & = () \\
\text{for} \ v \ \text{in} \ \text{vs}: \\
\quad & \text{if} \ v > 0: \\
\quad & \quad \text{ps} = \text{insert}(v, \text{ps})
\end{align*}
\]
Translation to constraints by example

vs

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

constraints
Translation to constraints by example

\[ (a, b) \]

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

\(a > 0 \wedge b > 0\)

\((a, b)\)

solve:

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

constraints

\(a > 0 \wedge b > 0\)
Design space of precise symbolic encodings

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

symbolic execution

bounded model checking
Design space of precise symbolic encodings

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

bounded model checking
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)

symbolic execution

bounded model checking

```python
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(\( vs \))
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:
```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

{$\{a > 0, b > 0, \text{true}\}$}
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)
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
A new design: type-driven state merging

solve:

\[
\text{ps} = ()
\]

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

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

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

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

Execute \text{insert} concretely on all lists in the union.

\[
g_0 = a > 0
\]

\[
g_1 = b > 0
\]

\[
\text{symbolic virtual machine}
\]

\[
\text{vs} \mapsto (a, b)
\]

\[
\text{ps} \mapsto ()
\]

\[
\neg g_0 \mapsto ()
\]

\[
g_0 \mapsto (a)
\]

\[
\text{ps} \mapsto ()
\]

\[
\text{ps} \mapsto (a)
\]

\[
\neg g_0 \mapsto (b)
\]

\[
g_1 \mapsto ()
\]

\[
\text{ps} \mapsto (a)
\]

\[
\neg g_0 \mapsto (b)
\]

\[
\text{ps} \mapsto (a, b)
\]
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
A new design: type-driven state merging

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

\[
\begin{align*}
g_0 & = a > 0 \\
g_1 & = b > 0 \\
g_2 & = g_0 \land g_1 \\
g_3 & = \neg (g_0 \leftrightarrow g_1) \\
g_4 & = \neg g_0 \land \neg g_1 \\
c & = \text{ite}(g_1, b, a)
\end{align*}
\]

\[
\text{symbolic virtual machine}
\]

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

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

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

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

\[
\begin{align*}
g_2 & \quad \text{ps} \mapsto (b, a) \\
g_3 & \quad \text{ps} \mapsto (c) \\
g_4 & \quad \text{ps} \mapsto ()
\end{align*}
\]
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)

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

g_0 = a > 0
\(\neg g_0\) \(\vdash (\ )\)

\(g_1 = b > 0\)
\(\neg g_0 \land g_1\)
\(\neg g_0 \vdash (\ )\)

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

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

g₀ = a > 0
g₁ = b > 0
g₂ = g₀ ∧ g₁
g₃ = ¬(g₀ ⇔ g₁)
g₄ = ¬g₀ ∧ ¬g₁
c = ite(g₁, b, a)
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
solver-aided programming for everyone
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.

![Instructions/Second vs Power](image1.png)

~100x

![GreenArrays GA144 Processor](image2.png)

Figure by Per Ljung
Chlorophyll: ultra low-power computing

GreenArrays GA144 Processor

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

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

Drawing by Mangpo Phothilimthana
Chlorophyll: ultra low-power computing

GreenArrays GA144 Processor

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

\[ c = a \times b \]

Drawing by Mangpo Phothilimthana
Chlorophyll: ultra low-power computing

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

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

```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

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

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

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

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

Configuring BGP is tricky

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

BGP configuration property

Bagpipe

policy violation

A BGP interpreter implemented in Rosette.
Bagpipe: verifying BGP router configurations

BGP configuration → property

Bagpipe

→

policy violation

Built by two grads in a few weeks

Konstantin Weitz and Doug Woos
Bagpipe: verifying BGP router configurations

Internet2 announcements are not leaked

Bagpipe

route leaks!

[under submission]
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

Therapy Control Software

Prescription

Sensors

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

**EPICS documentation / semantics**

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

- EPICS program
- safety property
- EPICS Verifier
- bug report

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

Calvin Loncaric
Neutrons: verifying a radiotherapy system

Therapy Control Software

EPICS Verifier

safety property

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

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