Synthesis and Verification for Everyone

Emina Torlak
University of Washington
emina@cs.washington.edu
homes.cs.washington.edu/~emina/
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

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

P(x) {

 specification

 ... ...

}
Programming ...

test case

```java
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))

42

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

verify

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.

\begin{align*}
P(x) \{ & \quad \text{v} = x + 2 \\
& \quad \text{...} \\
& \quad \text{assert safe(P(x))} \\
\end{align*}

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

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

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

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

∃v . safe(P(42, v))

Kaplan [Koksal et al, POPL'12]
PBN [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.

Sketch [Solar-Lezama et al., ASPLOS'06]
Comfusy [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

translator

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

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

verify
debg
solve
synth

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

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

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

verify debug solve synth

an interpreter or a library

ROSSETTÉ
A new, easy way to build tools

verify debug solve synth

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

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

an interpreter or a library

Rosette symbolic virtual machine
A new, easy way to build tools

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

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

symbolic virtual machine

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

[Torlak & Bodik, PLDI’14, Onward’13]
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**

```
C = A * B
```

**Associativity**

```
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    for (k = 0; k < p; k++)
      C[i][k] += A[i][j] * B[j][k]
```

**Library**

- Eigen / Matlab
- C / Java
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

interceptor

solver-aided host language

symbolic virtual machine

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

solver-aided domain-specific language (SDSL)

library interpreter

solver-aided host language

symbolic virtual machine

spatial programming
   Chlorophyll

data-parallel programming
   SynthCL

web scraping
   WebSynth

secure stack machines
   IFC

radiotherapy controllers
   Neutrons

BGP router configurations
   BagPipe

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

solver-aided domain-specific language (SDSL)

library

interpreter

solver-aided host language

symbolic virtual machine

spatial programming
  Chlorophyll

data-parallel programming
  SynthCL

web scraping
  WebSynth

secure stack machines
  IFC

radiotherapy controllers
  Neutrons

BGP router configurations
  BagPipe

[Rorlak & 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)

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
```

**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.

<table>
<thead>
<tr>
<th>Test</th>
<th>Usage</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>interpreter</td>
<td>[10 LOC]</td>
</tr>
<tr>
<td>2.</td>
<td>verifier</td>
<td>[free]</td>
</tr>
<tr>
<td>3.</td>
<td>debugger</td>
<td>[free]</td>
</tr>
<tr>
<td>4.</td>
<td>synthesizer</td>
<td>[free]</td>
</tr>
</tbody>
</table>
A tiny example SDSL: ROSETTA

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

```rust
(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)
```

```scheme
(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)

A tiny example SDSL:

```scheme
(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
```

```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)))
```

> bvmax(-2, -1)

{-1}

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

(intepret 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 (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)))
```

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

A tiny example SDSL:
A tiny example SDSL:

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

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

```
A tiny example SDSL:

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

```
A tiny example SDSL:

(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:

(define bvmax (0 -2)
  1 -1
  2
  3
  4
  5
  6
```

```
A tiny example SDSL:

(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:

(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)))

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

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

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

```
(define (interpret prog inputs) 6 -1
 (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)
-1
```

ROSETTE

```
(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)
  (define (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:

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

> \text{verify}(\text{bvmax}, \text{max})

\[
(\text{define-symbolic } n0 \ n1 \ \text{number}?)
(\text{define inputs (list } n0 \ n1))
(\text{verify}
  (\text{assert } (= \text{(interpret } \text{bvmax inputs})
  (\text{interpret } \text{max inputs})))
\]
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)`

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

**(define inputs (list n0 n1))**

**(verify**

    (assert (= (interpret bvmax inputs)
                (interpret max inputs))))

 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)`

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

Symbolic values can be used just like concrete values of the same type.
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)`
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
> bvmax(0, -2)
-1
```

(query)

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

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

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

(query)

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

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

(query)

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

```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 number?)
(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)
```

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

Torlak & Bodik, Onward’13
[Torlak & Bodik, PLDI’14]
How it all works: a big picture view

[Torlak & Bodik, Onward’13]

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

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

[Rosette, SDSL]

- symbolic
- virtual machine

[Theory of bitvectors]

[Results]

[Refs: Torlak & Bodik, Onward'13; Torlak & Bodik, PLDI'14]
Translation to constraints by example

vs
(3, 1, -2)

reverse and filter, keeping only positive numbers

ps
(1, 3)
Translation to constraints by example

vs
(3, 1, -2)

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

ps
(1, 3)
Translation to constraints by example

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

\[(a, b)\]

**solve:**

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

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)
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), ( ))
Design space of precise symbolic encodings

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

```
solve:
s ps = ()

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

assert len(ps) == len(vs)
```

symbolic execution

\[
\{ a \leq 0 \} \land \{ b \leq 0 \} \lor \{ a > 0 \} \land \{ b > 0 \} \land \{ \text{false} \}
\]

bounded model checking

\[
v s \mapsto (a, b)
\]
\[
ps \mapsto ()
\]
\[
a \leq 0
\]
\[
ps \mapsto ()
\]
\[
b > 0
\]
\[
ps \mapsto (a)
\]
\[
a > 0
\]
\[
ps \mapsto (b, a)
\]
\[
a > 0
\]
\[
ps = \text{ite}(a > 0, (a), ( ))
\]
\[
ps_0 \mapsto ps_0
\]
\[
ps_1 = \text{insert}(b, ps_0)
\]
Design space of precise symbolic encodings

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

symbolic execution

\[ \begin{align*}
& a \leq 0 \\
& b \leq 0 \\
& \text{false}
\end{align*} \quad \begin{align*}
& a \leq 0 \\
& b > 0 \\
& \text{false}
\end{align*} \quad \begin{align*}
& a > 0 \\
& b \leq 0 \\
& \text{false}
\end{align*} \quad \begin{align*}
& a > 0 \\
& b > 0 \\
& \text{true}
\end{align*} \]

bounded model checking

\[ \begin{align*}
& vs \mapsto (a, b) \\
& ps \mapsto () \\
& a \leq 0 \\
& a > 0 \\
& ps \mapsto () \\
& ps \mapsto (a) \\
& b \leq 0 \\
& b > 0 \\
& ps \mapsto (b) \\
& ps \mapsto (b) \\
& ps \mapsto (a) \\
& ps \mapsto (b, a) \\
& ps_0 = \text{ite}(a > 0, (a), ( )) \\
& ps_1 = \text{insert}(b, ps_0) \\
& ps_2 = \text{ite}(b > 0, ps_0, ps_1) \\
& \text{assert } \text{len}(ps_2) = 2
\end{align*} \]
A new design: type-driven state merging

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

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

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

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

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

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

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

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

\{ a > 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)
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.
A new design: type-driven state merging

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

Execute insert concretely on all lists in the union.

\[
\begin{align*}
g_0 &= a > 0 \\
g_1 &= b > 0
\end{align*}
\]
A new design: type-driven state merging

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

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

g₀ = a > 0
г₁ = b > 0
g₂ = g₀ ∧ g₁
г₃ = ¬(g₀ ↔ g₁)
g₄ = ¬g₀ ∧ ¬g₁
c = ite(g₁, b, a)

symbolic virtual machine

vs ⇔ (a, b)
ps ⇔ ()

¬ g₀

¬ g₁

ps ⇔ ()

ps ⇔ (a)

ps ⇔ { g₀ ⊨ (a),
        ¬g₀ ⊨ () }

ps ⇔ { g₀ ⊨ (b, a),
        ¬g₀ ⊨ (b) }

ps ⇔ { g₂ ⊨ (b, a),
        g₃ ⊨ (c),
        g₄ ⊨ () }
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 $g_2$.

$g_0 = a > 0$
$g_1 = b > 0$
$g_2 = g_0 \land g_1$
$g_3 = \neg(g_0 \leftrightarrow g_1)$
$g_4 = \neg g_0 \land \neg g_1$
$c = \text{ite}(g_1, b, a)$
assert $g_2$

symbolic virtual machine

$vs \mapsto (a, b)$
$ps \mapsto ()$
$\neg g_0 \mapsto ( )$
$g_0 \mapsto ( )$
$ps \mapsto ( )$
$ps \mapsto (a)$
$\neg g_0 \mapsto ( )$
$g_0 \mapsto ( )$
$ps \mapsto \{ g_0 \vdash (a), \neg g_0 \vdash ( ) \}$
$\neg g_1 \mapsto ( )$
$g_1 \mapsto ( )$
$ps \mapsto \{ g_0 \vdash (a), \neg g_0 \vdash ( ) \}$
$ps \mapsto \{ g_1 \vdash (a), \neg g_1 \vdash ( ) \}$
$ps \mapsto \{ g_2 \vdash (b, a), g_3 \vdash (c), g_4 \vdash ( ) \}$
A new design: type-driven state merging

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

\[ g_0 = a > 0 \]
\[ g_1 = b > 0 \]
\[ g_2 = g_0 \land g_1 \]
\[ g_3 = \neg (g_0 \leftrightarrow g_1) \]
\[ g_4 = \neg g_0 \land \neg g_1 \]
c = ite(g_1, b, a)
assert g_2
Effectiveness of type-driven state merging

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

- Number of unions
- Size of all unions

\[ R^2 = 0.9884 \]

\[ R^2 = 0.95 \]
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
Chlorophyll: ultra low-power computing

Instructions/Second vs Power

GreenArrays GA144 Processor

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

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

\[ c = a \times b \]

drawing by Mangpo Phothilimthana
Chlorophyll: ultra low-power computing

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

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

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

Synthesizes placement of code and data onto cores, by type-checking a program sketch.
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.
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

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

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

A BGP interpreter implemented in Rosette.

policy violation
Bagpipe: verifying BGP router configurations

BGP configuration property

Bagpipe

policy violation

Built by two grads in a few weeks

Konstantin Weitz and Doug Woos
Bagpipe: verifying BGP router configurations

Internet2

private 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 now being built.
Neutrons: verifying a radiotherapy system

Clinical Neutron Therapy System (CNTS) at UW

Prescription

Sensors

Therapy Control Software

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

Experimental Physics and Industrial Control System (EPICS) Dataflow Language

Prescription

Sensors

Therapy Control Software

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

**EPICS documentation / semantics**

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

- Prescription
- Sensors
- Therapy Control Software
- Beam, motors, etc.
Neutrons: verifying a radiotherapy system

An *end-to-end property* that spans the entire system, not just software.

**CNTS Couch Safety Property:**

The beam will turn off if the couch rotation angle moves out of tolerances during treatment and the operator has not issued the manual override command.
Neutrons: verifying a radiotherapy system

EPICS program

safety property

EPICS Verifier

bug report

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

Calvin Loncaric
Neutrons: verifying a radiotherapy system

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

[under submission]
thanks
ROSETTE
symbolic virtual machine

your SDSL

verify
dedigit
solve
synthesize