Program Synthesis for Forth
Forth Day 2012

Ras Bodik
Mangpo Phitchaya Phothilimthana
Tikhon Jelvis
Rohin Shah
Synthesis with “sketches”

Extend your language with two constructs

**spec:**
```c
int foo (int x) {
    return x + x;
}
```

**sketch:**
```c
int bar (int x) implements foo {
    return x << ??;
}
```

**result:**
```c
int bar (int x) implements foo {
    return x << 1;
}
```

Instead of **implements**, assertions over safety properties can be used.
Example: Parallel Matrix Transpose
Example: 4x4-matrix transpose with SIMD

a functional (executable) specification:

```c
int[16] transpose(int[16] M) {
    int[16] T = 0;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
            T[4 * i + j] = M[4 * j + i];
    return T;
}
```

This example comes from a Sketch grad-student contest
Implementation idea: parallelize with SIMD

Intel SHUFP (shuffle parallel scalars) SIMD instruction:

\[
return = \text{shufps}(x1, x2, \text{imm8} :: \text{bitvector8})
\]
High-level insight of the algorithm designer

Matrix $M$ transposed in two shuffle phases

**Phase 1:** shuffle $M$ into an intermediate matrix $S$ with some number of shufps instructions

**Phase 2:** shuffle $S$ into an result matrix $T$ with some number of shufps instructions

Synthesis with partial programs helps one to complete their insight. Or prove it wrong.
The SIMD matrix transpose, sketched

```c
int[16] trans_sse(int[16] M) implements trans {
    int[16] S = 0, T = 0;

    S[??::4] = shufps(M[??::4], M[??::4], ??);
    S[??::4] = shufps(M[??::4], M[??::4], ??);
    ...
    S[??::4] = shufps(M[??::4], M[??::4], ??);

    T[??::4] = shufps(S[??::4], S[??::4], ??);
    T[??::4] = shufps(S[??::4], S[??::4], ??);
    ...
    T[??::4] = shufps(S[??::4], S[??::4], ??);

    return T;
}
```
The SIMD matrix transpose, sketched

```c
int[16] trans_sse(int[16] M) implements trans {
    int[16] S = 0, T = 0;
    repeat (??) S[??::4] = shufps(M[??::4], M[??::4], ??);
    repeat (??) T[??::4] = shufps(S[??::4], S[??::4], ??);
    return T;
}
```

```c
int[16] trans_sse(int[16] M) implements trans { // synthesized code
    S[4::4]   = shufps(M[6::4],   M[2::4],  11001000b);
    S[0::4]   = shufps(M[11::4],  M[6::4],  10010110b);
    S[12::4]  = shufps(M[0::4],   M[2::4],  10001101b);
    S[8::4]   = shufps(M[8::4],   M[12::4], 11010111b);
    T[4::4]   = shufps(S[11::4],  S[1::4],  11010000b);
    T[12::4]  = shufps(S[3::4],   S[8::4],  11000011b);
    T[8::4]   = shufps(S[4::4],   S[9::4],  11100010b);
    T[0::4]   = shufps(S[12::4],  S[0::4],  10110100b);
}
```

From the contestant email:
Over the summer, I spent about 1/2 a day manually figuring it out.

Synthesis time: <5 minutes.
Demo: transpose on Sketch

Try Sketch online at http://bit.ly/sketch-language
Inductive Synthesis, Phrased as Constraint Solving
What to do with a program as a formula?

Assume a formula $S_P(x,y)$ which holds iff program $P(x)$ outputs value $y$

**program:** $f(x) \{ \text{ return } x + x \}$

**formula:** $S_f(x,y): y = x + x$

This formula is created as in program verification with concrete semantics [CMBC, Java Pathfinder, ... ]
With program as a formula, solver is versatile

Solver as an **interpreter**: given $x$, evaluate $f(x)$

$$S(x, y) \land x = 3 \quad \text{solve for } y \quad y \mapsto 6$$

Solver as a program **inverter**: given $f(x)$, find $x$

$$S(x, y) \land y = 6 \quad \text{solve for } x \quad x \mapsto 3$$

This solver “bidirectionality” enables synthesis
Search of candidates as constraint solving

\[ S_p(x, h, y) \text{ holds iff sketch } P[h](x) \text{ outputs } y. \]

- `spec(x) { return x + x }`
- `sketch(x) { return x << ?? }`  \( S_{sketch}(x, y, h): y = x \times 2^h \)

The solver computes \( h \), thus synthesizing a program correct for the given \( x \) (here, \( x=2 \))

\[ S_{sketch}(x, y, h) \land x = 2 \land y = 4 \quad \text{solve for } h \quad h \mapsto 1 \]

Sometimes \( h \) must be constrained on several inputs

\[
S(x_1, y_1, h) \land x_1 = 0 \land y_1 = 0 \land \\
S(x_2, y_2, h) \land x_2 = 3 \land y_2 = 6 \quad \text{solve for } h \quad h \mapsto 1
\]
Inductive synthesis

Our constraints encode **inductive synthesis:**

- We ask for a program $P$ correct on a few inputs.
- We hope (or test, verify) that $P$ is correct on rest of inputs.
Synthesis for Forth and ArrayForth
Applications of synthesis for ArrayForth

Synthesizing optimal code

**Input:** unoptimized code (the spec)
Search space of all programs

Synthesizing optimal library code

**Input:** sketch + spec
Search completions of the sketch

Synthesizing communication code for GreenArray

**Input:** program with virtual channels
Compile using synthesis
1) Synthesizing optimal code

unoptimized code (spec)

optimal code

slower

faster
Our Experiment

Register-based processor

naive

hand

spec

optimized

most optimal

Stack-based processor

slower

faster

bit trick

synthesizer
Our Experiment

Register-based processor

Stack-based processor

slower

faster

naive

hand

spec

bit trick

optimized

most optimal

synthesizer

most optimal
Comparison

Register-based processor

- naive
- optimized

Stack-based processor

- hand
- spec
- bit trick
- most optimal
- translation
- synthesizer

slower

faster
Preliminary Synthesis Times

Synthesizing a program with
  8 unknown instructions
takes 5 second to 5 minutes

Synthesizing a program up to
  ~25 unknown instructions
within 50 minutes
## Preliminary Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Approx. Speedup</th>
<th>Code length reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>x – (x &amp; y)</td>
<td>Exclude common bits</td>
<td>5.2x</td>
<td>4x</td>
</tr>
<tr>
<td>~ (x – y)</td>
<td>Negate difference</td>
<td>2.3x</td>
<td>2x</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>Inclusive or</td>
<td>1.8x</td>
</tr>
<tr>
<td>(x + 7) &amp; -8</td>
<td>Round up to multiple of 8</td>
<td>1.7x</td>
<td>1.8x</td>
</tr>
<tr>
<td>(x &amp; m)</td>
<td>(y &amp; ~m)</td>
<td>Replace x with y where bits of m are 1’s</td>
<td>2x</td>
</tr>
<tr>
<td>(y &amp; m)</td>
<td>(x &amp; ~m)</td>
<td>Replace y with x where bits of m are 1’s</td>
<td>2.6x</td>
</tr>
<tr>
<td>x’ = (x &amp; m)</td>
<td>(y &amp; ~m)</td>
<td>Swap x and y where bits of m are 1’s</td>
<td>2x</td>
</tr>
<tr>
<td>y’ = (y &amp; m)</td>
<td>(x &amp; ~m)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Code Length

<table>
<thead>
<tr>
<th>Program</th>
<th>Original Length</th>
<th>Output Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x - (x &amp; y))</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>(\neg(x - y))</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>(x \mid y)</td>
<td>27</td>
<td>15</td>
</tr>
<tr>
<td>((x + 7) &amp; -8)</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>((x &amp; m) \mid (y &amp; \neg m))</td>
<td>22</td>
<td>11</td>
</tr>
<tr>
<td>((y &amp; m) \mid (x &amp; \neg m))</td>
<td>21</td>
<td>8</td>
</tr>
<tr>
<td>(x' = (x &amp; m) \mid (y &amp; \neg m))</td>
<td>43</td>
<td>21</td>
</tr>
<tr>
<td>(y' = (y &amp; m) \mid (x &amp; \neg m))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2) Synthesizing optimal library code

Input:

Sketch: program with holes to be filled
Spec: program in any programing language

Output:

Complete program with filled holes
Example: Integer Division by Constant

Naïve Implementation:

- Subtract divisor until reminder < divisor.
- # of iterations = output value  **Inefficient!**

Better Implementation:

\[
\text{quotient} = (M \times n) \gg s
\]

- n       - input
- M       - “magic” number
- s       - shifting value

\(M\) and \(s\) depend on the number of bits and constant divisor.
Example: Integer Division by 3

Sketch in ArrayForth:

```
: div3 ?? a! 0 17 for +* unext
  push dup or pop
?? for +* unext a ;
```

Spec in C:

```c
int div3(int n) {
    return n/3;
}
```
## Preliminary Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Solution</th>
<th>Synthesis Time (s)</th>
<th>Verification Time (s)</th>
<th># of Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x/3$</td>
<td>$(43691 \times x) \gg 17$</td>
<td>2.3</td>
<td>7.6</td>
<td>4</td>
</tr>
<tr>
<td>$x/5$</td>
<td>$(209716 \times x) \gg 20$</td>
<td>3</td>
<td>8.6</td>
<td>6</td>
</tr>
<tr>
<td>$x/6$</td>
<td>$(43691 \times x) \gg 18$</td>
<td>3.3</td>
<td>6.6</td>
<td>6</td>
</tr>
<tr>
<td>$x/7$</td>
<td>$(149797 \times x) \gg 20$</td>
<td>2</td>
<td>5.5</td>
<td>3</td>
</tr>
<tr>
<td>deBruijn: $\log_2 x$ ($x$ is power of 2)</td>
<td>deBruijn = 46, Table = {7, 0, 1, 3, 6, 2, 5, 4}</td>
<td>3.8</td>
<td>N/A</td>
<td>8</td>
</tr>
</tbody>
</table>

Note: these programs work for 18-bit number except $\log_2 x$ is for 8-bit number.
3) Communication Code for GreenArray

Synthesize communication code between nodes

Interleave communication code with computational code such that

There is no deadlock.
The runtime of the synthesized program is minimized.
Future Roadmap

Language Design
- Good for partitioning
- Easy to compile to arrayForth

Partitioning
- Minimize number of communication
- Each block fits in each node

Placement & Communication
- Minimize communication cost
- Reason about I/O pins

Scheduling & Optimization
- Order that does not break dependency
- No Deadlock
- Find the fastest schedule

Language?

Comp1
Comp2
Comp3
Send X
Recv Y
Comp4
Comp5
Project Pipeline

1. L0: high-level language
   - Partitioning
2. L1: L0 + virtual communication channels
   - Placement & Routing
3. L2: L1 + layout + routing
   - Inserting communication
4. L3: L2 + real communication code
   - Code generation
5. L4: machine instructions
## Preliminary Results #1 (backup)

<table>
<thead>
<tr>
<th>Program</th>
<th>Approx Runtime (ns)</th>
<th>Program Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Optimized</td>
</tr>
<tr>
<td>$x - (x &amp; y)$</td>
<td>15.5</td>
<td>3</td>
</tr>
<tr>
<td>$\sim(x - y)$</td>
<td>14</td>
<td>6</td>
</tr>
<tr>
<td>$x \mid y$</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>$(x + 7) &amp; -8$</td>
<td>24</td>
<td>14</td>
</tr>
<tr>
<td>$(x &amp; m) \mid (y &amp; \sim m)$</td>
<td>33</td>
<td>16.5</td>
</tr>
<tr>
<td>$(y &amp; m) \mid (x &amp; \sim m)$</td>
<td>31.5</td>
<td>12</td>
</tr>
<tr>
<td>$x' = (x &amp; m) \mid (y &amp; \sim m)$</td>
<td>64.5</td>
<td>31.5</td>
</tr>
<tr>
<td>$y' = (y &amp; m) \mid (x &amp; \sim m)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Preliminary Results #1 (backup)

<table>
<thead>
<tr>
<th>Program</th>
<th>Original Program</th>
<th>Synthesized Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x - (x &amp; y) )</td>
<td>over and -1.+.+</td>
<td>- and</td>
</tr>
<tr>
<td>( \neg(x - y) )</td>
<td>-1.+.+</td>
<td>over-.+</td>
</tr>
<tr>
<td>( x</td>
<td>y )</td>
<td>over over or a! and a or</td>
</tr>
<tr>
<td>((x + 7) &amp; -8 )</td>
<td>7.+ 8 -1.+ and</td>
<td>7.+ 262136 and</td>
</tr>
<tr>
<td>((y &amp; m)</td>
<td>(x &amp; \neg m) )</td>
<td>a! over over a - and push a and pop over over or push and pop or push</td>
</tr>
<tr>
<td>((x &amp; m)</td>
<td>(y &amp; \neg m) )</td>
<td>a and push a - and pop over over or push and pop or pop</td>
</tr>
<tr>
<td>( x' = (x &amp; m)</td>
<td>(y &amp; \neg m) )</td>
<td>a! over over a - and push a and pop over over or push and pop or pop or pop</td>
</tr>
<tr>
<td>( y' = (y &amp; m)</td>
<td>(x &amp; \neg m) )</td>
<td>a! over over a - and push a and pop over over or push and pop or push a and push a - and pop over over or push and pop or pop</td>
</tr>
</tbody>
</table>
Log Base 2 of Power of 2 (backup)

Compute \( \lg x \), where \( x \) is a power of 2.

```
const uint64_t deBruijn = 0x022fdd63cc95386d;
const unsigned int convert[64] = {
    0, 1, 2, 53, 3, 7, 54, 27,
    4, 38, 41, 8, 34, 55, 48, 28,
    62, 5, 39, 46, 44, 42, 22, 9,
    24, 35, 59, 56, 49, 18, 29, 11,
    63, 52, 6, 26, 37, 40, 33, 47,
    61, 45, 43, 21, 23, 58, 17, 10,
    51, 25, 36, 32, 60, 20, 57, 16,
    50, 31, 19, 15, 30, 14, 13, 12};

r = convert[(x*deBruijn) >> 58];
```

Sketch:

dup dup or a!
?? !+ ?? !+ ?? !+ ?? !+ ?? !+ ?? !+ ?? !+ ?? !+
?? a! 0 17 for +* unext
a 2/ 2/ 2/ 2/ 2/ 7 and a! @