Sketching

CS294 Spring 2006

Work by: Armando Solar-Lezama, Liviu Tancau, David Turner, Rastislav Bodik, Sanjit Seshia UC Berkeley, Vijay Saraswat IBM

Administrativa

- Thanks for all the pointers to papers
  - will be added to the master list tonight
- Tue homework:
  - select a paper you’d like to lead a discussion on
  - add comments such as “should be preceded by paper X”
  - email me summary for the Prospector paper
  - suggested format to be posted
  - if all goes well, “coffee service” will start
  - 10 cappuccinos at 9:30
- Thu: 10-minute presentations on challenge problems
  - sign up by Mon; auditors can present

The sketching experience

spec + sketch ⇒ implementation (completed sketch)

specification sketch implementation

Programming with StreamBit

- Specification
  - executable: easy to debug, serves as a prototype
  - a reference implementation: simple and sequential
  - written by domain experts: crypto, bio, MPEG committee
- Sketched implementation
  - program with holes: filled in by synthesizer
  - programmer sketches strategy; machine provides details
  - written by performance experts: vector wizard; SSE guru

Example: divide and conquer parallelization

- Parallel algorithm:
  - Data rearrangement + parallel computation
- spec:
  - sequential version of the program
- sketch:
  - parallel computation
- automatically synthesized:
  - Rearranging the data (dividing the data structure)

Benefits of sketching

- productivity
  - many tedious details synthesized automatically
  - focus on creative process
- separation of roles:
  - domain expert, performance expert collaborate
- separation of aspects: correctness vs. performance
  - rapidly develop high-quality implementations
  - without fear of introducing bugs
Verification, synthesis, sketching

- **Verification**: does your program implement the spec?
  - user responsible for low-level implementation details
  - redundancy: implementation restates aspects of spec

- **Synthesis**: produce a program that implements the spec
  - say what not how; say it only once
  - hard to synthesize a good implementation

- **Sketching**: synthesis + partially described implementation

Sketching: uses and expectations

- **What it can do**:
  - synthesize hard-to-get-right expressions (masks, indices)
  - synthesize algebraic tricks when merging parallel results
  - synthesize data structures (re)layout for parallelization
  - and other parallelization machinery

- **What it is not designed to do**:
  - invent a good algorithm automatically (search space too large)
  - provide clever algorithm ideas (but helps in exploring them)
  - remove fun from programming (focus on clever ideas)

Sketching in StreamBit [PLDI’05]: Best Results

- Implementing a mini cipher, sketching vs. C:

**Example 1**

- bitvector parallelism
  - exploited thanks to some algebra
- sketches help reinvent the trick
- sketches are reusable

Sketching in StreamBit [PLDI’05]: Worst Result

- sketching easy to explain but mastering took a while
  - sketches not really programs, but meta-level rewrite rules
  - baseline compiler, its rules overridden by rewrite rules
  - implementation broken into multiple, hierarchical sketches
  - dataflow programming model useful but unfamiliar

- sketching limited in expressibility
  - implementations had to use instructions that were semi-permutations (bit shift ok, addition no)

SKETCH

- A language that addresses all the limitations
  - like C without pointers
  - sketching support: two simple constructs

- restricted to finite programs:
  - input size known at compile time, terminates on all inputs
- most high-performance kernels are finite:
  - matrix multiply: yes
  - binary search tree: no
Ex1: Isolate rightmost 0-bit. 1010 0111 → 0000 1000

```java
bit[W] isol0 (bit[W] x) {  // W: word size
  bit[W] ret = 0;
  for (int i = 0; i < W; i++)
    if (x[i] == 0) ret[i] = 1; break;
  return ret;
}

bit[W] isol0Fast (bit[W] x) implements isol0 {
  return ~x & (x+1);
}

bit[W] isol0Sketched (bit[W] x) implements isol0 {
  return ~(x + ??) & (x + ??);
}
```

Sketches are reusable

```java
bit[W] expression (bit[W] x) {
  return ~(x + ??) & (x + ??);
}

bit[W] isol0Sketched (bit[W] x) implements isol0 {
  return expression(x);
}

bit[W] isol1Sketched (bit[W] x) implements isol1 {
  return expression(x);
}
```

Programmer’s view of sketches

- the ?? operator replaced with a suitable chunk of bits
- as directed by the implements clause.

- the ?? operator introduces non-determinism
- the implements clause constrains it.

Meaning of sketches

- programs with ?? have many meanings
  - ~(x + ??) & (x + ??);
    means:
    - ~(x + 0) & (x + 1);
    - ~(x - 1) & (x + 0);
    ...
- loops are unrolled:
  - x = ??; loop (x) (y = y + ??; )
    means:
    - x = 2; y = y + 4; y = y + 0;
    - x = 3; y = y + 2; y = y + 4; y = y + 17;
    ...
- f implements g:
  - synthesizer “selects” the meaning of f that is functionally equivalent to g

Example 2

- divide and conquer parallelism
- SIMD parallelism, and how to emulate SIMD semantics
- sketching table-based implementations
- more on reusability
- rapidly prototyping multiple implementations

Ex 2: Population count. 0010 0110 → 3

```java
int pop (bit[W] x) {
  int count = 0;
  for (int i = 0; i < W; i++) {
    if (x[i]) count++;
  }
  return count;
}
```
Parallel pop, divide-and-conquer

- `pop(word x) = pop(1st half of x) + pop(2nd half of x)`
  - recurse until argument is single bit
- idea: execute simultaneously all operations of same size
  - store sums in the word itself, SIMD style
  - O(log W) steps
- tricky implementation
  - SIMD subword size different at each step
  - on non-SIMD, must “emulate” SIMD semantics with bitmasks
  - example, base case:
    \[ x = (x \& 0x5555) + ((x >> 1) \& 0x5555) \]

Sketch of the parallel pop

```c
int popSketched (bit[W] x) implements pop {
  loop (??) {
    x = (x & 0x5555) + ((x >> ??) & 0x5555);
  }
  return x;
}
```

Table-based implementation

```c
int popTable (bit[8] in) implements pop {
  int[256] table = ??;
  return table[in];
}
```

Table-based implementation

```c
int popTable (bit[W] in) implements pop {
  int[256] table = ??;
  int ret = 0;
  loop (??) { ret += table[in>>?? & ??]; }
  return ret;
}
```

Implementation for sparse populations

```c
int popSparseSketched (bit[W] in) implements pop {
  int ret;
  for (ret = 0; in; ret++) {
    in &= ~expression(in); // ~(x + ??) & (x + ??);
  }
  return ret;
}
```

Beyond synthesis of literals

- Synthesizing values of `??` already very useful
  - tricky expressions
  - parallelization machinery
- We can synthesize more than values
  - semi-permutations: functions that select and shuffle bits
  - polynomials: over one or more variables
Example 3: IP from DES.

```c
uint32_t IPsketched (uint64_t x) {
    uint32_t result;
    uint32_t table[8][16];
    x = (x >> 32) | (x << 32);
    for (int i = 0; i < 8; ++i) {
        result[0:31] |= table[i][x[i*4::4]];
        result[32:63] |= table[i][x[32+i*4::4]];
    }
    return result;
}
```

Template for an arbitrary permutation

```c
uint64_t permutation<int N>(uint64_t x) {
    uint64_t result;
    int i = 0;
    while (true) {
        result ^= x >> i & 1;
        result ^= x << i & 1;
        ++i;
    }
    return result;
}
```

More higher-level synthesis

- Synthesizing polynomials

```c
int spec (int x) {
    return 2*x*x*x*x + 3*x*x*x + 7*x*x + 10;
}
int p (int x) implements spec {
    return (x+1)*(x+2)*poly(3,x);
}
int poly(int n, int x) {
    if (n == 0) return 1;
    else return x * poly(n-1, x) + poly(1, x);
}
```

Sketch of Karatsuba

```c
void k <int N> (uint64_t x, uint64_t y) implements mult {
    if (N == 1) return x*y;
    if (N == 2) x1 = x[0:N-2-1]; y1 = y[0:N-2-1];
    if (N == 2) x2 = x[N/2:N-1]; y2 = y[N/2:N-1];
    if (N == 4) return multPolySparse<4,N/2>(x1, y1, x2, y2);
    if (N == 8) return multPolySparse<8,N/2>(x1, y1, x2, y2);
    return multPolySparse<2*N>(N/2, 11, 11, 11, 11);
    return multPolySparse<2*N>(N/2, 11, 11, 11, 11);
}
```

Karatsuba's multiplication

```
x = x1*b + x0  
y = y1*b + y0

x*y = b^3*x1*y1 + b*(x1*y0 + x0*y1) + x0*y0

O(N^1.5) vs. O(N^2)
```
Semantic view of sketches

- the ?? operator modeled as reading from an oracle

```c
int f (int y) {
    int f (int y, bit[K] oracle) {
        x = ??;
        return y;
    }
}
```

- synthesizer finds oracle satisfying \( f \) implements \( g \)

Synthesis as generalized SAT

- The sketch synthesis problem is an instance of 2QBF:
  \[
  \exists o \in \{0,1\}^k . \forall x \in \{0,1\}^m . \ P(x) = S(x,o)
  \]

- Counter-example driven solver:
  ```c
  I = {}
x = random()
do while x 
  c = synthesizeForSomeInputs(I)
  if c = nil then exit("buggy sketch")
x = verifyForAllInputs(c)  // x: counter-example
  return c
  ```

Scalability of the synthesizer

<table>
<thead>
<tr>
<th>Program</th>
<th>Input size, W</th>
<th>Synthesis time (s)</th>
<th>SAT unknowns</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES MixCol</td>
<td>32</td>
<td>443</td>
<td>2602</td>
</tr>
<tr>
<td>DES.IP</td>
<td>64</td>
<td>693</td>
<td>178</td>
</tr>
<tr>
<td>Tblcrc</td>
<td>24 (48)</td>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>Tblcrc2</td>
<td>8</td>
<td>245</td>
<td>2048</td>
</tr>
<tr>
<td>Reverse</td>
<td>64</td>
<td>193</td>
<td>522</td>
</tr>
<tr>
<td>Parity</td>
<td>24 (48)</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>Log2</td>
<td>24 (28)</td>
<td>1268</td>
<td>409</td>
</tr>
<tr>
<td>Pop</td>
<td>8 (16)</td>
<td>4</td>
<td>109</td>
</tr>
<tr>
<td>Polynomial</td>
<td>16</td>
<td>617</td>
<td>96</td>
</tr>
<tr>
<td>Karatsuba</td>
<td>6 (8)</td>
<td>11</td>
<td>63</td>
</tr>
</tbody>
</table>

Finite programs

- finite programs:
  - input size known at compile time, terminates on all inputs
  - matrix multiply: yes
  - binary search tree: no

- complete:
  - specification can specify any finite program
  - sketch can describe any implementation over given instructions
  - synthesizer can resolve any sketch in theory; in practice, scales to real-world problems

Beyond small finite programs

- Some finite programs are too large
  - Ex: big-integer multiplication
  - here, our synthesis works only for word size of \( W = 6 \)
  - but synthesized result is same for all \( W \)
  - with this knowledge, programmer can already use our system
  - hope to develop static analysis proving synthesis independent of \( W \)

- Some programs are not finite
  - streaming computation
  - but the kernel applied on the stream is typically finite

Example 5: DCT

```c
float[N] DCT<N>(float[N] x) implements DCTSpec<N>{
    float[N] t;
    loop(log2(N)) {
        loop(N) {
            t[??] = ?? * x[??] + ?? * x[??];
        }
        x=t;
    }
    return x;
}
```
Example 6: Data Rearrangement

**Problem:** vectorizing big integer addition

```c
bit[32*N]
bigAdd (bit[32*N] a1, bit[32*N] a2) {
    return a1 + a2;
}
```

```c
bit[32][N][4]
bigAdd4 (bit[32][N][4] a1, bit[32][N][4] a2) {
    bit[32][N][4] result;
    result[0] = bigAdd(a1[0], a2[0]);
    result[1] = bigAdd(a1[1], a2[1]);
    result[2] = bigAdd(a1[2], a2[2]);
    result[3] = bigAdd(a1[3], a2[3]);
    return result;
}
```

BigInt Addition: Sketch of Vectorized Code

```c
{T}[N] bigAddMMX({T}[N] a1, {T}[N] a2) {
    bit[32][4][N] result;
    {T} carry = 0, {T} tmp = 0;
    for (int i=0; i<N; ++i) {
        bit[32][4] tmp = a1[i] + a2[i];
        result[i] = tmp +/- carry;
        carry = (tmp < a1[i]);
    }
    return result;
}
```

```c
    {T} result;
    bit[32][4] a1t = permutation<32*4*N>(a1);
    bit[32][4] a2t = permutation<32*4*N>(a2);
    result = bigAddMMX(a1t, a2t);
    return permutation<32*4*N>(result);
}
```

Conclusion

- Sketching is more general than it appears
  - where do specifications come from?
    - most problems have a simple reference implementation
    - finite programs too restrictive?
      - many implementations have finite kernels

- Scalability of synthesis (key future work)
  - show independence of synthesis from input size
  - learn synthesized patterns and give hints to synthesizer
  - better solver (beyond bit-blasting)