Lecture 19: Program Synthesis, Part 1

17-355/17-655/17-819: Program Analysis

Rohan Padhye and Jonathan Aldrich

April 8, 2021

* Course materials developed with Claire Le Goues
With slide inspiration gratitude to Emina Torlak and Ras Bodik

Warm-up exercise: specify findMax(list)

- Specify a program that finds the maximum number in a list. How many different ways can you do it?

Program Synthesis Overview

- A mathematical characterization of program synthesis: prove that

$$\exists P \;.\; \forall x \;.\; \varphi(x, P(x))$$

- In constructive logic, the witness to the proof of this statement is a program P that satisfies property φ for all input values x

Program Synthesis Overview

- A mathematical characterization of program synthesis: prove that

$$\exists P \,.\, \forall x \,.\, \varphi(x, P(x))$$

- In constructive logic, the witness to the proof of this statement is a program P that satisfies property φ for all input values x
- What could the inferred program P be?
  - Historically, a protocol, interpreter, classifier, compression algorithm, scheduling policy, cache coherence policy, …
- How is property φ expressed?
  - Historically, as a formula, a reference implementation, input/output pairs, traces, demonstrations, a sketch, …

Expressing User Intent

- How do we constrain the program to be synthesized?
  - Express what we know about the problem and/or solution
  - Usually incomplete
- Two forms of specification can constrain synthesis
  - Observable behavior: input/output relations, executable specification, safety property
  - Structural properties: constraints on internal computation, such as a sketch, template, assertions about structure (e.g. number of iterations)

The Search Space of Programs

- Constraining the search space can help make synthesis feasible
  - Subset of a real programming language?
  - Grammar for combining fixed set of operators and control structures?
  - DSL?
  - Logic?

Two approaches to searching for programs

- Deductive synthesis
  - Maps a high-level specification to an implementation, using a theorem prover
  - Efficient, provably correct
  - Require complete specifications, sufficient axiomatization of the domain
    - Can be as complicated as writing the program itself!
  - Used for e.g. controllers
  - A lot like compilation!
- Inductive synthesis
  - Takes a partial, perhaps multi-modal specification and constructs a program that satisfies it
  - Flexible in specification requirements, require no axioms
  - May be less efficient, weaker guarantees on correctness/optimality
  - Search techniques: brute-force, probabilistic, genetic programming, logical reasoning
  - Major current focus of research

Deductive Synthesis Thought Exercise

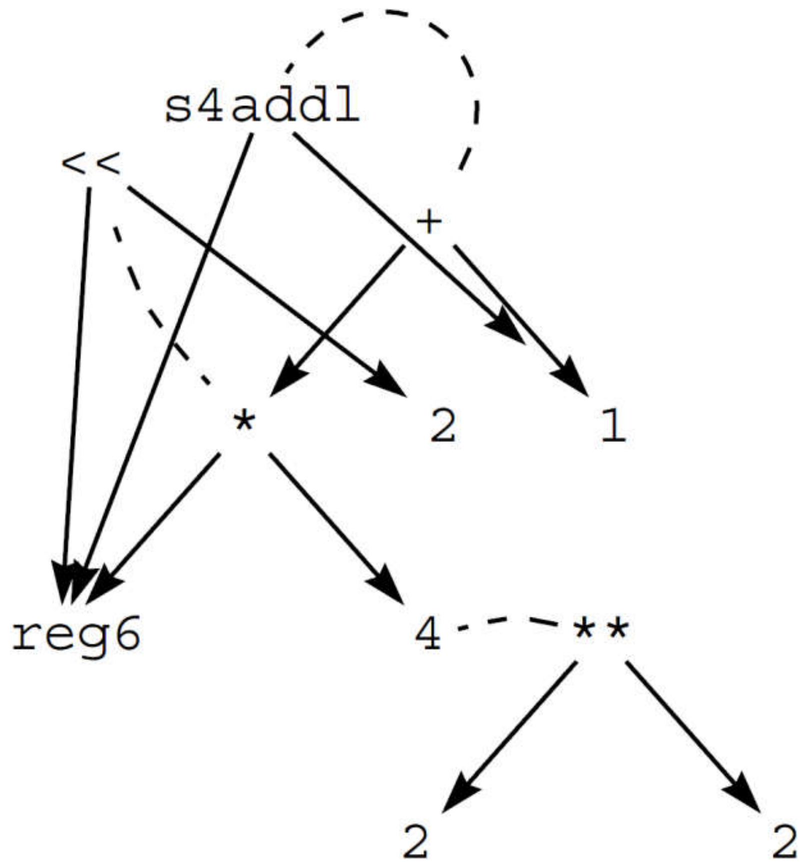
- Write a formula that can check whether a candidate program P correctly solves the task of finding the maximum number in a list

Denali – Deductive Synthesis for Superoptimization

- Goal: optimized compilation
  - generate short sequences of provably optimal loop-free machine instructions

Denali: synthesis with axioms and E-graphs

[Joshi, Nelson, Randall PLDI'02]



$$\forall\, n\,.\; 2^n = 2{*}{*}n$$

$$\forall\, k, n\,.\; k * 2^n = k{<}{<}n$$

$$\forall k, n{:}{:}\; k * 4 + n \;= \mathtt{s4addl}(k, n)$$

$\mathbf{reg6} * 4 + 1 \longrightarrow \mathtt{s4addl}(\mathbf{reg6}, 1)$

specification → synthesized program

Two kinds of axioms

Instruction semantics: defines (an interpreter for) the language

$$\forall\, k, n\,.\; k * 2^n = k\texttt{<<}n$$

$$\forall k, n{::}\; k * 4 + n \;= \texttt{s4addl}(k, n)$$

Algebraic properties: associativity of add64, memory modeling, math, …

$$\forall\, n\,.\; 2^n = 2{*}{*}n$$

$$(\forall\, x, y :: \texttt{add64}(x, y) = \texttt{add64}(y, x))$$

$$(\forall\, x, y, z :: \texttt{add64}(x, \texttt{add64}(y, z)) = \texttt{add64}(\texttt{add64}(x, y), z))$$

$$(\forall\, x :: \texttt{add64}(x, 0) = x)$$

$$\begin{aligned}&(\forall\, a, i, j, x \;:: i = j\\ &\qquad\qquad \vee\; \texttt{select}(\texttt{store}(a, i, x), j) = \texttt{select}(a, j))\end{aligned}$$

Compilation vs. synthesis

So where's the line between compilation & synthesis?

Compilation:

1) represent source program as abstract syntax tree (AST)

    (i) parsing, (ii) name analysis, (iii) type checking

2) lower the AST from source to target language

    eg, assign machine registers to variables, select instructions, …

Lowering performed with <u>tree rewrite rules,</u> sometimes based on <u>analysis of the program</u>

eg, a variable cannot be in a register if its address is in another variable

Properties of deductive synthesizers

Efficient and provably correct

- thanks to semantics-preserving rules
- only correct programs are explored
- Denali is scalable: prior super-optimizers gen-and-test

Successfully built for axiomatizable domains

- expression equivalence (Denali)
- linear filters (FFTW, Spiral)
- linear algebra (FLAME)
- statistical calculations (AutoBayes)
- data structures as relational DBs (P2; Hawkins et al.)

Inductive synthesis

Find a program correct on a set of inputs and hope (or verify) that it's correct on other inputs.

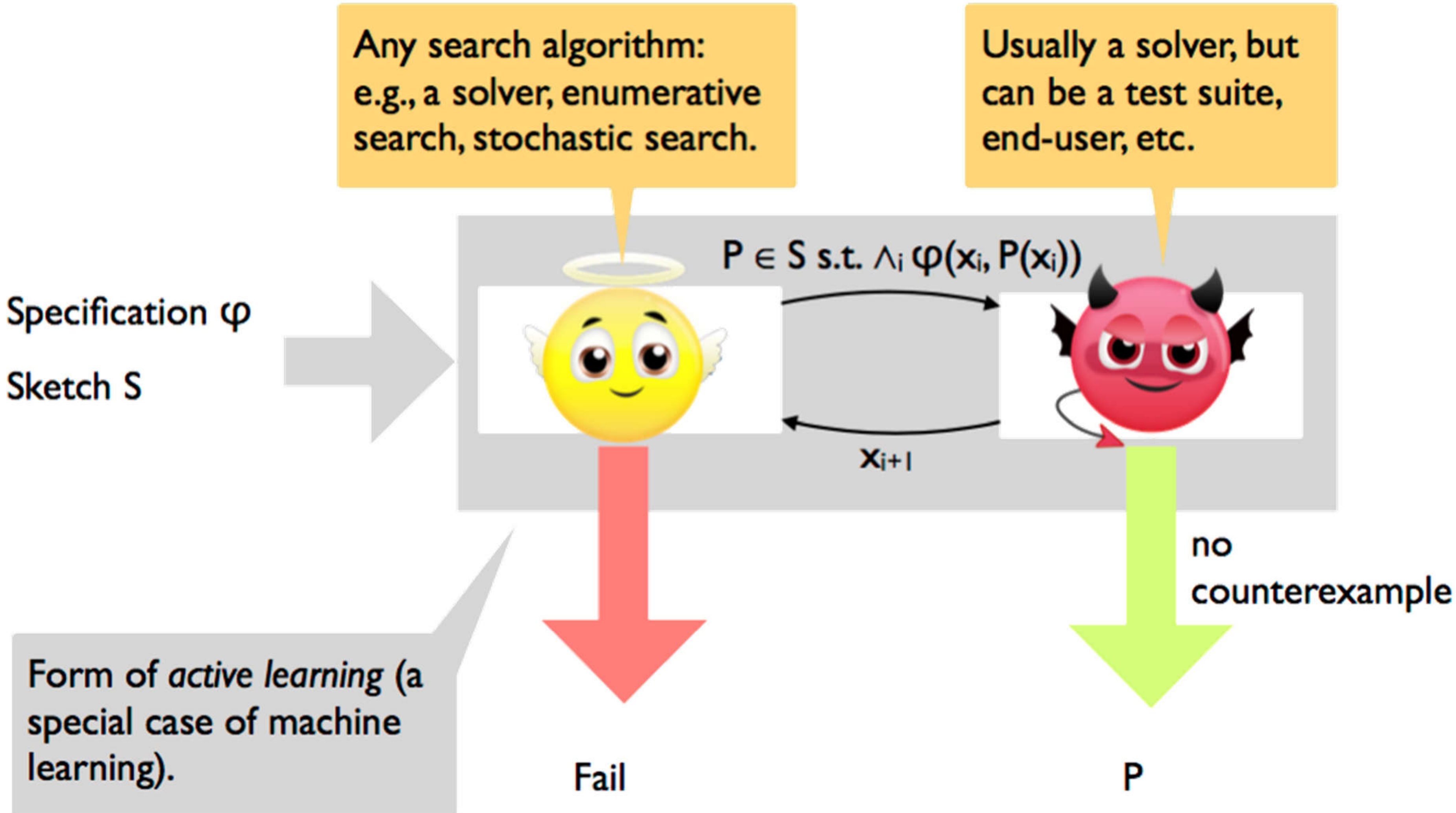
A **partial program** syntactically defines the candidate space.

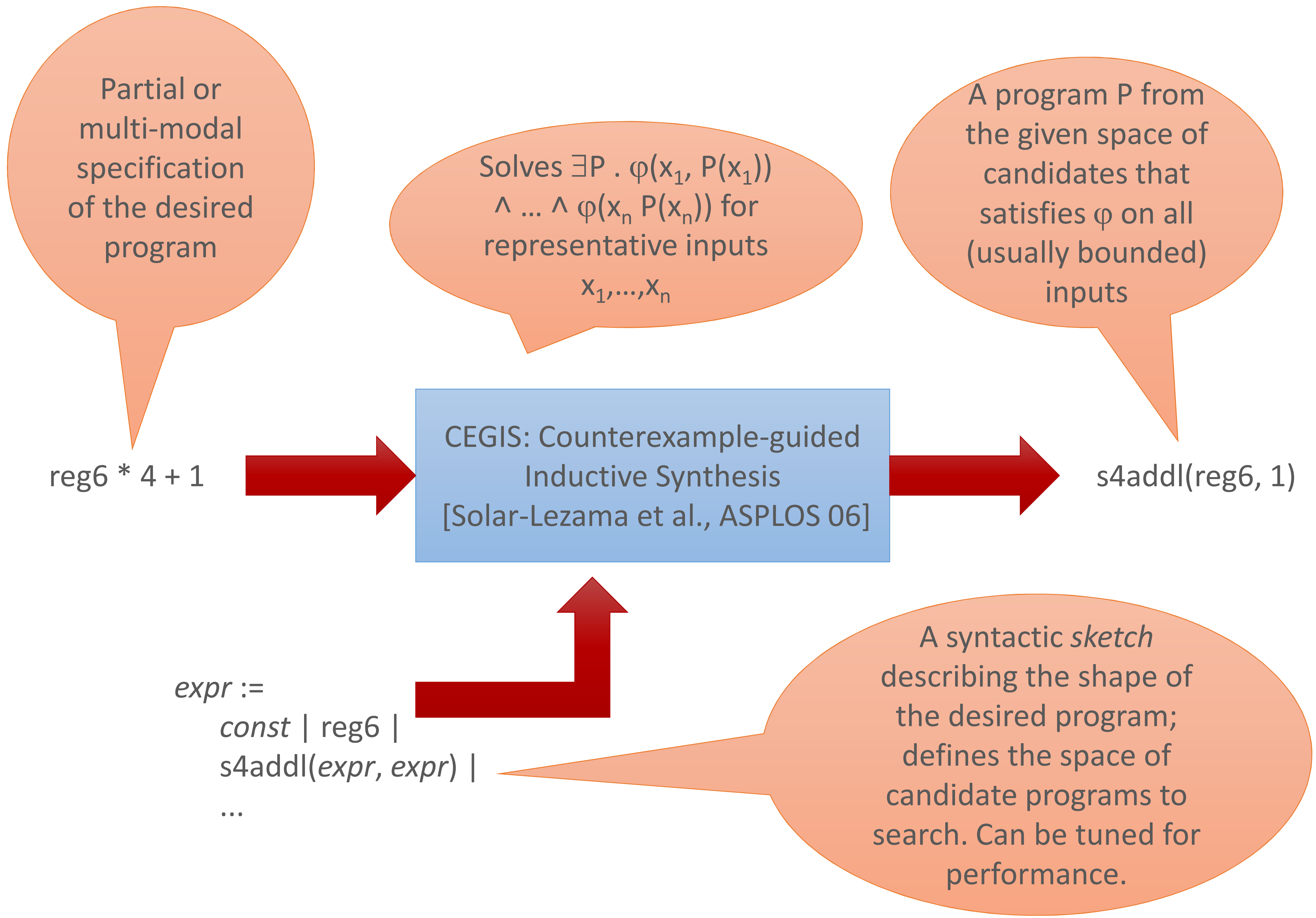
Inductive synthesis search phrased as a **constraint problem**.

Program found by (symbolic) interpretation of a (space of) candidates, not by deriving the candidate.

So, to find a program, we need only an interpreter,
not a sufficient set of derivation axioms.

Overview of CEGIS




Partial or multi-modal specification of the desired program
Solves ∃P . φ(x1, P(x1)) ∧ … ∧ φ(xn P(xn)) for representative inputs x1,…,xn
A program P from the given space of candidates that satisfies φ on all (usually bounded) inputs
reg6 * 4 + 1
CEGIS: Counterexample-guided Inductive Synthesis [Solar-Lezama et al., ASPLOS 06]
s4addl(reg6, 1)
expr :=
const | reg6 |
s4addl(expr, expr) |
...
A syntactic sketch describing the shape of the desired program; defines the space of candidate programs to search. Can be tuned for performance.

Sketching intuition

Extend the language with two constructs

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

$\phi(x, y): y = \mathbf{foo}(x)$

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

?? substituted with an int constant meeting ϕ

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

EXAMPLE: BIT COUNTING

```
bit[W] countBits(bit[W] x)
{
    int count = 0;
    for (int i = 0; i < W; i++) {
      if (x[i]) count++;
    }
    return count;
}
```

Intuition

```
Bit string: ABCDEFGHIJKLMNOP
Mask:       0101010101010101
Result:     0B0D0F0H0J0L0N0P
```

```
Bit string: ABCDEFGHIJKLMNOP
Bits >> 1:  0ABCDEFGHIJKLMNO
Mask:       0101010101010101
Result:     0A0C0E0G0I0K0M0O
```

```
        0B  0D  0F  0H  0J  0L  0N  0P
+       0A  0C  0E  0G  0I  0K  0M  0O
=       A+B C+D E+F G+H I+J K+L M+N O+P
```

Repeat

```
0 A+B+C+D 0 E+F+G+H 0 I+J+K+L 0 M+N+O+P

0000 A+B+C+D+E+F+G+H 0000 I+J+K+L+M+N+O+P

0000 0000 000 A+B+C+D+E+F+G+H+I+J+K+L+M+N+O+P
```

```
bit[W] countSketched(bit[W] x)
        implements countBits {
    loop (??) {
        x = (x & ??) +
                      ((x >> ??) & ?? );
    }
  return x;
}
```

```
bit[W] countSketched(bit[W] x)
{
 x = (x & 0x5555) +
          ((x >> 1) & 0x5555);
 x = (x & 0x3333) +
          ((x >> 2) & 0x3333);
 x = (x & 0x0077) +
          ((x >> 8) & 0x0077);
 x = (x & 0x000F) +
          ((x >> 4) & 0x000F);
 return x;
}
```

High level steps

- Write a program *sketch* with holes and a specification.
- A partial evaluator iteratively rewrites program, converts to a Quantified Boolean Formula Satisfiability problem (QBF); problem becomes:

  $\exists c \in \{0, 1\}^k$ s.t. $\forall x \in \{0, 1\}^m$ $P(x) = S(x, C)$
- (actually 2QBF – i.e. $\forall x \exists y . \varphi$, which makes it tractable)
- Use cooperating theorem provers to fill in holes.

Simple example

```
def f(int[4] in) {
    loop(??)
        f = f ^ in[??] ;
}
```

Partially evaluated

```
def f(int[4] in) {
  loop(??)
    f = f ^ in[??] ;
}
```

```
def f(int[4] in, int c1, int c2, int c3, int c4)
{
  let t0 = c1 in
    if(t0>0)
      f = f^in[c2];
      let t1 = t0-1 in
      if(t1>0)
        f = f^in[c3];
        let t2 = t1-1 in
        if(t2>0)
          f = f ^ in[c4];
          assert t2-1 == 0;
}
```

```
function synthesize (sketch S, spec P)
// synthesize control that completes S for a random input;
// check if it works for all other inputs
// if not, add counterexample input to set of inputs and repeat
I = {}
x = random()
do
  I = I ∪ { x }
  c = synthesizeForSomeInputs(I)
  if c = nil then exit (“buggy sketch”)
  x = verifyForAllInputs(c)
while x ≠ nil
return c
```

```
function synthesizeForSomeInputs(input set I)
// synthesize controls c that make the sketch equivalent to the
// specification on all inputs from I
if ∀_{x∈I} P(x,c) = S(x) is satisfiable then
  return c (the witness)
else
  return nil
```

```
function verifyForAllInputs(control c)
// verify if sketch S completed with controls c is functionally
// equivalent to the specification P.
// If not, return the counterexample
if P(x) ≠ S(x, c) is satisfiable then
  return x (the witness)
else
  return nil
```

Example: Parallel Matrix Transpose

Example: 4x4-matrix transpose with SIMD

a functional (executable) specification:

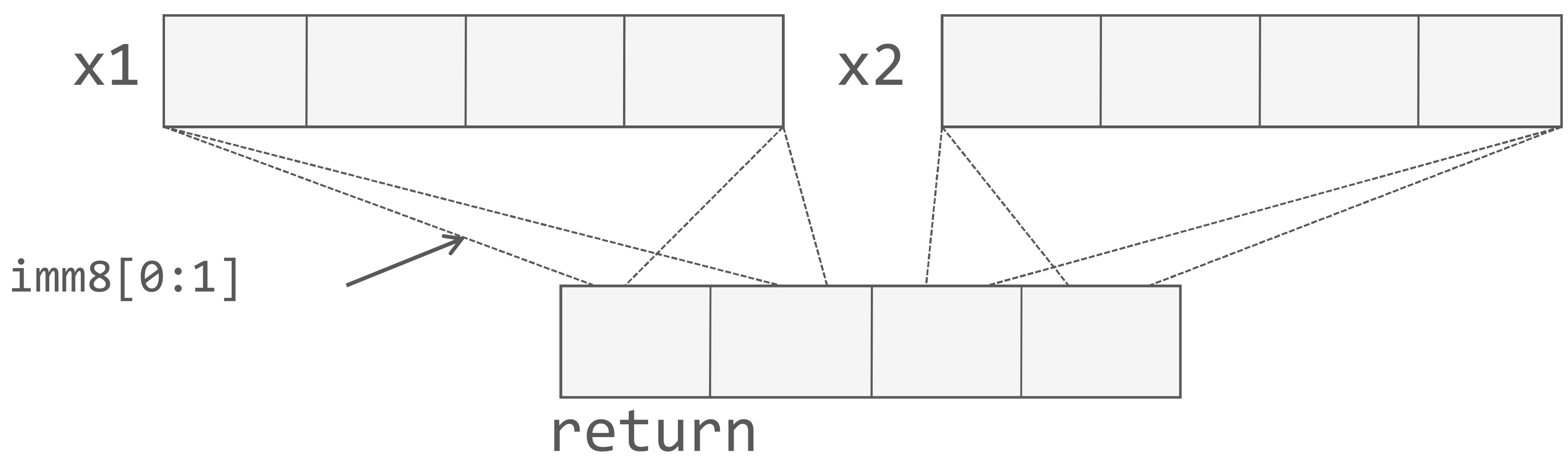
```
int[16] transpose(int[16] M) {
  int[16] T = 0;
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
      T[4 * i + j] = M[4 * j + i];
  return T;
}
```

(This example comes from a Sketch grad-student contest.)

Implementation idea: parallelize with SIMD

Intel SHUFP (shuffle parallel scalars) SIMD instruction:

```
return = shufps(x1, x2, imm8 :: bitvector8)
```



High-level insight: transpose as a 2-phase shuffle

- Matrix M can be 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 the result matrix T with some number of shufps instructions
- Synthesis with partial programs helps one to complete their insight. Or prove it wrong.

SIMD matrix transpose, sketched

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

  S[??::4] = shufps(M[??::4], M[??::4], ??);
  S[??::4] = shufps(M[??::4], M[??::4], ??);      Phase 1
  …
  S[??::4] = shufps(M[??::4], M[??::4], ??);

  T[??::4] = shufps(S[??::4], S[??::4], ??);
  T[??::4] = shufps(S[??::4], S[??::4], ??);      Phase 2
  …
  T[??::4] = shufps(S[??::4], S[??::4], ??);

  return T;
}
```

SIMD matrix transpose with more insight

```
int[16] trans_sse(int[16] M) implements transpose {
  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], ??);
  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], ??);
  T[??::4] = shufps(S[??::4], S[??::4], ??);

  return T;
}
```

4 shuffle instructions per phase

SIMD matrix transpose with even more insight

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

  S[0::4]  = shufps(M[??::4], M[??::4], ??);
  S[4::4]  = shufps(M[??::4], M[??::4], ??);
  S[8::4]  = shufps(M[??::4], M[??::4], ??);
  S[12::4] = shufps(M[??::4], M[??::4], ??);

  T[0::4]  = shufps(S[??::4], S[??::4], ??);
  T[4::4]  = shufps(S[??::4], S[??::4], ??);
  T[8::4]  = shufps(S[??::4], S[??::4], ??);
  T[12::4] = shufps(S[??::4], S[??::4], ??);

  return T;
}
```

1 shuffle instruction per row of output

SIMD matrix transpose, sketched

```
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;
}
```

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

```
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],  10111100b);
  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);
}
```

Summary

- Synthesis: deriving a program given a specification
  - Multiple ways to specify, to constrain the program, and to search the space of programs
- Denali: deductive synthesis for superoptimization
  - Efficient, provably correct – but a lot of work to specify and axiomatize
- Sketching
  - Provides design insight to synthesis tool
  - Effective when you know a program's shape but need to fill in the holes
- What if we don't have a sketch?
  - See the next lecture!

Some links with info on SHUFPS

- https://www.felixcloutier.com/x86/shufps
- http://www.jaist.ac.jp/iscenter-new/mpc/altix/altixdata/opt/intel/vtune/doc/users_guide/mergedProjects/analyzer_ec/mergedProjects/reference_olh/mergedProjects/instructions/instruct32_hh/vc293.htm