

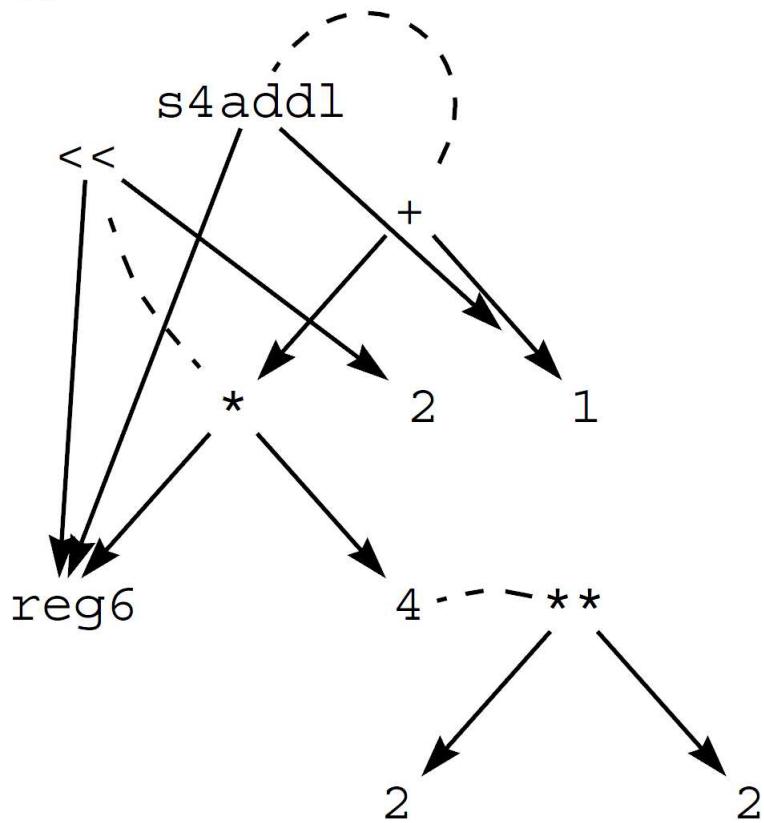
Overview of Program Synthesis

17-355/17-665/17-819:
Program Analysis
Jonathan Aldrich

Slides adapted from material by Claire Le Goues,
Emina Torlak, and Ras Bodik

Denali: synthesis with axioms and E-graphs

[Joshi, Nelson, Randall PLDI'02]



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

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

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

reg6 * 4 + 1
specification

s4addl(reg6,1)
synthesized program

Two kinds of axioms

Instruction semantics: defines (an interpreter for) the language

$$\forall k, n . \ k * 2^n = k \ll n$$

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

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

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

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

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

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

$$(\forall a, i, j, x :: i = j$$

$$\vee \text{select}(\text{store}(a, i, x), j) = \text{select}(a, j))$$

(c) 2019, Jonathan Aldrich and Claire Le

Goues

3

Inductive synthesis

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

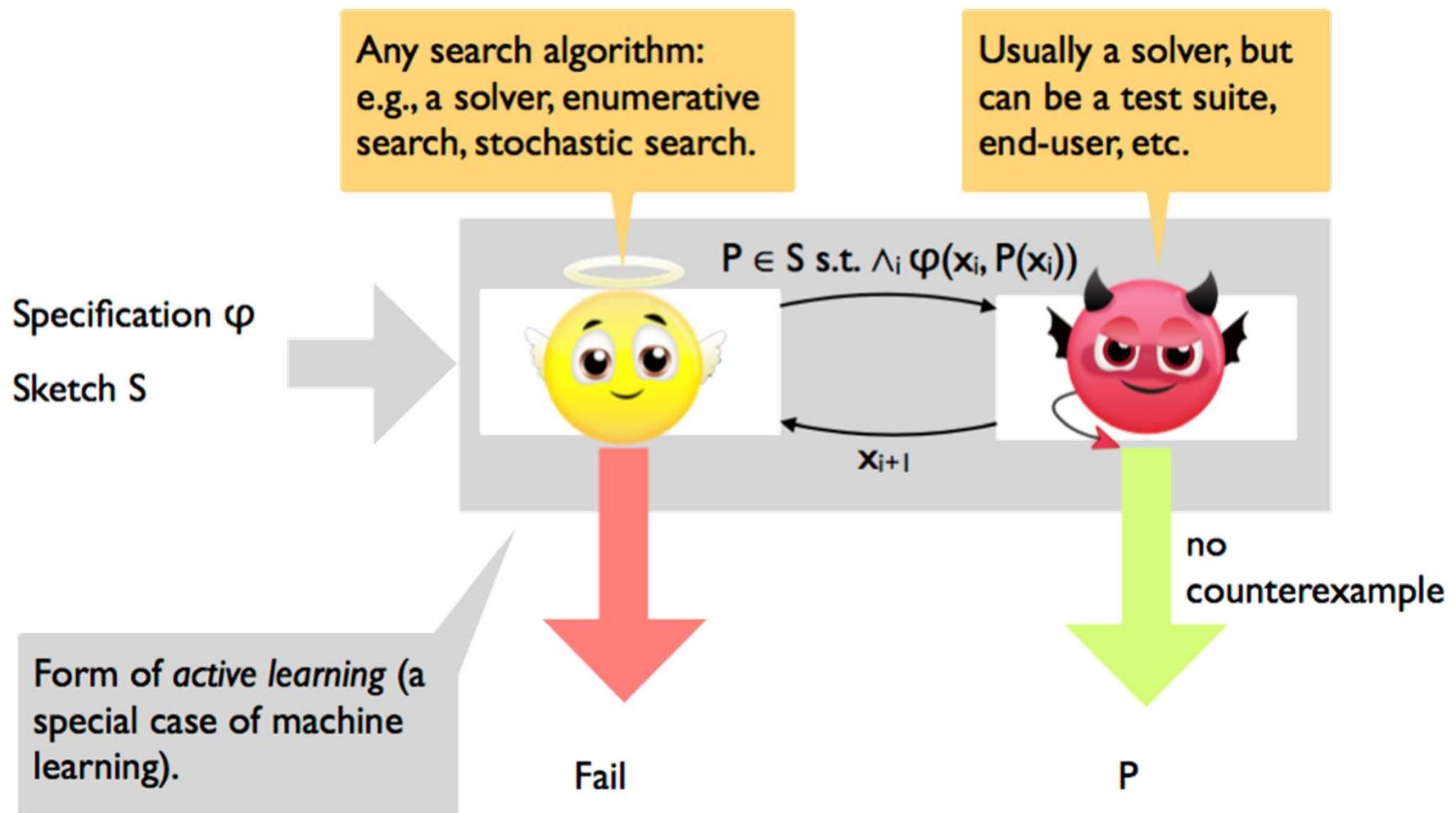
A **partial program** syntactically defines the candidate space.

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 $\exists P . \varphi(x_1, P(x_1)) \wedge \dots \wedge \varphi(x_n, P(x_n))$ for representative inputs
 x_1, \dots, x_n

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: POPULATION COUNTING

```
1. bit[W] pop (bit[W] x) pop
2. {
3.     int count = 0;
4.     for (int i = 0; i < W; i++) {
5.         if (x[i]) count++;
6.     }
7.     return count;
8. }
```

Intuition

Bit string: ABCDEFGHIJKLMNOP

Mask: 0101010101010101

Result: 0B0D0F0H0J0L0N0P

Bit string: ABCDEFGHIJKLMNOP

Bits >> 1: 0ABCDEGHIJKLMNOP

Mask: 0101010101010101

Result: 0A0C0E0G0I0K0M0O

$$\begin{array}{r} 0B \quad 0D \quad 0F \quad 0H \quad 0J \quad 0L \quad 0N \quad 0P \\ + \quad 0A \quad 0C \quad 0E \quad 0G \quad 0I \quad 0K \quad 0M \quad 0O \\ = \quad A+B \quad C+D \quad E+F \quad G+H \quad I+J \quad K+L \quad M+N \quad O+P \end{array}$$

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

```
1. bit[W] popSketched (bit[W] x)
2.     implements pop {
3.         loop (??) {
4.             x = (x & ??) +
5.                 ((x >> ??) & ??);
6.         }
7.         return x;
8.     }
```

```
1. bit[W] popSketched (bit[W] x)
2. {
3.     x = (x & 0x5555) +
4.           ((x >> 1) & 0x5555);
5.     x = (x & 0x3333) +
6.           ((x >> 2) & 0x3333);
7.     x = (x & 0x0077) +
8.           ((x >> 8) & 0x0077);
9.     x = (x & 0x000F) +
10.        ((x >> 4) & 0x000F);
11.    return x;
12. }
```

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

Simple example

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

Partially evaluated

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

```
1. def f(int[4] in, int c1, int c2, int c3, int c4)  
2. {  
3.   let t0 = c1 in  
4.   if(t0>0)  
5.     f = f^in[c2];  
6.   let t1 = t0-1 in  
7.   if(t1>0)  
8.     f = f^in[c3];  
9.   let t2 = t1-1 in  
10.  if(t2>0)  
11.    f = f ^ in[c4];  
12.  assert t2-1 == 0;  
13. }
```

```

1.   function synthesize (sketch S, spec P)
2.     // synthesize control that completes S for a random input;
3.     // check if it works for all other inputs
4.     // if not, add counterexample input to set of inputs and repeat
5.     I = {}
6.     x = random()
7.     do
8.       I = I ∪ { x }
9.       c = synthesizeForSomeInputs(I)
10.      if c = nil then exit ("buggy sketch")
11.      x = verifyForAllInputs(c)
12.    while x ≠ nil
13.    return c

14.   function synthesizeForSomeInputs(input set I)
15.     // synthesize controls c that make the sketch equivalent to the
16.     // specification on all inputs from I
17.     if  $\forall_{x \in I} P(x, c) = S(x)$  is satisfiable then
18.       return c (the witness)
19.     else
20.       return nil

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

```

Example: Parallel Matrix Transpose

Example: 4x4-matrix transpose with SIMD

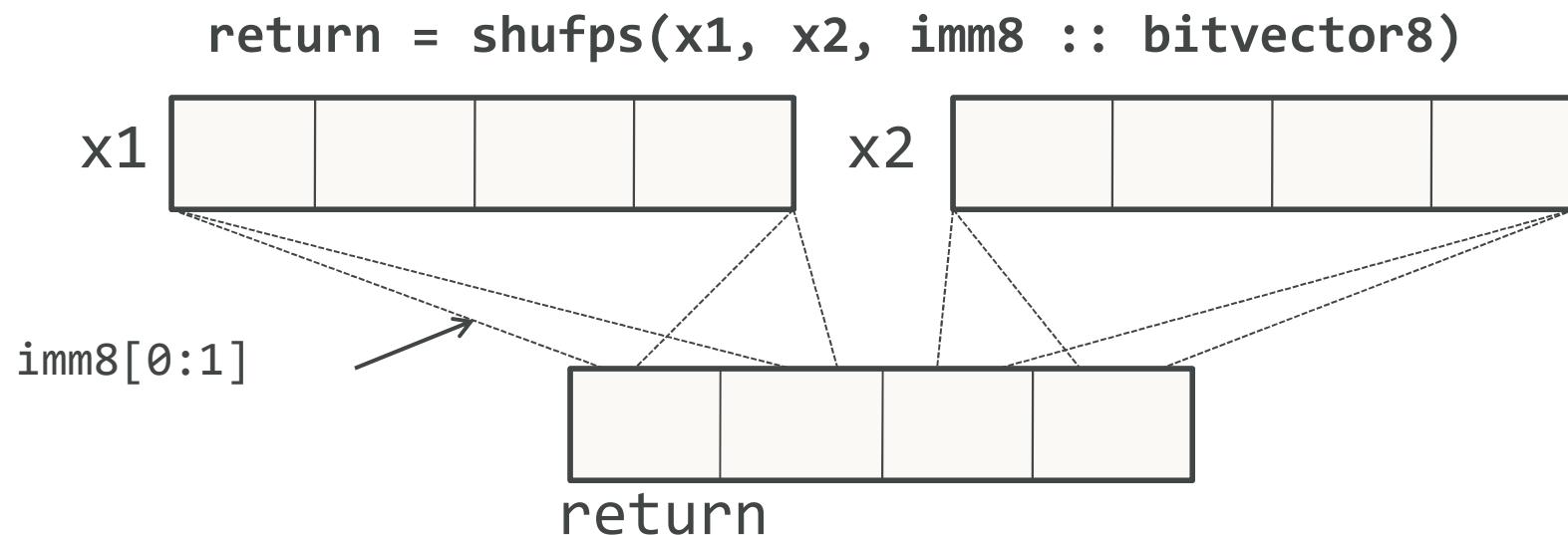
a functional (executable) specification:

```
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:



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 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], ??); } Phase 1

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

    return T;
}
```

SIMD matrix transpose with more insight

```
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], ??);
    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);  
}
```