Program Synthesis, Part 1

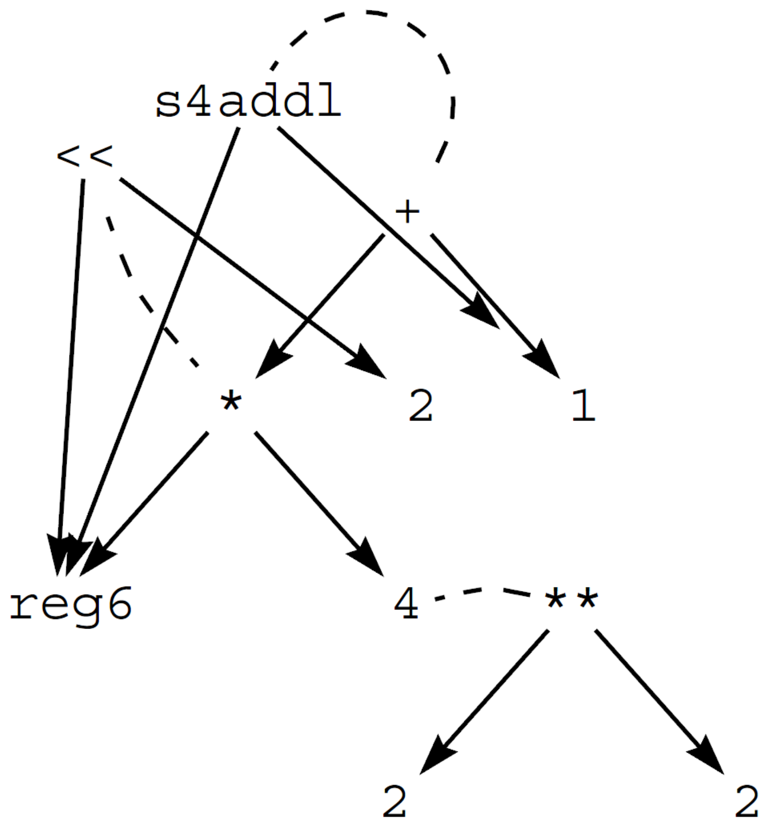
Claire Le Goues

With slide inspiration gratitude to Emina Torlak and Ras Bodik

institute for SOFTWARE RESEARCH

Denali: synthesis with axioms and E-graphs

[Joshi, Nelson, Randall PLDI'02]



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

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

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

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

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, …

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

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

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

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

$$(\forall\, a, i, j, x\ :: i = j \\ \lor \texttt{select}(\texttt{store}(a, i, x), j) = \texttt{select}(a, j))$$

institute for SOFTWARE RESEARCH

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

isr institute for SOFTWARE RESEARCH

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.)

isr institute for SOFTWARE RESEARCH

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.

Partial or multi-modal specification of the desired program

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

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]

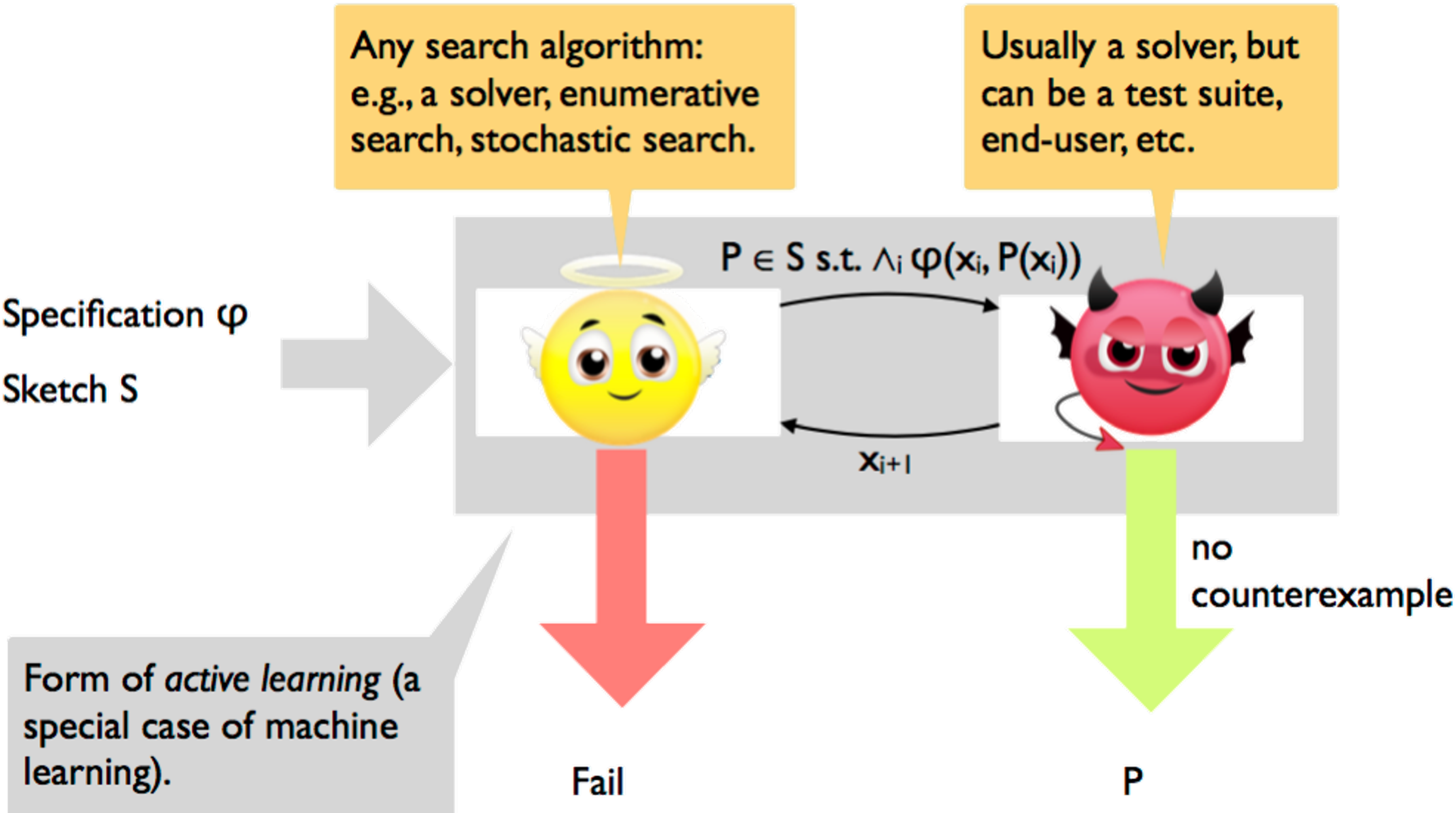
s4addl(reg6, 1)

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.

Institute for SOFTWARE RESEARCH

Overview of CEGIS



institute for SOFTWARE RESEARCH

Sketching intuition

Extend the language with two constructs

spec:

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

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

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

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

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

```
bit[W] popSketched (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;
}
```

(c) 2020, Claire Le Goues

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:

  $\exists\, c \in \{0, 1\}^k, \forall\, x \in \{0, 1\}^m; P(x) = S(x,C)$

  (actually 2QBF, which makes it tractable)
- Use cooperating theorem provers to fill in holes.

institute for SOFTWARE RESEARCH

Simple example

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

(c) 2020, Claire Le Goues

institute for SOFTWARE RESEARCH

Partially evaluated

```
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 counter example 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(inputs set I)
// synthesize controls C that make the sketch equivalent to the
// specification on all inputs from I
if ∧_{x ∈ I} P(x) = S(x,c) is satisfiable then
  return a satisfying c
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 satisfying x
else
  return nil
```


institute for SOFTWARE RESEARCH

Example: Parallel Matrix Transpose

Example: 4x4-matrix transpose with SIMD

a functional (executable) specification:

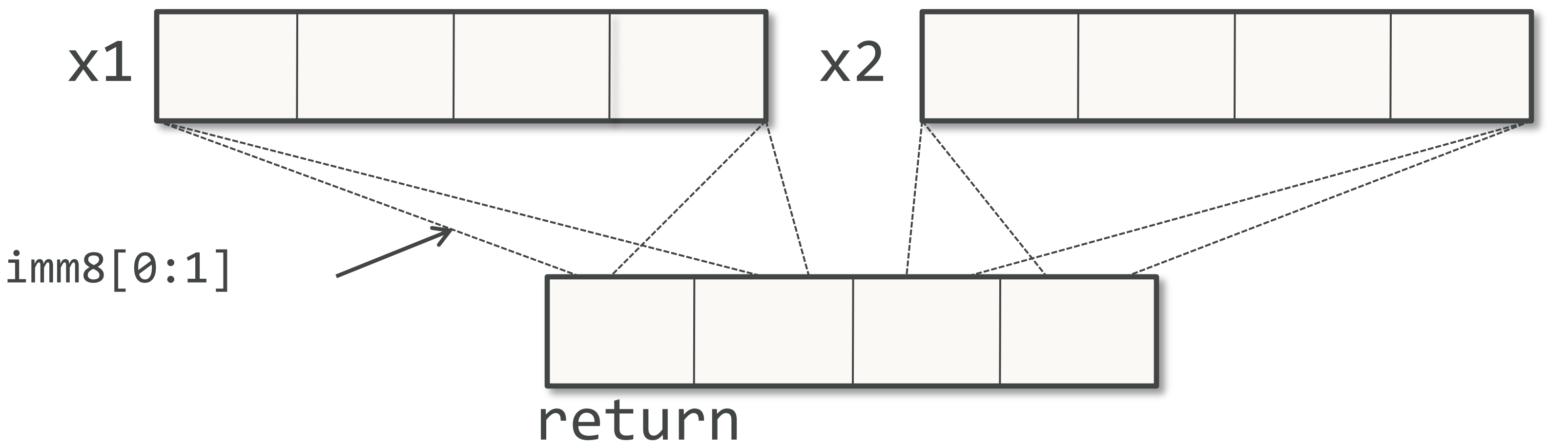
```
int[16] transpose(int[16] M) {
  int[16] T = 0;
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
      T[4 * i + j] = M[4 * j + i];
  return T;
}
```

(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 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;
}
```

Phase 1

Phase 2

isr institute for SOFTWARE RESEARCH

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

institute for SOFTWARE RESEARCH

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

isr institute for SOFTWARE RESEARCH

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);
}
```