

Syntax-Guided Synthesis

Agenda

Let us switch gears to talk about the target of program synthesis

What is a program?

How to represent a program?

How to represent the space of programs?

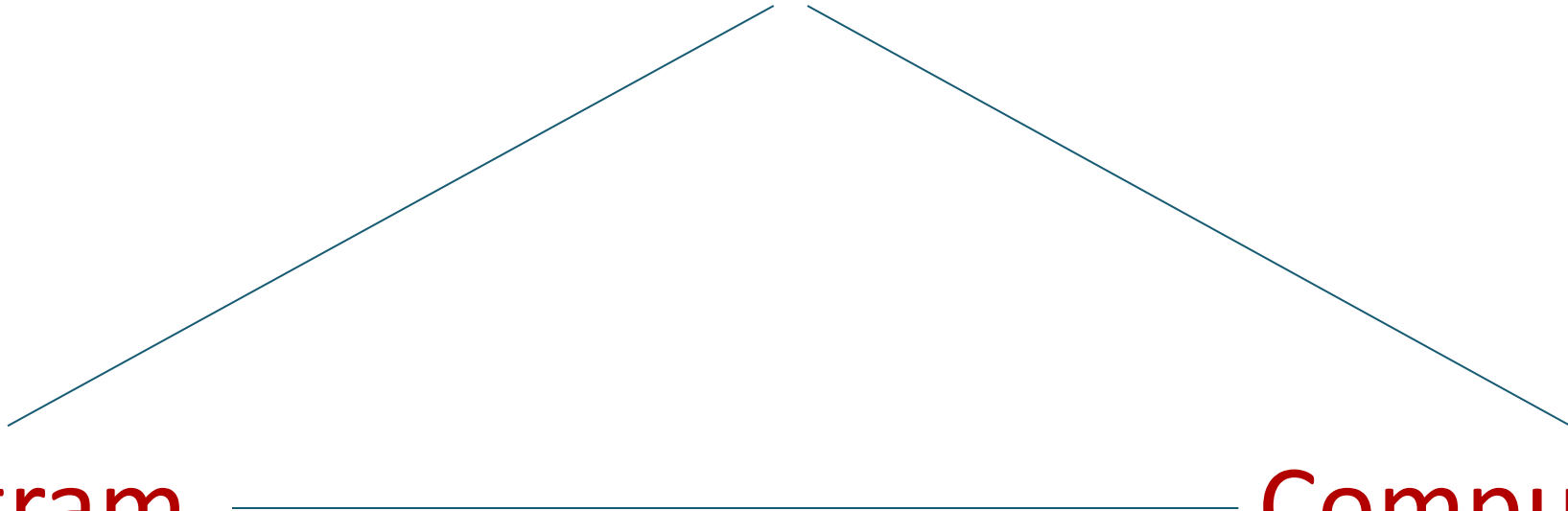
How to get help from the user?

What are we synthesizing?

Algorithm (abstract)

Program
(implementation)

Computation
(theory)



What do we mean by "program?"

- General-purpose programming languages such as Java or Python (e.g. JSketch)
- Arithmetic/Boolean expressions (e.g., SyGuS solvers)
- Domain-specific languages (e.g., FlashFill, sketch-n-sketch)
- Hardware circuit (e.g., reactive synthesis)
- You name it!

How to represent a target program?



Reactive Programs

- The oldest class of programs considered for synthesis (Church 1957)
- **Representation:** Finite State Machines
- **Formal Specs:** Temporal Logics (MSO, LTL, CTL, etc.)
- **Examples:** Event/Action Sequences (finite or infinite)

How to represent a target program?

sorts:

Int, Bool, Id, AExp, BExp, Block, Stmt, Pgm

subsorts:

Int, Id < *AExp*

Bool < *BExp*

Block < *Stmt*

operations:

$_ + _ : AExp \times AExp \rightarrow AExp$

$_ / _ : AExp \times AExp \rightarrow AExp$

$_ <= _ : AExp \times AExp \rightarrow BExp$

$_ ! _ : BExp \rightarrow BExp$

$_ \&\& _ : BExp \times BExp \rightarrow BExp$

$\{ \} : \rightarrow Block$

$\{ _ \} : Stmt \rightarrow Block$

$_ = _ ; _ : Id \times AExp \rightarrow Stmt$

$_ - _ : Stmt \times Stmt \rightarrow Stmt$

$if(_) _ else _ : BExp \times Block \times Block \rightarrow Stmt$

$while(_) _ : BExp \times Block \rightarrow Stmt$

$int _ ; _ : List\{Id\} \times Stmt \rightarrow Pgm$

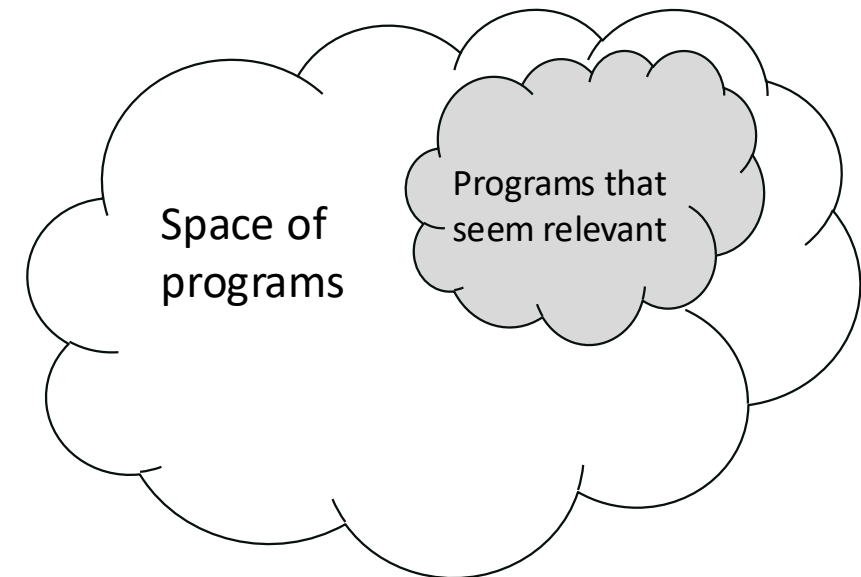
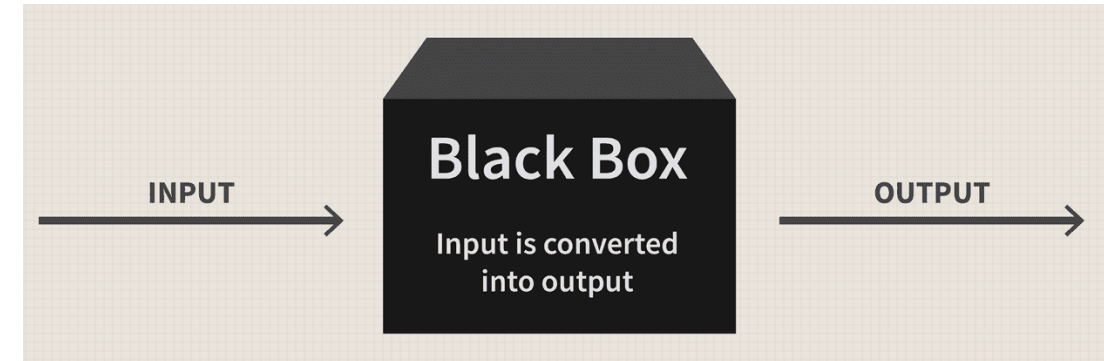
Programming Languages

- Many different choices (imperative, functional, domain-specific)
- **Representation:** Context-free grammars (CFGs) or Regular tree grammars (RTGs)
- **Formal Specs:** FOL, separation logic, etc.
- **Examples:** input/output pairs

How can the user help?

So far, what the user specifies is *all semantic*

- Even a carefully chosen program space is still too big
- It is extremely helpful if the user can also specify the syntactic aspect of the target program (e.g., how the program looks like)
- The user may provide concise grammars, or program sketches
- The form of the syntactic hint is tightly related to the Invention pillar (i.e., how to search)



Example

Semantic specification:

$\text{Process}(1,4,2,0,7,9,2,5,0,3,2,4,7) =$
 $(1,2,4,0,2,5,7,9,0,2,3,4,7,0)$

Syntactic constraint:

$\text{lstExpr} := \text{sort}(\text{lstExpr})$
 $\text{lstExpr}[\text{intExpr}, \text{intExpr}]$
 $\text{lstExpr} + \text{lstExpr}$
 $\text{recursive}(\text{lstExpr})$
 $[0]$
 in
 $\text{intExpr} := \text{firstZero}(\text{lstExpr})$
 $\text{len}(\text{lstExpr})$
 0
 $\text{intExpr} + 1$

Syntax-guided
Synthesizer

$\text{Process}(\text{in}) := \text{sort}(\text{in}[0, \text{firstZero}(\text{in})]) + [0] + \text{recursive}(\text{in}[\text{firstZero}(\text{in})+1, \text{len}(\text{in})]);$

What is the search space?

Delineated by the grammar

- A domain-specific language (DSL) for list manipulation
- `lstExpr` represents a much smaller search space than a general-purpose language
- Still not searchable in practice?

```
lstExpr := sort(lstExpr)
         lstExpr[intExpr,intExpr]
         lstExpr + lstExpr
         recursive(lstExpr)
         [0]
         in
intExpr := firstZero(lstExpr)
         len(lstExpr)
         0
         intExpr + 1
```

What if we know the following:

Sort is never called more than once in a sub-list.

Recursive calls should be made on lists whose length is less than `len(in)`

We'll never have to add one multiple times in a row

Program Sketching

Sketch: maximizing syntactic guidance

- A simple imperative language with unique features allowing expert users to carefully engineer a program space
- Also served as a target language to which higher-level languages are encoded
- The underlying synthesis engine is constraint-based and well engineered
- Frontend: <https://github.com/asolarlez/sketch-frontend>
- Backend: <https://github.com/asolarlez/sketch-backend>
- Manual: <http://people.csail.mit.edu/asolar/manual.pdf>

Language Design Strategy

Extend base language with one construct

Constant hole: ??

```
int bar (int x)
{
    int t = x * ??;
    assert t == x + x;
    return t;
}
```



```
int bar (int x)
{
    int t = x * 2;
    assert t == x + x;
    return t;
}
```

Synthesizer replaces ?? with a constant

High-level constructs defined in terms of ??

Integer Generator $\rightarrow$ Sets of Expressions

Expressions with ?? == sets of expressions

linear expressions

$x^{*??} + y^{*??} + ??$

polynomials

$x^{*}x^{*??} + x^{*??} + ??$

sets of variables

$?? \ ? \ x : y$

Example: Registerless Swap

Swap two words without an extra temporary

```
int W = 32;
```

```
void swap(ref bit[W] x, ref bit[W] y) {  
    if(??) { x = x ^ y; } else { y = x ^ y; }  
    if(??) { x = x ^ y; } else { y = x ^ y; }  
    if(??) { x = x ^ y; } else { y = x ^ y; }  
}
```

```
harness void main(bit[W] x, bit[W] y) {  
    bit[W] tx = x; bit[W] ty = y;  
    swap(x, y);  
    assert x==ty && y == tx;  
}
```

Generators

From simple to complex holes

We need to compose ?? to form complex holes

Borrow ideas from generative programming

Define generators to produce families of functions

Use partial evaluation aggressively

Generators

Look like a function

but are partially evaluated into their calling context

Key feature:

Different invocations → Different code

Can recursively define arbitrary families of programs

Sample Generator

```
/**
 * Generate the set of all bit-vector expressions
 * involving +, &, xor and bitwise negation (~).
 * the bnd param limits the size of the generated expression.
 */

generator bit[W] gen(bit[W] x, int bnd) {
    assert bnd > 0;
    if(??) return x;
    if(??) return ??;
    if(??) return ~gen(x, bnd-1);
    if(??) {
        return { | gen(x, bnd-1) (+ | & | ^) gen(x, bnd-1) | };
    }
}
```

Example: Least Significant Zero Bit

```
bit[W] isolate0 (bit[W] x) {
    bit[W] ret = 0;
    for (int i = 0; i < W; i++)
        if (!x[i]) { ret[i] = 1; return ret; }
}

generator bit[W] gen(bit[W] x, int bnd) {
    assert bnd > 0;
    if(??) return x;
    if(??) return ??;
    if(??) return ~gen(x, bnd-1);
    if(??) return { | gen(x, bnd-1) (+ | & | ^) gen(x, bnd-1) | };
}

bit[W] isolate0sk implements isolate0 (bit[W] x) {
    return gen(x, 3);
}
```

Higher Order Generators

```
/*  
 * Generate code from f n times  
 */  
generator void rep(int n, fun f) {  
    if(n>0) {  
        f();  
        rep(n-1, f);  
    }  
}
```

Closures + Higher Order Generators

```
generator void rep(int n, fun f){
    if(n>0){
        f();
        rep(n-1, f);
    }
}

bit[16] reverseSketch(bit[16] in) {
    bit[16] t = in;
    int s = 1;
    generator void tmp(){
        bit[16] m = ??;
        t = ((t << s) & m) | ((t >> s) & (~m));
        s = s*??;
    }
    rep(??, tmp);
    return t;
}
```

Syntactic Sugar

`{ | RegExp | }`

RegExp supports choice `'|'` and optional `'?'`

can be used arbitrarily within an expression

to select operands `{ | (x | y | z) + 1 | }`

to select operators `{ | x (+ | -) y | }`

to select fields `{ | n(.prev | .next)? | }`

to select arguments `{ | foo( x | y, z) | }`

Set must respect the type system

all expressions in the set must type-check

all must be of the same type

Repeat

Avoid copying and pasting

`repeat(n){s}` → `s;s;...s;`

each of the n copies may resolve to a distinct stmt
 n can be a hole too.

Example: Reversing Bits

```
pragma options "--bnd-cbits 3 ";

int W = 32;
bit[W] reverseSketch(bit[W] in) {
    bit[W] t = in;
    int s = 1;
    int r = ??;
    repeat(??){
        bit[W] tmp1 = (t << s);
        bit[W] tmp2 = (t >> s);
        t = tmp1 {||} tmp2; // Syntactic sugar for m=??, (tmp1&m | tmp2&~m).
        s = s*r;
    }
    return t;
}
```

Algebraic Data Types

Algebraic Data Types

Sketch supports

Algebraic Data Types

Polymorphic Synthesis Constructs

Natural for sketching recursive transformations over structured data (e.g., abstract syntax trees for compilers)

Example: Lambda Calculus

```
adt dstAST {  
    VarD { int name;}  
    AbsD { VarD var; dstAST a;}  
    AppD { dstAST a; dstAST b;}  
}
```

```
dstAST interpretDstAST(dstAST s){  
    if (s == null) return null;  
    switch(s){  
        case AppD: {  
            dstAST s_a = interpretDstAST(s.a);  
            dstAST s_b = s.b;  
            if (s_a == null || s_b == null) return null;  
            switch (s_a) {  
                case AbsD:{  
                    dstAST s_new = substitute(s_a.var, s_a.a, s_b);  
                    if(s_new == null) return null;  
                    return interpretDstAST(s_new); } ...  
            }}...  
    }...}
```

Goal: Church Encoding

```
adt srcAST{  
  VarS { int name;}  
  AbsS { VarS var; srcAST a;}  
  AppS { srcAST a; srcAST b;}  
  AndS { srcAST a; srcAST b;}  
  OrS { srcAST a; srcAST b;}  
  NotS { srcAST a;}  
  TrueS {}  
  FalseS {}  
}
```

desugar



```
adt dstAST {  
  VarD { int name;}  
  AbsD { VarD var; dstAST a;}  
  AppD { dstAST a; dstAST b;}  
}
```

Synthesizing the Desugaring

```
dstAST desugar(srcAST s){
  if(s == null) return null;

  switch(s){
    repeat_case:{
      dstAST v1 = desugar(s.??);
      dstAST v2 = desugar(s.??);
      return new ??(a = getPart(v1, v2, 2),
                    b = getPart(v1, v2, 2),
                    name = {|s.?? | ?? |},
                    var = getVar(s.??));
    }
  }
}
```

```
generator VarD getVar(VarS var){
  if (var == null) return new VarD(name = ??);
  else return new VarD(name = {|var.name | ?? |});
}

generator dstAST getPart(dstAST c1, dstAST c2, int bnd){
  dstAST z = new VarD(name = ??);
  if (??) return {| c1|c2|z|};
  else if (bnd > 1){
    return new ??( a = getPart(c1,c2,bnd-1),
                  b = getPart(c1,c2,bnd-1),
                  name = ??,
                  var = new VarD(name = ??));
  }
  return null;
}
```

Synthesizing the Desugaring

```
harness void main(int[20] arr){  
    int idx =0;  
    srcAST s = produce(arr, idx, 2);  
    srcAST c1 = interpretSrcAST(s);  
    if(c1!=null){  
        dstAST c2  = desugar(s);  
        dstAST c3 = interpretDstAST(c2);  
        assert(c3!=null);  
        assert(equals(c1,c3));  
    }  
}
```