Multi-output Program Synthesis with Sharing

Ruijie Fang, Kexin Jin
Princeton University

COS 516 Automated Reasoning about Software
Professor Aarti Gupta, November 2020

Content of the Talk

Main Problem: How to do program synthesis for programs that have multiple outputs such that we allow sharing of common subexpressions and the depth of the resulting AST is as shallow as possible.

Application: To program synthesis of network switches (smaller depth = easier to fit into specialized hardware).

Agenda:

- What is syntax-guided synthesis
- Chipmunk: Why use synthesis for networked devices
- What is Sketch
- Min-depth single-output synthesis in Sketch
- Multi-output synthesis with sharing of common subexpressions
- Pending work

Syntax-guided Synthesis (SyGuS)

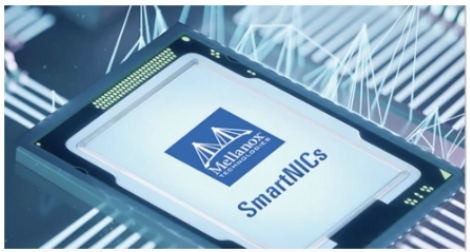
- A background theory T
- A typed function f
- A formula φ over the vocabulary of T
- Context-free grammar

Goal: synthesize an expression according to the grammar for the function f such that the formula $\varphi[f/e]$ is valid modulo T.

Programmable Network Devices

Programmable network devices enable new types of intelligence on the network. Examples include high-speed programmable switches, multicore SoC SmartNICs, FPGAs, software middleboxes, and the networking stack within servers.

100 Gbits/s



100 Gbits/s

6.5 Tbits/s

However, it is hard to write fast packet-processing code that fits with resource constraints of specialized hardware architecture. It requires familiarity with the underlying architecture and on sophisticated compilation techniques.

A Synthesis-based Approach

In synthesis-based techniques for compilation for programmable network switches, the network program is treated as the specification and the grammar represents the functionality available in switch hardware.

- X. Gao, T. Kim, A. K. Varma, A. Sivaraman, and S. Narayana (2019): Autogenerating Fast Packet-Processing Code Using Program Synthesis.
- X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta (2020): Switch Code Generation Using Program Synthesis.

Chipmunk: A Sketch-based synthesis tool for network switches

- X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta (2020): Switch Code Generation Using Program Synthesis.

Main Idea: The user supplies a program in Domino, a high-level language for packet processing. The program is then divided into different components; each component is compiled into a Sketch specification. Then Sketch is used to generate the output according to a predefined grammar structure.

Sketch: Synthesis by Sketching

Sketch is a program synthesizer that takes as input a **partial program** and user-specified constraints, and aims to output a complete program. A Sketch partial program input is presented in the Sketch language, which bears a high resemblance to C++ and Java, with a new construct called **holes**, which represent unspecified variable assignments. The vanilla output will be in the Sketch language, although it can also output concrete code in C++ via command-line flags.

Sketch Internals

More formally: hole filling $\iff$ QBF solving

Let $\vec{v} \in 0,1^n$ be an n-bit vector representing all input variables to a specification $S(\vec{x})$ of the partial program P. Let $\vec{c} \in 0,1^m$, $m < n$ be a vector that specifies the location of the "holes" in $\vec{x}$. Sketch solves the following **quantified boolean formula**:

$$\exists \vec{c} \in 0,1^m . \forall \vec{x} \in 0,1^n . S(\vec{x}) = P(\vec{x}, \vec{c})$$

.

How? CEGIS (Counterexample-Guided inductive synthesis)

- QBF solvers not as performant in this setting
- Idea of CEGIS: a) synthesize a small set of test inputs; b) do a search to fill in the holes; c) verify the test inputs given to the filled program matches the specification; d) If not, add it to the counterexample set and return to b).

Programming in Sketch

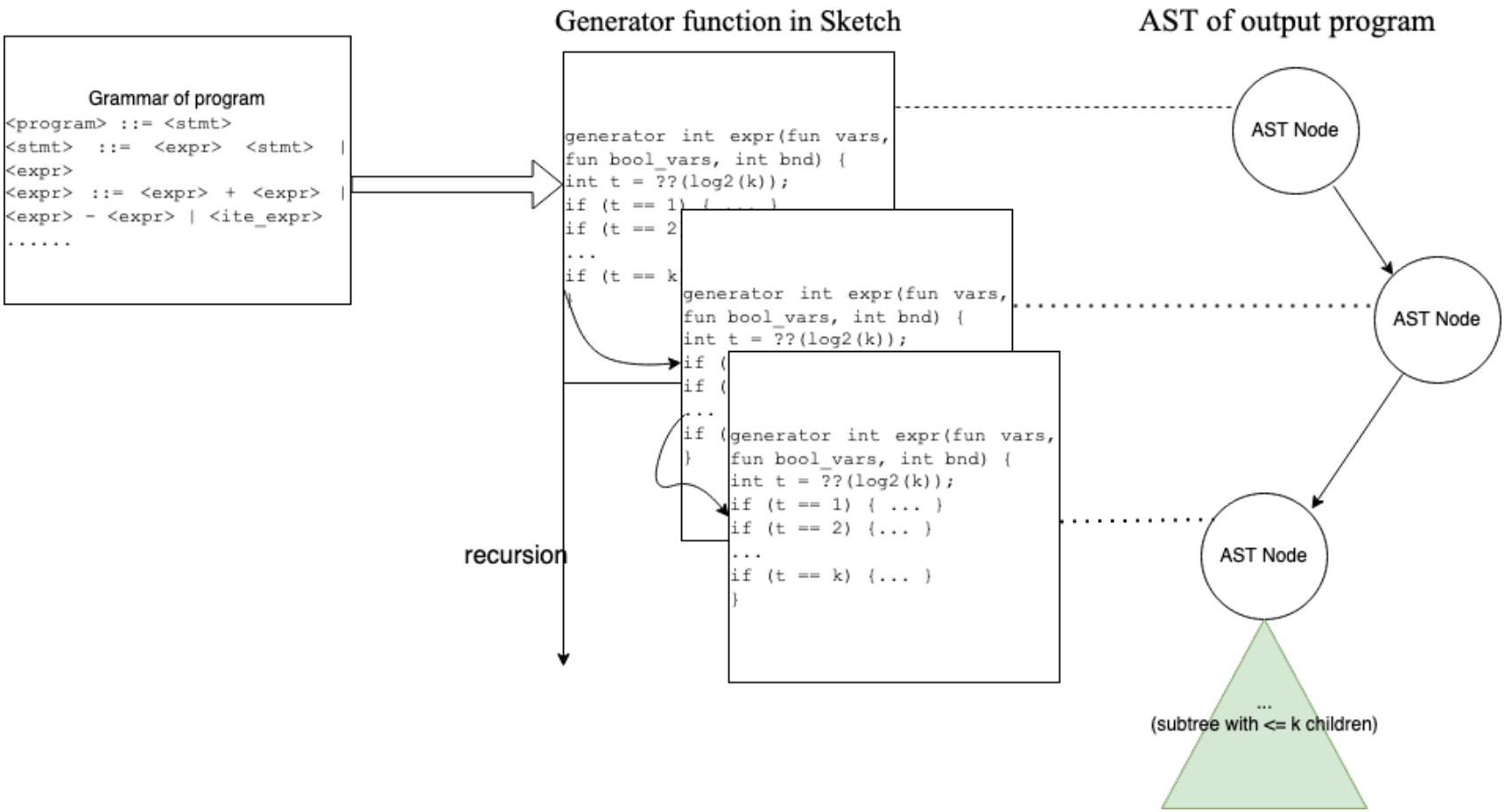
Two main parts: A harness containing holes, and a function containing the specification.
The last assertion asserts semantic equality of the completed program and the program specified by specification.

```
int doubleSketchSpec(int x) { return x + x; }
harness void doubleSketch(int x) {
int y = x * ??;
assert y == doubleSketchSpec(x); }
```

Handling Grammar in Sketch

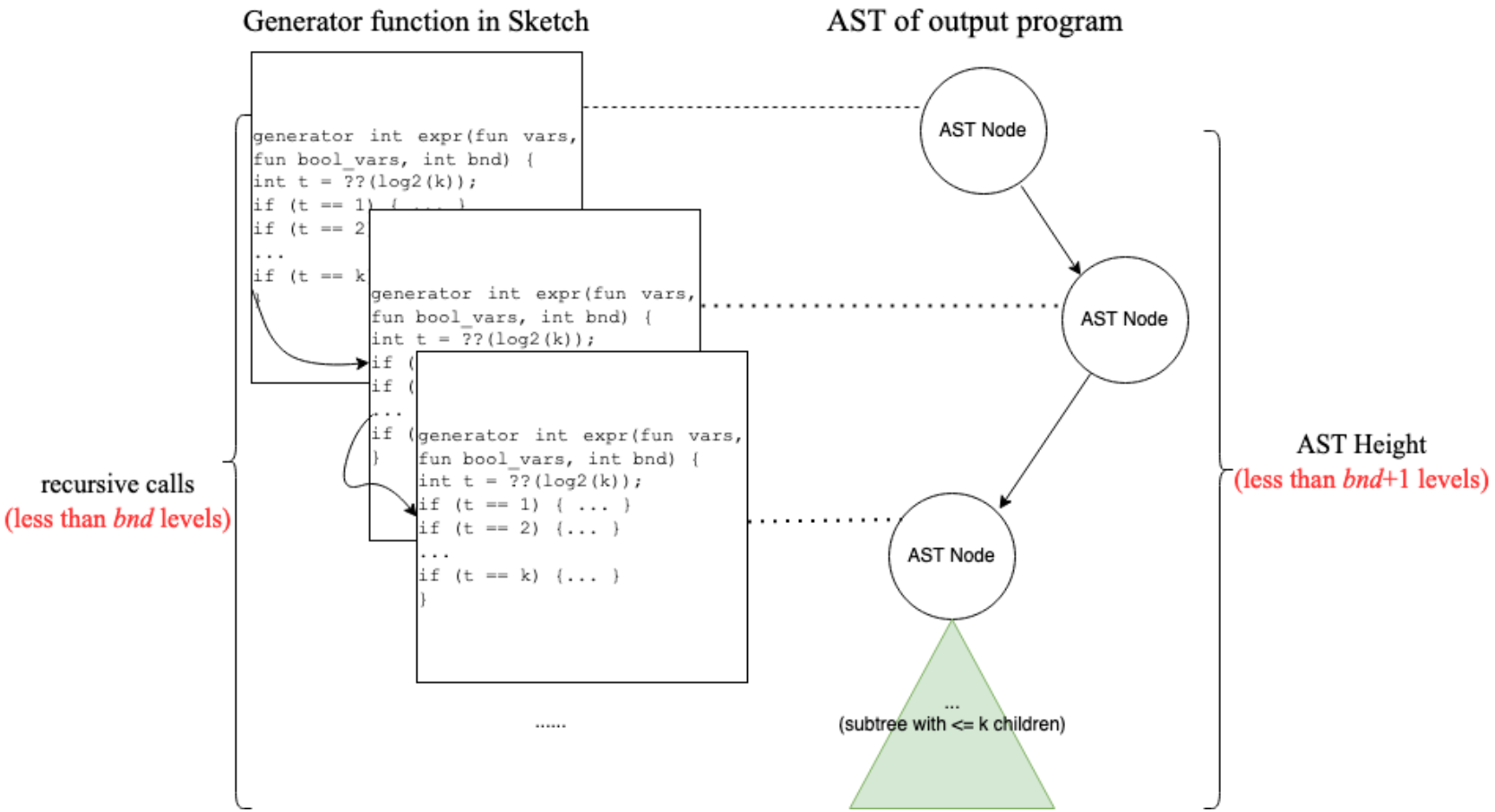
Specifying a Grammar. Constrain the structure of the output Sketch generates via defining generators that describe the abstract syntax tree (AST) structure of a grammar. Inside the harness of the partial Sketch program, call the generator to obtain a synthesized expression, and then use assertions in Sketch to check if it conforms with the specification.
Each recursion level in the generator function corresponds to a node in the AST. At each recursion level in the AST, we will have k choices on what to recurse according to the grammar specification. We use a variable associated with a fixed-width hole `int v = ??(`$\lceil \log_2 k \rceil$`)` for Sketch to automatically search through all possible enumerations.

Handling Grammar in Sketch



Min-depth Synthesis in Sketch

Idea: Min-Depth $\equiv$ Min-Depth of AST. Specify a recursion depth parameter `bnd` in generator. Stopping the recursion whenever `bnd` reaches 0 will constrain the depth of the AST.



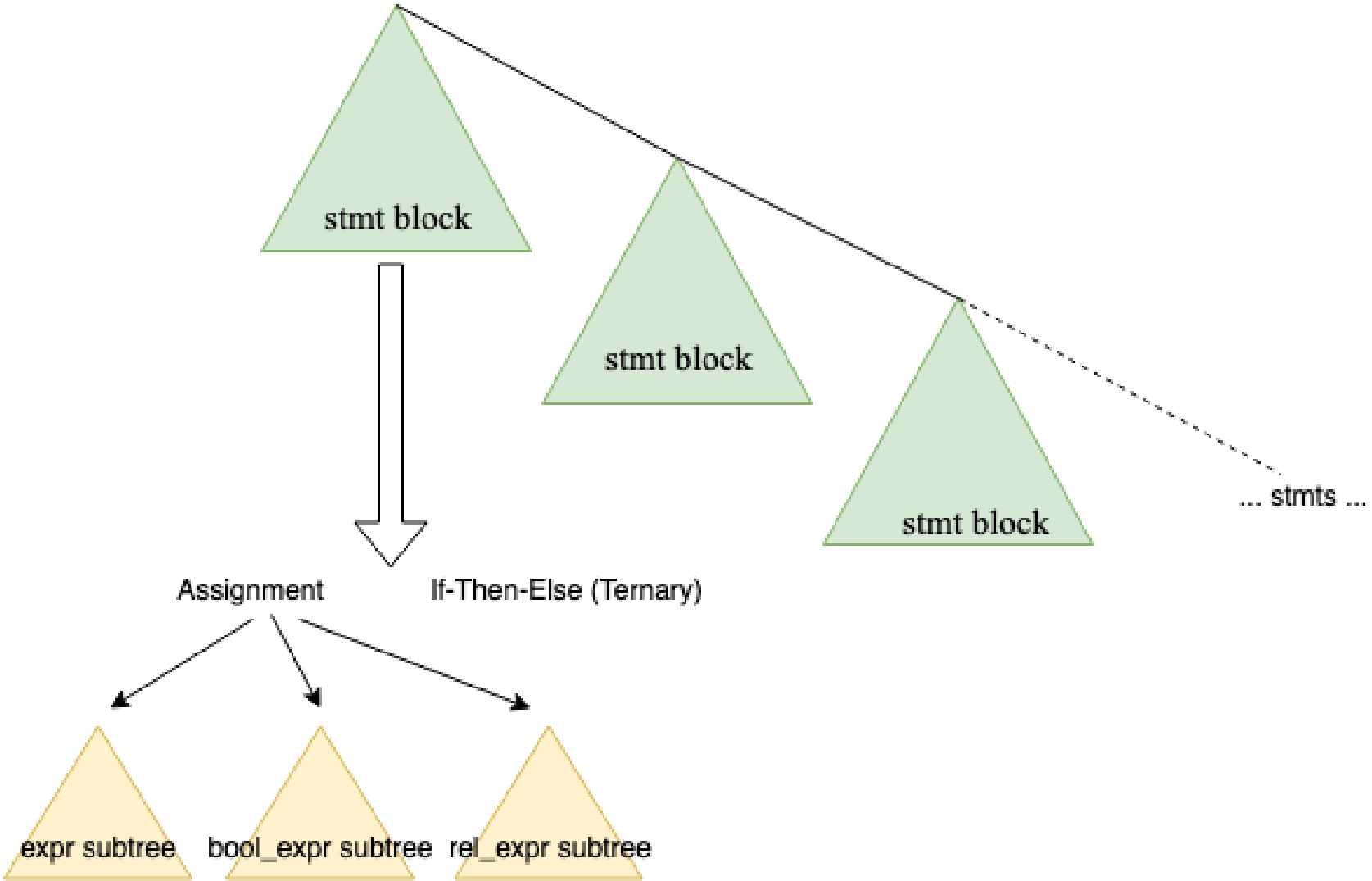
High-level description of Grammar

Description. The grammar we'll be working with represents a simple straight-line program grammar with ITE expressions.

```
generator int expr(fun vars, fun bool_vars, int bnd) {
                        assert bnd >= 0;
                        int t = ??(3);
                        if (t == 0) {
                                return vars();
                        }
                        if (t == 1) {
                                return ??;
                        }
                        if (t == 2) {
                                return -??;
                        }
                        if (t == 3) {
                                return ite_expr(vars, bool_vars, bnd);
                        }
        else {
                                tempVar tmp = new tempVar();
                                tmp.temp = {| expr(vars, bool_vars, bnd-1) (+ | -) expr(vars, bool_vars, bnd-1) |};
                                return tmp.temp;
                        }
                }
```

High-level description of Grammar

Description. The grammar we'll be working with represents a simple straight-line program grammar with ITE expressions.



Multi-output synthesis

Gist of problem:

- We know how to do synthesis in sketch for a program that returns a single output.
- What about multi-output programs (e.g. programs that return a pair/tuple, or an array of elements).
- Trivial solution: Do generation for each output and merge them (but cannot reuse overlapping substructures in the code.)

Formal description of the problem: The program $P(\vec{x}, \vec{c})$ will now return a p-dimensional array, so does the specification $S(\vec{x})$. Sketch will assert that each element of the output conforms to the specification:

$$\forall i. P(\vec{x}, \vec{c})[i] = S(\vec{x})[i]$$

.

Allowing Sharing for Multi-Output

Problem. Trivial solution for multi-output synthesis not min-depth! Need to find a way to reuse common subexpressions between outputs.

Two Solutions:

- Sharing subexpressions $\iff$ sharing of states between calls to Sketch generators. Find a way to share mutable states between Sketch generators: Use structs/array refs.
- Postprocessing of Sketch output: Use a compiler pass to do common subexpression elimination and dead code elimination of trivial multi-output synthesis result.

Main idea

Modify AST generator in Sketch to allow sharing of up to k common variables. Sketch supports pass-by-reference using the `ref` keyword or a struct. Pass the reference object into the generator so that different calls to the generator can read/modify the same struct/ref array.

Postprocessing of output. After Sketch outputs the complete program, use an extra compiler pass to do DCE and common subexpression elimination.

Caveats: Our approach implies a slower synthesis time: Modify the generator in Sketch to create k more branches (for assignment of common variables) at each recursion level $\Longrightarrow$ Much more time spent on enumerating the subtrees of AST.

An Example (without sharing)

Specification:

```
bit[2] component_0(int pkt_arrival0, int pkt_last_time_pkt_id0_0) {
    bit pkt_br_tmp0;
    bit pkt_br_tmp1;
    pkt_br_tmp0 = (pkt_arrival0 - pkt_last_time_pkt_id0_0 > 3);
    pkt_br_tmp1 = (pkt_arrival0 - pkt_last_time_pkt_id0_0 > 2);
    return {pkt_br_tmp0, pkt_br_tmp1};
}
```

Synthesize component_0 with depth 2 for both output.

```
harness void sketch_component0(int pkt_arrival0, int pkt_last_time_pkt_id0_0) {
    generator bit bool_vars() {
        return {| 0 | 1 |};
    }

    generator int vars() {
       return {| pkt_arrival0 | pkt_last_time_pkt_id0_0 |};
    }

    assert bool_expr(vars, bool_vars, 2) == component_0(pkt_arrival0, pkt_last_time_pkt_id0_0)[0];
    assert bool_expr(vars, bool_vars, 2) == component_0(pkt_arrival0, pkt_last_time_pkt_id0_0)[1];

}
```

Output:

```
void sketch_component0 (int pkt_arrival0, int pkt_last_time_pkt_id0_0)/*f
{
  tempBitVar@ANONYMOUS tmp = new tempBitVar();
  tempVar@ANONYMOUS tmp_0 = new tempVar();
  tmp_0.temp = pkt_arrival0 + -3;
  int _out_s115 = tmp_0.temp;
  tmp.temp = pkt_last_time_pkt_id0_0 < _out_s115;
  bit _out_s16 = tmp.temp;
  bit[2] _out_s8 = {0,0};
  component_0(pkt_arrival0, pkt_last_time_pkt_id0_0, _out_s8);
  assert (_out_s16 == (_out_s8[0])); //Assert at flow.sk:189 (0)
  tempBitVar@ANONYMOUS tmp_1 = new tempBitVar();
  tempVar@ANONYMOUS tmp_2 = new tempVar();
  tmp_2.temp = pkt_last_time_pkt_id0_0 - pkt_arrival0;
  int _out_s119 = tmp_2.temp;
  tmp_1.temp = -3 >= _out_s119;
  bit _out_s16_0 = tmp_1.temp;
  bit[2] _out_s12 = {0,0};
  component_0(pkt_arrival0, pkt_last_time_pkt_id0_0, _out_s12);
  assert (_out_s16_0 == (_out_s12[1])); //Assert at flow.sk:190 (0)
}
/*flow.sk:180*/
```

output[0]

output[1]

Program Synthesis with Sharing

With the same specification, our method can synthesize component_0 such that output[0] is of depth 2, but output[1] is of depth 0, i.e. a single statement.

```
harness void sketch_component0(int pkt_arrival0, int pkt_last_time_pkt_id0_0) {

  S3 s = new S3();
  generator bit bool_vars() {
      return {| 0 | 1 | s.a | s.b |};
  }

  generator int vars() {
     return {| pkt_arrival0 | pkt_last_time_pkt_id0_0 |};
  }

  assert bool_expr(vars, bool_vars, 2, s) == component_0(pkt_arrival0, pkt_last_time_pkt_id0_0)[0];
  assert bool_expr(vars, bool_vars, 0, s) == component_0(pkt_arrival0, pkt_last_time_pkt_id0_0)[1];
```

Output:

```
void sketch_component0 (int pkt_arrival0, int pkt_last_time_pkt_id0_0)/*stat
{
  S3@ANONYMOUS s = new S3();
  tempVar@ANONYMOUS tmp = new tempVar();
  tmp.temp = pkt_arrival0 - 31;
  int _out_s163 = tmp.temp;
  tempVar@ANONYMOUS tmp_0 = new tempVar();
  tmp_0.temp = pkt_last_time_pkt_id0_0 + -28;
  int _out_s165 = tmp_0.temp;
  s.b = _out_s163 >= _out_s165;
  tempBitVar@ANONYMOUS tmp_1 = new tempBitVar();
  tempVar@ANONYMOUS tmp_2 = new tempVar();
  tmp_2.temp = -28 - pkt_arrival0;
  int _out_s197 = tmp_2.temp;
  tempVar@ANONYMOUS tmp_3 = new tempVar();
  tmp_3.temp = -31 - pkt_last_time_pkt_id0_0;
  int _out_s199 = tmp_3.temp;
  tmp_1.temp = _out_s197 < _out_s199;
  bit _out_s19 = tmp_1.temp;
  bit[2] _out_s11 = {0,0};
  component_0(pkt_arrival0, pkt_last_time_pkt_id0_0, _out_s11);
  assert (_out_s19 == (_out_s11[0])); //Assert at statefu..owlets.sk:151 (0)
  bit _out_s17 = s.b;
  bit[2] _out_s15 = {0,0};
  component_0(pkt_arrival0, pkt_last_time_pkt_id0_0, _out_s15);
  assert (_out_s17 == (_out_s15[1])); //Assert at statefu..owlets.sk:152 (0)
}
/*statefu..owlets.sk:132*/
```

output[0]

output[1]

Can the vanilla version generate the second output with depth 0?

```
harness void sketch_component0(int pkt_arrival0, int pkt_last_time_pkt_id0_0) {
    generator bit bool_vars() {
        return {| 0 | 1 |};
    }

    generator int vars() {
       return {| pkt_arrival0 | pkt_last_time_pkt_id0_0 |};
    }

    assert bool_expr(vars, bool_vars, 2) == component_0(pkt_arrival0, pkt_last_time_pkt_id0_0)[0];
    assert bool_expr(vars, bool_vars, 0) == component_0(pkt_arrival0, pkt_last_time_pkt_id0_0)[1];

}
```

Rejected!

```
*** Rejected
    [1606108779.0320 - ERROR] [SKETCH] Sketch Not Resolved Error:   UNSATISFIABLE ASSERTION

*** Rejected
The sketch could not be resolved.
```

Another example

```
bit[2] boolexpr_constraint(bit a, bit b, bit c) {
  bit d;
  bit e;
  d = (a & b) & c;
  e = (a & b) | c;
  return {d, e};
}

harness void boolexpr(bit a, bit b, bit c) {
  S3 s = new S3();
  generator bit bool_vars() {
    return {| s.a | s.b | s.c | a | b | c |};
  }
  generator int vars() { return 0; }
  assert bool_expr(vars, bool_vars, 2, s) == boolexpr_constraint(a, b, c)[0];
  assert bool_expr(vars, bool_vars, 1, s) == boolexpr_constraint(a, b, c)[1];
}
```

Output:

```
void boolexpr (bit a, bit b, bit c)/*abc.sk:147*/
{
  S3@ANONYMOUS s = new S3();
  s.b = c & a;
  bit _out_s39 = s.b;
  s.c = a & b;
  bit _out_s41 = s.c;
  s.b = _out_s39 & _out_s41;
  bit _out_s11 = s.b;
  bit[2] _out_s13 = {0,0};
  boolexpr_constraint(a, b, c, _out_s13);
  assert (_out_s11 == (_out_s13[0])); //Assert at abc.sk:153 (0)
  bit _out_s19 = s.c;
  s.a = _out_s19 | c;
  bit _out_s15 = s.a;
  bit[2] _out_s17 = {0,0};
  boolexpr_constraint(a, b, c, _out_s17);
  assert (_out_s15 == (_out_s17[1])); //Assert at abc.sk:154 (0)
}
/*abc.sk:147*/
```

reuse s.c

Next steps

- Do postprocessing using compiler techniques to further process the code generated by sketch: dead code elimination and common subexpression elimination.
- Experiment with another synthesis solver called DryadSynth and see if we can get a running time improvement.