MinSizeSketch: Minimum-Size Program Synthesis with Sketch[*]

Ruijie Fang <ruijief@princeton.edu>
Kexin Jin <kexinj@math.princeton.edu>

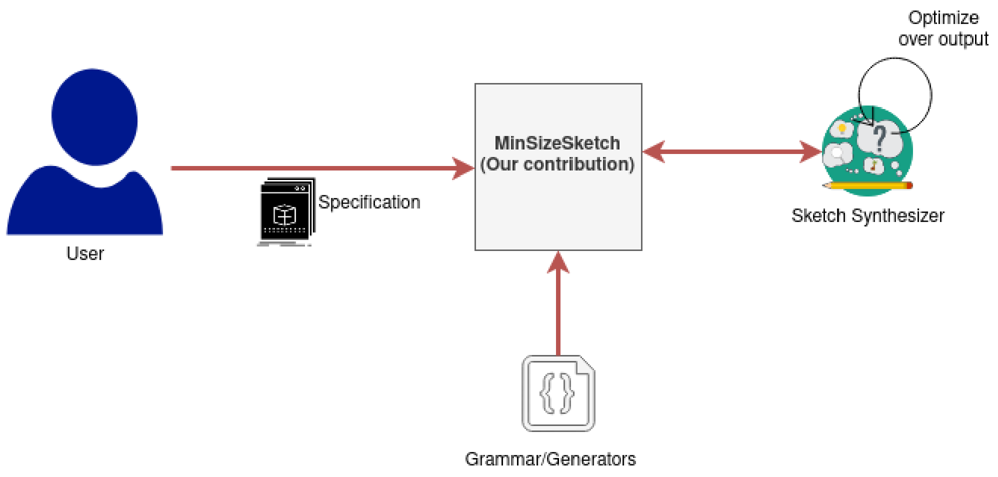
1 Introduction

The syntax-guided synthesis (SyGuS) problem, first defined by [1], is stated informally as follows: given a background theory T, a typed function f, a formula φ over the vocabulary of T and a context-free grammar, synthesize an expression according to the grammar for the function f such that the formula $\varphi[f/e]$ is valid modulo T. Relevant recent advances include applications of the problem to synthesizing programmable network switches [2] as well as improved synthesis solvers, such as Sketch and DryadSynth [4, 3].

The Sketch program synthesizer [4] is a well-known synthesizer for automatically generating complete programs from a given input program with "holes," which are unspecified variables that are unknown to the synthesizer. The input language of Sketch is flexible and allows the user the specify a wide range of inputs, including generator functions that enforce the Sketch output to conform to a given grammar. However, by default, Sketch does not provide capabilities to optimize over its output solution, although in many scenarios, minimizing the AST depth and the overall size of the output program is desired. Internally, Sketch is divided into the front-end, a compiler for the Sketch language that does a series of source-to-source code transformations, and the back-end, a CEGIS solver for QBF formulae.

We present MinSizeSketch, a tool based on the Sketch synthesizer front-end, for performing size-optimized multi-output program synthesis given a specification file and a target grammar. Our tool is a modification of the original Sketch front-end and open-sourced at `https://github.com/ruijiefang/sketch-frontend`. A compiled distribution, as well as documentation for installation and running may be found at `https://cs.princeton.edu/~ruijief/MinSizeSketch`. We also present a preliminary evaluation of the quality of output programs generated by MinSizeSketch and consider certain use-cases of our tool.

[*]This is a final project report for COS516 Fall 2020 at Princeton University, taught by Professor Aarti Gupta. We thank Professor Gupta and Divya Raghunathan for providing us with the project idea and helpful inputs. Both authors contributed equally to this project.



2 Overview

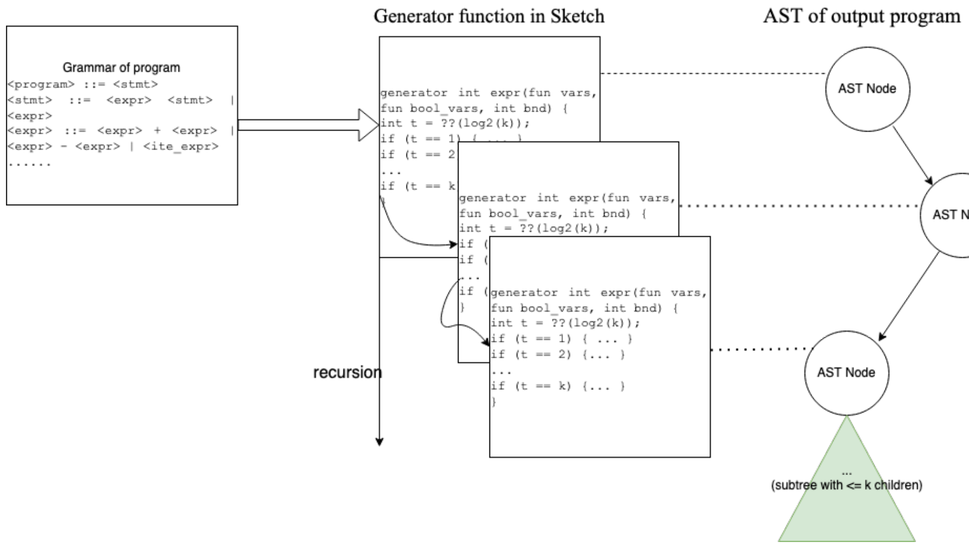
How Sketch works. The central concept in Sketch relies on holes, which are variable assignment statements with the right-hand side expression left unspecified, but replaced with a hole expression `??`. The synthesizer, supplied with a partial program, will attempt to fill in appropriate expressions for the holes present in the program, and output a complete program.

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

The Sketch synthesizer software has two parts: A front-end compiler that resembles a source-to-source compiler in the Sketch programming language, a Java-like high level language with OOP support, and a back-end solver that solves the QBF formula passed in by the front-end compiler in via a counterexample-guided synthesis (CEGIS) loop [4]. The final output program is then checked via an assert statement to be equivalent to the specification.

One potential pitfall of Sketch is the lack of optimization capabilities exposed to the user: the entire synthesizer functions as a black-box, and although the synthesizer might output a semantically correct solution, certain properties of the solution (such as the number of statements in the straight-line program or the depth of the program AST) may be not optimized.

MinSizeSketch: A high-level overview. The user supplies two files, both written in the Sketch language, to our program. For the purposes of this report, we call a target function that needs to be synthesized a *program component*. One file contains a series of specifications for each program component. Each specification is expressed as a Sketch function. The other file contains Sketch generator functions that encode the syntax of the target grammar. Our tool, MinSizeSketch, takes in both files and performs a series of static analysis passes over their abstract syntax tree (AST) representations to generate a final AST representation of a Sketch program, complete with harness functions for each component. The tool will then feed the Sketch program into the synthesizer and optimize for the depth of the output AST, as well as the number of temporary variables used.



```
<program> ::= <stmt>
<stmt>  ::=  <expr>  <stmt>  |
<expr>
<expr>  ::= <expr> + <expr>  |
<expr> - <expr> | <ite_expr>
......
```

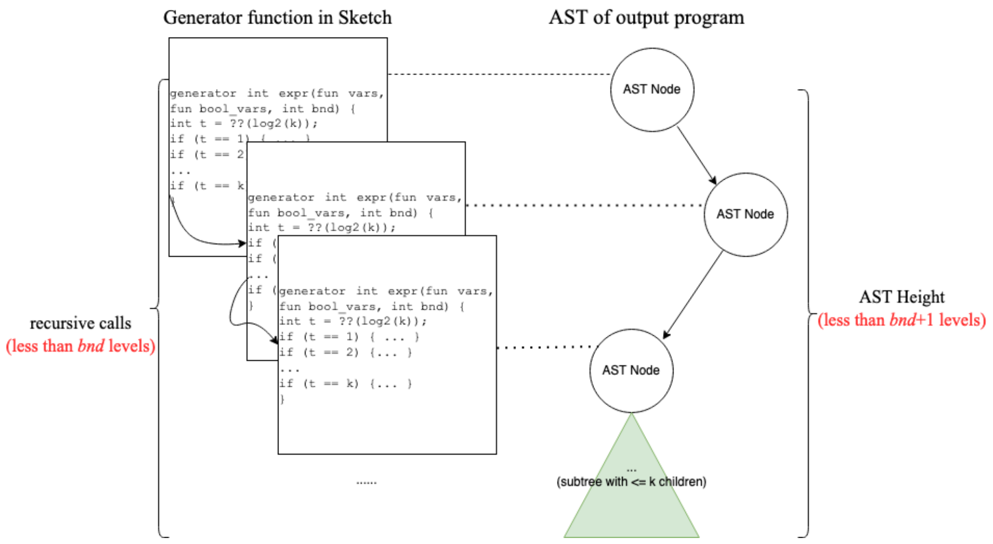
```
generator int expr(fun vars,
fun bool_vars, int bnd) {
int t = ??(log2(k));
if (t == 1) { ... }
if (t == 2) {... }
...
if (t == k) {... }
}
```

Figure **A**: Translating a context-free grammar into a Sketch generator function.

3 Minimum-Depth Synthesis in Sketch

Encoding of context-free grammar in the Sketch language. A typical Sketch input program may be described as containing several parts: A **harness** function containing assert statements that assert the semantic equality between a partial program (given as statements inside the harness and calls to other functions) on the left-hand side, and a **specification** function that on the right-hand side that contains semantic constraints of the final generated program. To encode context-free grammars in the Sketch language, we use **generators**, which could be reasoned as functions that can induce grammar-conforming expressions and statements that can then be used as the left-hand side of the assertion inside the Sketch harness. For a graphical illustration of this approach, refer to Figure **A**: Each nonterminal of the input grammar is translated into a mutually recursive generator function, which encodes the production rules for the nonterminal as conditional branches, with the conditional expression being a Sketch hole, allowing Sketch to freely choose between which production rule to use when searching for a specification-conforming output. Each recursion level of the generator function, then, corresponds to a vertex inside the abstract syntax tree representation of the output program.

Minimum-depth synthesis of single-output functions. Given the method of encoding grammar as Sketch generators above, we can require the AST of the output program to be of a certain depth k by augmenting the generator functions with an extra parameter indicating the recursion depth of the current recursion level. Since each recursion level coressponds to a vertex in the final output AST, bounding the recursion level inside generators is equivalent to bounding the depth of the output AST.



```
generator int expr(fun vars,
fun bool_vars, int bnd) {
int t = ??(log2(k));
if (t == 1) { ... }
if (t == 2) {... }
...
if (t == k) {... }
}
```

Figure **B**: The recursion level of the generator function corresponds to the height of the output AST.

Having solved the decision problem of deciding if a depth-k output AST exists, we can now solve the problem of outputing an abstract syntax tree representation of minimum-depth by searching over values of k and minimizing its value. Although in conventional settings, one would expect using binary search to be theoretically most effective to minimize the value of the depth parameter k, we found both exponential binary search and conventional binary search to be impractical in this setting: Both might give a very large recursion depth bound as input to Sketch in when trying to minimize the value of k, which is usually shallow (≤ 10 on small grammars) in the program synthesis setting, resulting in impractically long running times when invoking Sketch on these large recursion depths. Hence, we found simply doing a linear search and incrementally trying $k = 1, 2, \ldots$ until Sketch successfully performs the synthesis to be a better search strategy.

Synthesis of multi-output functions in Sketch. Multi-output functions can be understood in our setting as functions that return a fixed-size array of values. Their specification, for instance, would have a type signature similar to the following:

A trivial solution for a multi-output synthesis problem with d outputs is to write exactly d assertions in the harness function, thus generating a specification-conforming set of statements for each element of the output array. However, one could expect that in many settings, different elements of the output array might share common subexpressions. This trivial approach would not permit the syntehsizer to generate intermediate variables representing shared common subexpressions and reuse them during synthesis.

Minimum-depth synthesis of multi-output functions: A first approach. The problem of sharing common subexpressions between different elements of the output program can be reduced to the problem of making global, mutable state variables available to different invocations of the generator function: If one can supply to the generator function a list of temporary variables that can both be used as right-hand-side values and be assigned to as right-hand-side, then Sketch can choose to share variables during synthesis, and we simply have to minimize the AST height of the output program, just like in single-output synthesis.

In Sketch, there are to ways to share mutable variables: Having the harness function supply a `Struct` instance that contains all the mutable variables whose values would persist during different calls to the generator function, or an array reference, which is of the `array ref` type. Our tool, MinSizeSketch, uses the second approach: Given a generator function, it performs static analysis of the source code to put a global array containing the variables that could be used as storage for shared subexpressions into generator function calls and create a final harness.

The first approach to synthesizing multi-output functions with common subexpression sharing can be summarized as follows: We augment the input generator functions to take in an extra array reference, which contains possible intermediate variables for Sketch to use, and leave it to Sketch for deciding when to use non-shared temporary variables and when to use the shared temporary variables from the array reference. The output program can thus be viewed like a directed acyclic graph (DAG) instead of an AST: different outputs might share common, intermediate temporaries. For each output variable, we can minimize the depth of its corresponding DAG depth using the following greedy procedure:

1. Initialize an array of depths $D_1...D_d$ to 0 initially, with D_k being the depth for the k-th output variable. Maintain another array $O[1...d]$ of booleans with $O[i]$ set to `true` once the i-th variable is minimized.

2. If Sketch cannot find a solution, then uniformly increase $D_1...D_d$ by 1 and iterate this process until Sketch finds a solution.

3. Minimize the depth from the d-th variable backward to the first variable; at each stage, find $\arg\max\{i; O[i] = \texttt{false}\}$ and decrease its entry in D by 1. Mark $O[i] = \texttt{true}$ whenever $D[i] = 0$ or the change results in Sketch being unable to find a solution.

4. Terminate when $\forall i. O[i] = \texttt{true}$.

Note that this strategy only finds a local minimum over all possible minima of the array $D_1...D_d$; if, for instance, a larger value for an entry in the middle of D would result in much smaller values for other entries, the algorithm would fail to find this global minimum.

Minimum-depth synthesis of multi-output functions: A refined approach. In our first approach, we allowed Sketch to use an unlimited number of temporary variables that are unshared, while maintaining a fixed-size global array of shared variables between different outputs. In situations where the depth of the generated output program AST is required to be large, one tricky situation might happen: Say, if the minimum viable depth for the output program AST is 3, then both a straight-line program with 8 statements and a straight-line program with 3 statements (structured succinctly such that it resembles a directed path of length 3) would both qualify as outputs. However, in our empirical evaluation process, we found that the Sketch solver would prefer the lengthier output program among all candidate solutions of the same depth.

A solution to overcome this issue, both in single-output and multi-output synthesis, is to constrain Sketch to *only use shared temporaries inside the array reference.* Obviously, then, the size of the shared array matters: Should the size be too small no solution would be possible; in addition, inside the generator we must use a hole variable to select over different shared temporaries as right-hand side expressions; so a large array size would result in a blow-up in the number of possible branches to take in the generated AST, and would impact the running time of Sketch quite significantly.

However, since the program we synthesize are often small straight-line programs, the aforementioned solution could perform well in small test-cases, by having the user supply the size of the overall number of shared temporaries to use. In practice, we found that the number of shared temporaries rarely ever exceed 10; and in the case it exceeds 8, the running time of the synthesizer becomes impractical. Our tool, MinSizeSketch, utilizes this approach, and the user will be required to supply a parameter indicating the maximum number of shared temporaries used as a command line argument.

4 Empirical Evaluation

We evaluate the quality of output programs generated by MinSizeSketch on a sample language grammar supporting statements, arithmetic expressions with plus and minus, boolean expressions, and if-then-else (ITE) statements encoded as ternary expressions. There are two types in our sample language: `int` and bit, both are primitive types in Sketch. Our language grammar can be expressed in BNF notation as follows (for brevity, the statement structure is expressed in the expression itself):

$$\mathsf{lit} \rightarrow \texttt{[a-zA-Z0-9]}^*$$

$$\mathsf{lhs}_{\mathsf{int}} \rightarrow \texttt{int}\ \mathsf{lit}$$

$$\mathsf{lhs}_{\mathsf{bool}} \rightarrow \texttt{bit}\ \mathsf{lit}$$

$$\mathsf{relExpr} \rightarrow \mathsf{relExpr{<}relExpr} \mid \mathsf{relExpr{>}relExpr} \mid \mathsf{relExpr{=}relExpr} \mid \mathsf{lit} \mid z \in \mathbb{Z}$$

$$\mathsf{boolExpr} \rightarrow \mathsf{lhs}_{\mathsf{bool}}\mathsf{:=boolExpr;boolExpr} \mid \mathsf{boolExpr} \wedge \mathsf{boolExpr} \mid \mathsf{boolExpr} \vee \mathsf{boolExpr} \mid \mathsf{relExpr} \mid \mathsf{lit} \mid b \in \{0,1\}$$

$$\mathsf{expr} \rightarrow \mathsf{lhs}_{\mathsf{int}}\mathsf{:=expr;expr} \mid \mathsf{expr} + \mathsf{expr} \mid \mathsf{expr} - \mathsf{expr} \mid \texttt{ite}(\mathsf{boolExpr}, \mathsf{expr}, \mathsf{expr}) \mid \mathsf{lit} \mid z \in \mathbb{Z}$$

See Figure **C** for a graphical illustration of the above grammar. Programs generated by our grammar will be straight-line programs with if/else statements.

We show, in detail below, the quality of MinSizeSketch output evaluated on a selected subset of programs included in the distribution tarball and real-world workloads. On average, when the output size is measured as the size of the Sketch-generated output, our

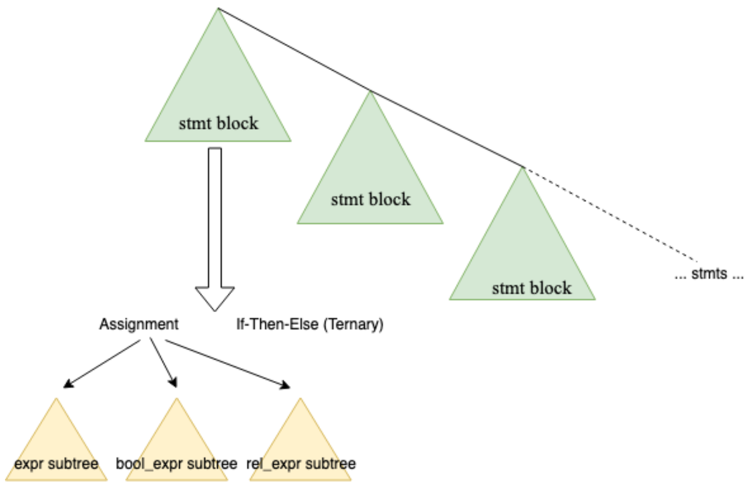


Figure **C**: A graphical illustration of our grammar.

tool achieves a 10 percent code size reduction.

Example Programs in the Distribution Tarball. Our first example, which is given as `simpleSpec.sk` in the program distribution, is as follows:

```
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};
}
```

One can see that the code involves sharing of the common subexpression $a \wedge b$ in the two outputs.

Our program, MinSizeSketch, when run with the maximum number of shared variables set to 3, generates the following output:

```
 { memoBit__ANONYMOUS_s152[0] = b &
    a;
  bit _out_s18 =
     memoBit__ANONYMOUS_s152[0];
  memoBit__ANONYMOUS_s152[1] =
     _out_s18 & c;
  bit[2] _out_s8 = {0,0};
  boolexpr_constraint(a, b, c,
     _out_s8);
  assert ((memoBit__ANONYMOUS_s152
     [1]) == (_out_s8[0]));
  bit _out_s14 =
     memoBit__ANONYMOUS_s152[0];
  memoBit__ANONYMOUS_s152[1] =
     _out_s14 | c;
  bit[2] _out_s12 = {0,0};
  boolexpr_constraint(a, b, c,
     _out_s12);
  assert ((memoBit__ANONYMOUS_s152
     [1]) == (_out_s12[1])); }
```

As one can see by examining the output, the statements before the second assertion statement on the last line reused the variable `__out_s14`, which represents the shared expression $a \wedge b$. Here, MinSizeSketch manages to generate an output that achieves maximal sharing of common subexpressions.

Our second example, which is given as `simpleSpec2.sk` in the program distribution, is as follows:

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

One can see both outputs `d, e` share the common subexpression $a \wedge b$, although now the expression to yield d is a bit more complicated. Invoking MinSizeSketch with the max number of shared variables set to 3, we obtain the following (truncated) output:

```
{
  _out = ((bit[2])0);
  memoBit_s152[2] = a & b;
  bit _out_s18 = memoBit_s152[2];
  memoBit_s152[2] = _out_s18 & c;
  bit[2] _out_s8 = {0,0};
  boolexpr_constraint(a, b, c,
     _out_s8);
  assert ((memoBit_s152[2]) == (
     _out_s8[0]));
  bit _out_s14 = memoBit_s152[2];
  memoBit_s152[2] = _out_s14;
  bit[2] _out_s12 = {0,0};
  boolexpr_constraint(a, b, c,
     _out_s12);
  assert ((memoBit_s152[2]) == (
     _out_s12[1]));
}
```

Here one can see that the output also achieves sharing of common subexpressions; in addition, MinSizeSketch optimized the depth of the first element to 2, and the depth of the second element to 1.

MinSizeSketch Output versus Vanilla Sketch Output. We will now show a real-world example where our tool achieves a 20% code size reduction versus the vanilla output of Sketch. We are given the following specification, which arises from the specification of network switches; it has two outputs, and the two outputs share a common subexpression of the type `int`.

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

The vanilla output without sharing generated by Sketch is below:

```
void sketch_component0 (int
    pkt_arrival0, int
    pkt_last_time_pkt_id0_0)
{
  tempBitVar tmp = new tempBitVar();
  tempVar tmp_0 = new tempVar();
  tmp_0.temp =
     pkt_last_time_pkt_id0_0 + -7;
  int _out_s113 = tmp_0.temp;
  tempVar tmp_1 = new tempVar();
  tmp_1.temp = -10 + pkt_arrival0;
  int _out_s115 = tmp_1.temp;
  tmp.temp = _out_s113 < _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 tmp_2 = new tempBitVar
     ();
  tempVar tmp_3 = new tempVar();
  tmp_3.temp =
     pkt_last_time_pkt_id0_0 -
     pkt_arrival0;
  int _out_s119 = tmp_3.temp;
  tmp_2.temp = -3 >= _out_s119;
  bit _out_s16_0 = tmp_2.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)
}
```

The output generated by our tool is below:

```
void sketch_component0 (int
    pkt_arrival0, int
    pkt_last_time_pkt_id0_0)
{
  S3 s = new S3();
tempVar tmp = new tempVar();
tmp.temp = pkt_last_time_pkt_id0_0
    + 3;
int _out_s142 = tmp.temp;
s.a = pkt_arrival0 >= _out_s142;
tempBitVar tmp_0 = new tempBitVar
   ();
tempVar tmp_1 = new tempVar();
tmp_1.temp = -3 + pkt_arrival0;
int _out_s195 = tmp_1.temp;
tmp_0.temp =
   pkt_last_time_pkt_id0_0 <
   _out_s195;
bit _out_s19 = tmp_0.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]))
   ;

bit _out_s17 = s.a;
bit[2] _out_s15 = {0,0};
component_0(pkt_arrival0,
   pkt_last_time_pkt_id0_0,
   _out_s15);
assert (_out_s17 == (_out_s15[1]))
   ;
}
```

Measuring the size of output by the number of characters in the program, the code size reduction is roughly 20%.

5 Conclusion

We presented a tool, MinSizeSketch, based on the Sketch synthesizer front-end to perform multi-output program synthesis in Sketch while minimizing the size as well as AST depth of the output program. On a small, manually crafted set of input examples, its output achieved sharing of common subexpressions between different output variables, and its output solution is shorter than vanilla Sketch outputs. One possible future application of our tool would be in the area of synthesis for network switches [2], where a size-minimized output solution is often desired for the output program to be fitted into specialized hardware.

Future Work. We hope to further investigate the applications of our tool in the future, as well as improve the quality of the code generation process in our tool. We also plan to investigate the performance of our tool on more real-world workloads, as well as improve it to take in other input formats such as the SyGuS-Lib format [1].

References

[1] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD 2013*.

[2] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta. Switch Code Generation Using Program Synthesis. In *ACM SIGCOMM 2020*.

[3] K. Huang, X. Qiu, P. Shen, and Y. Wang. Reconciling enumerative and deductive program synthesis. In *ACM SIGPLAN 2020*.

[4] A. Solar-Lezama. Program synthesis by sketching. Ph.D. dissertation, University of California, Berkeley. 2008.