Combinatorial Sketching for Finite Programs

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat*, Sanjit Seshia

UC Berkeley
{asolar,tancau,bodik,sseshia}@eecs.berkeley.edu

*IBM T.J. Watson Research Center
vsaraswa@us.ibm.com

Abstract

Sketching is a software synthesis approach where the programmer develops a partial implementation — a sketch — and a separate specification of the desired functionality. The synthesizer then completes the sketch to behave like the specification. The correctness of the synthesized implementation is guaranteed by the compiler, which allows, among other benefits, rapid development of highly tuned implementations without the fear of introducing bugs.

We develop SKETCH, a language for finite programs with linguistic support for sketching. Finite programs include many high-performance kernels, including cryptocodes. In contrast to prior synthesizers, which had to be equipped with domain-specific rules, SKETCH completes sketches by means of a combinatorial search based on generalized boolean satisfiability. Consequently, our combinatorial synthesizer is complete for the class of finite programs: it is guaranteed to complete any sketch in theory, and in practice has scaled to realistic programming problems.

Freed from domain rules, we can now write sketches as simple-to-understand partial programs, which are regular programs in which difficult code fragments are replaced with *holes* to be filled by the synthesizer. Holes may stand for index expressions, lookup tables, or bitmasks, but the programmer can easily define new kinds of holes using a single versatile synthesis operator.

We have used SKETCH to synthesize an efficient implementation of the AES cipher standard. The synthesizer produces the most complex part of the implementation and runs in about an hour.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Software Architectures, Design Tools and Techniques; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design, Performance

Keywords Sketching, SAT

1. Introduction

When programming by sketching, the programmer develops only a skeleton of the desired implementation, called a *sketch*, and a synthesizer completes the sketch such that it meets a separate specification of the desired behavior. Sketching encourages clean specifications because it relies on specification only to define the functionality. Sketching also promises to enable complex implementations that may be too tedious to develop and maintain without automatic synthesis of low-level detail.

The concept of sketching was introduced in StreamBit, a system for bit-stream programming [13]. In StreamBit, sketching proved to be very effective: in a single afternoon, a sketching programmer implemented a DES cipher that nearly matched the performance of the best public-domain DES implementation. In another experiment, a sketched implementation of a cipher was produced twice as a fast as a C implementation, and ran 50% faster.

Unfortunately, this productivity came at the expense of significant training: Only one of the authors of [13] was able to write non-trivial sketches. The programmability problem stems from StreamBit's reliance on *transformational* synthesis, wherein the specification is transformed into the desired implementation with domain-specific rewrite rules. The transformational setting in StreamBit complicates programming because sketches cannot be expressed directly as partial implementations but instead must be given as partial rewrite rules; the sketching language can hide this fact only partly. Furthermore, in StreamBit, a desired implementation often has to be painfully decomposed into a hierarchy of sketches, one for each rewrite rule. Section 7 elaborates, but to summarize, the awareness of the rewrite rules made programming with StreamBit's sketching challenging.

To address these limitations, this paper develops a *combinatorial* synthesizer. Its key feature is how it synthesizes a correct implementation, i.e., an implementation that is functionally equivalent to the specification. While a transformational synthesizer generates correct implementations by transforming the specification with semantics-preserving domain rules, combinatorial synthesizer relies on a verifier. The verifier "filters out" incorrect implementations, which allows the synthesizer to enumerate all possible candidate implementations, not just those derivable with given domain rules. Sketching thus allows us to use the verifier not just to prove that the program is correct, but to help us write it, by searching the space of sketch completions.

The first benefit of combinatorial synthesis is *completeness*: we can both *specify* any finite program and *sketch* any implementation of it. A finite program is one whose input is bounded and which terminates on all inputs after a bounded number of operations. Many high-performance kernels have this property. On the contrary, systems based on domain-specific rewrite rules are often incomplete [2, 6, 8, 9]. Furthermore, the rules may not easily reveal which implementations they are able to generate.

The second benefit is that we can express sketches as genuine partial programs, i.e., programs with "holes" that will be filled by the synthesizer. Programmers use holes to ask the synthesizer to insert various *kinds* of hard-to-write code fragments, such as index expressions, more general polynomial functions, constants, bit-masks, lookup-table content, or expressions dividing work in divide-and-conquer algorithms. Further kinds of holes can be defined by the programmer. Sketching with holes is supported by the combinatorial synthesizer as it searches over all possible "hole

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

completions," respecting given constraints on the kind of the hole, until it finds a correct implementation.

We develop a programming language SKETCH for sketching implementations of finite programs. Our language design is guided by two goals. First, we seek a language that makes sketches easy to write. In SKETCH, sketches look like hand-written implementations, except that the code fragments to be synthesized are replaced with holes. The idea is that if a programmer would know how to write the final program, he should also know how to sketch it since the sketch is merely an incomplete implementation. We hope that the programmer thus need not change his mindset and instead simply think of the synthesizer as a "hole-filling" assistant.

Our second goal is to provide an expressive but small set of language constructs to specify various kinds of holes. We support a single synthesis operator, denoted **??**. The operator represents the elementary hole: it is replaced by the synthesizer with a suitably chosen constant. We show how richer kinds of holes can be easily built from this versatile operator.

The SKETCH language is supported by a new synthesis algorithm that is complete in that it can complete an arbitrary sketch. The algorithm reduces the synthesis problem to a quantified boolean satisfiability (QBF) problem with one quantifier alternation. Our solver for this problem uses a counterexample-driven iteration over a synthesize-verify loop built from two communicating SAT solvers. We show that although the problem is harder than NP-complete, the counterexample-driven search terminates on real problems after solving only a few SAT instances.

We present an empirical evaluation of our system. We sketched an implementation of the AES cipher, the current block cipher standard. We show that the sketch describes a high-performance implementation of the cipher very concisely, and that our solver produces the complex implementation in a scalable way, in about an hour. We also implemented a few kernels that are smaller in size but stress test the solver's synthesis abilities.

In summary, this paper makes the following contributions:

- We develop the first combinatorial code synthesizer. The synthesizer works in the context of sketching. The synthesizer is complete in that it can complete all sketches in theory; in practice, it scales well for realistic problems. It scales surprisingly well also for some hard problems that we expected to be beyond its limits, such as synthesis of polynomial expressions.
- We develop a language for sketching implementations of finite programs. Sketches are expressed as partial programs with holes, which can be of various kinds. A single, versatile synthesis operator can be used to synthesize various code fragments to insert into the hole.
- We implemented the AES cipher and showed the scalability of the solver for real problems.

Section 2 presents an example and Section 3 presents a tutorial on programming with sketches. Section 4 defines the language formally. Section 5 describes the synthesizer. Section 6 evaluates the solver and describes our programming experience. Section 7 discusses related work.

2. Example

To illustrate sketching, we use an example from [13]. The problem at hand is to efficiently implement the *IP* permutation from the DES cipher. Such bit-level permutations form important building blocks of block ciphers. The *IP* permutation is shown in Figure 1(a); the corresponding specification in the SKETCH language is given here:

```
bit[64] IP (bit[64] x) {
    int[64] P = { 63, 3, ... };
    bit[64] ret = 0;
    for (int i=0; i < 64; i++)
        if (x[i]) ret[P[x]] = 1;
    return ret;
}
```

When implementing a bit permutation, the programmer can exploit a spectrum of solutions. These include implementing the permutation as a sequence of shifts and masks, or as a sequence of table lookups whose results are *or*ed together. However, the *IP* permutation is too irregular to permit a small number of shifts operations, and requires too much space when implemented with tables, so the programmer must resort to more clever optimizations.

Let us now show how sketching helps in developing the tuned implementation. First, the programmer may notice that the permutation has certain regularity. Specifically, it turns out it is possible to first perform a simple permutation, implementable with a few shift operations, after which one performs two identical permutations of half width; this would reduce the table storage four-fold. The clever implementation is shown in Figure 1(b). The sketch for this implementation is shown below.

```
bit[64] IPsketched (bit[64] x) implements IP {
   bit[64] result;
   bit[32] table[8][16] = ??;
   x = permute(x, 2);
   for (int i=0; i<8; ++i) {
      result[0:31]  |= table[i][x[i*4::4]];
      result[32:63] |= table[i][x[32+i*4::4]];
   }
   return result;
}
```

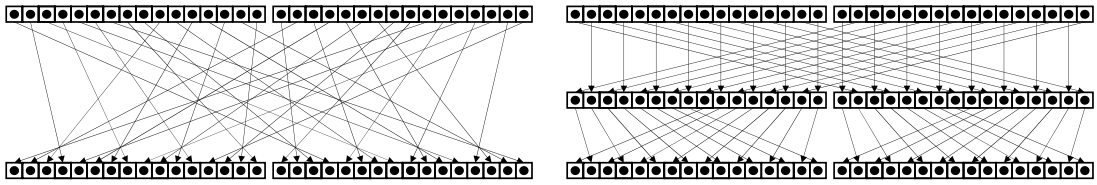
The sketch contains two holes, in the arguably hardest parts of the program. The first hole initializes the table `table`; the content of the table will be synthesized, so the programmer need not write (and debug) a script for filling the table, as is common today. The second hole (`permute(x,2)`) will be replaced with a shift-and-or operation sequence that will suitably permute the bits in `x`. We show how this hole is defined in the next section; for now it suffices to say that the argument `2` limits the number of shift operations that will be synthesized.

Note that the sketch looks like an ordinary program with hard parts omitted in favor of holes. All constraints needed to complete the sketch are given in the code. For example, the loop encodes the requirement that the second permutation is composed of two identical, half-sized permutations.

3. Programming with Sketches

The SKETCH language is a procedural language with no pointers but with support for sketching. The language is targeted towards integer kernels over finite inputs. The features of SKETCH have been selected so that the language can express any *finite program*. A program is finite if (i) its input is bounded and (ii) the program terminates on all inputs. For example, a matrix multiplication over matrices of sizes known at compile time is a finite program, but a search on an arbitrary binary tree is not. Important for our work is that finite programs can be viewed as boolean functions that map a vector of input bits to a vector of output bits. (Note that this functional view does not preclude finite programs from implementing an internal finite state machine, if that's what the programmer desires.)

This section gives a tutorial of the SKETCH language, going from simple to more complex kernels. All sketches in this section



(a) (b)

Figure 1. (a) A truncated version of the DES *IP* permutation. (b) Same permutation decomposed into a simple permutation implementable with shifts, followed by two identical permutations. The two permutations can be implemented using a single table, reducing table storage four-times.

are beyond the power of [13], but can be completed by our new compiler for reasonable word-sizes.

Isolate Rightmost Bit The following example is simple, but it already benefits from the ability of the SKETCH language to verify and sketch implementations.

The problem at hand is to isolate the rightmost 0-bit, if there is any. For example, given a word `1010 0111`, we return the bit mask `0000 1000`. The function `isolate0` below is a straightforward specification of the task. Like a good specification, the function is readable at the cost of efficiency.

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

Like `isolate0`, each SKETCH specification is executable and can be invoked by clients until a better implementation of the specification is developed.

The function `isolate0Fast` is such a better implementation; it exploits bit-vector parallelism by relying on a little bit of algebra [15]. (The SKETCH language support arrays and vector operations are provided for arrays of bits, to give access to bitwise integer machine instructions.)

```
bit[W] isolate0Fast (bit[W] x) implements isolate0 {
   return ~x & (x+1);
}
```

The implementation achieves performance at the expense of clarity. While the correctness of this implementation is not immediately obvious, the keyword **`implements`** insists that the function `isolate0Fast` implements specification `isolate0`. The equivalence of the two functions is verified by the compiler, which guarantees that if i **`implements`** s, implementation i must produce the same output as the specification s on all inputs, and thus be free of all bugs.

The main contribution of SKETCH, however, is not the ability to verify but the power to synthesize an implementation from a sketch. The function `isolate0Sketched` illustrates a sketch. The "holes" in the sketch are indicated by the **`??`** operators. These operators will be replaced with a value, in this example, a bit vector.

```
bit[W] isolate0Sketched(bit[W] x) implements isolate0{
   return ~(x + ??) & (x + ??);
}
```

In `isolate0Sketched`, the first **`??`** will be synthesized to the value `0`, while the second one will be synthesized to the value `1`.

In addition to sparing the user from having to derive some of the low level details of the implementation, the **`??`** operator also makes the sketches more reusable. For example, the user can excise the sketch above into a separate function and use it to produce an implementation not just for `isolate0`, but also for the dual problem of isolating the rightmost 1-bit, specified by `isolate1`, whose code we do not show.

```
bit[W] expression (bit[W] x) {
   return ~(x + ??) & (x + ??);
}

bit[w] isolate0Sketched (bit[W] x) implements isolate0 {
   return expression(x);
}

bit[w] isolate1Sketched (bit[W] x) implements isolate1 {
   return expression(x);
}
```

We have two call sites of `expression`: In the first, the synthesizer completes `expression` to `~x & (x + 1)`; in the second to `~(x - 1) & x`. The semantics of calling a function in SKETCH is thus that of cloning, which can be implemented by inlining the function into the call site. `expression` alone is an *unrestricted* sketch, one that is not asked to implement a particular specification; such a sketch can be thought of having many different behaviors from which the synthesizer must select one when the sketch is bound.

Population Count We now show how sketching can be used to synthesize a tricky divide-and-conquer algorithm. The problem at hand is to compute the population count of 1-bits in a word. The obvious specification is here:

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

An efficient implementation uses a divide and conquer strategy, in which the original problem of summing k bits is divided into two problems of summing $k/2$ bits, and so on recursively. After the subproblems are solved, their results are added [12, 15].

The key to efficiency is solving the smaller problems (of the same size) all in parallel, SIMD-style. Let us illustrate on the

smallest problem: we want to sum the number of 1-bits in the 0th bit (the sum is either 0 or 1) with the number of bits in the 1st bit, and store the result in these two bits; the same for all adjacent pairs of bits. To perform these sums simultaneously with a single addition instruction, the programmer must make the instruction mimic SIMD semantics: even and odd bits must be aligned by shifting and suitable bit masks must prevent the propagation of the carry bit across the pairs of bits. The same must be accomplished for the larger subproblems, only with different shift amounts and bitmasks.

With sketching, writing the algorithm is easy. The SKETCH compiler synthesizes the loop bound and the suitable masks and shift amounts for each iteration of the loop.

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

Notice that the sketch does not spell out details of the "divide" strategy: in particular, the desire to divide the problem recursively in two equal halves is not made explicit. Also note that the holes corresponding to the masks and shift amounts will get different values on each iteration. This is because when the sketch is completed, the loop construct will be unrolled before the holes in its body are replaced with values (the details of this are explained in section 4). For word size $\mathtt{W} = 16$, the loop is unrolled 4 times. The synthesized code is shown below.

```
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;
```

Because the sketch offers a lot of freedom, the synthesized code is not identical to the textbook version — which shifts in the expected sequence $(1, 2, 4, 8)$ rather than in $(1, 2, 8, 4)$ — but the algorithm behaves as desired and is equally efficient.

Another implementation, suitable when the word is populated sparsely, is to keep resetting the rightmost 1-bit until the word is zero [16]. The sketch below accomplishes this by invoking the sketched function `expression`, which we previously used to synthesize an implementation of `isolate0` and also of `isolate1`.

```
int popSparseSketched (bit[W] in) implements pop {
   int ret;
   for (ret = 0; in; ret++) { in &= ~expression(in); }
   return ret;
}
```

Notice that `expression` will be completed to isolate the rightmost 1-bit even though the synthesizer is not instructed to do so; the only constraint given to the synthesizer is that `popSparseSketched` must implement `pop`.

Richer holes It should now be easy to see how the programmer can define "richer holes," i.e., holes that will be replaced not with a constant but with a synthesized expressions or sequence of statements. We start with a *bit permutation hole*, which we used in Section 2 to synthesize a bit permutation implemented with a sequence of shift operations. The *generator* of this kind of hole is given here (assume that `x>>y` with a negative y shifts x to the left):

```
bit[N] permute(bit[N] x, int count) {
   bit[N] result;
   loop (count) result ^= x>>?? & ??;
   return result;
}
```

Another kind of hole is *polynomial*. Synthesizing a polynomial expression comes handy when the implementations involves a complicated array index expression. This example also illustrates an important pattern where recursive functions are used to define complex expressions. We illustrate the polynomial generator `poly` using a contrived example that divides two polynomials; the hole is replaced with the result of the division:

```
int spec (int x) {
   return x*x*x*x + 6*x*x*x + 11*x*x + 6*x;
}
int p (int x) implements spec {
   return (x+1)*(x+2)*poly(3,x);
}
int poly(int n, int x) {
   if (n==0) return ??;
   else return x * poly(n-1, x) + ??;
}
```

The recursive calls to the sketch `poly` will be inlined and the holes are completed to produce a function shown below, which will in turn be inlined into `p`.

```
int poly(int n, int x) {
    int rv1;
      int rv2;
        int rv3;
        rv3 = 0;
      rv2 = x * rv3 + 1;
    rv1 =  x * rv2 + 3;
  return x * rv1 + 0;
}
```

It is easy to see that with a little bit of constant folding, the function above will turn into $(x * (x + 3))$, which is the desired expression.

In Section 2 we have also shown a hole that initialized a lookup table. This hole does not need a generator, but the programmers will probably consider it a "richer hole."

Using Rich Holes (Karatsuba Multiplication) The Karatsuba multiplication algorithm is used to multiply large integers. Its complexity is $O(N^{1.585})$, as opposed to $O(N^2)$ for the standard long multiplication algorithm, and it is the building block of many public key ciphers. The algorithm is recursive, and is generally used on integers of unbounded size. For this example, we will assume integers of a small fixed size, but it is worth pointing out that although the synthesized code will have been proven correct only for this small size, it will actually work for arbitrary input sizes. Proving this mechanically in general will be the subject of future work, but for now, the algorithm will serve as an example of the use of rich holes to express complex code patterns.

The algorithm uses a divide-and-conquer approach. We are given two N-digit numbers x and y which we split in half bitwise: $x = x_1 b + x_0$, $y = y_1 b + y_0$, where $b = 2^k$ is a base. The standard multiplication can be defined in this fashion as shown here.

$$x * y = b^2 x_1 * y_1 + b(x_1 * y_0 + x_0 * y_1) + x_0 * y_0$$

```
bit[N*2] k<int N>(bit[N] x, bit[N] y) implements mult {
   if (N<=1) return x*y;

   sbit[N/2] x1 = x[0:N/2-1]; sbit[N/2] x2=x[N/2:N-1];
   sbit[N/2] y1 = y[0:N/2-1]; sbit[N/2] y2=y[N/2:N-1];

   sbit[2*N]  t11 = x1 * y1;
   sbit[2*N]  t12 = poly(1,x1,x2,y1,y2)*poly(1,x1,x2,y1,y2);
   sbit[2*N]  t22 = x2 * y2;

   return multPolySparse<2*N>(2, N/2, t11)
        + multPolySparse<2*N>(2, N/2, t12)
        + multPolySparse<2*N>(2, N/2, t22);
}
bit[2*N] poly<int N>(int n, sbit[N] x0, x1, x2, x3) {
   if (n<=0) return ??;
   else return  (??*x0 + ??*x1 + ??*x2 + ??*x3)
              * poly<N>(n-1, x0, x1, x2, x3);
}
bit[2*N] multPolySparse<int N>(int n, int x, sbit[N] y) {
   if (n<=0) return 0;
   else return  y<<x*?? + multPolySparse<N>(n-1, x, y);
}
```

Figure 2. Sketch for Karatsuba's multiplication. The type sbit[N] stands for a signed bit-vector, and includes a sign bit in addition to its N bits.

We denote the expensive (big-integer) multiplication with the $*$ operator. The multiplication with the base terms is implemented with shifts, so it is not an expensive operation.

Let us illustrate how Karatsuba might have been able to invent (and implement) his algorithm with the assistance from sketching. He would first observe that it may be possible to replace the four expensive multiplications with three expensive multiplications. He would guess that one cannot avoid computing terms $x_0 * y_0$ and $x_1 * y_1$, so he would focus on replacing the term $x_1 * y_0 + x_0 * y_1$ with a one-multiplication term. This optimization would be performed at the expense of adding big-integer additions or subtractions, a good trade-off. To express his idea, he would write the following sketch, expressed mathematically. Note that we ask to synthesize new base terms, too.

$$\begin{aligned} x * y = & \; poly(??, b) * (x_1 * y_1) \\ & + poly(??, b) * (poly(1, x_1, x_0, y_1, y_0) * poly(1, x_1, x_0, y_1, y_0)) \\ & + poly(??, b) * (x_0 * y_0) \end{aligned}$$

It turns out that the idea for this optimization is correct and the algorithm is synthesized as follows.

$$\begin{aligned} x * y = & \; (b^2 + b) * (x_1 * y_1) \\ & + b * ((x_1 - x_0) * (y_1 - y_0)) \\ & + (b + 1) * (x_0 * y_0) \end{aligned}$$

The sketch given the in the SKETCH language, shown in Figure 2, is only a little more complex than the clean mathematical sketch. The key reason is that we make explicit the fact that multiplications with the base terms are performed with shifts. Our system currently completes and verifies the sketch for $N = 12$. The resulting program is correct for all N, though, as was said before, it is up to the programmer to verify this since our system can only make claims about the finite version of the program it was asked to verify.

4. Language Definition

This section defines the meaning of a sketch, by which we mean the set of functions that the sketch can be made to compute. We give the meaning of a sketch through a partial evaluation procedure that non-deterministically transforms a given sketch into any of its possible completions. Functions computable by a sketch are thus defined via programs that can be produced from it. We first describe the meaning of sketches in a general, language-independent way, and then make the definition specific to the SKETCH language.

A sketch is any program containing holes. The meaning of a sketch p is the set of programs $C(p)$ that can be obtained by completing p. The set of completions $C(p)$ is computed by evaluating p with a semantics-preserving partial evaluator (PE) that evaluates the hole **??** to a non-deterministically selected constant. The PE thus non-deterministically produces all possible completions of a sketch. An *s-correct* completion of a sketch p is a program $p' \in C(p)$ that is behaviorally equivalent to a specification s. A sketching compiler for a language L is a program that takes as input a sketch p in L, a specification s in L and produces an s-correct completion of p.

While the above definition permits any semantics-preserving PE, the choice of PE determines the semantics of holes in terms of how many different values a syntactic instance h of a hole is allowed to generate. For example, if we desire an extreme semantics in which h always evaluates to the same value, we would use a PE that never duplicates a program fragment (and hence does not unroll loops or inline functions). Such a PE syntactically replaces, in the original sketch, each instance of h with a literal. On the other hand, if we desire the ability to define rich holes by means of recursive functions, as was done in Section 3, we need a more powerful semantics for holes: we want the PE to inline functions, so that multiple copies of a hole can be created, allowing each to be completed to a different value.

The partial evaluator underlying the SKETCH language unrolls all **loop**-loops, does *not* unroll **while**-loops, and inlines all function calls, except for calls to functions with the **implements** clause (we call these functions *restricted* sketches). We arrived at this design after expressing our benchmarks under various semantics and observing that this choice led to the most natural sketches. (The rationale for not inlining restricted sketches can be seen in the Karatsuba Multiplication example in Section 3, where we want each recursive invocation of the sketch to completed the same way; i.e., we want to synthesize only one copy of the sketch.)

We now elaborate on the PE in the SKETCH language. We do so with a small imperative language extended with the sketching constructs found in SKETCH. The sketch-free subset of SKETCH is a standard imperative language over booleans and integers. It includes a counting loop **loop** e **do** c which evaluates the integer expression e once and executes c exactly e times. A function `f` returns a value by assigning to the output variable `f`, named after the function. The abstract syntax for the language is shown below; it does not include all SKETCH constructs; in particular it omits arrays, **for** loops, and functions with multiple arguments. They are omitted because their semantics is standard even in the presence of sketching operators, although it is important to point out that **for** loops are handled like **while** loops.

expressions	e	::=	n | **true** | **false** | x | e **op** e | $f(e)$
statements	c	::=	**let** x = e **in** c | x := e | **skip** | **if** e **then** c **else** c | c ; c | **while** e **do** c | **loop** e **do** c
functions	f	::=	**def** f(x) c
programs	p	::=	p f | f

Sketching is enabled with two constructs:

expressions	e	::=	**??**
functions	f	::=	**def** f(x) **implements** g c

We define the partial evaluator with rewrite rules. The following notation applies to statements.

$$\langle c, \sigma\rangle \rightarrow \langle c', \sigma'\rangle$$

The rule states that, given a state σ, a statement c is partially evaluated to a residual statement c'. The state σ maintains values of variables in the spirit of constant propagation: variables found to be constant are mapped to their constant values; non-constant variables are mapped to a special value $\perp$. Partial evaluation of c under the initial state σ produces a modified state σ'. The special state $\sigma_\perp$ maps all variables to $\perp$. Given two different values v_1 and v_2, we define $v_1 \cap v_1 = v_1$, $v_1 \cap v_2 = \perp$, and $v_1 \cap \perp = \perp$. We define $\sigma_1 \cap \sigma_2 = [x \mapsto \sigma_1(x) \cap \sigma_2(x) \mid x \in \sigma_1 \cap \sigma_2]$.

The following notation applies to expressions.

$$\langle e, \sigma\rangle \rightarrow \langle v, e'\rangle$$

Partial evaluation of e in state σ returns its residual expression e' as well as a value v. When e evaluates to a constant n, then $v = n$; otherwise, $v = \perp$. Expressions are free of side effects, so they do not modify σ.

The partial evaluator rules are all shown in Figure 3, except for some symmetric and the obvious rules. To understand the partial evaluator, it is useful to start with rules E-Var, E-Op1, and E-Op2. We see that a non-constant variable x is evaluated to $\perp$; it produces code fragment x. A constant variable is evaluated to a constant n; it generates the literal n. The rules for **op** evaluate the operator if both operands are constants. Otherwise, the residual expression is generated.

The key non-deterministic rule is E-hole, where the code generated from a hole is a non-deterministically chosen literal. Together with the non-deterministic rules for loop unrolling and function call inlining, discussed below, this rule makes the evaluator produce all sketch completions.

Unrolling of counting loops deserves special attention. When the iteration-count expression e is a constant n, the loop body is replicated n times (S-Loop1). Otherwise, the loop is unrolled with a loop-exit test before each iteration (S-Loop2). The unroll factor is determined speculatively: the synthesizer unrolls the loop by some amount, terminating unrolling with rule S-Loop3 (formally, there is a non-deterministic choice between S-Loop2 and S-Loop3). In order to determine if the loop was unrolled sufficiently, an assertion is inserted by S-Loop3. This assertion is statically verified by the synthesizer described in the next section. If the assertion may fail on some input to the completed sketch, the loop needs to be unrolled more. In this case, the partial evaluation process is repeated with a larger unrolling factor.

Inlining of function calls is handled with four rules. When the called function evaluates to a constant, the function call disappears (E-Call1a). Otherwise, we inline the residual code of the function body (E-Call1b). In the latter rule, we model inlining as function cloning: a new function called g is generated and called from the original call site. Note that in c', the body of function g obtained by partially evaluating the body of f, we must rename the return variable f to g. E-Call3 handles calls to restricted sketches. These calls are not inlined, since in SKETCH, we wish to complete only one copy of these sketches. Also note that we evaluate the call to the specification of these sketches, whose behavior is known before synthesis is completed Analogously to E-Loop3, Rule E-Call3 non-deterministically terminates the inlining with an assertion that checks whether the partial evaluator performed potentially recursive inlining deep enough for all inputs.

$$\frac{\sigma(x) = \perp}{\langle x, \sigma\rangle \rightarrow \langle \perp, x\rangle} \qquad \frac{\sigma(x) \neq \perp}{\langle x, \sigma\rangle \rightarrow \langle \sigma(x), \sigma(x)\rangle} \tag{E-Var}$$

$$\frac{}{\langle n, \sigma\rangle \rightarrow \langle n, n\rangle} \tag{E-Lit}$$

$$\frac{\text{choose}() \Rightarrow n}{\langle \texttt{??}, \sigma\rangle \rightarrow \langle n, n\rangle} \tag{E-Hole}$$

$$\frac{\langle e_1, \sigma\rangle \rightarrow \langle n_1, e_1'\rangle \quad \langle e_2, \sigma\rangle \rightarrow \langle n_2, e_2'\rangle \quad n_1 \ \textbf{op}\ n_2 \Rightarrow n}{\langle e_1 \ \textbf{op}\ e_2, \sigma\rangle \rightarrow \langle n, n\rangle} \tag{E-Op1}$$

$$\frac{\langle e_1, \sigma\rangle \rightarrow \langle \perp, e_1'\rangle \ \langle e_2, \sigma\rangle \rightarrow \langle n_2, e_2'\rangle}{\langle e_1 \ \textbf{op}\ e_2, \sigma\rangle \rightarrow \langle \perp, e_1' \ \textbf{op}\ e_2'\rangle} \tag{E-Op2}$$

$$\frac{(\textbf{def}\ f(in)\ c) \ \ \langle e, \sigma\rangle \rightarrow \langle v', e'\rangle \quad \langle c, \sigma_\perp[in \mapsto v']\rangle \rightarrow \langle c', \sigma'\rangle \ \ \sigma'(f) \neq \perp}{\langle f(e), \sigma\rangle \rightarrow \langle \sigma'(f), \sigma'(f)\rangle} \tag{E-Call1a}$$

$$\frac{(\textbf{def}\ f(in)\ c) \ \ \langle e, \sigma\rangle \rightarrow \langle v', e'\rangle \quad \langle c, \sigma_\perp[in \mapsto v']\rangle \rightarrow \langle c', \sigma'\rangle \quad \sigma'(f) = \perp \quad \text{fresh}\ g}{\langle f(e), \sigma\rangle \rightarrow \langle \perp, g(e')\rangle \quad \text{emit}\ \textbf{def}\ g(in)\ c'[g/f]} \tag{E-Call1b}$$

$$\frac{\begin{array}{c}(\textbf{def}\ \text{f(in)}\ \textbf{implements}\ \text{g}\ c) \\ (\textbf{def}\ \text{g(in)}\ c_2) \\ \langle e, \sigma\rangle \rightarrow \langle v', e'\rangle \ \ \langle c_2, \sigma_\perp[in \mapsto v']\rangle \rightarrow \langle c_2', \sigma'\rangle\end{array}}{\langle f(e), \sigma\rangle \rightarrow \langle \sigma'(g), f(e')\rangle} \tag{E-Call2}$$

$$\frac{}{\langle f(e), \sigma\rangle \rightarrow \langle \perp, \textbf{assert false}\rangle} \tag{E-Call3}$$

$$\frac{\langle e, \sigma\rangle \rightarrow \langle v, e'\rangle}{\langle x := e, \sigma\rangle \rightarrow \langle x := e', \sigma[x \mapsto v]\rangle} \tag{S-Asgn}$$

$$\frac{\langle e, \sigma\rangle \rightarrow \langle \textbf{true}, e'\rangle \ \ \langle c_1, \sigma\rangle \rightarrow \langle c_1', \sigma_1'\rangle}{\langle \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \sigma\rangle \rightarrow \langle c_1', \sigma_1'\rangle} \tag{S-If1}$$

$$\frac{\begin{array}{c}\langle e, \sigma\rangle \rightarrow \langle \perp, e'\rangle \\ \langle c_1, \sigma\rangle \rightarrow \langle c_1', \sigma_1'\rangle \quad \langle c_2, \sigma\rangle \rightarrow \langle c_2', \sigma_2'\rangle \quad \sigma_u = \sigma_1' \cap \sigma_2'\end{array}}{\langle \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \sigma\rangle \rightarrow \langle \textbf{if}\ e'\ \textbf{then}\ c_1'\ \textbf{else}\ c_2', \sigma_u\rangle} \tag{S-If2}$$

$$\frac{\langle c_1, \sigma\rangle \rightarrow \langle c_1', \sigma_1'\rangle \ \ \langle c_2, \sigma_1'\rangle \rightarrow \langle c_2', \sigma_2'\rangle}{\langle c_1; c_2, \sigma\rangle \rightarrow \langle c_1'; c_2', \sigma_2'\rangle} \tag{S-Seq}$$

$$\frac{\langle e, \sigma\rangle \rightarrow \langle n, e'\rangle \ \ \langle \overbrace{c; \ldots; c}^{n}, \ \sigma\rangle \rightarrow \langle c', \sigma'\rangle}{\langle \textbf{loop}\ e\ \textbf{do}\ c, \sigma\rangle \rightarrow \langle c', \sigma'\rangle} \tag{S-Loop1}$$

$$\frac{\begin{array}{c}\langle e, \sigma\rangle \rightarrow \langle \perp, e'\rangle \\ S \stackrel{def}{=} \textbf{let}\ t = e'\ \textbf{in if}\ t > 0\ \textbf{then}\ \{c; \textbf{loop}\ t - 1\ \textbf{do}\ c\} \\ \langle S, \sigma\rangle \rightarrow \langle c', \sigma'\rangle\end{array}}{\langle \textbf{loop}\ e\ \textbf{do}\ c, \sigma\rangle \rightarrow \langle c', \sigma'\rangle} \tag{S-Loop2}$$

$$\frac{\langle e, \sigma\rangle \rightarrow \langle \perp, e'\rangle}{\langle \textbf{loop}\ e\ \textbf{do}\ c, \sigma\rangle \rightarrow \langle \textbf{assert}\ e' \leq 0, \sigma\rangle} \tag{S-Loop-3}$$

$$\frac{\langle e, \sigma_\perp\rangle \rightarrow (v, e') \ \ \langle c, \sigma_\perp\rangle \rightarrow (c', \sigma')}{\langle \textbf{while}\ e\ \textbf{do}\ c, \sigma\rangle \rightarrow \langle \textbf{while}\ e'\ \textbf{do}\ c', \sigma \cap \sigma'\rangle} \tag{S-While}$$

$$\frac{\langle c, \sigma_\perp\rangle \rightarrow \langle c', \sigma'\rangle \quad \text{fresh}\ h \quad (\textbf{def}\ h(in)\ c'[h/f]) \quad h \equiv g}{\langle \textbf{def}\ f(in)\ c\ \textbf{implements}\ g, \sigma_\perp\rangle \rightarrow \langle \textbf{def}\ f(in)\ c', \sigma'\rangle} \tag{S-Impl}$$

Figure 3. The partial evaluation rules for producing sketch completions in SKETCH.

Finally, rule S-impl translates definitions of restricted sketches. In contrast with other rules, this rule does not generate all possible completions of a sketch f. Instead, it selects a completion from $C(f)$ that behaves like the specification g. Formally, this selection assumes that the partial evaluator can guess hole values that will make the completed sketch behave suitably. The next section shows how the holes are computed in our system.

5. Synthesis of Holes

In this section, we describe how our synthesizer computes the values of holes. We first describe how this hole synthesizer collaborates with the partial evaluator defined in the previous section. Next, we phrase synthesis of holes as a quantified boolean satisfiability problem. We then show how SKETCH programs are translated to boolean functions suitable for this problem. Finally, we present a new counterexample-driven solver for the boolean satisfiability problem, as well as our scalability optimizations.

5.1 The SKETCH Compiler

In Section 4 we defined the semantics of sketches in terms of a partial evaluator which non-deterministically replaces holes with constant values that are then propagated through the program. In practice, the SKETCH compiler implements this transformation using a three-stage process. First, the partial evaluation algorithm in Figure 3 is invoked, slightly modified: the holes are replaced with free variables and evaluated to the unknown value $\perp$.

$$\frac{\text{fresh } c}{\langle \texttt{??}, \sigma\rangle \rightarrow \langle \perp, c\rangle} \ .$$

These free variables are added to the function parameters of the sketch and are referred to as *control inputs*.

Second, the synthesizer computes an assignment of values for these control inputs. After these computed values are assigned to the control inputs, a fully completed sketch is obtained. As an aside, it is in this step where a sketch may be shown to be buggy (i.e., it cannot be completed to behave like the specification) or an assertion may be found to fail (e.g., some loop should have been unrolled more). In the latter case, the first step must be repeated to unroll loops further.

The last step optimizes the completed sketch by partially re-evaluating it with Figure 3. At this point, all holes have been replaced with constants, allowing code improvements. In fact, the code produced will be the same as if the E-Hole rule in Figure 3 was magically able to guess a suitable value in the first step. An example of code improvement possible in the step: if a loop iteration count becomes known once the hole are filled in, a loop initially unrolled with E-Loop2 will be partially evaluated (with rule S-If1) into the more efficient form normally produced by rule E-Loop1.

5.2 Synthesis as QBF

The synthesizer represents SKETCH programs as boolean functions. Both specifications and sketches are translated to boolean functions with the same procedure. A specification with m input bits and n output bits is translated to a function $P : \{0,1\}^m \rightarrow \{0,1\}^n$. A matching sketch with k control inputs is translated to a function $S : \{0,1\}^{(m+k)} \rightarrow \{0,1\}^n$. As will be shown in Section 5.3, these boolean functions model program semantics without approximation.

Having obtained functional description of the specification and the sketch, we phrase synthesis as an instance of the Quantified Boolean Formula Satisfiability problem (QBF). QBF is a generalization of the boolean satisfiability problem (SAT) in which both existential and universal quantifiers can be applied to a boolean formula. Formally, the sketch S can be completed to the specification P if there exists a control c such that the specification and sketch are functionally identical, i.e., when the following formula is satisfiable.

$$\exists c \in \{0,1\}^k, \forall x \in \{0,1\}^m; P(x) = S(x,c) \qquad (5.1)$$

This problem is decidable, because c and x range over finite domains. Consequently, for any sketch expressed in our language, we can either complete the sketch or show that the sketch is buggy.

5.3 Programs as Boolean Functions

Our translation to boolean functions largely follows the standard approach used by other verifiers [3, 17]. Expressions computing k-bit integers are modeled as a boolean function that computes a k-bit output vector from the program inputs and controls.

To efficiently model array operations and switch statements, we represent some integer expressions with a less standard sparse encoding, detailed below. Each expression has a type; either `bin` if it is to be represented with the (binary) encoding, or `spar` if it is encoded sparsely. Currently, we assign these types syntactically, based on operations performed, but one can imagine a smarter selection of representations.

The translation of a program into its boolean representation computes a symbolic value for each program variable. The symbolic value is a boolean function over the inputs to the program. The boolean representation of a function is the symbolic value of its return variable. A symbolic state π maintains symbolic values of variables. We describe the translation of SKETCH programs into boolean functions with rewrite rules in the following notation: $\langle p, \pi\rangle \rightarrow \langle b, \pi'\rangle$ means that a program fragment p in state π produces a boolean function b in state π'. Since expressions do not affect the state, we abbreviate to $\langle e, \pi\rangle \rightarrow b$. Similarly, since statements do not produce values, we abbreviate to $\langle c, \pi\rangle \rightarrow \pi'$.

We first discuss the translation of a function call. Assume function `f` has parameters $in_1, \ldots, in_m$; recall that this function returns the last value assigned to variable `f` in the body of `f`.

$$\frac{\texttt{def } f(in_1, \ldots, in_m)\ c \qquad \langle c, [in_1 \mapsto \pi(x_1), \ldots, in_m \mapsto \pi(x_m)]\rangle \rightarrow \pi'}{\langle f(x_1, \ldots, x_m), \pi\rangle \rightarrow \pi'(f)}$$

There is an analogous rule for the case where `f` is a restricted sketch; in that case, instead of translating the body of `f`, as in rule E-Call2 in Figure 3, we translate the body of its specification. The specification has the same behavior and its body contains no holes, which simplifies matters.

To translate a definition of a function, we invoke the rule for translating a function call but we initialize π to functions that map parameters to themselves. For example, given

```
def f(x,y) {x:=y; f:=x+y;}
```

we set π to $[x \mapsto \lambda x\, y.x, y \mapsto \lambda x\, y.y]$. After the call is translated, the function $\pi'(f) = \lambda x\, y.y + y$, expressed as a boolean circuit, is the function representation of function f.

5.3.1 Arrays and sparse integer expressions

The sparse integer representation is primarily intended to efficiently model array operations. The sparse encoding is very similar to guarded location sets used by Saturn [17] to represent pointers (Saturn does not model arrays). The sparse encoding represents an integer as a set of *guarded values* of the form (v, b), where v is an integer constant and the *guard* b is a boolean function over input variables. If b is true, then the integer has value v. We maintain an invariant that for all inputs at most one guard will be true. Similarly, we maintain uniqueness of the guarded values by taking a disjunction of terms guarding the same value.

Let us assume that $\mathbf{x} = in_1, \ldots, in_m$ is a vector of program inputs. An integer constant n represented sparsely uses a guard function that returns true on any input:

$$\frac{}{\langle n : \texttt{spar}, \pi\rangle \rightarrow [(n, \lambda\, \mathbf{x}.\mathbf{true})]}$$

Arithmetic expressions are handled by applying the arithmetic operation on pairs of values from the two operands and guarding them with the conjunction of their respective conditions.

$$\frac{\langle e_1 : \texttt{spar}, \pi\rangle \rightarrow [(v_1^1, b_1^1), \ldots (v_k^1, b_k^1)] \quad \langle e_2 : \texttt{spar}, \pi\rangle \rightarrow [(v_1^2, b_1^2), \ldots (v_l^2, b_l^2)]}{\langle e_1 \text{ op } e_2 : \texttt{spar}, \pi\rangle \rightarrow [(v_i^1 \text{ op } v_j^2, b_i^1 \wedge b_j^2)]}$$

It is useful to note here that the sparse representation makes it easy to discover constants. If an expression $e : \texttt{spar}$ does not depend on any inputs or controls, then $e = [(n, \lambda\, \mathbf{x}.\mathbf{true})]$ for some constant n. (In practice, we only do trivial minimizations on the guards as we construct them, so the sparse representation for a constant could be $e = [(n_1, f_1), (n_2, f_2), \ldots (n_k, f_k)]$, where some f_i is a tautology while the remaining guards are unsatisfiable.)

The conversion from binary to sparse representation incurs a potentially exponential size explosion since the sparse form is, in the worst case, a unary representation.

$$\frac{\langle e : \texttt{bin}, \pi\rangle \rightarrow [b_1, \ldots, b_k]}{\langle e : \texttt{bin}, \pi\rangle \rightarrow [(i, \lambda \mathbf{x}.e(\mathbf{x}) = i) \mid \forall i \in \{0 \ldots 2^k - 1\}] : \texttt{spar}}$$

To alleviate the exponential explosion, one can examine how the sparse value is to be used. For example, if the sparse integer is used only to index an n-element array, then the sparse representation needs to maintain at most $n + 1$ terms: n terms for the array elements and a term to discover out-of-bounds accesses. (Here, we generalize the sparse representation somewhat in that the latter term represents not a single constant but all out-of-bounds index values.)

We are now ready to translate array accesses. An n-element array is modeled as a (dense) tuple of n boolean functions, which can be either all in the binary form

$$\texttt{ar} : \texttt{bin} = [[b_1^1, \ldots, b_k^1], \ldots, [b_1^n, \ldots, b_k^n]]$$

or all in the sparse form

$$\texttt{ar} : \texttt{spar} = [[(v_1^1, b_1^1), \ldots, (v_{l_1}^1, b_{l_1}^1)], \ldots, [(v_1^n, b_1^n), \ldots, (v_{l_n}^n, b_{l_n}^n)]]$$

Index expressions are always in sparse form.

Given an index expression $e : \texttt{spar}$ and an array $\texttt{ar} : \texttt{bin}$, the translation produces $\texttt{ar}[e] : \texttt{bin}$. The translation proceeds bitwise: the j-th bit in $\texttt{ar}[e] : \texttt{bin}$ will be true if the j-th bit of $\texttt{ar}[p]$ is true, and e evaluates to p. The value of e will equal p if for some i, the pair (x_i, f_i) in the sparse representation of e has $x_i = p$ and f_i evaluates to true.

$$\frac{\langle e : \texttt{spar}, \pi\rangle \rightarrow [(x_1, f_1), \ldots (x_l, f_l)] \quad \pi(\texttt{ar}) = [[b_1^1, \ldots, b_k^1], \ldots, [b_1^n, \ldots, b_k^n]]}{\langle \texttt{ar}[e] : \texttt{bin}, \pi\rangle \rightarrow [(\bigvee_{i \le l}(b_1^{x_i} \wedge f_i)), \ldots, (\bigvee_{i \le l}(b_k^{x_i} \wedge f_i))]}$$

Given an index expression $e : \texttt{spar}$ and an array $\texttt{ar} : \texttt{spar}$, the translation produces $\texttt{ar}[e] : \texttt{spar}$. To understand the translation, shown below, consider the conditions under which a value v_i^p stored in the array becomes the value of $\texttt{ar}[e]$. (1) Since the value v_i^p belongs to the values of the sparse array element $\texttt{ar}[p]$, the sparse index expression e must be able to evaluate to p. In other words, $e : \texttt{spar}$ must contain a value $x_j = p$. (2) The guard function f_j of x_j must evaluate to true, meaning that we indeed want to read the element $\texttt{ar}[p]$. (3) To ensure that v_i^p, is actually the value of $\texttt{ar}[p]$, its guard function $b_i^{x_j}$ must evaluate to true. To obtain the translation, these conditions are collected for all values that $\texttt{ar}[p]$ can evaluate to (iteration over all i's), as well as for all possible index values (iteration over all j's).

$$\frac{\langle e : \texttt{spar}, \pi\rangle \rightarrow [(x_1, f_1), \ldots (x_k, f_k)] \quad \pi(\texttt{ar}) = [[(v_1^1, b_1^1), \ldots, (v_{l_1}^1, b_{l_1}^1)], \ldots, [(v_1^n, b_1^n), \ldots, (v_{l_n}^n, b_{l_n}^n)]]}{\langle \texttt{ar}[e] : \texttt{spar}, \pi\rangle \rightarrow [(v_i^{x_j}, b_i^{x_j} \wedge f_j) \mid 1 \le j \le k, 1 \le i \le l_{x_j}]}$$

We can now illustrate why the sparse representation models common array operations more efficiently than the binary representation. There are two reasons for this, and both have to do with the way arrays are used in programs. The first reason is that array index expressions often range over a small set of values, a property not exploited by the binary representation. For example, the following statements appear in one of the benchmarks we studied:

```
tmp = (i+??)%4;
idx = input[(tmp)*4+j];
```

In the benchmark, i and j are constant so the index can take only four different values, but that information would be lost if the expressions were represented in bit-vector form.

The second reason is that when arrays are used in loops, their index is often the sum of a loop invariant expression and an induction variable (which after unrolling the loop becomes a constant).

```
ofst = ??;
for(int i=0; i<N; ++i){
  out[i] = (ofst+i)<N ? in[i+ofst] : 0;
}
```

This pattern is handled very efficiently by the sparse values. For example, in the code above `out[i]` evaluates to:

$$(f_0 \wedge in[i]) \vee (f_1 \wedge in[i+1]) \vee \ldots \vee (f_{n-i-1} \wedge in[n-1])$$

The f_i are the guard variables of the sparse `ofst`. If `ofst` were represented in binary, each access to array **`in`** would involve evaluating an expensive boolean function. The function would compute `ofst+i` with an adder circuit and then perform the selection of the array element. With the sparse representation, the guard functions are computed only once for each assignment to `ofst`. Also, the addition of constants such as `i` is free, since they are folded into the guarded values.

5.3.2 Conditionals and Loops

The treatment of conditionals is fairly standard; we convert them into predicated code, as in [3, 17]. Loops are also handled in essentially the same way as in [3, 17]. They are unrolled by a fixed factor and assertions are inserted to ensure that the loop was unrolled sufficiently. The rules in figure 3 already unrolled the **`loop`** construct, but when generating boolean functions, all remaining kinds of loops need to be unrolled, too.

An interesting point is that even though our handling of loops is fairly standard, the sparse representation of integers leads to more efficient representations for loops, particularly for the **`loop`** construct. This is because when such loops get unrolled, they produce **`if`** statements with conditions of the form $t - i > n$. These conditions have very small boolean representations when t is a sparse integer and n and i are constants. In fact, when t is sparse, we can often statically determine branch directions, even if we cannot determine the precise value of t. For example, if we know $t = [(0, v1)(3, v1)]$, then if we also know that $t > 0$, it must be that $t = 3$. This way, we can statically eliminate most of the branches produced when generating functions for **`loop`**`(t) c`.

5.3.3 A Simple Example

As an example of the application of these rules, consider the sketch:

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

Then, after applying the rules in Figure 3, the function becomes

```
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;
}
```

Assume the unknowns are fixed at two bits (see Section 5.5). Then we have that

$$\texttt{c1} \rightarrow [\hat{c_1}, \hat{c_2}]$$

When **??** is converted to sparse form, we get that **??** becomes

$$t_0 = [(0, \neg\hat{c_1} \wedge \neg\hat{c_2}), (1, \hat{c_1} \wedge \neg\hat{c_2}), (2, \neg\hat{c_1} \wedge \hat{c_2}), (3, \hat{c_1} \wedge \hat{c_2})]$$

and the expressions `t1>0` and `t2>0` become $(\hat{c_1} \wedge \neg\hat{c_2}) \vee (\neg\hat{c_1} \wedge \hat{c_2}) \vee (\hat{c_1} \wedge \hat{c_2})$ and $(\neg\hat{c_1} \wedge \hat{c_2}) \vee (\hat{c_1} \wedge \hat{c_2})$ respectively .

5.4 Counterexample-Driven Solver

Problem (5.1) is decidable but intractable. In general, QBF is PSPACE-complete, and can be solved in time exponential in the number of quantified variables. However, Problem (5.1) is a restricted form of QBF with only one quantifier alternation (of the form $\exists\ \forall$), a problem known as 2QBF. The computational complexity of this problem is Σ_2-complete, falling in the polynomial hierarchy between NP and PSPACE.

Our solver relies on two SAT solvers, co-operating in a synthesize-verify loop. First, a random input x is generated and the synthesizing solver attempts to find control c that makes the sketch equal to the specification on the input x. If such control cannot be found, the sketch is buggy. Otherwise, the control c is given to the verifying solver, which attempts to verify that the sketch is equivalent to the specification on all inputs. If so, the sketch can be completed and control c is the result of the synthesis. Otherwise, a counterexample input provided by the verifier is added to the set of inputs considered by the synthesizer and the process repeats. The algorithm is shown in Figure 4.

The algorithm will terminate because, in the worst case, each of the 2^m inputs will appear as a counterexample, at which point the synthesizer's answer no longer needs to be verified. This reduction of 2QBF to two SAT solvers does not come free: the algorithm requires more than polynomial space but the trade-off is that we can employ the efficient techniques embedded in modern SAT solvers. Others have used two cooperating SAT solvers to solve 2QBF problems before [11]; however, none of the solvers presented in [11] uses counterexamples to guide the synthesis of controls in the way we do.

5.5 Scalability Optimizations

In order to improve scalability, our solver performs the following optimizations.

```
function synthesize(sketch S, specification P)
    // Synthesize control that completes S for a random input;
    // check if the control also works for all other inputs. If not,
    // add counterexample input to the 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(set of inputs I)
    // Synthesize controls c that make the sketch equivalent to the
    // specification on all inputs from I, i.e., ∀x ∈ I.P(x) = S(x, c)
    if ⋀_{x∈I} P(x) = S(x, c) is satisfiable then
        return c that satisfies the formula
    else // sketch S cannot be completed
        return nil
    end if
function verifyForAllInputs(control c)
    // Verify if sketch S completed with controls c is functionally
    // equivalent to the specification P. If not, return the witness
    // (counterexample) x, i.e., P(x) ≠ S(x, c).
    if P(x) ≠ S(x, c) is satisfiable then
        return x satisfying the formula
    else
        return nil
    end if
```

Figure 4. The counterexample-driven synthesis algorithm.

Increasing Ranges of Holes. The translation of some language constructs leads to exponentially large boolean functions, for example in `loop(??)`, where the loop is controlled by a value unknown at the time of translation. Our solution is to initially restrict the range of holes and attempt synthesis. When synthesis fails, we re-translate the sketch with a larger range.

Specialization of the Boolean Formula For each new counterexample input x_i that we add to the SAT problem that handles the synthesis, we have to add a new set of clauses corresponding to $P(x_i) = S(x_i, c)$. However, when generating these clauses, the input is a known constant, so we can specialize the formula, producing many fewer clauses than what we would need to represent $P(x) = S(x, c)$ for an arbitrary input x.

ABC For most SAT problems, we use the MiniSAT [4] solver. However, for the hardest problems our solver can switch to ABC [10], a system which does logic optimization based on And-Inverter Graphs originally designed for circuit verification and optimization. For the hardest problems we've encountered, ABC provides order of magnitude improvement over the standard SAT solver.

6. Evaluation

To evaluate our system, we coded a set of small benchmarks from various domains, including networking, cryptography, and coding. The benchmarks are described in Section 6.1. Section 6.2 describes the performance of the solver and analyzes the factors that determine the solving time. Subsection 6.3 takes the `pop` benchmark and shows how the user can improve the running time for the solver by providing the compiler with additional information about the contents of holes. Finally, subsection 6.4 shows a case study of the SKETCH solver for a real benchmark, the AES encryption standard, and describes how the SKETCH compiler is able to incorporate multiple sketches to produce a single implementation. The

section also shows that the code produced by the SKETCH compiler is competitive with a real hand-tuned implementation of the benchmark available in the public domain.

6.1 The Benchmarks

The following benchmarks were used to test our solver.
log2 computes the log of a number using only $log(n)$ operations for an n bit word, by performing what amounts to a binary search. The sketch leaves unspecified a loop iteration count as well as two masks and a shift amount in the body of the loop.
pop is the population count benchmark described in Section 3.
parity computes the parity of an n-bit word in a divide-and-conquer fashion using $log(n)$ operations. The sketch leaves unspecified a loop iteration count, and a shift amount in the loop body.
reverse reverses the order of the bits in an n-bit word in a divide-and-conquer fashion using $log(n)$ operations. The sketch contains a loop with an unspecified iteration count, and in the body of the loop is a statement `t = ((t >> s)& m) | ((t << s)&~m)`, where `m` is set with a hole on each iteration, and `s` is multiplied by an unknown value on each iteration.
crc is a widely used error detection code based on division of polynomials. The input message is interpreted as coefficients $mod\ 2$ of a polynomial $M(x)$. The checksum for the message is defined as the reminder of dividing this polynomial by a fixed polynomial $K(x)$ of degree n.

This benchmark is interesting because it is an example of a streaming program. It is technically not finite, because it manipulates a message of unbounded size, but it was easily refactored into an outer loop which iterates over the entire message and a finite procedure which takes the current checksum and the next word of input, and produces a new checksum. The sketch uses a well-known trick to produce a table-based implementation. Our sketch has a parameter S, which determines the number of tables.
poly This is the polynomial example from section 3.
karat is the karatsuba multiplication algorithm from section 3.

6.2 Synthesizer Performance

Table 1 summarizes the completion times for the benchmarks described above on a 1.3 GHz Pentium 3 with 1GB of RAM. All benchmarks were run using 5 bits for each integer hole and 8 as the default unroll factor for loops. The table lists the number of bits for both the input and the controls, and the number of nodes in the boolean dag that represents the sketch and the spec after partial evaluation. The table also lists data about the behavior of the solver, including the number of iterations of synthesis and verification, and the total time spent on these two.

The first observation that jumps out from the data is that the solution time for most benchmarks is dominated by the synthesis time, as opposed to verification. This is expected considering most benchmarks have a lot more controls than inputs, so the search space for synthesis is much larger. The predominance of the synthesis time also explains the very weak correlation between the solution time and the size of the spec. This is because the synthesis phase searches for controls that work for a given set of inputs, so in this phase the spec is completely evaluated away into a set of output values. This makes the internal structure of the spec irrelevant for the synthesis time. The fact is of practical importance because users should not attempt to modify their specifications to please the solver.

Two exceptions for which verification time dominates are `crc` and `poly-15`. `poly` is a challenging benchmark involving addition and multiplication, which are known to cause problems for SAT solvers, so the what is surprising is not that verification is slow but that synthesis is so fast. This is likely due to the fact that even

Bench	In	ctrl	spec	sketch	Iter	Synth	Verify
crcS2	32	512	816	1208	25	2.28	77.90
crcS4	32	1024	816	1396	61	6.10	65.20
crcS8	32	8192	816	8402	511	223.52	589.04
crcS2	16	128	216	316	13	0.08	0.27
crcS4	16	256	216	362	31	0.18	0.53
crcS8	16	2048	216	2113	255	8.22	22.78
kar	4	59	22	427	7	0.52	0.02
kar	8	61	131	950	8	21.55	0.19
kar	12	63	324	1540	8	51.70	1.62
log2	4	109	24	501	7	0.07	0.02
log2	8	173	126	957	17	1.06	0.12
log2	16	301	570	1869	25	70.75	0.71
log2	24	429	1334	2781	52	1006.93	2.72
pop	4	77	35	478	8	0.08	0.02
pop	8	109	149	966	13	1.10	0.09
pop	16	173	617	1942	21	1016.93	2.27
parity	4	45	8	178	4	0.01	0.00
parity	8	45	16	266	7	0.02	0.01
parity	16	45	32	442	10	0.24	0.10
parity	32	45	64	794	24	52.00	3.12
poly	5	30	322	834	3	0.06	0.09
poly	10	60	1527	3739	5	0.35	1.56
poly	15	90	3632	8744	4	0.85	97.93
rev	16	138	32	1039	14	16.86	2.10
rev	32	266	64	2063	27	28.23	2.50
rev	64	522	128	4111	59	90.88	8.15

Table 1. Synthesis and verification time for selected benchmarks.

though we are synthesizing 15-bit integers, it is easy for the solver to discover that the 12 most significant bits of each of these should be zero. In CRC, controls also dominate inputs, but the unknowns, which correspond to table entries, are to a large degree independent, which implicitly decomposes the search.

Another interesting observation is the very strong correlation between the number of iterations and the number of unknowns. This roughly linear correlation seems of limited practical importance, though, since there is little correlation between the number of iterations of the solver and its solution time. However, it is interesting to realize that the number of counter-example inputs needed to find controls is not proportional to the size of the input space but to the number of bits of controls.

6.3 Impact of Holes on Scalability

In this section, we use the `pop` benchmark with input size 16 to analyze how the user can improve the solution time by providing additional information about the holes present in the benchmark.

The original benchmark contains four holes: one for the loop bound, one for the shift amount and two for the masks. For our experiments, we recorded the solution time for the original benchmark, and then we recorded how the solution time varied when we added extra information in the following way:

- By specifying the loop bound fully (s) instead of leaving it unspecified (u).
- By fully specifying the masks (0), or by stating that on each iteration the two masks contain the same value (1), instead of leaving them as two independent holes (2).
- By specifying the shift ammount fully (f), or partially (p), instead of leaving it unspecified (u). The shift is specified partially by stating that it is initially one, and on each iteration `shift=shift*??`.

Table 2 summarizes the solution times for the experiments. For these experiments, loops were unrolled by 4 by default, and the size of holes was not determined a priori, but was set to grow as

loop	mask	shift	iters	synth	verify	total
u	2	u	31	1588	5.80	1593.80
u	2	p	29	16.50	5.32	21.80
s	2	u	32	311.80	7.14	318.96
u	1	u	16	173.06	4.93	177.99
u	2	f	28	14.86	5.31	20.18
u	1	f	12	0.30	8.97	9.31
s	1	f	10	0.28	2.36	2.64
s	1	p	12	0.37	3.80	4.24
s	2	p	25	8.21	5.20	13.41
s	0	u	3	0.52	10.63	11.16

Table 2. Running times for population count with varying levels of unknowns. Synth corresponds to the time spent in synthesis and verify is the time spent of verification.

```
int[4] roundSK(bit[32][4] in, bit[32][4] rkey)
    implements round{
  int[4][256] T = ??;
  int[4] output = 0;
  bit[32] mask = 0x000000FF;
  int[4][4] ch = {{0,1,2,3},{1,2,3,0},
                  {2,3,0,1},{3,0,1,2}};
  for(int i=0; i<4; ++i){
    int i0 = (int) in[ch[i][0]] &  mask;
    int i1 = (int)(in[ch[i][1]] >> 8) & mask;
    int i2 = (int)(in[ch[i][2]] >> 16) & mask;
    int i3 = (int)(in[ch[i][3]] >> 24) & mask;
    output[i] = T[0][i0] ^ T[1][i1]
              ^ T[2][i2] ^ T[3][i3];
    output[i] = output[i] ^ rkey[i];
  }
  return output;
}
```

Figure 5. Sketch for one round of AES.

needed (see Section 5.5). The table does not contain all possible combinations of the optimizations, because some of them were not possible—in specifying the mask we also had to specify the number of iterations—and others were not very interesting because their solution times were already quite small—such as (s,0,p) or (s,0,f).

The most striking observation is how by adding even partial information about the shift amounts, the solution time is reduced dramatically. For this benchmark, adding information about the shifts is more valuable than for either the masks or the number of iterations. For example, (u,2,p) is an order of magnitude faster than (s,2,u).

Partial information about the masks is also quite valuable, as one can see from comparing (u,2,u) with (u,1,u), although not as much as the shift amounts. The masks, however, have a direct impact on the number of iterations of the solver, reinforcing the observation made earlier on the relationship between control bits and number of iterations.

6.4 Case Study: AES

As a case study, we used our system to create a full implementation of the AES cipher [5]. The core of the cipher consists of 14 rounds which take a 128-bit input block and a round key and processes it, followed by a final round.

```
bit[W] round(bit[W] in, bit[W] rkey){
  bit [W] t1 = ByteSub(in);
  bit [W] t2 = ShiftRows(t1);
  bit [W] t3 = MixColumns(t2);
  return t3 ^ rkey;
}
```

Total Synth:	791 sec	= 13.183 min
Total Verify:	3942 sec	= 65.7 min
Synth easy:	1.17 sec	avg time per SAT problem
Synth hard:	3.4 sec	avg time per SAT problem
Verify easy:	5.33 sec	avg time per SAT problem
Verify hard:	50 sec	avg time per SAT problem

Table 3. Solution time for roundSK in AES benchmark. The times for Synth and Verify hard correspond to the times for the last 10 iterations.

The `ByteSub` transformation performs a set of table lookups to do a substitution on each byte; `ShiftRows` does a permutation of the bytes in the block; and `MixColumns` treats each word as a 4 element vector in the Galois field $GF(2^8)$, then transforms it by multiplying it with a matrix whose elements are also in $GF(2^8)$. The final round is like the other rounds but without the `MixColumns` transformation.

In the optimized version, all the operations in the round are folded into a set of table lookups. A programmer implementing AES by traditional means would have to figure out the formula for generating the table entries. This may be difficult if one is not familiar with the algebra involved, and one must then write a generator for the table from the specification, incorporate it into the code, and check the correctness of the cipher using known input/output pairs. In our approach, the tables are completely synthesized and correctness is guaranteed. Figure 5 shows the sketch for the regular round. The sketch for the final round is similar, except it uses only one table instead of four, and it combines outputs from the tables using masks—which are left unspecified— instead of xors.

The `roundSK` sketch places a lot of stress on the solver since there are 32768 bits in the table that have to be generated. Furthermore, each input considered by the solver helps complete only a small number of table entries, so the synthesize/verify loop has to iterate 655 times. Nonetheless, the solver is able to complete the sketch in about an hour. Table 3 shows the exact times spent by the two SAT solvers involved. All instances of synthesis were solved using MiniSat. For verification, we used MiniSat for the first 645 iterations. For the last 10 iterations we switched our SAT solver to ABC [10] because it provides much better performance for hard SAT problems.

Performance of generated code. The resulting code was run against a hand optimized AES implementation from open SSL. The runtime for 50000 encryptions was as follows:

OpenSSL AES	19.652 ms
Sketch	21.307 ms
Spec	19936.100 ms

The difference between the hand coded AES and the sketched version is less than 10%. We can also see that the original specification, which is very close to the specification of AES [5], is over 1000 times slower.

7. Related Work

StreamBit. This submission attacks programmability challenges observed in StreamBit [13]: (**1**) In StreamBit, programmers could not express sketches directly in the implementation language. Instead, they had to sketch the desired implementation by means of meta-level rewrite rules that translated the specification into the desired implementation. (**2**) The implementation strategy had to be

often decomposed hierarchically into multiple sketches with onerous dependences. For example, the sketch specifying word-level parallelism had to plan the implementation carefully so that the sketch for bit-level parallelism would apply. (**3**) Sketches had to be inserted into the rewrite sequence of a baseline compiler. The awareness of the baseline compiler made the meta-level nature of rewrite rules even more confounding. (**4**) Sketching was embedded into a dataflow programming language [14]. While the dataflow programming model helped synthesis and subsequent parallelization, novice programmers faced sketching simultaneously with another new programming model.

We present a combinatorial synthesizer as well as linguistic support that sidesteps these four issues. First, sketches are expressed as *partial programs*, or programs with "holes." As a result, sketches are not meta-rules but straightforward code templates. Second, the desired implementation can now be sketched in a single sketch, without decomposition. Third, there is no baseline compiler to cooperate with. Finally, sketching is embedded into an imperative language with a familiar programming model.

Besides programmability limitations, sketching in [13] was also restricted in expressiveness. (**1**) Except for some high-level refactorings, sketching worked primarily by decomposing (semi)-permutations of bit-vectors. While this addressed the main performance bottlenecks of most ciphers–rearranging bits so there is better use of word-level parallelism–it precluded some optimizations that required use of boolean identities. (**2**) The sketched implementations themselves could not implement permutations using non-permutations instructions, such as additions.

Synthesis with Partial Programs. Synthesis based on partial programs has been explored in the AI community. For example, ALisp [1], developed by Andre and Russell to program Reinforcement Learning Agents is a form of Lisp extended with non-deterministic constructs. In ALisp, the behavior of the non-deterministic branches is defined through learning, a domain-specific approach.

The sketch completion problem is a constraint satisfaction problem similar to those studied by the constraint programming community [7].

Schema-based program synthesizers automatically compile a high-level declarative specification into code; an example is the AUTOBAYES system which compiles a statistical model into code [6]. However, these synthesizers are highly domain-specific and are not based on a general formal notion of partial programs as introduced in this paper.

The Denali superoptimizer [8] was one of the first systems to leverage the progress in SAT solving for code optimization. Denali is still a classical transformational synthesizer, as its use of SAT solvers is restricted to optimizing and scheduling of the synthesized code. Also, Denali focused on optimizing straight-line code. While our system does not look for the optimal way to resolve a sketch, it applies to a more general class of programs.

Many recent program verification projects are also SAT-based, including CBMC [3] and Saturn [17]. Our work uses SAT solving not just for program verification, but also for program synthesis.

8. Conclusion

We have designed a language that supports sketches in a natural way, designed a general solver for synthesizing sketched implementations, and evaluated the generality of the linguistic support as well as the scalability of the solver. We implemented the AES cipher standard; the sketch is surprisingly concise, the solver scales well on this real benchmark, and the generated code runs almost as fast as a good hand-coded implementation.

Acknowledgment

We are grateful to Manu Sridharan, Gilad Arnold, Mooly Sagiv, and the anonymous referees for their helpful comments. David Turner contributed to benchmark development. This work is supported in part by the National Science Foundation with grants CCF-0085949, CCR-0105721, CCR-0243657, CNS-0225610, CCR-0326577, and CNS-0524815, the University of California MICRO program, the MARCO Gigascale Systems Research Center, an Okawa Research Grant, and a Hellman Family Faculty Fund Award. This work has also been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056. The views expressed herein are not necessarily those of DARPA.

References

[1] D. Andre and S. Russell. Programmable reinforcement learning agents. *Advances in Neural Information Processing Systems*, 13, 2001. MIT Press.

[2] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[3] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, pages 368–371, May 2003.

[4] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[5] Advanced encryption standard (AES). U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology, November 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[6] B. Fischer and J. Schumann. Autobayes: a system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 13(3):483–508, May 2003.

[7] P. V. Hentenryck and V. Saraswat. Strategic directions in constraint programming. *ACM Comput. Surv.*, 28(4):701–726, 1996.

[8] R. Joshi, G. Nelson, and K. H. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, pages 304–314, 2002.

[9] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.

[10] A. Mishchenko, S. Chatterjee, and R. Brayton. Dag-aware AIG rewriting: A fresh look at combinational logic synthesis. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 532–535, New York, NY, USA, 2006. ACM Press.

[11] D. P. Ranjan, D. Tang, and S. Malik. A comparative study of 2qbf algorithms. In *The Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, May 2004.

[12] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms – Theory and Practice*. Prentice-Hall, 1977.

[13] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 281–294, New York, NY, USA, 2005. ACM Press.

[14] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr. 2002.

[15] H. S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[16] P. Wegner. A technique for counting ones in a binary computer. *Commun. ACM*, 3(5):322, 1960.

[17] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 351–363, 2005.